HAUFE.
Ausgabe 05/2024

ACCESS
IM UNTERNEHMEN

HALBAUTOMATISCHE FEHLERBEHANDLUNG

Lernen Sie, wie Sie per Knopfdruck Fehlerbehandlungsroutinen und Zeilennummern zu Prozeduren hinzufügen (ab Seite 19).

In diesem Heft:

DATENBANK-VERSIONEN UNTER SQL SERVER IM GRIFF

Finden Sie Unterschiede zwischen Datenbankversionen mit den SQL Server Data Tools

SEITE 52

SEITENRÄNDER IN BERICHTEN

Manipulieren Sie Seitenränder in Berichten, um beispielsweise spiegelverkehrte Seitenränder zu realisieren.

SEITE 5

FLEXIBLE SCHNELLSUCHE

Fügen Sie Access mit einem COM-Add-In eine flexible Schnellsuche für die Datenblattansicht hinzu.

SEITE 65

Mat-Nr. 01583-5145

Fehlerbehandlung per Mausklick

Das Hinzufügen einer Fehlerbehandlung zu den Prozeduren des VBA-Projekts einer Access-Datenbank ist obligatorisch – spätestens, wenn man die Anwendung an andere Benutzer weitergibt und diese nicht mit kryptischen Fehlermeldungen konfrontieren möchte. Da das Schreiben der Codezeilen für die Fehlerbehandlung aufwendig ist, macht man dies tendenziell eher nur dort, wo es unbedingt nötig erscheint. Damit zumindest der Aufwand nicht mehr als Ausrede herhalten kann, stellen wir in dieser Ausgabe kleine Lösungen vor, mit denen Sie Ihre Prozeduren per Mausklick um Fehlerbehandlungen und Zeilennummern anreichern können.

Dem Thema Fehlerbehandlung widmen wir uns gleich in vier Beiträgen. Der erste zeigt, wie man eine Fehlerbehandlung per VBA-Code zu einer Prozedur hinzufügen kann. Dabei braucht man nur die entsprechende Prozedur zu markieren und die Routine zum Hinzufügen einer individuell auf die Prozedur angepassten Fehlerbehandlung aufzurufen. Wie das gelingt, beschreiben wir unter dem Titel **Fehlerbehandlung per VBA hinzufügen** ab Seite 19.

Wichtig zur Lokalisierung des Fehlers sind Zeilennummern. Diese müssen wir tatsächlich manuell hinzufügen – gut, dass wir im Beitrag **Zeilennummern per VBA hinzufügen** ab Seite 28 zeigen, wie das ebenfalls per Mausklick gelingt.

Mit der Fehlerbehandlung können wir Fehlermeldungen plus Fehlerort protokollieren. Passend dazu zeigen wir in **Fehlerhafte Zeilen anzeigen lassen** ab Seite 38, wie wir die so vorliegenden Fehler direkt an der entsprechenden Stelle im Code anzeigen können, um diesen zu untersuchen.

Schließlich liefern wir noch eine Strategie, um auch in Anwendungen ohne umfassende Fehlerbehandlung schnell herauszufinden, wo sich Fehler verbergern, die nur eine nichtssagende Meldung liefern: **Fehler in der Runtime von Access finden** ab Seite 42.

In den beiden Beiträgen **Seitenränder in Access-Berichten** (ab Seite 5) und **Bericht mit unterschiedlichen Seitenrändern** (ab Seite 11) beschreiben wir, wie wir per VBA die Seitenränder von Berichten beeinflussen können.

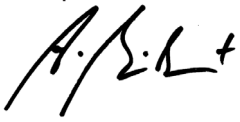
Rund um das Thema SQL Server drehen sich die folgenden drei Beiträge. In **SQL Server Data Tools installieren und starten** zeigen wir ab Seite 52, wir Sie die SQL Server Data Tools installieren und starten können

Im Beitrag **SQL Server-Datenbank kopieren** ab Seite 56 zeigen wir, wie Sie direkt im SQL Server komplette Datenbanken kopieren können.

Die Kopien nutzen wir wiederum, um im Beitrag **Schemata von SQL Server-Datenbanken vergleichen** ab Seite 60 zu zeigen, wie man mit den SQL Server Data Tools die Unterschiede zwischen zwei SQL Server-Datenbanken untersuchen kann.

Schließlich liefern wir noch ein praktisches Tool, das sich nahtlos in die Benutzeroberfläche von Access integriert und mit dem wir komfortabel in der Datenblattansicht von Access-Objekten suchen können. Das alles erfahren Sie im Beitrag **Flexible Schnellsuche** ab Seite 65.

Viel Spaß beim Lesen!



Ihr André Minhorst

Optionen einfach in der Registry speichern

In einem weiteren Beitrag namens »Registryeinträge für VBA-Anwendungen« (www.access-im-unternehmen.de/1508) haben wir dir grundlegenden Techniken für das Speichern von Anwendungsdaten in der Registry vorgestellt. Im vorliegenden Beitrag gehen wir noch einen Schritt weiter und vereinfachen diesen Vorgang, sodass die Befehle zum Lesen und Schreiben der Daten noch einfacher werden. Das Verwalten von Informationen wie beispielsweise von Anwendungsdaten in der Registry ist eine Alternative zum Verwenden einer Optionentabelle oder auch einer Textdatei im Anwendungsverzeichnis. Je nachdem, an wie vielen Stellen man lesend oder schreibend auf diese Daten zugreift, möchte man den Zugriff auf die Registry möglichst einfach gestalten. Dazu stellen wir nachfolgend ein paar geeignete Werkzeuge vor.

Die im obigen Artikel vorgestellten Befehle dienen dazu, Einträge zu erstellen, zu aktualisieren, zu lesen oder zu löschen. Die Befehle zum Schreiben und Lesen der Einträge haben bis zu vier Parameter.

Die ersten beiden davon wiederholen sich vermutlich für die meisten oder sogar für alle Aufrufe.

Die Parameter **AppName**, **Section**, **Key** und **Setting** braucht man normalerweise zum Anlegen, aber für die Einstellungen einer einzigen Anwendung ist zumindest der Wert für **AppName** immer gleich. Und wenn die Anwendung nicht allzu umfangreich ist, sodass man mehrere Bereiche für die Registry-Einträge benötigt, verwendet man auch immer den gleichen Namen für den Parameter **Section**.

In der Lösung dieses Beitrag machen wir nicht viel mehr, als für die immer gleich bleibenden Parameter Konstanten zu definieren und diese immer automatisch zu übergeben, wenn wir eine Einstellung vornehmen oder auslesen wollen. Außerdem müssen wir natürlich, um die Handhabung wirklich zu vereinfachen, noch ein paar neue Funktionen um die eigentlichen Aufrufe bauen, damit wir tatsächlich nur die notwendigsten Daten angeben müssen.

Also legen wir in einem neuen, leeren Modul namens **mdlRegistryApp** zunächst einmal die folgenden beiden Konstanten an und hinterlegen für diese gleich die entsprechenden Werte:

```
Public Const cStrAppName As String = "MeineAnwendung"
Public Const cStrSection As String = "MeinAnwendungsbereich"
```

Einfaches Schreiben in die Registry

Die einfachere der beiden Funktionen zum Schreiben und Lesen ist die zum Schreiben von Daten in die Registry. Hier brauchen wir neben **AppName** und **Section** einfach nur den Namen des Schlüssels und den Wert als Parameter anzugeben. Also bauen wir eine Funktion namens **SaveAppSetting**, die nur noch die Parameter **strKey** und **strSetting** entgegennimmt und damit und mit den Werten der beiden Konstanten den Aufruf der Anweisung **SaveSetting** durchführt:

```
Public Sub SaveAppSetting(strKey As String, _
        strSetting As String)
    SaveSetting cStrAppName, cStrSection, strKey, strSetting
End Sub
```

Wenn wir zum Beispiel drei Werte in die Registry schreiben wollen, erledigen wir das mit der folgenden Beispiel-

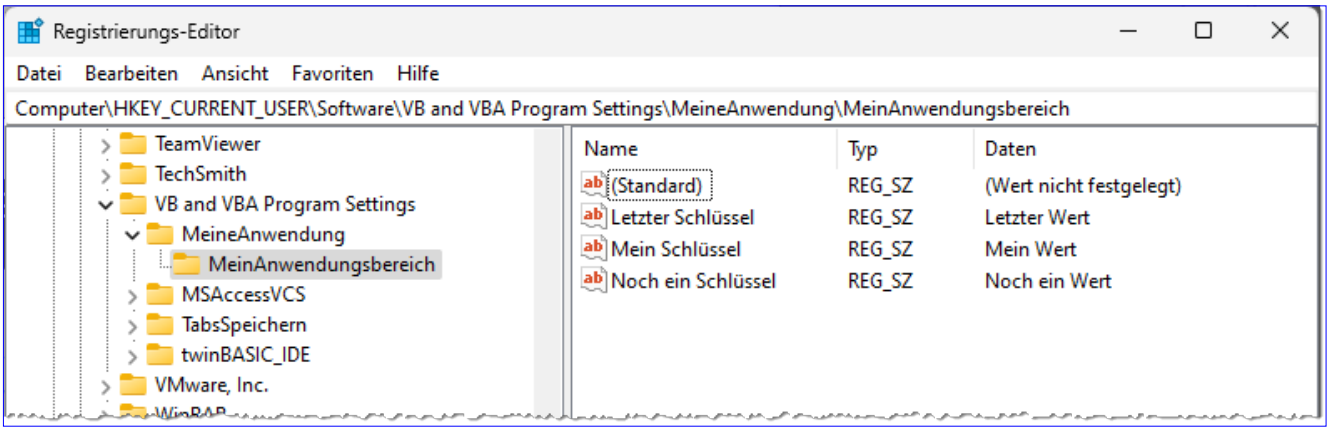


Bild 1: Frisch angelegte Registryschlüssel

prozedur. Diese ruft einfach nur drei Mal die Prozedur **SaveAppSetting** auf und übergibt Schlüssel und Wert:

```
Public Sub Test_SaveAppsetting()
    SaveAppSetting "Mein Schlüssel", "Mein Wert"
    SaveAppSetting "Noch ein Schlüssel", "Noch ein Wert"
    SaveAppSetting "Letzter Schlüssel", "Letzter Wert"
End Sub
```

Das Ergebnis können wir in der Registry im Zweig **Computer\HKEY_CURRENT_USER\Software\VB and VBA Program Settings\MeineAnwendung\MeinAnwendungsbereich** betrachten (siehe Bild 1).

Einfaches Lesen aus der Registry

Nun wollen wir diese Werte auch noch auf einfache Weise auslesen. Dazu erstellen wir eine weitere Prozedur namens **GetAppSetting**.

Diese verwendet die folgenden drei Parameter:

- **strKey**: Erwartet den Namen des Schlüssels des auszulesenden Wertes.
- **strDefault**: Erwartet einen Standardwert, der zurückgegeben werden soll, wenn es den mit **strKey** angegebenen Schlüssel noch nicht gibt.
- **bolSaveIfNotExists**: Wenn der Schlüssel noch nicht vorhanden oder leer ist, kann **GetAppSetting** diesen auch direkt anlegen. Dazu stellt man diesen Parameter auf **True** ein.

Die Prozedur finden wir in Listing 1.

Sie prüft zunächst, ob der Parameter **bolSaveIfNotExists** den Wert **True** enthält. Ist das der Fall, rufen wir mit **GetSetting** den Wert für den mit **strKey** übergebenen Schlüssel ab, ohne den mit **strDefault** angegebenen Standardwert mit zu übergeben.

Nur so können wir feststellen, ob der Schlüssel noch nicht existiert oder leer ist. Enthält **strTemp** danach eine Zeichenkette mit einer Länge von 0 Zeichen, schreiben wir den mit **strDefault** übergebenen Wert in einen neuen Schlüssel namens **strKey**.

In diesem Fall tragen wir den Wert aus **strDefault** in die Variable **strTemp** ein, deren Wert wir später als Funktionsergebnis zurückgeben.

Wenn **bolSaveIfNotExists** den Wert **False** enthält, lesen wir einfach mit **GetSetting** den Wert für den mit **strKey** angegebenen Schlüssel aus und speichern das Ergebnis in **strTemp**. Sollte das Ergebnis leer sein, erhalten wir stattdessen den Inhalt von **strDefault** als Funktionsergebnis.

Seitenränder in Access-Berichten

Im Beitrag »Bericht mit unterschiedlichen Seitenrändern« (www.access-im-unternehmen.de/1517) haben wir untersucht, wie wir einem Bericht für gerade und ungerade Zahlen unterschiedliche Seitenränder zuweisen können. Das ist zunächst vor allem daran gescheitert, dass die in der Seiteneinrichtung zugewiesenen Seitenränder größer waren als die per VBA eingestellten. Diese konnten wir zwar korrigieren, aber scheinbar willkürlich wurden diese wieder auf Werte eingestellt, die nicht mit unseren Anpassungen harmonierten. Also schauen wir uns im vorliegenden Beitrag einmal an, woher diese Daten überhaupt kommen, wo sie gespeichert werden und wie wir dafür sorgen können, dass sie uns nicht ins Gehege kommen, wenn wir die Seitenränder einmal auf kleinere Werte einstellen, als wir in den Seiteneinstellungen vorfinden.

Das Problem in dem oben genannten Beitrag ist, dass wir die Seitenränder für die linke und die rechte Seite abhängig davon, ob es sich um eine gerade oder eine ungerade Seite handelt, einstellen wollen. Für die Seiten 1, 3, 5 und so weiter soll links ein 25 mm-Rand vorliegen und rechts ein 10 mm-Rand. Die Seiten 2, 4, 6 und so weiter wollten wir mit einem linken Rand von 10 mm und einem rechten Rand von 25 mm ausstatten.

Das klappt auch, aber wir haben zwischenzeitlich einmal im Dialog **Seite einrichten** den linken Rand auf 25 mm und den rechten Rand auf 10 mm eingestellt. Diese Ränder werden von Access immer eingehalten. Auch wenn wir für den Bericht per VBA einen linken Rand von 10 mm einstellen – dann werden die Inhalte zwar um 15 mm nach links verschoben, aber der im Dialog **Seite einrichten** vorgegebene Seitenrand überschreibt diese Inhalte einfach.

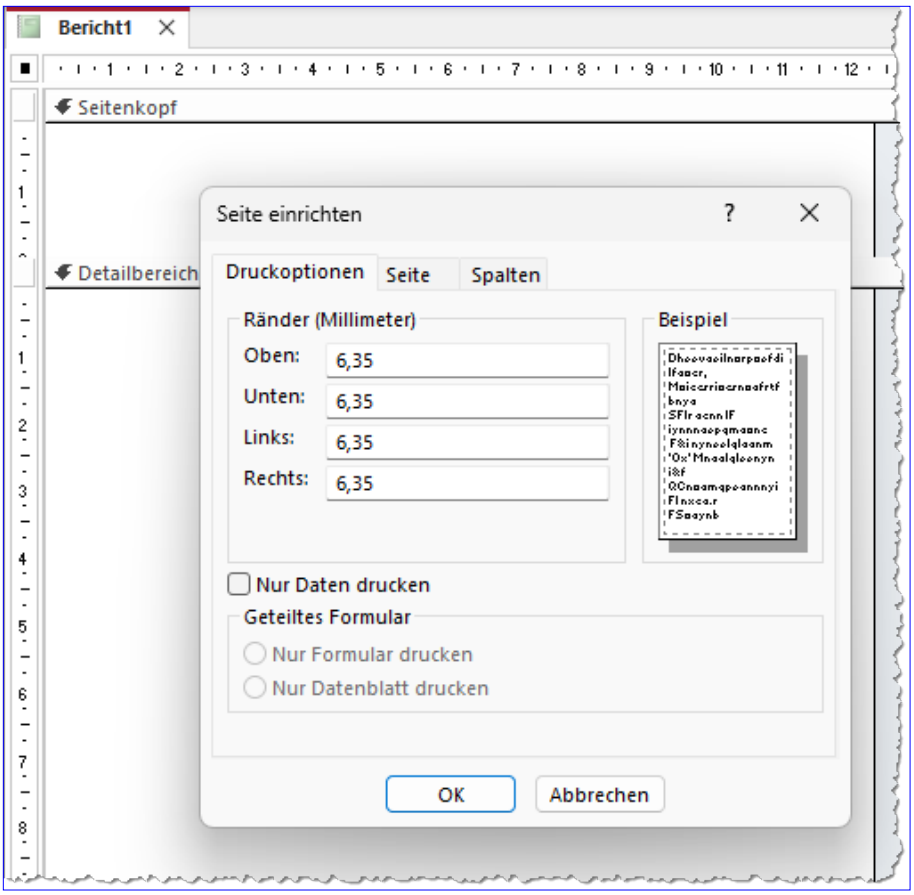


Bild 1: Standardeinstellung der Seitenränder

Da sich die Werte in der Seiteneinrichtung scheinbar zufällig ändern, wollen wir dieses Verhalten einmal grundlegend beleuchten.

Standardwerte für einen neuen Bericht in einer neuen, leeren Datenbank

Wenn wir eine neue, leere Datenbank öffnen, finden wir für einen ebenfalls neuen Bericht die Einstellungen aus Bild 1 im Dialog **Seite einrichten** vor. Diese liegen für alle Seitenränder bei 6,35 mm. Diesen Dialog öffnen Sie, indem Sie den Bericht in der Entwurfsansicht öffnen und im Ribbon unter **Seite einrichtenlSeitenlayout** den Befehl **Seite einrichten** betätigen.

Um zu prüfen, wie diese Werte nun unter VBA erscheinen, hinterlegen wir für das Ereignis **Bei Seite** die folgende Ereignisprozedur:

```
Private Sub Report_Page()
    Debug.Print "Linker Rand:   " & Me.Printer.LeftMargin
    Debug.Print "Rechter Rand: " & Me.Printer.RightMargin
    Debug.Print "Oberer Rand:   " & Me.Printer.TopMargin
    Debug.Print "Unterer Rand: " & Me.Printer.BottomMargin
End Sub
```

Nach einem Wechsel in die Seitenansicht erhalten wir im Direktbereich die folgenden Werte:

```
Linker Rand:   360
Rechter Rand: 360
Oberer Rand:   360
Unterer Rand: 360
```

Wir erhalten also erstmal die korrekten Werte, hier in der Einheit Twips. 567 Twips entsprechen einem Zentimeter.

Nun schauen wir uns an, was geschieht, wenn wir die Seitenränder alle auf 1 cm einstellen (siehe Bild 2).

Wechseln wir wieder in die Seitenansicht, erhalten wir nun die folgende Ausgabe:

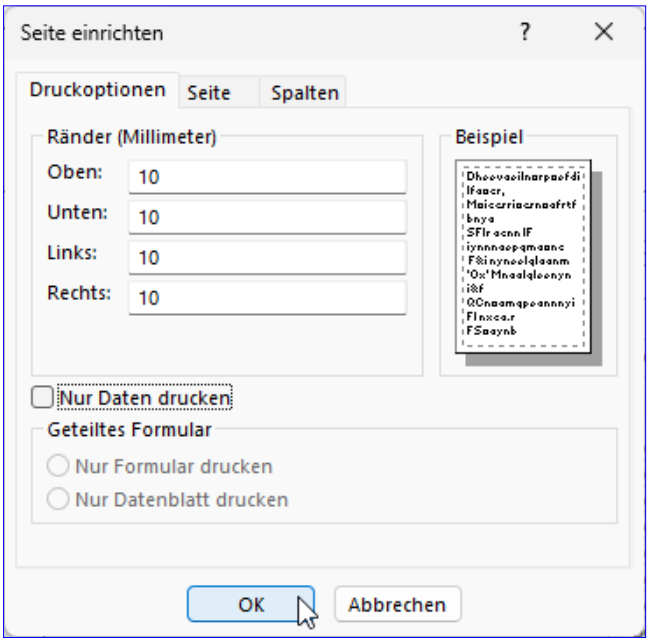


Bild 2: Manuell geänderte Seitenränder

```
Linker Rand:   567
Rechter Rand: 567
Oberer Rand:   567
Unterer Rand: 567
```

Wir speichern und schließen den Bericht nun unter dem Namen **rpt10mm**. Nach dem erneuten Öffnen erhalten wir wieder die Seitenränder von 10 mm.

Einstellungen in einem weiteren neuen, leeren Bericht

Wir legen nun einen weiteren neuen, leeren Bericht an und öffnen den Dialog **Seite einrichten**. Hier finden wir nun wieder die Seitenränder von 6,35 mm vor.

Es scheint also so zu sein, dass diese Einstellungen mit dem Bericht gespeichert werden und nicht als Standardeinstellungen übernommen werden, wenn man diese in einem Bericht ändert.

Wo werden die Seitenränder gespeichert?

Nun möchten wir herausfinden, wo diese Einstellungen gespeichert werden. Dazu speichern wir den Bericht im

Textformat auf der Festplatte, und zwar mit dem folgenden Befehl:

```
SaveAsText acReport, "rpt10mm", _
    Currentproject.Path & "\rpt10mm.txt"
```

Dies legt eine Textdatei mit dem kompletten Code für den Bericht an.

Auf den ersten Blick finden wir hier keine Einstellung, die nach den Seitenrändern für den Bericht aussieht. Also ändern wir diese nun auf jeweils 20 mm und speichern den Bericht erneut, diesmal unter dem Dateinamen **rpt20mm.txt** (allerdings unter dem gleichen Berichtsnamen):

```
SaveAsText acReport, "rpt10mm", _
    Currentproject.Path & "\rpt20mm.txt"
```

Nun öffnen wir die beiden Dateien in einem Texttool, welches das Vergleichen von Dateien erlaubt – zum Beispiel **Notepad++** mit dem Plugin **ComparePlus**.

Die Unterschiede sehen wir in Bild 3. Die Checksumme stimmt nicht überein, was in Anbetracht weiterer folgender Unterschiede logisch ist.

Der erste Unterschied sind die beiden Eigenschaften **Right** und **Bottom**:

Die Eigenschaft **Right** hat bei der 10 mm-Variante den Wert **22.245** und bei 20 mm den Wert **21.990**.

Die Differenz beträgt **255**. Bei der Eigenschaft **Bottom** beträgt die Differenz ebenfalls **255**.

Außerdem gibt es noch Unterschiede bei den Eigenschaften **PrtMip** und **PrtDevMode**. Diese enthalten verschiedene Informationen für den Drucker:

- **PrtMip** (Printer MIP - Mode Information Packet): Diese Eigenschaft enthält verschiedene Druckereinstellungen und -parameter, die in einer kompakten binären Form gespeichert sind. **PrtMip** speichert Informationen wie

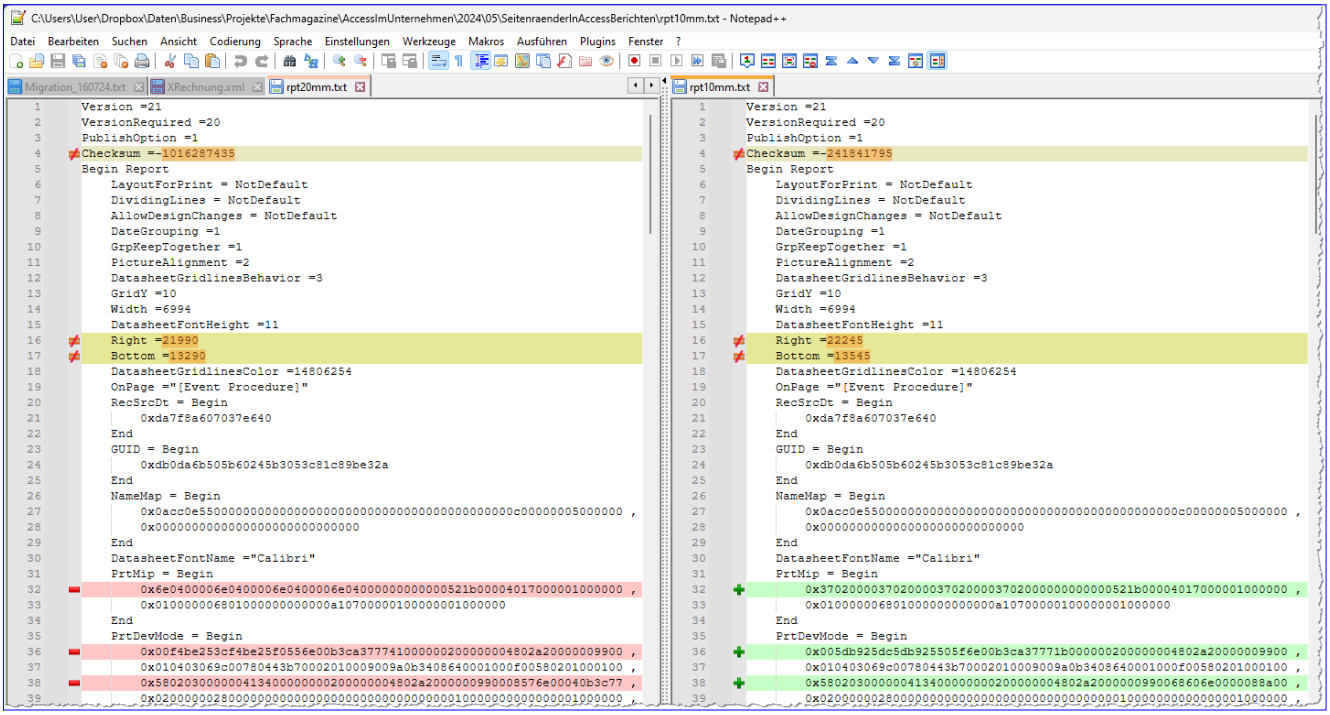


Bild 3: Unterschiede der als Datei gespeicherten Berichte

Bericht mit unterschiedlichen Seitenrändern

Berichte werden in der Regel so ausgelegt, dass sie immer auf der linken Seite einen Rand zum Abheften haben. Bei den meisten Dokumenten ist das völlig ausreichend, zum Beispiel für Rechnungen oder Angebote. Es gibt jedoch auch wesentlich anspruchsvollere Aufgaben, die mit einer Access-Anwendung samt Bericht erledigt werden. Diese sollen dann so ausgedruckt werden können, dass Vorder- und Rückseite eines Blatts bedruckt werden und beim aufgeklappten Dokument der breitere Rand immer zur Heftung hin zeigt. Auch wenn die dazu notwendigen Einstellungen selten angewendet werden: Es gibt sie und in diesem Beitrag zeigen wir, wie man einen Bericht so druckt, dass die Seiten als Broschüre geheftet werden können.

Bericht vorbereiten

Als Erstes benötigen wir einen geeigneten Bericht. Eigentlich eignet sich dazu jeder Bericht, der vier und mehr Seiten enthält. Als Beispiel verwenden wir einen Bericht, der die Adressen der Mitarbeiter aus unserer Mitarbeiterverwaltung als Liste ausgibt.

Wir erstellen also einen neuen, leeren Bericht, dem wir die Tabelle **tblMitarbeiter** als Datensatzquelle zuweisen. Wir ziehen die Felder **MitarbeiterID**, **AnredeID**, **Vorname**, **Nachname**, **Strasse**, **PLZ** und **Ort** in den Detailbereich des Entwurfs.

Dann wollen wir diese möglichst effizient in die Tabellenform überführen, also so, dass die Bezeichnungsfelder als Spaltenüberschriften im Seitenkopf-Bereich landen und die eigentlichen Felder im Detailbereich verbleiben.

Dazu markieren wir alle Steuerelemente und betätigen im Ribbon den Befehl **AnordnenlTabellelTabelle** (siehe Bild 1).

Damit erhalten wir die Ansicht aus Bild 2. Damit können wir bereits fast arbeiten.

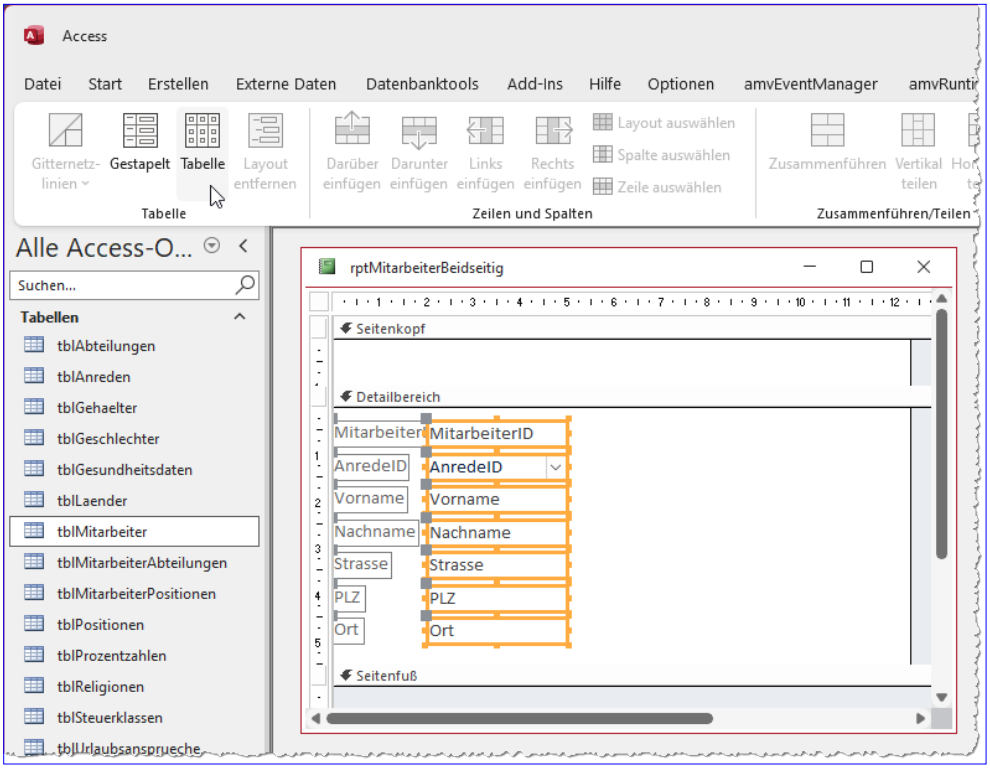


Bild 1: Hinzufügen und Anordnen der anzuzeigenden Felder

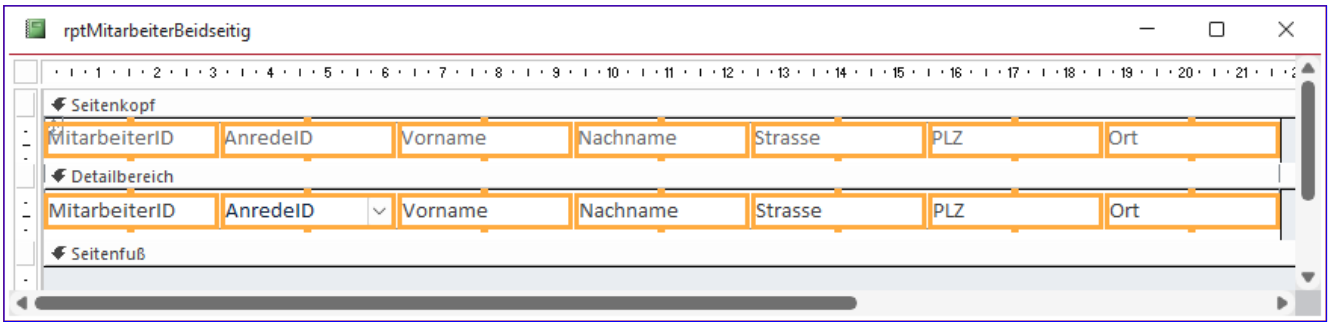


Bild 2: Tabelle als Basis unseres Berichts

Wenn wir jedoch in die Seitenansicht wechseln, stellen wir fest, dass wir noch ein paar optische Verbesserungen vornehmen müssen (siehe Bild 3):

- Die Zeilen werden noch mit wechselnder Hintergrundfarbe angezeigt.
- Die einzelnen Textfelder haben noch Ränder.
- Die Breite der Textfelder muss noch optimiert werden, damit alle Einträge auf die Breite des bedruckbaren Bereichs passen.

Die wechselnden Hintergründe stellen wir über die Eigenschaft **Alternative Hintergrundfarbe** ein, der wir den gleichen Wert hinzufügen, den die Eigenschaft **Hintergrundfarbe** aufweist.

Die Ränder der Textfelder entfernen wir, indem wir diese markieren und die Eigenschaft **Rahmenart** auf **Transparent** einstellen. Schließlich wechseln wir in die überaus praktische Layoutansicht des Berichts. Diese zeigt die Daten so an, wie sie auch in der Berichtsansicht erscheinen und so, wie es ungefähr der Seitenansicht entspricht – zumindest bezüglich der Spaltenbreiten. Diese Ansicht bietet aber gleichzeitig die Möglichkeit, Einstellungen an den Elementen vorzunehmen. Dazu gehört auch das Definieren der Spaltenbreiten. Wir brauchen nur wie in Bild 4 den rechten Rand eines der Steuerelemente einer Spalte nach links oder rechts zu ziehen, um die Breite an den Inhalt anzupassen.

Hier müssen wir die Spaltenbreiten nun insgesamt so verkleinern, dass auf der rechten Seite kein Element mehr aus der gestrichelten Markierung herausragt. Die Spaltenüberschriften der Felder **MitarbeiterID** und **AnredeID** ändern wir noch auf **ID** und **Anrede**, sodass diese weniger Platz benötigen. Dazu brauchen wir diese nur doppelt anzuklicken und anzupassen. Die Inhalte der Spalte **MitarbeiterID**, die nun die Überschrift **ID** trägt, wollen wir

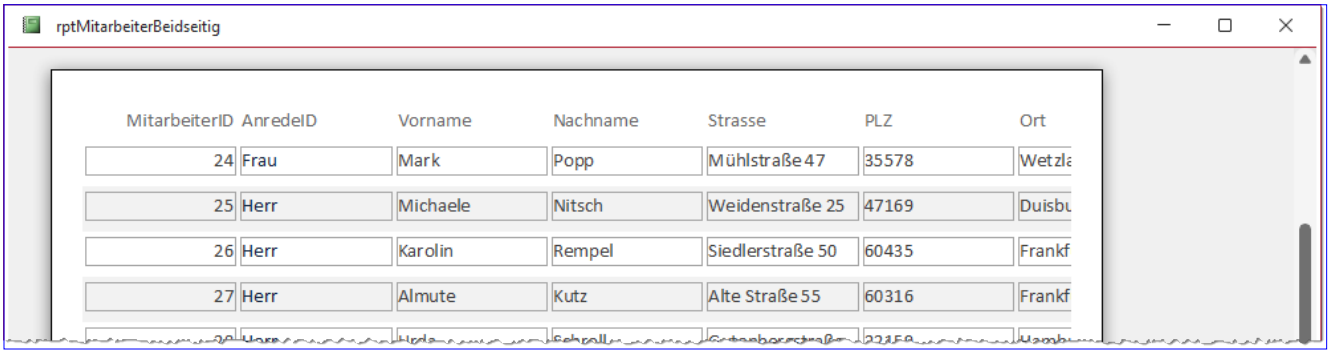


Bild 3: Die Optik muss noch optimiert werden.

außerdem noch linkszentriert ausrichten. Dazu wechseln wir im Ribbon zum Bereich **Format|Schriftart**, markieren die Spaltenüberschrift und stellen für diese als Ausrichtung **Linksbündig** ausrichten ein. Das Gleiche erledigen wir dann noch für die Feldinhalte dieser Spalte. Hier sehen wir beim Anklicken eines der Einträge, dass direkt alle Elemente der Spalte markiert werden (siehe Bild 5).

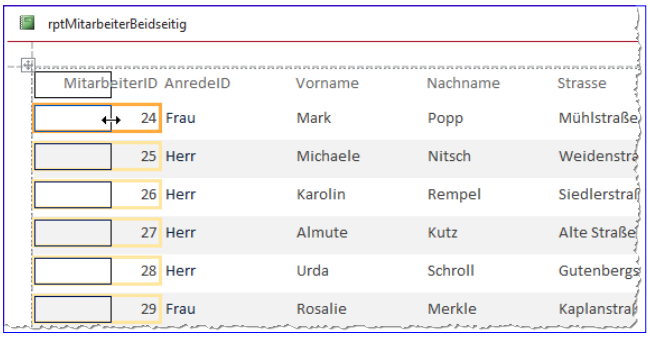


Bild 4: Anpassen der Spaltenbreiten

Überschriften hervorheben

Schließlich wollen wir die Überschriften noch fett drucken und unter den Überschriften eine Linie hinzufügen. Das erledigen wir wieder in der Entwurfsansicht (siehe Bild 6). Hier sehen wir gegebenenfalls auch noch, dass die Seite wesentlich breiter als der Platz ist, der durch die Steuerelemente beansprucht wird – in diesem Fall ziehen wir den rechten Seitenrand bis an den rechten Rand des ganz links platzierten Steuerelements heran.

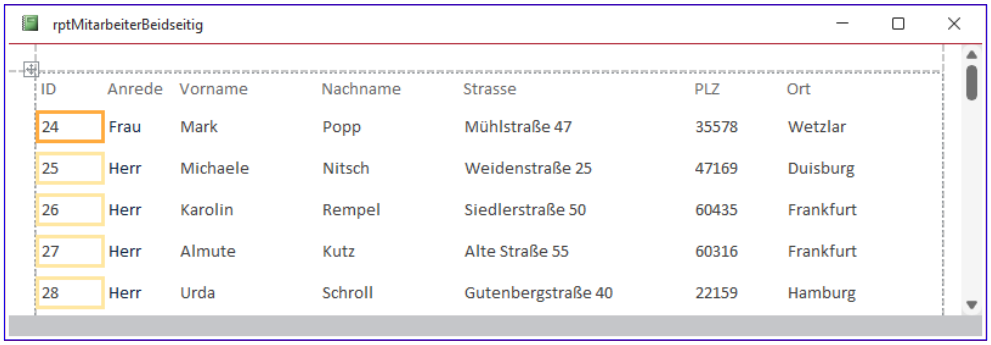


Bild 5: Ausrichten der IDs am linken Spaltenrand

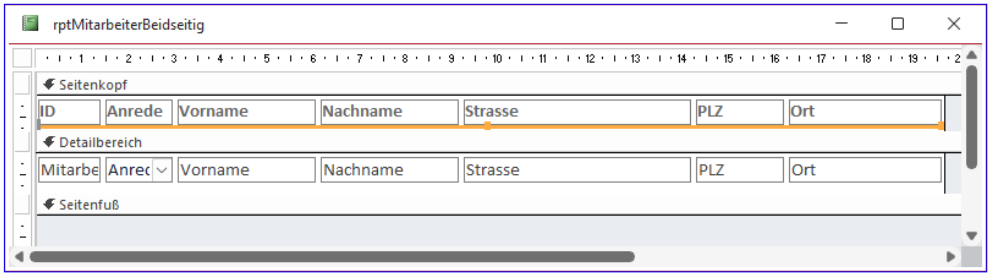


Bild 6: Der Bericht in der Entwurfsansicht

Seitenränder korrekt einrichten

Wechseln wir nun zur Seitenansicht, erhalten wir die Ansicht aus Bild 7. Das sieht schon gut aus, aber der Seitenrand beträgt auf beiden Seiten nur weniger als einen Zentimeter.

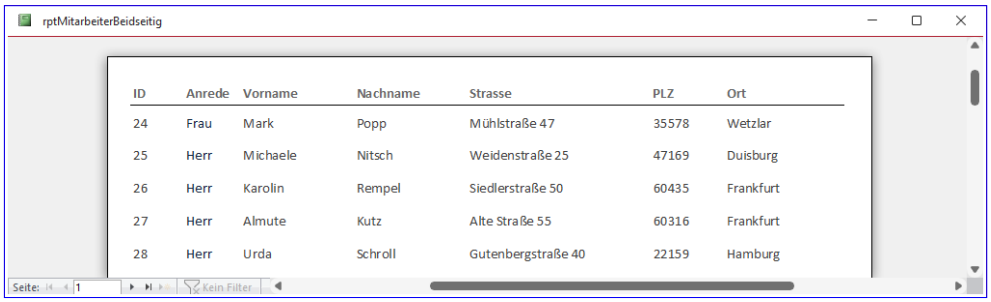


Bild 7: Der Bericht in der Seitenansicht

Die genauen Einstellungen finden wir im Dialog **Seite einrichten**, den wir in der Entwurfsansicht des Berichts im Ribbon unter **Seiten einrichten|Seitenlayout** mit dem Befehl **Seite einrichten** aufrufen können. Hier sehen wir nun, dass der Abstand für alle Seiten auf 6,35 Millimeter eingestellt ist (siehe Bild 8).

Wir werden die Seitenränder gleich ohnehin per VBA einstellen. Dieser soll für den äußeren Rand 10 mm betragen und für den inneren Rand 25 mm. Wenn das keine Rolle spielt, warum gehen wir nicht direkt zur Programmierung der Seitenränder über? Weil wir aktuell links und rechts in Summe nur 13 mm Seitenrand sehen und wir später insgesamt 35 mm benötigen. Das heißt, dass der Inhalt aktuell noch zu viel Platz benötig. Also stellen wir in diesem Dialog zunächst die linke Seite auf 25 mm und die rechte auf 10 mm Seitenrand ein.

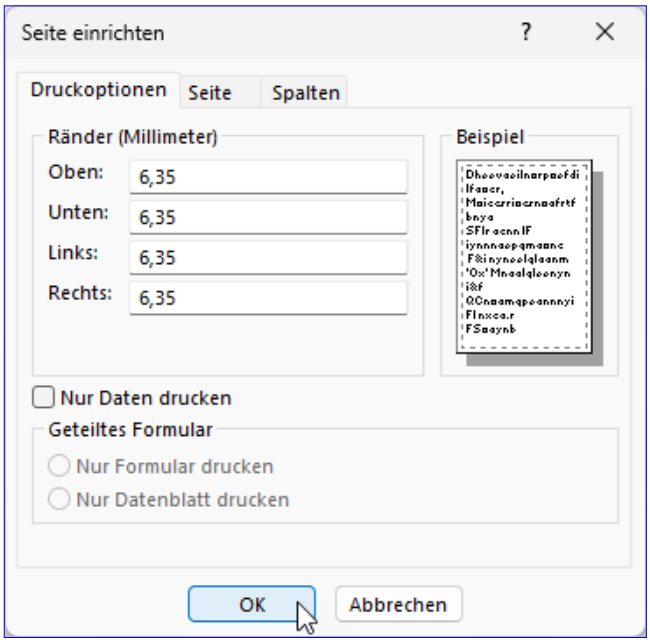


Bild 8: Dialog zum Einrichten der Seite

Wechseln wir nun zur Seitenansicht, stellen wir fest, dass auf der ersten Seite der rechte Rand der rechten Spalte abgeschnitten wird. Um dies zu korrigieren, wechseln wir erneut zur Layoutansicht und justieren die Spaltenbreiten so, dass diese wieder in den gestrichelt markierten Bereich hineinpassen (siehe Bild 9).

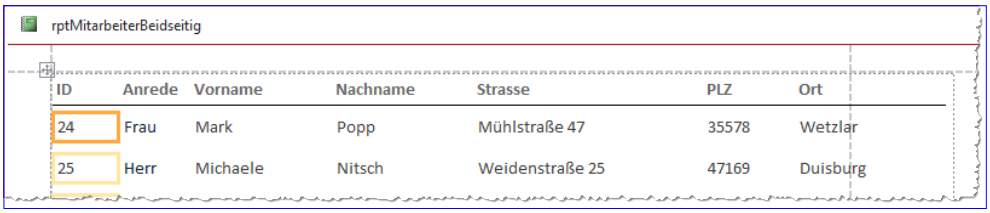


Bild 9: Die Spalten sind aktuell wieder zu breit.

Nach einigen Korrekturen und dem Durchlaufen aller Inhalte zur Prüfung, ob diese noch vollständig sichtbar sind, sieht der Entwurf in der Layoutansicht wie in Bild 10 aus. Lediglich die Linie im Seitenkopf müssen wir hier noch kürzen.

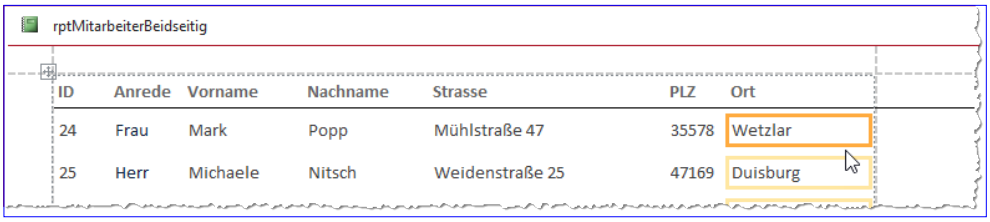


Bild 10: Spalten mit perfekter Breite

Abwechselnde Seitenränder

Danach hat der Bericht die Form, mit der wir in die Programmierung der alternierenden Seitenränder einsteigen können. Das Ziel ist, dass der Seitenrand für die erste Seite links 25 mm beträgt und rechts 10 mm – diese Seite ist die Titelseite. Die zweite und die dritte Seite sollen dann die folgenden Seitenränder aufweisen:

- Zweite Seite: Links 10 mm, rechts 25 mm
- Dritte Seite: Links 25 mm, rechts 10 mm.
- Für die folgenden Seiten wechseln die Seitenränder entsprechend.

Das heißt also: Für alle Seiten mit einer ungeraden Seitennummer beträgt der linke Rand 25 mm und der rechte Rand 10 mm und alle Seiten mit einer geraden Seitennummer erhalten die umgekehrten Seitenränder. Aktuell betragen die Seitenränder für alle Seiten links 25 mm und rechts 10 mm. Das werden wir nun ändern.

Fehlerbehandlung per VBA hinzufügen

Eine Fehlerbehandlung ist Teil einer professionellen Datenbankanwendung auf Basis von Microsoft Access. Sobald wir eine Datenbank an einen Kunden oder Mitarbeiter weitergeben, also an einen anderen Benutzer als uns selbst, ist dies praktisch ein Pflichtprogramm. Die Fehlerbehandlung soll den Benutzer über einen Fehler informieren und diesem die Möglichkeit geben, dem Entwickler Informationen über den Fehler zukommen zu lassen. In diesem Beitrag wollen wir die Hauptarbeit bei der Implementierung einer Fehlerbehandlung erledigen. Dazu wollen wir eine Prozedur schreiben, die beliebige Routinen, also Sub-, Function- und Property-Prozeduren, mit einer einfachen Fehlerbehandlung ausstattet – und überdies noch mit einer Zeilennummerierung. Diese ist wichtig, wenn wir schnell herausfinden wollen, an welcher Stelle ein Fehler aufgetreten ist. Zusammen mit dem Namen des Moduls, dem Namen der Prozedur, der Fehlernummer und der Fehlerbeschreibung erhalten wir so Informationen, die in der Regel zum Aufdecken des Fehlers führen.

Wir wollen in diesem Beitrag eine minimale Fehlerbehandlung verwenden, die nur die nötigsten Informationen aufnimmt. Dazu gehören die folgenden:

- Fehlernummer
- Fehlerbeschreibung
- Zeile, in welcher der Fehler aufgetreten ist
- Prozedur, in welcher der Fehler aufgetreten ist
- Modul, in dem der Fehler aufgetreten ist

Woher bekommen wir diese Informationen?

Die Fehlernummer und die Fehlerbeschreibung erhalten wir per VBA grundsätzlich nur, wenn wir die eingebaute Fehlerbehandlung von VBA ausgeschaltet haben. Das erledigen wir mit einer der folgenden Anweisungen:

```
On Error Resume Next
On Error GoTo Sprungmarke
```

Dabei ist Sprungmarke eine Zeile, die den Namen der Sprungmarke und einen Doppelpunkt enthält, zum Beispiel:

```
Errorhandling:
```

Wenn wir die eingebaute Fehlerbehandlung wieder einschalten wollen, können wir dies mit der folgenden Anweisung erledigen:

```
On Error GoTo 0
```

Während die eingebaute Fehlerbehandlung ausgeschaltet ist, werden Fehler entweder einfach ignoriert (**On Error Resume Next**) oder der Code wird an der angegebenen Sprungmarke fortgesetzt, wenn eine angegeben wurde (**On Error Goto Errorhandler**).

Fehlernummer und Fehlerbeschreibung ermitteln

In jedem Fall können wir in den darauffolgenden Zeilen die Eigenschaften des **Err**-Objekts auslesen. Das können wir zwar immer, aber wenn kein unbehandelter Fehler aufgetreten ist, liefert dies auch keine Fehlerinformationen.

Gehen wir jedoch davon aus, dass wir die Informationen nach Auftreten eines Fehlers abfragen, können wir dem **Err**-Objekt die folgenden Informationen entnehmen:

- **Err.Number**: Liefert die Fehlernummer, zum Beispiel **11**.
- **Err.Description**: Liefert die Fehlerbeschreibung, zum Beispiel **Division durch Null**.

Name des VBA-Projekts ermitteln

Wir könnten mit der **Err**-Eigenschaft **Err.Source** sogar noch den Namen des VBA-Projekts ermitteln, in dem der Fehler ausgelöst wurde. Dies ist jedoch nicht unbedingt notwendig, da sich die Fehlerbehandlung meist auf das gleiche VBA-Projekt bezieht, in dem sich diese befindet.

Zeilennummer ermitteln

Die Nummer der Zeile, in welcher der Fehler aufgetreten ist, finden wir offiziell gar nicht. Allerdings gibt es eine nicht dokumentierte Funktion namens **Erl**, die uns im Falle eines Fehlers die Nummer der Zeile mit der auslösenden Anweisung liefert. Dies geschieht jedoch nur, wenn es auch eine Zeilennummerierung gibt. Diese müssen wir selbst hinzufügen, es gibt keine eingebaute Funktion, um dies einfach zu erledigen.

Wie wir die Zeilennummer hinzufügen, haben wir deshalb ausführlich beschrieben, und zwar im Beitrag **Zeilennummern per VBA hinzufügen** (**www.access-im-unternehmen.de/1515**). Hier finden Sie sogar Prozeduren, mit denen Sie die Zeilennummern für eine Prozedur, alle Prozeduren eines Moduls und sogar für alle Module in einem VBA-Projekt per Mausklick anlegen können.

Prozedurname und Modulname ermitteln

Den Prozedurnamen und den Modulnamen können wir ebenfalls nicht automatisch ermitteln. Es ist jedoch dennoch sinnvoll, diesen in einer Fehlerbehandlung auszugeben – dabei spielt es keine Rolle, ob die Fehlerbehandlung nur eine Meldung anzeigt, die Fehlerinformationen in eine Tabelle schreibt oder direkt eine E-Mail an den Entwickler der Anwendung schickt.

In jedem Fall werden wir in einer Fehlerbehandlung landen, die sich innerhalb der Prozedur befindet, in welcher der Fehler ausgelöst wurde. Diese befindet sich in der Regel hinter einer Sprungmarke am Ende der jeweiligen Prozedur. Und wenn wir ohnehin für jede Prozedur eine eigene Fehlerbehandlung hinzufügen müssen, dann können wir hier auch direkt die Information unterbringen, in welchem Modul und in welcher Prozedur wir uns gerade befinden.

Das gilt auch, wenn sich die Fehlerbehandlungsanweisungen nicht selbst in der Prozedur befinden, sondern in einer eigenen Routine. Dann müssen wir dafür sorgen, dass der Modul- und der Prozedurname zur dortigen Routine gelangen und können diese im Aufruf der fehlerauslösenden Prozedur platzieren.

Das sieht beispielsweise wie folgt aus:

```
Public Sub Test_WriteError()
10    On Error GoTo Errorhandling
20    Debug.Print 1 / 0
30    Exit Sub
Errorhandling:
    HandleError Err.Number, Err.Description, Erl, _
        "Test_WriteError", "mdlErrors"
End Sub
```

Die Prozedur hat bereits Zeilennummern und die eingebaute Fehlerbehandlung wird mit **On Error Goto Errorhandling** deaktiviert. Der Fehler durch die Division durch **0** in Zeile **20** führt somit dazu, dass die Prozedur direkt zur Sprungmarke **Errorhandling** springt. Hier rufen wir eine Prozedur namens **HandleError** auf, die unsere Fehlerbehandlungsroutine enthält. Dieser übergeben wir alle Daten, die wir in der Fehlerbehandlungsroutine verarbeiten wollen – die Fehlernummer, die Fehlerbeschreibung, die Zeilennummer, den Modulnamen und den Prozedurna-

```
Public Sub HandleError(lngNumber As Long, strMessage As String, lngLine As Long, strProcedure As String, _
        strModule As String)
    Dim db As DAO.Database
    Set db = CurrentDb
    db.Execute "INSERT INTO tblErrors(ErrorNumber, ErrorMessage, ErrorLine, ErrorProcedure, ErrorModule) VALUES(" _
        & lngNumber & ", '" & Replace(strMessage, "'", "''") & "', " & lngLine & ", '" & strProcedure & "', '" _
        & strModule & "')", dbFailOnError
End Sub
```

Listing 1: Fehlerbehandlungsroutine

men. An dieser Stelle wird klar: Wenn wir diese Fehlerbehandlung zu vielen Prozeduren hinzufügen wollen, können wir zwar mit Copy und Paste arbeiten, aber wir müssen dennoch viele Anpassungen vornehmen. In diesem Fall müssen wir jeweils den Modulnamen und den Prozedurnamen anpassen. Beim Modulnamen muss man dies nur einmal pro Modul erledigen und kann die entsprechenden Zeilen dann kopieren und braucht nur noch den Prozedurnamen anzupassen. Viel Aufwand bleibt es dennoch, und es ist auch nicht wenig fehleranfällig.

Fehlerbehandlungsroutine

Die Routine namens **HandleError**, die wir von allen Prozeduren mit einer Fehlerbehandlung aus aufrufen wollen, finden Sie in Listing 1. In diesem Fall wollen wir aufgetretene Fehler einfach in eine Tabelle namens **tblErrors** schreiben, die sich in der gleichen Datenbank befindet. Diese enthält Felder für alle erwähnten Fehlerinformationen und zuzüglich noch ein Feld mit Datum und Uhrzeit.

Für dieses legen wir als Standardwert **Jetzt()** fest, damit wir dieses Feld nicht per Code füllen müssen (siehe Bild 1).

Die Fehlerbehandlungsroutine enthält im Wesentlichen eine **INSERT INTO**-Anweisung, die einen neuen Datensatz in die Tabelle **tblErrors** schreibt und die mit der **Execute**-Methode des **Database**-Objekts ausgeführt wird.

Wenn wir unsere Fehlerbehandlung einige Male mit der obigen Beispielprozedur auslösen, finden wir Einträge wie in Bild 2 in der Tabelle **tblErrors** vor.

Alternativen zum Eintrag in die Fehlertabelle

Wenn Sie in Ihrer Anwendung einmal alle Prozeduren, bei denen es sinnvoll ist (in der Regel alle), mit einer Fehlerbehandlung ausgestattet haben, können Sie die im Falle eines Fehlers durchzuführenden Aktionen einfach ändern. Dazu brauchen Sie nur die Anweisungen in der Prozedur

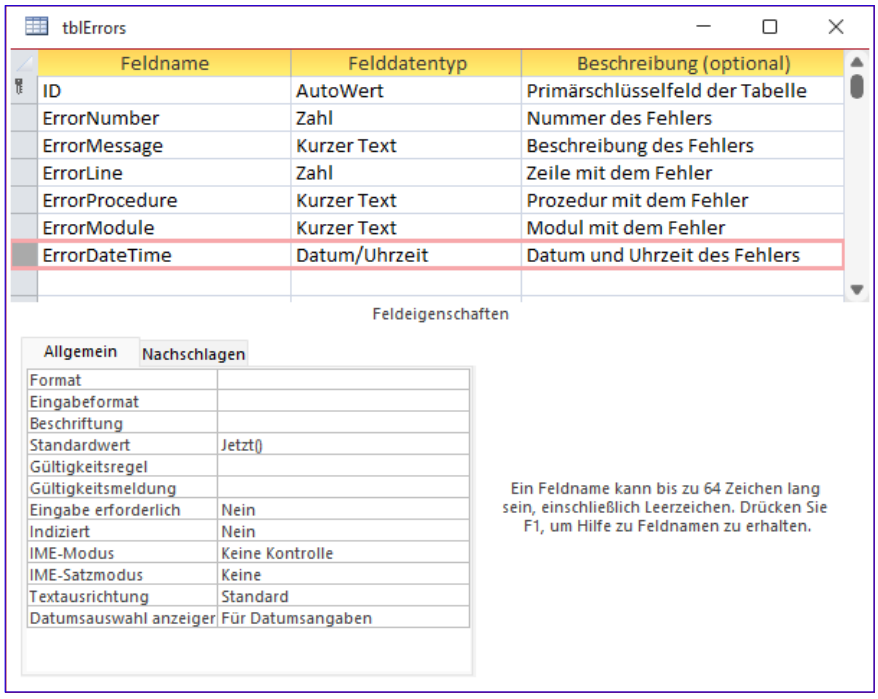
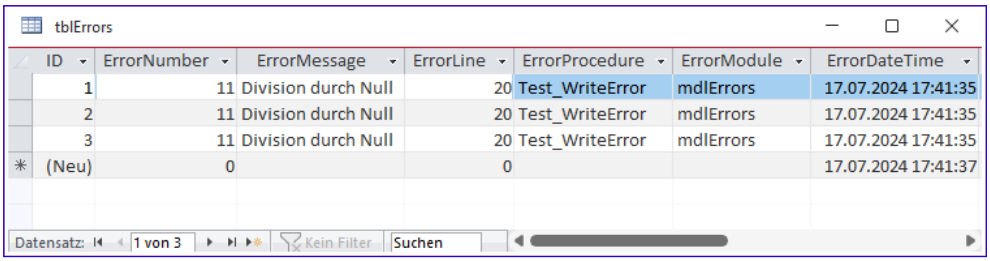


Bild 1: Tabelle zum Speichern der Fehlerinformationen

HandleError anzupassen. Der Fantasie sind hier keine Grenzen gesetzt – Sie können die Daten wir hier in eine Tabelle schreiben, eine E-Mail mit Fehlerinformationen an den Entwickler senden, einfach nur eine Fehlermeldung anzeigen oder auch Fehlermeldungen in eine Textdatei außerhalb der Datenbankdatei schreiben.


tblErrors

ID	ErrorNumber	ErrorMessage	ErrorLine	ErrorProcedure	ErrorModule	ErrorDateTime
1	11	Division durch Null	20	Test_WriteError	mdlErrors	17.07.2024 17:41:35
2	11	Division durch Null	20	Test_WriteError	mdlErrors	17.07.2024 17:41:35
3	11	Division durch Null	20	Test_WriteError	mdlErrors	17.07.2024 17:41:35
(Neu)	0		0			17.07.2024 17:41:37

Datensatz: 1 von 3 | Kein Filter | Suchen

Bild 2: Tabelle zum Speichern der Fehlerinformationen mit einige Fehlern

Sie können auch verschiedene dieser Vorschläge kombinieren. In der Regel ist es immer von Vorteil, wenn man den Benutzer über einen Fehler informiert, damit dieser gegebenenfalls weitere Schritte durchführen kann – wie den Entwickler zu informieren.

Fehlerbehandlung anwendungsweit integrieren

Nun stehen wir nur noch vor einer Fleißaufgabe. Wir müssen die Zeilen, die für die Funktion der Fehlerbehandlung nötig sind, in jede Prozedur eintragen.

Das sind aus der Beispielprozedur von oben alle Anweisungen außer Prozedurkopf, -fuß und der eigentlichen Anweisung:

```
Public Sub Test_WriteError()
10    On Error GoTo Errorhandling
20    Debug.Print 1 / 0
30    Exit Sub
Errorhandling:
    HandleError Err.Number, Err.Description, Erl, _
        "Test_WriteError", "mdlErrors"
End Sub
```

Und wenn wir, wie im Beitrag **Zeilennummern per VBA hinzufügen** (**www.access-im-unternehmen.de/1515**) beschrieben, auch Zeilennummern automatisiert zu Prozeduren hinzufügen können, warum sollten wir dann nicht automatisch die immer wiederkehrenden Zeilen hinzufügen?

Prozedur zum automatisierten Hinzufügen einer Fehlerbehandlung

Hier verwenden wir die Prozedur **AddErrorHandler** aus Listing 2. Sie erwartet die folgenden Parameter:

- **strModule**: Name des Moduls
- **strProcedure**: Name der Prozedur, in der die Fehlerbehandlung hinzugefügt werden soll

Die Prozedur füllt die Variable **objVBProject** mit einem Verweis auf das aktuelle VBA-Projekt. Dann ermittelt sie das **VBComponent**-Objekt zu dem mit **strModule** angegebenen Modul und referenziert es mit der Variablen **objVBComponent**. Über ihre Eigenschaft **CodeModule** holt sie sich einen Verweis auf das **CodeModule**-Objekt des Moduls.

Nun ermittelt sie einige Zeilennummern:

- **lngCountOfLines**: Wird mit der Anzahl der Zeilen des Moduls gefüllt.
- **lngProcBodyLine**: Wird mit der Nummer der ersten Zeile der Prozedur aus **strProcedure** gefüllt. Für den Pflichtparameter **ProcKind** übergeben wir die leere Variable **intProcType**. Diese erhält einen Wert für den Typ der Prozedur, den wir aber nicht benötigen.
- **lngFirstStatementLineNumber**: Hier ermitteln wir mit der Funktion **GetFirstStatementLineNumber** die Nummer der Zeile mit der ersten Anweisung der Prozedur, also die erste Zeile hinter der gegebenenfalls auch aus

Zeilennummern per VBA hinzufügen

Wer eine wirklich professionelle Fehlerbehandlung zu seiner Access-Anwendung hinzufügen möchte, kommt nicht um das Anlegen von Zeilennummern herum. Wenn man Zeilennummern festgelegt hat, kann man diese im Falle eines Fehlers mit der nicht dokumentierten Erl-Funktion auslesen. Das heißt, dass wir neben der Fehlernummer und der Fehlerbeschreibung noch auf die Zeilennummer zurückgreifen können, um den Fehler zu lokalisieren. Dazu gehört allerdings auch, dass wir in der Fehlermeldung Informationen über das Modul und die Prozedur unterbringen, in denen der Fehler aufgetreten ist – doch dies lässt sich einfach realisieren. Sehr aufwändig ist es hingegen, alle Prozeduren von umfangreichen Anwendungen mit Zeilennummern zu versehen. Um dies zu implementieren, nutzen wir die Bibliothek »Microsoft Visual Basic for Applications Extensibility 5.3«, die alle Möglichkeiten bietet, um auf die enthaltenen Module zuzugreifen und den Code automatisiert nach unseren Wünschen anzupassen.

Funktion zum Hinzufügen von Zeilennummern

Wichtig für die Fehlerbehandlung sind auch die Zeilennummern. Diese fügen wir mit der Funktion **AddLineNumbers** zu einer Prozedur hinzu. Diese Prozedur statten wir mit den folgenden Parametern aus:

- **strModule**: Name des Moduls, in dem sich die Prozedur befindet
- **strProcedure**: Name der Prozedur, die mit Zeilennummern versehen werden soll

Die Prozedur aus Listing 1 referenziert als Erstes das aktuelle VBA-Projekt und die mit **strModule** angegebene Komponente. Dazu holt sie sich über die **CodeModule**-Eigenschaft noch das passende **CodeModule**-Objekt, mit dem wir auf den Code zugreifen können. Mit **lngProcBodyLine** speichern wir die Zeile, in der die Prozedur aus **strProcedure** beginnt. Wir wollen nur die eigentlichen Anweisungen der Prozedur mit Zeilennummern versehen, also ermitteln wir die Nummer der ersten Zeile hinter dem Prozedurkopf und die Nummer der letzten Zeile vor der **End...**-Zeile der Prozedur. Dazu nutzen wir die beiden weiter hinten beschriebenen Funktionen **GetFirstStatementLineNumber** und **GetLastStatementLineNumber**.

Die Prozedur durchläuft nun in einer ersten **For...Next**-Schleife alle Zeilen der Prozedur, um eventuell bereits mit Zeilennummern versehene Prozeduren zu erkennen. Findet sie während dieses Durchlaufs in irgendeiner Zeile einen numerischen Wert in den ersten sechs Zeichen, wird die Variable **bolNumbers** auf **True** eingestellt und die Schleife verlassen. In diesem Fall legt die Prozedur keine neuen Zeilennummern an, sondern gibt eine entsprechende Meldung aus.

Anderenfalls durchläuft die Prozedur alle Zeilen von der ersten Anweisung bis zur letzten Anweisung der mit Zeilennummern zu versehenden Routine. Dies geschieht in einer **For...Next**-Schleife mit der Laufvariablen **lngLine**.

Innerhalb der Schleife prüfen wir in einer **Select Case**-Bedingung, ob verschiedene Bedingungen den Wert **True** haben. Dazu weisen wir der **Select Case**-Anweisung das Kriterium **True** zu. Der erste der folgenden **Case**-Zweige, der den Wert **True** liefert, wird demzufolge angesteuert.

```
Public Sub AddLineNumbers(strModule As String, strProcedure As String)
    Dim objVBProject As VBIDE.VBProject
    Dim objVBComponent As VBIDE.VBComponent
    Dim objCodemodule As VBIDE.CodeModule
    Dim intProcType As vbext_ProcKind
    Dim lngProcBodyLine As Long
    Dim lngFirstStatementLineNumber As Long
    Dim lngLastStatementLineNumber As Long
    Dim lngLine As Long
    Dim bolNumbers As Boolean
    Dim strLine As String
    Set objVBProject = VBE.ActiveVBProject
    Set objVBComponent = objVBProject.VBComponents(strModule)
    Set objCodemodule = objVBComponent.CodeModule
    lngProcBodyLine = objCodemodule.ProcBodyLine(strProcedure, intProcType)
    lngFirstStatementLineNumber = GetFirstStatementLineNumber(objCodemodule, lngProcBodyLine)
    lngLastStatementLineNumber = GetLastStatementLineNumber(objCodemodule, lngProcBodyLine)
    For lngLine = lngFirstStatementLineNumber To lngLastStatementLineNumber
        strLine = left(objCodemodule.Lines(lngLine, 1), 6)
        Select Case True
            Case IsNumeric(strLine)
                bolNumbers = True
                Exit For
        End Select
    Next lngLine
    If bolNumbers = False Then
        For lngLine = lngFirstStatementLineNumber To lngLastStatementLineNumber
            Select Case True
                Case left(LTrim(objCodemodule.Lines(lngLine, 1)), 5) = "Case "
                    objCodemodule.ReplaceLine lngLine, Space(6) & objCodemodule.Lines(lngLine, 1)
                Case IsLabel(objCodemodule.Lines(lngLine, 1))
                Case Else
                    objCodemodule.ReplaceLine lngLine, Format(lngLine, "00000") & " " & objCodemodule.Lines(lngLine, 1)
            End Select
        Next lngLine
    Else
        MsgBox "Die Prozedur '" & strProcedure & "' aus Modul '" & strModule _
            & "' scheint bereits Zeilennummern zu enthalten.", vbOKOnly + vbInformation, "Zeilennummern vorhanden"
    End If
End Sub
```

Listing 1: Prozedur zum Hinzufügen von Zeilennummern zu einer Prozedur

Die erste Bedingung prüft, ob die um führende und folgende Leerzeichen erleichterte aktuelle Zeile mit **Case** beginnt. **Case**-Zeilen sind eine der wenigen Zeilen, die nicht mit einer Zeilennummer versehen werden können. Eine **Case**-Zeile soll genau wie die nummerierten Zeilen um sechs Zeichen nach rechts eingerückt werden, auch wenn diese nicht nummeriert wird. Auf diese Weise bleibt die Einrückung gegenüber den nummerierten Zeilen erhalten.

Um die Zeile um sechs Zeichen einzurücken, verwenden wir die **ReplaceLine**-Anweisung des **CodeModule**-Objekts. Diese erwartet als ersten Parameter die Nummer der zu ersetzenden Zeile und als zweiten Parameter den neuen Inhalt der Zeile. Diesen stellen wir aus sechs mit der **Space**-Funktion ermittelten Leerzeichen und der vorherigen Zeile zusammen. Der zweite **Case**-Fall untersucht, ob es sich bei der Zeile um eine Sprungmarke handelt. Dazu verwendet sie die **IsLabel**-Funktion, welche die übergebene Zeile untersucht. Diese Funktion stellen wir später vor. Liefert **IsLabel** den Wert **True**, geschieht nichts weiter mit dieser Zeile. Sprungmarken sollten weder nummeriert noch eingerückt werden (letzteres wird ohnehin wieder rückgängig gemacht).

Der **Case Else**-Zweig tritt für alle weiteren Fälle ein. Dieser sorgt dafür, dass die aktuelle Zeile durch eine neue Zeile ersetzt wird, die aus der aktuellen Zeilennummer im Format **00000**, einem Leerzeichen und der ursprünglichen Zeile besteht. Warum als Format **00000**? Weil wir davon ausgehen, dass wir keine Module mit mehr als 99.999 Zeilen verwenden (falls doch, können Sie das Format und die Einrückungen so anpassen, dass diese mehr Stellen abdecken).

Wenn wir die Zeilenzahl **123** mit dem Format **00000** einfügen, wird daraus **123** plus zwei folgende Leerzeichen. Aus **1** wird **1** plus vier folgende Leerzeichen. Es erfolgt also unabhängig von der Stellenzahl der Zeilennummer eine konstante Einrückung. Auf diese Weise durchlaufen wir alle Zeilen von der ersten bis zur letzten Anweisung der angegebenen Prozedur.

Funktion zum Hinzufügen von Zeilennummern aufrufen

Um die Funktion **AddLineNumbers** aufzurufen, verwenden wir beispielsweise den folgenden Befehl:

```
Call AddLineNumbers("mdlTest", "Test")
```

Damit fügen wir genau einer Prozedur Zeilennummern hinzu.

Zeilennummern zu allen Prozeduren eines Moduls hinzufügen

Gegebenenfalls wollen wir jedoch nicht nur einer, sondern allen Prozeduren eines Moduls Zeilennummern hinzufügen – oder vielleicht sogar allen Modulen des VBA-Projekts. Dazu bauen wir uns schnell ein paar Routinen. Die erste soll den Namen des Moduls entgegennehmen, deren Prozeduren mit Zeilennummerierungen versehen werden sollen (siehe Listing 2).

Sie referenziert über das **VBProject**-Objekt das **VBComponents**-Objekt für das Modul und verwendet dann das **CodeModule** dieses Objekts. Sie ermittelt die Anzahl der Zeilen des Moduls und durchläuft dann alle Zeilen des Moduls. Dabei vergleicht sie jeweils den Namen der Prozedur der aktuellen Zeile mit dem in **strProcedure** gespeicherten Namen.

Diese ist zunächst leer. Sobald die Routine auf die erste Prozedur stößt, wird der Name der Prozedur in **strProcedure** geschrieben und die Prozedur **AddLineNumber** wird für dieses Modul und diese Prozedur aufgerufen. Auf diese Weise durchläuft die Prozedur **AddLineNumbersToModule** alle Prozeduren des Moduls:

Zeilennummern zu allen Prozeduren des VBA-Projekts hinzufügen

Schließlich schauen wir uns noch an, wie wir allen Prozeduren in allen Modulen eines VBA-Projekts Zeilennummern hinzufügen können. Dazu schreiben wir eine weitere Prozedur, welche wiederum die Prozedur **AddLineNumbersToModule** aufruft:

```
Public Sub AddLineNumbersToActiveVBProject()
    Dim objVBProject As VBIDE.VBProject
    Dim objVBComponent As VBIDE.VBComponent
    Set objVBProject = VBE.ActiveVBProject
    For Each objVBComponent In objVBProject.VBComponents
        Call AddLineNumbersToModule(objVBComponent.Name)
    Next objVBComponent
End Sub
```

```
Public Sub AddLineNumbersToModule(strModule As String)
    Dim objVBProject As VBIDE.VBProject
    Dim objVBComponent As VBIDE.VBComponent
    Dim objCodemodule As VBIDE.CodeModule
    Dim intProcType As vbext_ProcKind
    Dim lngCountOfLines As Long
    Dim lngLine As Long
    Dim strProcedure As String
    Set objVBProject = VBE.ActiveVBProject
    Set objVBComponent = objVBProject.VBComponents(strModule)
    Set objCodemodule = objVBComponent.CodeModule
    lngCountOfLines = objCodemodule.CountOfLines
    For lngLine = 1 To lngCountOfLines
        If Not strProcedure = objCodemodule.ProcOfLine(lngLine, intProcType) Then
            strProcedure = objCodemodule.ProcOfLine(lngLine, intProcType)
            Call AddLineNumbers(strModule, strProcedure)
        End If
    Next lngLine
End Sub
```

Listing 2: Prozedur zum Hinzufügen von Zeilennummern zu allen Prozeduren eines Moduls

Die Prozedur referenziert das aktive VBA-Projekt und durchläuft alle Elemente der **VBComponents**-Auflistung, die sie dabei mit der Variablen **objVBComponent** referenziert. Sie ruft für jeden Schleifendurchlauf die Prozedur **AddLineNumbersToModule** mit dem Namen des aktuellen **VBComponent**-Objekts auf.

Funktion zum Entfernen von Zeilennummern aus einer Prozedur

Gegebenenfalls möchte man auch einmal die Zeilennummern wieder entfernen. Beispielsweise dann, wenn man Zeilen zum Code hinzufügen oder entfernen will und nicht die Nummerierung von Hand anpassen möchte. Dazu können wir die Prozedur **RemoveLineNumbers** aus Listing 3 nutzen.

Achtung: Die Prozedur geht davon aus, dass die Prozedur, aus der die Zeilennummern entfernt werden sollen, mit der weiter oben beschriebenen Prozedur **AddLineNumbers** mit Zeilennummern versehen wurde.

Die Prozedur erwartet die folgenden Parameter:

- **strModule**: Name des Moduls, in dem sich die Prozedur befindet
- **strProcedure**: Name der Prozedur, aus der die Zeilennummern entfernt werden sollen

Sie referenziert das aktuelle VBA-Projekt und dann mit **objVBComponent** die mit **strModule** übergebene Komponente. Mit **objCodeModule** referenzieren wir das **CodeModule**-Element, das wir anschließend nutzen, um mit **ProcBodyLine** die erste Zeile der Prozedur aus **strProcedure** zu ermitteln. Mit dieser können wir nun die beiden Funktionen **GetFirstStatementLineNumber** und **GetLastStatementLineNumber** aufrufen, um die Nummern der ersten und der letzten Zeile innerhalb der Prozedur zu ermitteln. Diese Werte verwenden wir als Bereich von zunächst einer **For...Next**-Schleife.

In dieser untersuchen wir die ersten sechs Zeichen der Zeile. Enthalten diese einen numerischen Wert, ist die Prozedur offensichtlich mit einer Zeilennummerierung ausgestattet. Sobald wir einen numerischen Wert in den ersten sechs

Fehlerhafte Zeilen anzeigen lassen

Im Beitrag »Fehlerbehandlung per VBA hinzufügen« (www.access-im-unternehmen.de/1514) zeigen wir, wie man Fehlerinformationen direkt per VBA in eine Fehlertabelle schreibt, um diese später zu kontrollieren. Wenn man nun als Entwickler eine fehlerhafte Datei mit einigen protokollierten Fehlern vom Benutzer erhält, möchte man vielleicht direkt die fehlerhaften Stellen einsehen. Dazu muss man allerdings erst nachsehen, welches Modul, welche Prozedur und welche Zeile betroffen sind, dann die entsprechende Stelle im VBA-Editor suchen und so weiter. Bei einer umfangreichen Datenbank kann das sehr mühselig werden, vor allem wenn man sich von Fehler zu Fehler hangelt. Deshalb liefern wir in diesem Beitrag noch eine praktische Ergänzung, wenn Sie ohnehin schon eine Tabelle wie aus dem oben genannten Beitrag zum Speichern der Fehler verwenden: Ein Formular, dass diese Fehler anzeigt und mit dem Sie per Mausklick auf einen Fehler direkt die entsprechende Stelle im VBA-Editor anzeigen können.

Ziel des Beitrags

Ziel dieses Beitrags ist es, die Inhalte der Tabelle aus Bild 1 in einem Unterformular in der Datenblattansicht anzuzeigen, um diese dann zu selektieren und die Stelle zu der Fehlermeldung direkt im Quellcode anzuzeigen.

Formular und Unterformular erstellen

Dazu legen wir zunächst ein Unterformular an, dem wir als Datensatzquelle die Tabelle **tblErrors** zuweisen. Wir ziehen alle Felder dieser Tabelle bis auf das Feld **ID** in den Detailbereich des Formularentwurfs und stellen die Eigenschaft **Standardansicht** auf **Datenblatt** ein.

Dann schließen wir das Formular und speichern es unter dem Namen **sfmErrors**. Anschließend legen wir das Hauptformular an. Diesem fügen wir das Unterformular **sfmErrors** per Drag and Drop aus dem Navigationsbereich hinzu.

Damit es seine Größe anpasst, wenn wir die Größe des Formulars ändern, stellen wir die beiden Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** auf den Wert **Beide** ein. Die beiden automatisch geänderten entspre-

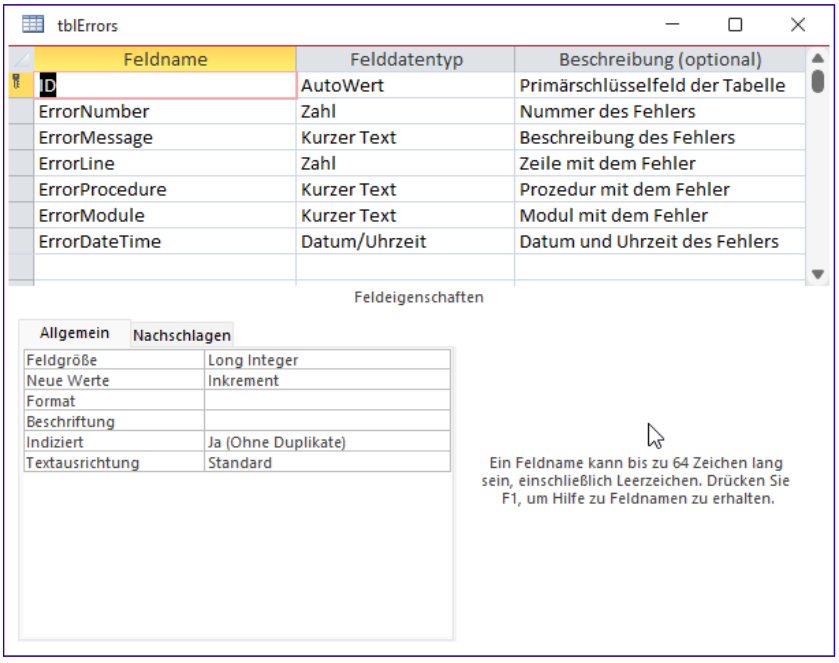


Bild 1: Die Inhalte dieser Tabelle sollen im Formular angezeigt werden

chenden Eigenschaften des Bezeichnungsfeldes stellen wir wieder auf die ursprünglichen Werte **Links** und **Oben** ein.

Außerdem stellen wir die Eigenschaften **Datensatzmarkierer**, **Navigationsschaltflächen**, **Trennlinien** und **Bildlaufleisten** für das Hauptformular auf **Nein** ein.

Schließlich benötigen wir zumindest eine Schaltfläche, mit der wir den aktuell markierten Fehler im VBA-Editor anzeigen können. Dieser geben wir den Namen **cmdShowInVBAEditor** und hinterlegen dann eine Ereignisprozedur für das Ereignis **Beim Klicken**.

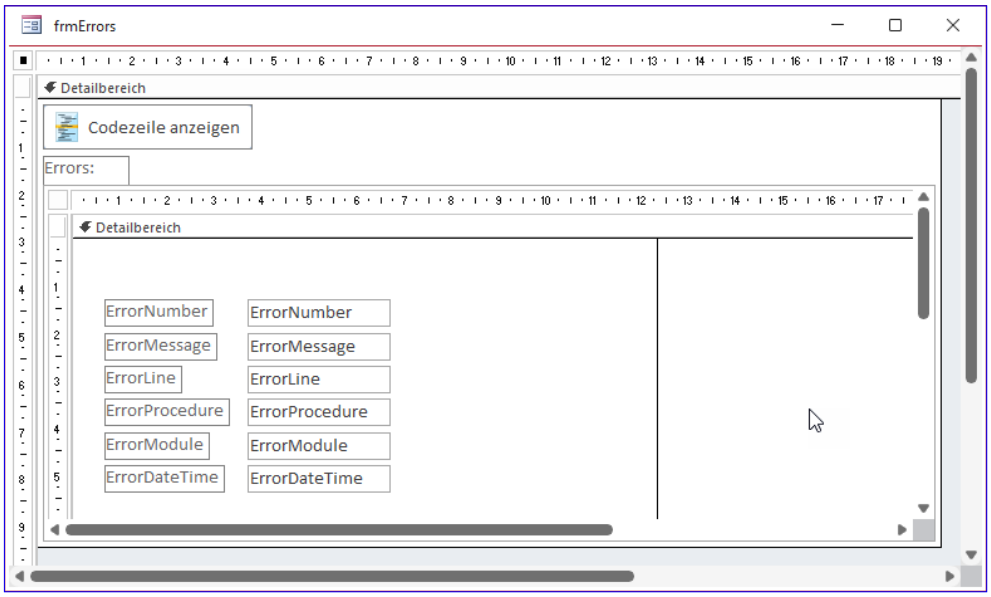


Bild 2: Das Formular **frmErrors** mit Unterformular

Anschließend sieht das Formular in der Entwurfsansicht wie in Bild 2 aus.

Verweis zum Programmieren des VBA-Editors

Bevor wir diese füllen, fügen wir, sofern noch nicht vorhanden, noch einen Verweis auf die Bibliothek Microsoft **Visual Basic Extensibility 5.3 Object Library** hinzu. Diese liefert das Objektmodell, mit dem wir den VBA-Editor programmieren können.

Prozedur zum Anzeigen und Markieren einer Zeile

Anschließend füllen wir die Ereignisprozedur **cmdShowInVBAEditor_Click** mit der Prozedur aus Listing 1.

Die Prozedur liest zuerst die relevanten Informationen für den aktuell markierten Datensatz aus und trägt diese in die Variablen **strErrorModule**, **strErrorProcedure** und **lngErrorLine** ein.

Dann erstellt sie einen Verweis auf das aktuelle VBA-Projekt und holt sich ebenfalls einen Verweis auf das **VBComponent**-Objekt mit dem Namen aus **strErrorModule** sowie auf dessen **CodeModule**-Element.

Dann beginnen wir bereits, die Zeile zu finden, die den in der Tabelle **tblError** enthaltenen Fehler ausgelöst hat. Dazu durchlaufen wir in einer **For...Next**-Schleife alle Zeilen des Moduls von der ersten Zeile bis zu der mit **objCodeModule.CountOfLines** ermittelten letzten Zeile.

Dabei ermitteln wir mit der **Lines**-Funktion die aktuelle Zeile und schreiben diese in die Variable **strLine**. Außerdem ermitteln wir jeweils die Prozedur, zu der die aktuelle Zeile gehört.

Das erledigen wir mit der Funktion **ProcOfLine**, der wir als ersten Parameter die Zeilennummer und als zweiten die leere Variable **intProcType** übergeben (den damit zurückgelieferten Wert benötigen wir nicht, aber es ist ein Pflichtparameter).

Wir prüfen dann für jede Zeile, ob die Nummer der fehlerhaften Zeile mit der Zeilennummer der aktuellen Zeile

Fehler in der Runtime von Access finden

Wenn wir eine Datenbank, die unter der Vollversion von Access fehlerfrei läuft, in der Runtime öffnen, kann es zu unerklärlichen Fehlern kommen. Die Runtime verabschiedet sich dann in der Regel mit einer Meldung wie »Die Ausführung dieser Anwendung wurde wegen eines Laufzeitfehlers angehalten.« Damit können wir natürlich erst einmal nicht viel anfangen. Anlass zu diesem Beitrag ist ein konkretes Problem einer Kundin, deren Anwendung auf der Runtime-Version von Access nicht wie gewünscht funktioniert. Beim Testen der Anwendung in der Runtime kam ich jedoch gar nicht erst soweit wie die Kundin. Es tauchte bereits vorher die besagte Meldung auf. Wie können wir nun herausfinden, was genau den Fehler verursacht und welche Zeile ihn auslöst? Vorgehensweisen dazu stellen wir in diesem Beitrag vor.

Optimale Lösung: Die Anwendung ist vollständig mit einer Fehlerbehandlung ausgestattet.

Probleme wie das geschilderte sollten eigentlich gar nicht in dieser Form auftreten. »In dieser Form« bedeutet, dass es durchaus einmal vorkommen kann, dass eine Anwendung unter der Runtime-Version von Access anders arbeitet als unter der Vollversion.

Aber wenn dabei Probleme auftauchen, werden diese in der Regel durch Fehler im VBA-Code ausgelöst. Und wenn die Meldung aus Bild 1 erscheint, ist die Wahrscheinlichkeit hoch, dass es in der Runtime einen Fehler gibt, der durch eine VBA-Anweisung ausgelöst wurde.

Eine professionell programmierte Anwendung würde in jeder Prozedur eine Fehlerbehandlung enthalten, die beim Auftreten eines Fehlers dafür sorgt, dass diese nicht die eingebaute Access-Fehlermeldung auslöst. Genau diese wird nämlich durch die hier gezeigte Fehlermeldung der Runtime-Version kaschiert.

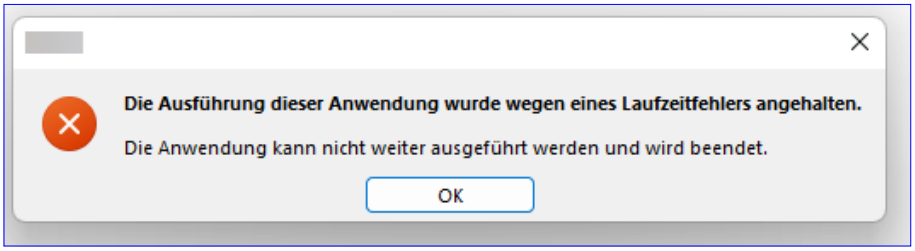


Bild 1: Typische Fehlermeldung der Runtime

Fehler im Griff mit umfassender Fehlerbehandlung

Wenn wir flächendeckend unsere eigene Fehlerbehandlung in den Routinen der Anwendung untergebracht hätten, wäre die eingebaute Fehlermeldung der Runtime gar nicht nötig. Eine solche Fehlerbehandlung ist relativ einfach zu realisieren, wenn man dies gleich von Beginn an berücksichtigt. Wenn man jedoch erst einmal eine umfangreiche Anwendung programmiert hat und anschließend die Fehlerbehandlung hinzufügen will, kommt eine Menge Arbeit auf einen zu.

Fehlerbehandlung per MZ-Tools

Auch diese kann man weitgehend vereinfachen, indem man ein Werkzeug wie **MZ-Tools** zum halbautomatischen Anlegen der Fehlerbehandlung nutzt. Mit MZ-Tools können wir sogar automatisch Zeilennummern hinzufügen, was in einem Runtime-Setting essenziell ist, denn sonst können wir kaum herausfinden, wo in einer längeren Prozedur genau der Fehler aufgetreten ist.

Fehlerbehandlung per vbWatchDog

Die nächste Möglichkeit ist der **vbWatchDog**, ein Add-In von Access-Spezialist Wayne Philips. Das ist ein Tool, mit dem man nur durch Hinzufügen einiger weniger Objekte und ein wenig Code die komplette Anwendung mit einer allgemeinen Fehlerbehandlung ausstattet, ohne den einzelnen Prozeduren auch nur eine einzige Zeile Code hinzuzufügen (es sei denn, es ist eine spezielle Fehlerbehandlung erwünscht).

Problem bei der Runtime: Kein Debug.Print möglich

Wenn wir eine Anwendung in der Vollversion testen, um ein eventuell fehlerhaftes Verhalten zu prüfen, hilft uns oft die **Debug.Print**-Anweisung weiter. Damit können wir praktisch vor oder nach jeder Zeile Informationen im Direktbereich des VBA-Editors ausgeben.

Leider hilft uns auch das in einer unter der Runtime-Version von Access ausgeführten Anwendung nicht weiter, denn hier gibt es gar keinen VBA-Editor, mit dem wir uns die Ausgabe ansehen könnten.

Herausfinden, wo ein Fehler stattfindet, ohne alles mit Fehlerbehandlungen auszustatten

Typischerweise funktionieren Dinge, die unter einer Access-Vollversion reibungslos laufen, gelegentlich in der Runtime-Version nicht. Wenn wir keine Fehlerbehandlung eingebaut haben und vielleicht auch gerade nicht die Ressourcen verfügbar sind, um das nachzuholen, müssen wir ein wenig kleinschrittiger an das Problem herangehen.

Wir nehmen nun zusätzlich noch an, dass wir es mit einer Anwendung eines Kunden zu tun haben, die wir nicht so genau kennen wie unsere selbst programmierten Anwendungen.

Dann sehen wir vielleicht vor dem Erscheinen der ominösen Fehlermeldung noch, welche Formulare angezeigt werden, aber welche Prozedur den Fehler ausgelöst hat, finden wir auf diese Weise auch nicht heraus.

Wenn wir nicht alle Prozeduren mit einer Fehlerbehandlung versehen wollen, müssen wir einen anderen Weg gehen, um herauszufinden, welche Stelle im Code den Fehler auslöst. Dazu gehen wir wie folgt vor:

- Wir fügen jeder Prozedur eine Anweisung hinzu, die unmittelbar hinter der Kopfzeile der Prozedur angelegt wird. Diese soll eine Hilfsfunktion aufrufen, die den Namen des Moduls, den Namen der Prozedur und Datum und Uhrzeit des Fehlers notiert. Die Hilfsfunktion soll diese Daten in eine eigens dafür angelegte Tabelle schreiben.

- Dann kopieren wir die Datenbank auf den Rechner mit der Runtime-Version von Access und führen diese dort aus. Wir sollten nun den Fehler wie gewohnt erhalten.

- Der Unterschied ist nun jedoch, dass wir in der Tabelle zum Aufzeichnen der aufgerufenen Prozeduren einige Einträge vorfinden, von denen der letzte vermutlich die Prozedur enthält, die den Fehler ausgelöst hat. Um diese zu lesen, kopieren wir die Datenbank wieder auf den Rechner mit der Vollversion von Access zurück.

- Für die gefundene Prozedur fügen wir nun eine Fehlerbehandlung hinzu sowie eine Zeilennummerierung. Auch die hierbei anfallenden Informationen schreiben wir wieder in eine Tabelle.

- Wenn wir die Datenbank nun wieder auf den Runtime-Rechner verschieben und ausführen, wird Access den Fehler nicht erneut anzeigen, da dieser ja nun behandelt wird. Es ist jedoch wahrscheinlich, dass nun ein Folgefehler dazu führt, dass die von der Runtime gelieferte Fehlermeldung erscheint. Das ist jedoch erst einmal irrelevant – wir wollen nun erst einmal diesen Fehler untersuchen und beheben.

- Auf diese Weise arbeiten wir uns durch die Fehler, die im Runtime-Betrieb auftauchen und beheben diese.

```
Public Sub WriteCall(strProcedure As String, strModule As String, bolBeginning As Boolean)
    Dim db As DAO.Database
    Set db = CurrentDb
    db.Execute "INSERT INTO tblCalls(CallProcedure, CallModule, CallTime, Beginning) VALUES('" & strProcedure _
        & "', '" & strModule & "', " & Format(Now, "\#yyyy-mm-dd hh:nn:ss\#") & ", " & CInt(bolBeginning) & ")", _
        dbFailOnError
End Sub
```

Listing 1: Prozedur zum Schreiben der Informationen zu einem Prozeduraufruf

Optimalerweise fügt man einer Anwendung für Runtime jedoch dennoch bei nächster Gelegenheit eine Fehlerbehandlung zu.

Untersuchen, welche Prozedur vermutlich den Fehler auslöst

Damit kommen wir zum ersten Schritt. Wir fügen einer Prozedur den folgenden Aufruf hinzu:

```
Public Sub BeispielCall()
    Call WriteCall("BeispielCall", _
        "mdlErrors", True)
    '... weitere Anweisungen
End Sub
```

Die dadurch aufgerufene Prozedur sieht wie in Listing 1 aus. Sie erwartet den Namen der Prozedur und des Moduls als Parameter sowie einen Parameter namens **bolBeginning**.

Diesen stellen wir auf True ein, wenn wir den Aufruf von **WriteCall** am Anfang der Prozedur platzieren und auf **False**, wenn wir das Prozedurende verwenden.

Die Tabelle, die wir mit der **INSERT INTO**-Anweisung füllen wollen, finden wir in der Entwurfsansicht in Bild 2.

Wenn wir den Aufruf der Prozedur **WriteCall** in einige Prozeduren geschrieben und die Datenbank ausgeführt haben, sieht ihr Inhalt beispielsweise wie in Bild 3 aus. Nun stellt sich noch die Frage: Wie bekommen wir den Aufruf schnell in alle Prozeduren?

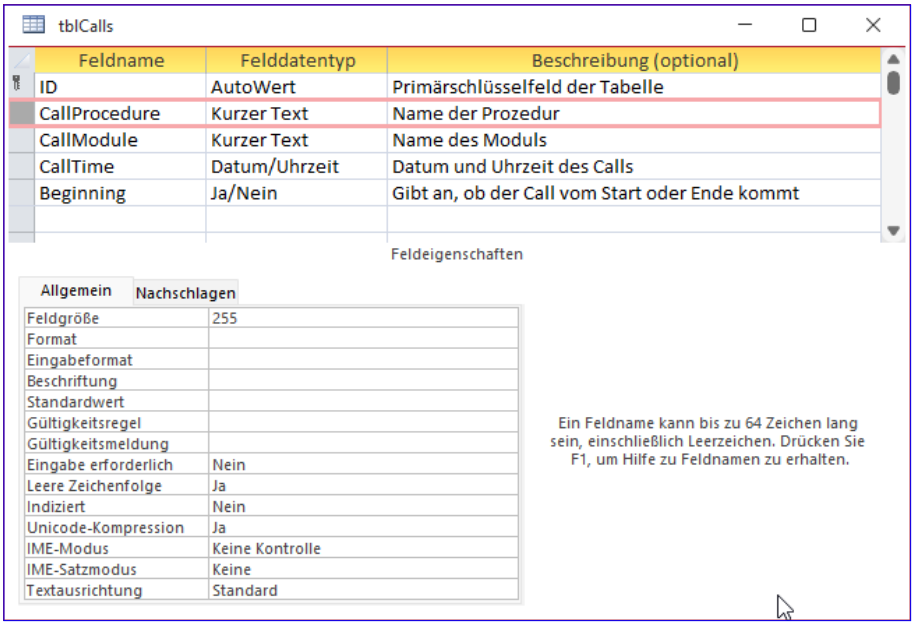


Bild 2: Entwurf der Tabelle für die Aufrufe

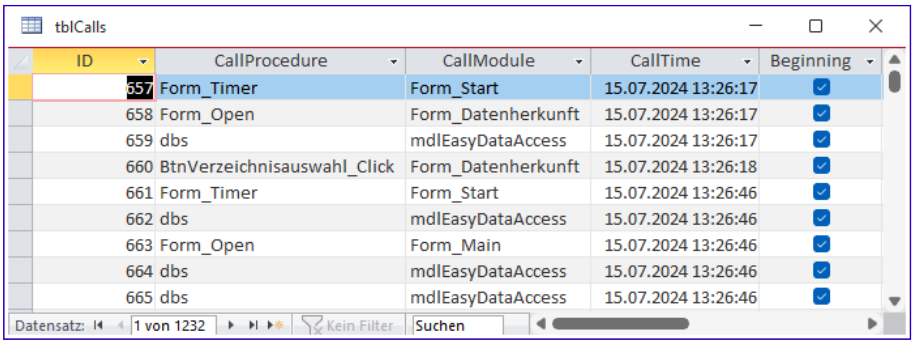


Bild 3: Tabelle mit einigen aufgezeichneten Prozedur-Calls

Aufruf der Prozedur WriteCall in alle Routine schreiben

Wir haben bereits davon gesprochen, dass unsere Anwendung recht viele Prozeduren hat. Wie also sollen wir mit vertretbarem Aufwand den Aufruf der Prozedur **WriteCall** in alle Routinen der Anwendungen schreiben?

Logisch: Copy und Paste nimmt uns bereits eine Menge Arbeit ab, und wenn wir den Namen des Moduls bereits vorab eintragen, brauchen wir nach dem Kopieren und Einfügen nur noch den Namen der jeweiligen Routine hinzuzufügen:

```
Call WriteCall("BeispielCall", "mdlErrors",
True)
```

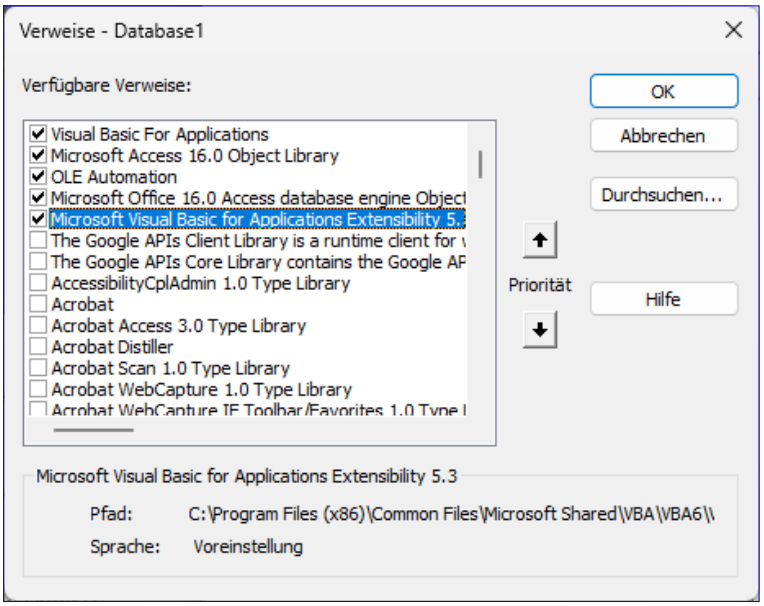


Bild 4: Verweis auf die Bibliothek **Microsoft Visual Basic for Applications 5.3**

Allerdings ist das bei ein paar hundert Routinen immer noch ein hoher Zeitaufwand und zweitens wäre diese Vorgehensweise recht fehleranfällig.

Also programmieren wir uns eine eigene Prozedur, mit der wir diese Zeile zu allen Prozeduren hinzufügen. Mit dieser greifen wir auf das VBA-Projekt der aktuellen Datenbank zu, also brauchen wir das entsprechende Objektmodell. Dieses holen wir uns durch Setzen des Verweises aus Bild 4 hinzu.

Danach programmieren wir die Prozedur aus Listing 2. Diese referenziert zunächst das aktuelle VBA-Projekt und speichert es in **objVBProject**. Das VBA-Projekt enthält für jedes Modul, egal ob Standard- oder Klassenmodul, ein **VBComponent**-Objekt. Diese durchlaufen wir in einer **For Each**-Schleife über alle Elemente der Auflistung **VBComponents**.

Uns interessiert das im **VBComponent**-Element enthaltene **CodeModule**-Objekt, das wir mit **objCodeModule** referenzieren. In **strModul** speichern wir den Namen dieses Moduls. Die folgenden Schritte sollen für alle Module mit Ausnahme des aktuellen Moduls namens **mdlErrors** ausgeführt werden.

Wir durchlaufen nun alle Zeilen und identifizieren jeweils die erste Zeile der nächsten Prozedur. Dazu stellen wir **strProcedure** zunächst auf eine leere Zeichenkette ein. Dann ermitteln wir die Anzahl der Zeilen des Moduls und speichern diese in **lngCountOfLines**. Die **Boolean**-Variable **bolGeaendert**, die angeben soll, ob wir in dem Modul mindestens eine Zeile geändert haben, stellen wir zuerst auf **False** ein.

Dann durchläuft die Prozedur alle Zeilen von **1** bis **lngCountOfLines** in einer **For...Next**-Schleife. Dabei prüft sie jeweils, ob der in **strProcedure** gespeicherte Prozedurname mit dem Namen der Prozedur aus der aktuellen Zeile übereinstimmt. Ist das der Fall, befinden wir uns noch in der aktuellen Prozedur (oder, wenn wir uns am Modulanfang befinden, im Deklarationsbereich – hier enthält **strProcedure** immer noch eine leere Zeichenkette).

Den Namen der Prozedur, zu der die aktuelle Zeile gehört, ermitteln wir mit der Funktion **ProcOfLine**. Dieser übergeben wir die Zeilennummer und eine Variable, die den Typ

SQL Server Data Tools installieren und starten

Die SQL Server Data Tools sind eine Erweiterung für Visual Studio, mit denen wir interessante Aufgaben rund um die Verwaltung von SQL Server-Datenbanken erledigen können, die teilweise nicht mit dem SQL Server Management Studio möglich sind. Dazu gehört unter anderem das Vergleichen der Schemata zweier Datenbanken. Diese werden jedoch nicht zwingend direkt mit Visual Studio installiert, sodass wir uns in diesem Beitrag kurz ansehen, wie wir dies nachholen können – bevor wir uns in weiteren Beiträgen ansehen, was wir mit den SQL Server Data Tools alles erledigen können.

Das SQL Server Management Studio ist ein extrem mächtiges Werkzeug zum Verwalten von SQL Server-Datenbanken. Damit können wir nicht nur die Datenbanken selbst erstellen und bearbeiten, sondern auch sehr viele administrative Aufgaben erledigen.

Allerdings gibt es ein paar Funktionen, die man hier vermisst – zum Beispiel das Vergleichen der Schemata zweier Datenbanken, um herausfinden, welche Änderungen zwischen zwei Versionen durchgeführt wurden. Solche Funktionen liefert allerdings Visual Studio mit, zumindest wenn man die SQL Server Data Tools installiert hat.

Wenn man Visual Studio bereits installiert hat, kann man leicht prüfen, ob die SQL Server Data Tools installiert sind – dann finden wir nämlich den Eintrag **Ansicht|SQL Server-Objekt-Explorer** im Menü von Visual Studio. Finden wir diesen Eintrag nicht, können wir diesen jedoch noch hin-

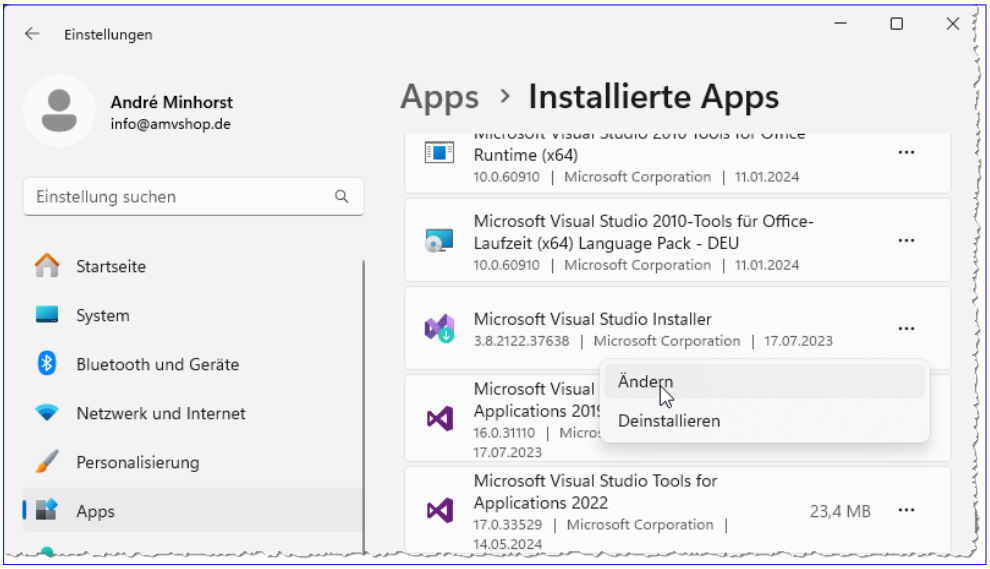


Bild 1: Starten der Änderung von Visual Studio

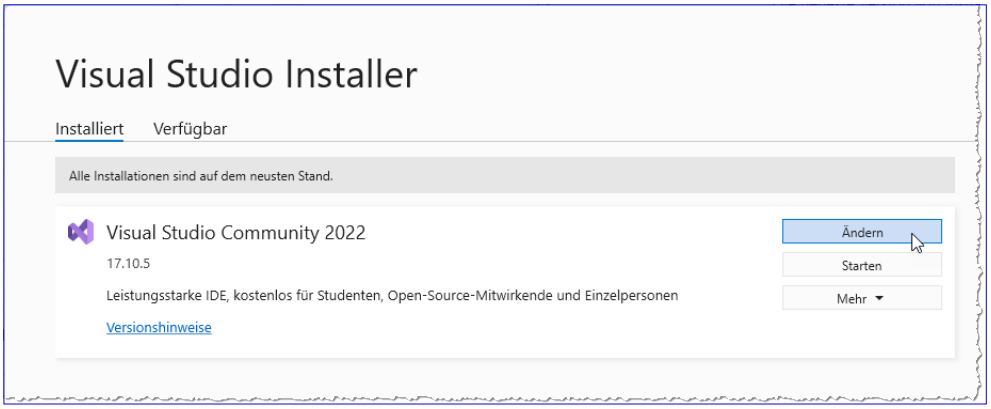


Bild 2: Ändern von Microsoft Visual Studio

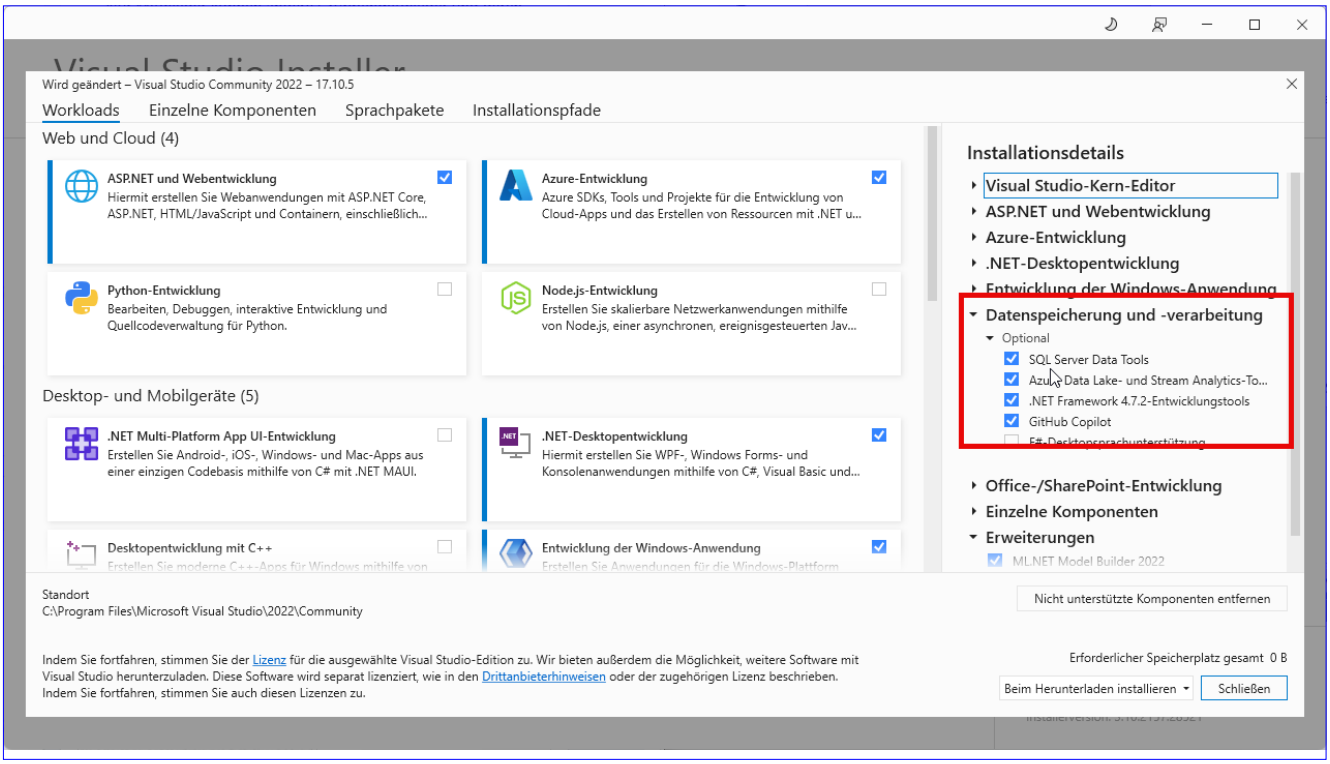


Bild 3: Hinzufügen der SQL Server Data Tools

zufügen. Dazu geben wir zunächst den Begriff **Programme** in die Windows-Suche ein und klicken doppelt auf den nun erscheinenden Eintrag **Programme hinzufügen oder entfernen**.

Dies öffnet den Dialog aus Bild 1, in dem wir den Eintrag **Microsoft Visual Studio Installer** auswählen und auf die Schaltfläche mit den drei Punkten auf der rechten Seite klicken. Im nun erscheinenden Menü klicken wir auf **Ändern**.

Dies öffnet wie in Bild 2 den Visual Studio Installer, wo wir die Schaltfläche **Ändern** betätigen.

Der folgende Dialog zeigt die Seite **Workload** an (siehe Bild 3). Hier sollten wir nun auf der rechten Seite unter Installationsdetails den Punkt **Datenspeicherung und -verarbeitung** finden, den wir aufklappen können. Hier befindet sich ein Unterpunkt namens **SQL Server Data Tools**, den wir nun aktivieren. Dies wandelt die hier noch sichtbare Schaltfläche **Schließen** in die Schaltfläche **Ändern** um, die wir nun zum Installieren anklicken.

Bild 4: Starten ohne Projekt

SQL Server Data Tools öffnen

Um die SQL Server Data Tools nun zu nutzen, starten wir Visual Studio. Da wir kein Projekt dazu benötigen, klicken wir im Startdialog auf den kleinen Link **Ohne Code fortfahren** (siehe Bild 4).

Nun finden wir im Menü Ansicht den Eintrag **SQL Server-Objekt-Explorer** vor (siehe Bild 5).

SQL Server-Datenbank kopieren

Wer für einen Kunden Datenbanken auf Basis von SQL Server programmiert, tut gut daran, nicht am offenen Herzen zu operieren, sprich: Änderungen an einer Datenbank erst an einer Kopie dieser Datenbank auszuprobieren. Wer seine eigenen Datenbanken auch auf dem SQL Server verwaltet, gerät da schon leichter in Versuchung, mal eben schnell eine Änderung am laufenden Produktivsystem durchzuführen. Auch wenn man meint, man wüsste genau, was man tut, macht es doch Sinn, zumindest eine Sicherheitskopie der zu bearbeitenden Datenbank anzulegen. Nun bietet der SQL Server keine einfache Lösung zum Herstellen einer Kopie á la Strg + C, Strg + V an. Allerdings wird man dennoch schnell fündig, wenn man nach einer entsprechenden Lösung sucht – in diesem Fall allerdings nur in höheren SQL Server-Editionen, also nicht in der Express Edition.

Logisch: Bevor man einem Kunden eine neue Version einer Datenbank übermittelt, wird diese erst einmal ordentlich getestet. Ohnehin werden Änderungen ja meist weniger durch Kopieren der kompletten Datenbank übertragen, sondern durch Hinzufügen der neuen Elemente und Entfernen oder Ändern vorhandener Elemente durch Methoden wie einem SQL-Skript realisiert.

Wenn man seine eigene Datenbank auf dem eigenen Rechner betreibt, kann es schon eher passieren, dass man mal eben schnell eine Änderung durchführt. Ein neues Feld, eine neue Tabelle, eine neue Beziehung sind schnell angelegt. Was aber, wenn man eine Änderung durchführt, die dazu führt, dass die Anwendung nicht mehr wie gewohnt funktioniert? Wenn man dann nicht regelmäßig eine Sicherung angelegt hat, kann die Wiederherstellung der letzten funktionierenden Variante schon aufwendiger werden.

Was man aber tun könnte, ist das Anlegen einer Kopie der SQL Server-Datenbank, mit der man die geplanten

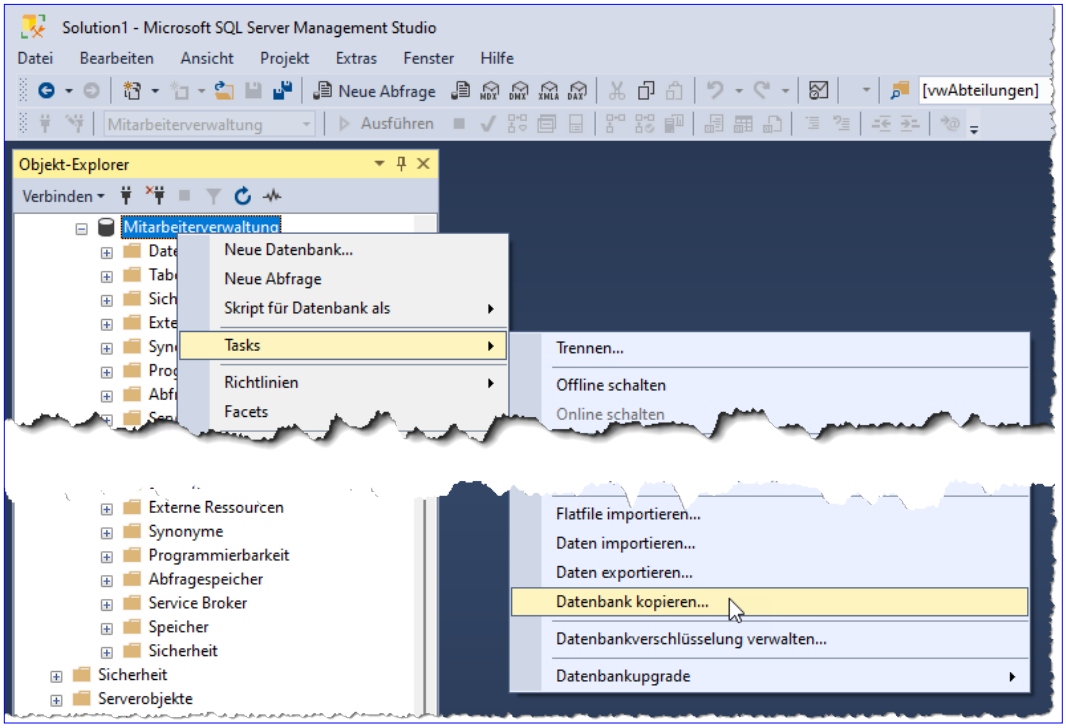


Bild 1: Kopieren einer SQL Server-Datenbank

Änderungen erst einmal durchführt und diese dann nach erfolgreichem Test auf die Produktdatenbank überträgt.

Dazu fehlt nur noch eine einfache Methode, mal eben eine Kopie der Datenbank anzulegen. Diese Methode ist allerdings, wenn man das SQL Server Management Studio verwendet, gar nicht so weit weg. Wir brauchen dazu auch keine Sicherung anzulegen und diese unter einem neuen Namen wiederherzustellen, sondern wir können einfach die eingebaute Funktion zum Kopieren einer Datenbank verwenden.

Datenbank kopieren als Task

Diese Methode finden wir, wenn wir das Kontextmenü für die zu kopierende Datenbank aufklappen und dort den Eintrag **Tasks|Datenbank kopieren...** auswählen (siehe Bild 1).

Auswählen des Quellservers

Im nächsten Schritt aus Bild 2 wählen wir den Quellserver aus, also den Datenbankserver, auf dem sich die zu kopierende

Bild 2: Erster Schritt im Assistenten zum Kopieren von Datenbanken

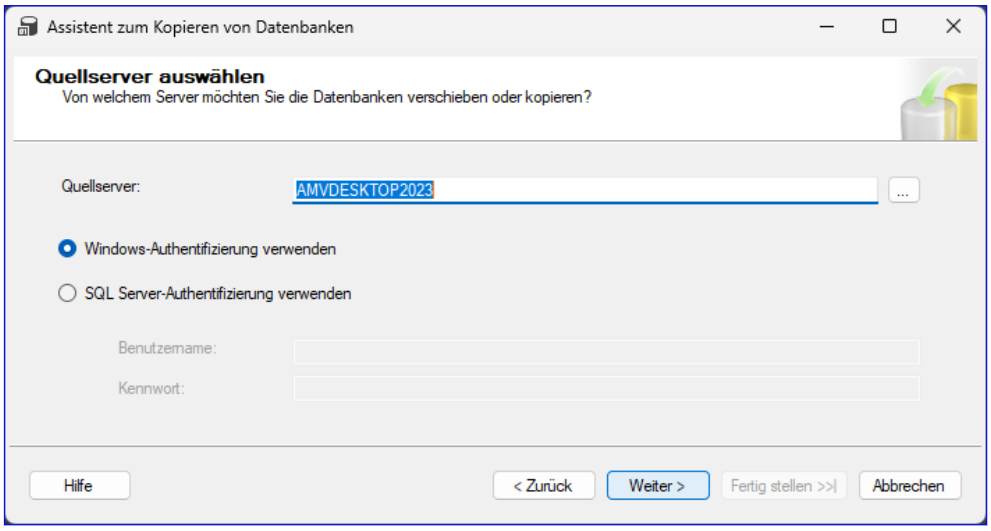


Bild 3: Festlegen des Quellservers

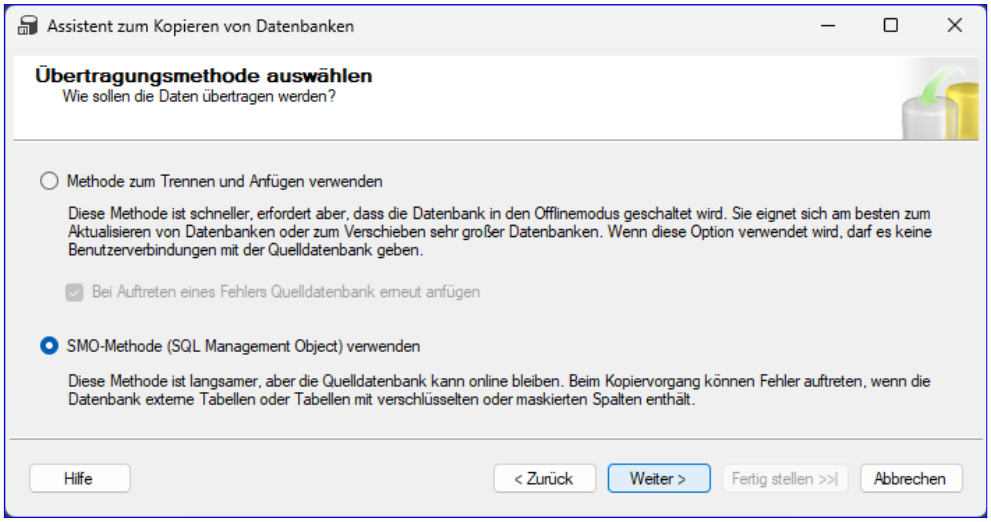


Bild 4: Auswählen der Übertragungsmethode

Schemata von SQL Server-Datenbanken vergleichen

Es kann in verschiedenen Szenarien sinnvoll sein, einmal den Unterschied zwischen zwei Versionen einer SQL Server-Datenbank zu vergleichen. Gibt es überhaupt Unterschiede? Welche Unterschiede sind das? Wurden Tabellen, Felder, Sichten, gespeicherte Prozeduren oder Trigger hinzugefügt oder entfernt? Wenn wir das herausfinden, können wir zum Beispiel identifizieren, welche Änderungen seit der letzten Veröffentlichung einer Datenbank als Produktivdatenbank in der Entwicklungsdatenbank durchgeführt wurden oder wir können herausfinden, wodurch ein Fehler ausgelöst werden könnte, der seit einem bestimmten Versionsstand immer wieder auftritt. In diesem Beitrag schauen wir uns zunächst einmal an, wie wir die Unterschiede zwischen den Schemata zweier Datenbanken ermitteln können. Dazu nutzen wir die SQL Server Data Tools, die wir in einem anderen Artikel bereits vorgestellt haben.

Unter dem Titel **SQL Server Data Tools installieren und starten** (**www.access-im-unternehmen.de/1520**) haben wir uns bereits angesehen, wie wir die SQL Server Data Tools installieren und starten können. Die SQL Server Data Tools sind ein Bestandteil von Visual Studio.

Die SQL Server Data Tools liefern unter anderem eine komfortable Möglichkeit zum Vergleichen von Datenbankschemata. Diese schauen wir uns auf den nächsten Seiten als Erstes an. Danach gehen wir noch einen Schritt weiter und untersuchen, ob und wie wir die Kenntnis dieser Unterschiede dafür nutzen können, eine der Datenbanken auf den Stand der anderen Datenbank zu bringen.

Datenbankschemata vergleichen

Warum sollte man überhaupt zwei Datenbankschemata vergleichen? Meist handelt es sich dabei um zwei Versionen einer Datenbank. Es kann sein, dass wir eine Version einer SQL Server-Datenbank entwickelt und beim Kunden oder auch bei uns selbst in den Produktivbetrieb übernommen haben. Nun entwickelt man solche Datenbanken normalerweise in einer Entwicklungsversion weiter, bis man den gewünschten Stand erreicht hat. Dann möchte man natürlich die Änderungen, die man in der Zwischenzeit durchgeführt hat, irgendwann auch auf die Produktivdatenbank übertragen. Hier gibt es zwei Wege:

- Wir legen die Entwicklungsdatenbank als neue Produktivdatenbank an und kopieren die Daten der alten Version der Produktivdatenbank in die neue Version.
- Oder wir übertragen die Änderungen, die am Datenbankschema durchgeführt wurden, in die aktuelle Version der Produktivdatenbank und bringen diese somit auf den aktuellen Entwicklungsstand.

In diesem Beitrag schauen wir uns die zuletzt genannte Variante an.

Es gibt auch noch weitere Gründe, sich die Unterschiede zwischen zwei Datenbankschemata anzuschauen. Es kann auch passieren, dass man versehentlich (oder auch absichtlich) an zwei verschiedenen Versionen unterschiedliche neue Elemente hinzufügt. Wenn man dies nicht sauber dokumentiert, kann man auch hier im Nachhinein den nachfolgend vorgestellten Schemavergleich nutzen, um die Unterschiede herauszufinden und die gewünschten Elemente bei einer der Datenbanken nachträglich hinzuzufügen.

Beispieldatenbanken

Um die nachfolgend vorgestellten Techniken zu testen, benötigen wir zwei Datenbanken, deren Schemata möglichst vielfältige Unterschiede aufweisen. Vielleicht finden Sie zu einer aktuellen SQL Server-Datenbank noch eine alte Sicherung, die Sie zu diesem Zweck einmal wiederherstellen können (unter anderem Namen, natürlich).

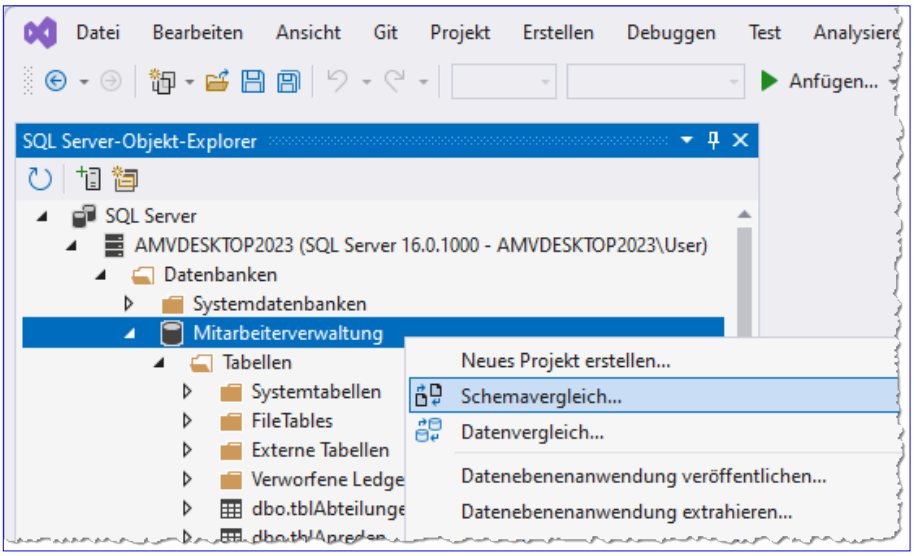


Bild 1: Starten eines Schemavergleichs

Diese beiden Datenbanken sollten von der gleichen Instanz der SQL Server Data Tools aus erreichbar sein, was aber in der Regel kein Problem sein sollte.

Schemavergleich á la SSDT

Dann können wir bereits beginnen und wie im oben genannten Beitrag beschrieben Visual Studio und den SQL Server-Objekt-Explorer starten. Hier wählen wir nun die erste Datenbank aus, deren Schema wir vergleichen wollen. Wir sollten hier direkt die ältere Datei wählen, denn später bekommen wir die Gelegenheit die Unterschiede aus der Sicht dieser Datenbank als Basis für eine Aktualisierung zu nutzen. Dann klicken wir mit der rechten Maustaste auf den Datenbanknamen. Im nun erscheinenden Kontextmenü finden wir den Eintrag **Schemavergleich...** vor, den wir nun betätigen (siehe Bild 1).

Es erscheint ein neuer Bereich, der oben zwei Auswahlfelder bereithält, mit denen wir die Quelldatenbank und die Zieldatenbank auswählen können. Die Auswahlliste für die Quelldatenbank wurde bereits mit der Datenbank gefüllt, für die wir den Kontextmenübefehl aufgerufen haben. Die Zieldatenbank wählen wir aus, indem wir die Liste ausklappen und wie in Bild 2 auf **Ziel auswählen...** klicken.

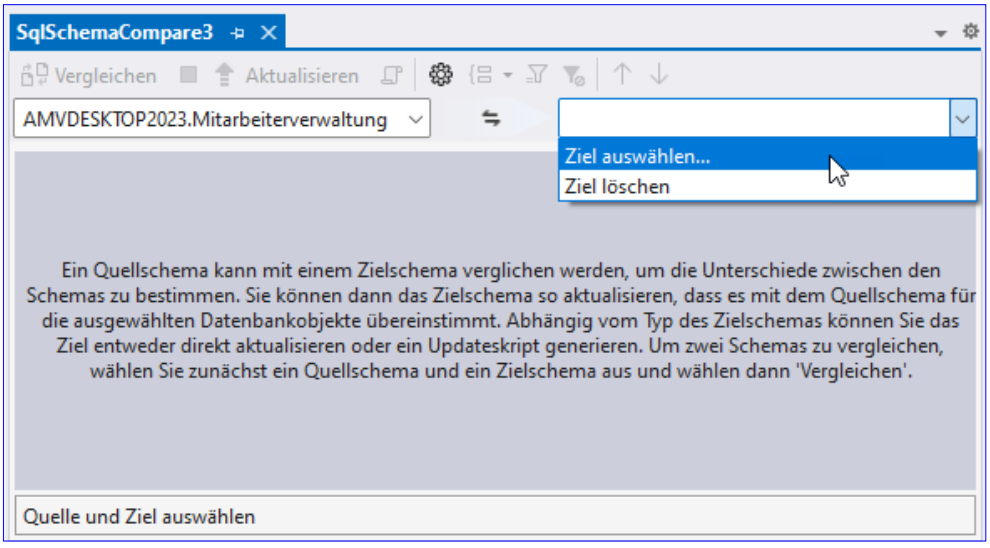


Bild 2: Auswählen von Quelle und Ziel des Schemavergleichs

Im Text im grauen Bereich können wir bereits lesen, dass wir hier nicht nur die Unterschiede zwischen den beiden Schemata ermitteln können. Wir haben anschließend die Möglichkeit, das Ziel zu aktualisieren

Flexible Schnellsuche

Wer kennt das nicht als Entwickler: Man möchte mal eben schnell in einer Tabelle nach einem bestimmten Datensatz suchen. Und das auch noch wiederholt, sodass man immer noch den Filter entfernen und diesen neu setzen muss – wozu wir die eingebauten Filter-Elemente der Datenblattansicht nutzen. Viel schöner wäre es doch, wenn wir mal eben das Kriterium und weitere Einstellungen über das Ribbon steuern könnten – während wir die Daten beim Filtern betrachten. Genau dazu haben wir uns für den Eigenbedarf eine kleine Schnellsuche gebaut, die wir in diesem Beitrag vorstellen. Das Beste: Selbst wenn Sie gar nicht genau erfahren wollen, wie es funktioniert, finden Sie die Lösung für 32-Bit- und 64-Bit-Access im Download zu diesem Beitrag.

Das Tool soll gar keine aufwendigen Aufgaben erledigen: Wir wollen schlicht und einfach im Ribbon ein Feld haben, in das wir einen Vergleichswert eingeben können.

Die Suche wollen wir durch Betätigen der Eingabetaste oder einer Schaltfläche starten.

Zum einen möchten wir nur das aktuell markierte Feld oder alle Felder nach dem gesuchten Inhalt filtern und zum anderen möchten wir festlegen, ob der gesamte Feldinhalt mit dem Suchbegriff übereinstimmen muss oder ob der Suchbegriff nur im Feld enthalten sein soll. All dies sehen wir in Bild 1. Hier haben wir das Textfeld zur Eingabe des Filters, die beiden Schaltflächen zum Aktivieren und Deaktivieren des Filters sowie die beiden Optionen für das Durchsuchen des kompletten Feldinhalts und für das aktuelle Feld oder alle Felder.

Eine solche Erweiterung können wir einer einzelnen Access-Anwendung hinzufügen, indem wir die ent-

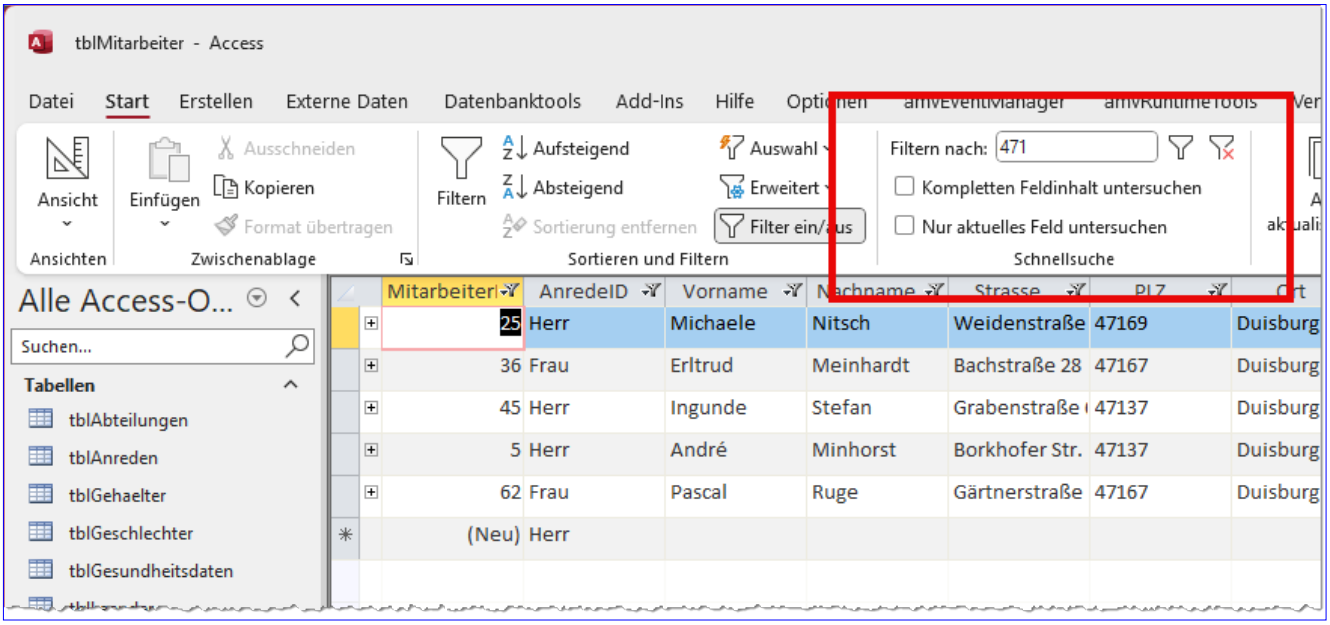


Bild 1: Ribbon-Funktion zur Schnellsuche in Datenblättern und Formularen

sprechende Ribbondefinition anlegen und die dadurch aufgerufenen Funktionen hinzufügen.

COM-Add-In mit twinBASIC

Wir wollen diese Funktion allerdings in allen Access-Anwendungen nutzen, jedoch nicht die Funktion in alle Access-Anwendungen einbauen. Deshalb haben wir ein COM-Add-In auf Basis von twinBASIC programmiert. Wie das COM-Add-In funktioniert, zeigen wir in diesem Beitrag. Wenn Sie das COM-Add-In anpassen oder erweitern oder ein neues COM-Add-In auf Basis dieses Add-Ins erstellen wollen, sind Sie nach der Lektüre dieses Beitrags bestens gerüstet. Alles, was man braucht, ist die aktuelle Version der für 32-Bit-Anwendungen kostenlosen twinBASIC-Entwicklungsumgebung, zum Beispiel von hier:

```
https://github.com/twinbasic/twinbasic/releases
```

Dabei handelt es sich um eine **.exe**-Datei mit einigen weiteren Dateien, die Sie nur auf Ihre Festplatte extrahieren und starten müssen, zum Beispiel durch einen Doppelklick auf die Datei **amvFastSearch.twinproj** aus dem Download zu diesem Beitrag.

Code des COM-Add-Ins

Das Projekt enthält nur zwei Module. Das erste namens **DllRegistration.twin** enthält lediglich die beiden Funktionen, die beim Registrieren und beim Aufheben der Registrierung durchgeführt werden sollen. Diese werden sowohl ausgeführt, wenn wir in twinBASIC den Befehl **File|Build** aufrufen als auch dann, wenn wir die fertig erstellte **.dll**-Datei mit dem Befehl **Regsvr32.exe** registrieren. Diese enthalten die Informationen, die dann in die Registry geschrieben werden sollen und beim Start von Access ausgelesen werden. Weiter unten zeigen wir jedoch auch noch, wie die Registrierung über die Benutzeroberfläche funktioniert.

Beim Start des COM-Add-Ins

Ist das COM-Add-In einmal registriert, wird es beim Start von Access eingelesen und verschiedene Ereignisprozeduren der beiden Schnittstellen des COM-Add-Ins werden ausgelöst. Das COM-Add-In implementiert die beiden Schnittstellen **IDTExtensibility2** und **IRibbonExtensibility**. Die erste sorgt dafür, dass das COM-Add-In beim Starten von Access ebenfalls gestartet wird und einen Verweis auf die Access-Instanz erhält. Die zweite enthält nur eine Funktion, mit der sich Access eine Ribbondefinition holen kann, die wir im COM-Add-In zusammenstellen.

Bevor wir uns die Prozeduren ansehen, die beim Start ausgelöst werden, werfen wir einen Blick auf einige Variablen, die wir im Kopf der Klasse **amvFastSearch** unterbringen:

```
Private objAccess As Access.Application
Private objRibbon As IRibbonUI
Private bolKompletterFeldinhalt As Boolean
Private bolNurAktuellesFeld As Boolean
Private strVergleichswert As String
```

Die erste nimmt einen Verweis auf die Access-Instanz auf, die das COM-Add-In verwendet. Die zweite erhält nach dem Erstellen des Ribbons einen Objektverweis auf die Definition, damit wir darüber auf bestimmte Ereignisse reagieren können.

Die übrigen Variablen sind bereits für das Speichern der Informationen vorgesehen, die wir über die Ribbon-Benutzerschnittstelle des Tools eingeben.

Beim Verbinden des COM-Add-Ins

Wenn Access gestartet wird und das COM-Add-In über die Registry als aktiv identifiziert, wird dieses aufgerufen und erhält über die Ereignisprozedur **OnConnection** mit dem Parameter Application einen Verweis auf die aufrufende Access-Instanz. Dieser wird direkt in der Variablen **objAccess** gespeichert:

```
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
```

Es gibt noch weitere Ereignisprozeduren für das COM-Add-In, aber die sind für uns nicht weiter interessant. Wichtig ist nur, dass sie überhaupt implementiert werden.

Ribbondefinition zusammenstellen

Die Funktion **GetCustomUI** stellt die Ribbondefinition zusammen und gibt diese an Access zurück, damit diese dort angewendet werden kann (siehe Listing 1). Das Ribbon besteht aus einem **customUI**-Element mit den Callback-Ereignissen **loadImage** und **onLoad**. Die für **onLoad** hinterlegte Prozedur wird einmalig beim Einlesen der Ribbondefinition ausgeführt.

Diese nimmt einen Verweis auf die Ribbondefinition entgegen und speichert diese in der Variablen **objRibbon**:

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

Die für das Attribut **loadImage** hinterlegte Funktion **loadImage** erwartet für den mit **imageId** übergebenen Namen ein entsprechendes Objekt des Typs **IPictureDisp**. Dieses holen wir mit der Funktion **LoadResPicture** aus den Ressourcen des twinBASIC-Projekts:

```
Private Function GetCustomUI(ByVal RibbonID As String) As String Implements IRibbonExtensibility.GetCustomUI
    Dim strXML As String
    strXML &= "<customUI xmlns=""http://schemas.microsoft.com/office/2009/07/customui"" loadImage=""LoadImage"" " _
        & "onLoad=""onLoad"">" & vbCrLf
    strXML &= "  <ribbon startFromScratch=""false"">" & vbCrLf
    strXML &= "    <tabs>" & vbCrLf
    strXML &= "      <tab idMso=""TabHomeAccess"">" & vbCrLf
    strXML &= "        <group id=""grpFastSearch"" label=""Schnellsuche"" insertAfterMso=""GroupSortAndFilter"">" _
        & vbCrLf
    strXML &= "          <box id=""box1"">" & vbCrLf
    strXML &= "            <editBox label=""Filtern nach:"" id=""txtFilter"" sizeString=""aaaaaaaaaaaaaaaaa"" " _
        & "onChange=""onChange"" getText=""getText""/>" & vbCrLf
    strXML &= "            <button imageMso=""FilterToggleFilter"" id=""btnFilter"" onAction=""onAction""/>" & vbCrLf
    strXML &= "            <button imageMso=""FilterClearAllFilters"" id=""btnFilterAus"" onAction=""onAction""/>" _
        & vbCrLf
    strXML &= "          </box>" & vbCrLf
    strXML &= "          <checkBox label=""Kompletten Feldinhalt untersuchen"" id=""chkKompletterFeldinhalt"" " _
        & "onAction=""OnAction_Checkbox""/>" & vbCrLf
    strXML &= "          <checkBox label=""Nur aktuelles Feld untersuchen"" id=""chkNurAktuellesFeldUntersuchen"" " _
        & "onAction=""OnAction_Checkbox""/>" & vbCrLf
    strXML &= "        </group>" & vbCrLf
    strXML &= "      </tab>" & vbCrLf
    strXML &= "    </tabs>" & vbCrLf
    strXML &= "  </ribbon>" & vbCrLf
    strXML &= "</customUI>" & vbCrLf
    Return strXML
End Function
```

Listing 1: Zusammenstellen der Ribbondefinition

Wie speichert man Objekte und Module in Access?

Auch uns von Access im Unternehmen fallen immer mal wieder Techniken oder Dinge auf, die uns zuvor noch nicht bekannt waren – und die andere Entwickler schon lange wie selbstverständlich nutzen. Wenn uns so etwas auffällt, berichten wir sofort darüber, denn: Wenn wir diese Technik noch nicht kennen, gibt es sicher noch andere Leser, die sich über einen neuen Trick freuen. In diesem Fall geht es um das Speichern von Objekten in Access. Wir zeigen erst einmal, wie das Speichern in Access überhaupt funktioniert und danach, welche für uns neue Technik wir dabei noch kennengelernt haben.

Einzelne Access-Objekte speichern über die Benutzeroberfläche

Wenn wir in der Access-Benutzeroberfläche Objekte wie Tabellen, Abfragen, Formulare, Berichte oder Makros ändern, können wir die Änderungen an dem aktuell im Fokus befindlichen Objekt wie folgt speichern:

- wir betätigen die Tastenkombination **Strg + S** oder
- wählen aus dem Kontextmenü der Titelleiste des Objekts den Befehl **Speichern** aus (siehe Bild 1).

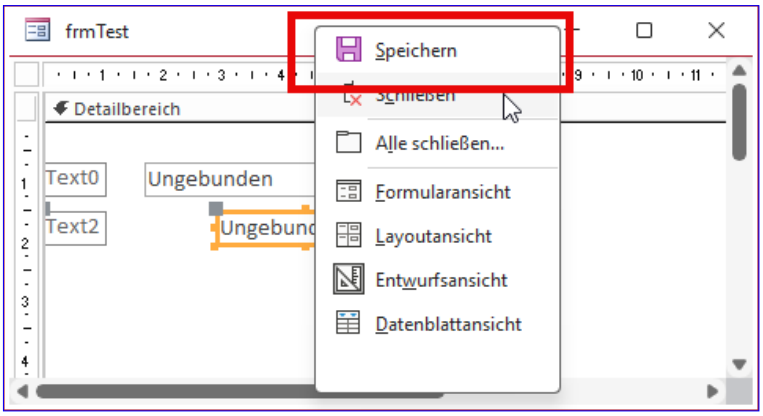


Bild 1: Speichern eines einzelnen Objekts

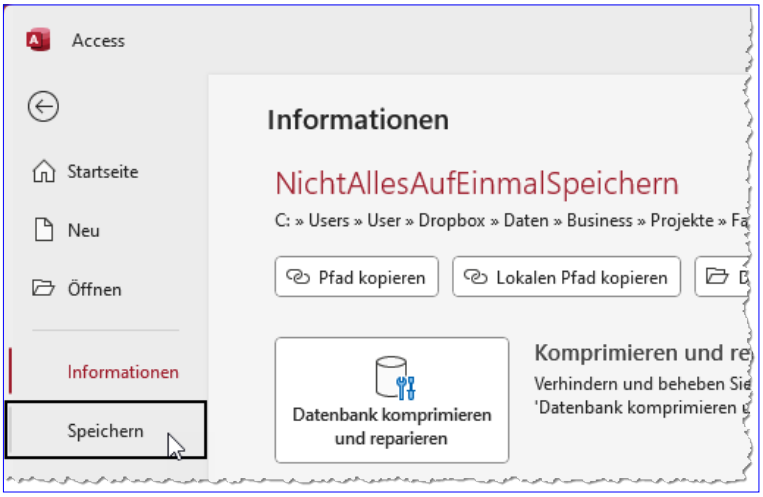


Bild 2: Speichern eines einzelnen Objekts per Ribbon-Befehl

Alle Änderungen speichern über die Benutzeroberfläche

Wenn wir hingegen mehrere Elemente geändert haben und speichern wollen (oder wenn wir grundsätzlich alle Änderungen speichern wollen), nutzen wir dazu den Befehl **Speichern** aus dem Ribbon-Bereich **Datei** (siehe Bild 2).

Dies speichert ohne weitere Rückfrage einfach alle aktuell nicht gespeicherten Access-Objekte und auch die nicht gespeicherten Module aus dem VBA-Editor.

Speichern beim Schließen eines Objekts

Ein einzelnes Objekt können wir auch über die Benutzeroberfläche speichern, indem wir dieses schließen und die dann erscheinende Meldung aus Bild 3 bestätigen.

In diesem Fall wird ebenfalls nur das eine Objekt gespeichert.