HAUFE.
Ausgabe 04/2023

ACCESS IM UNTERNEHMEN

DATENBANKEN VERGLEICHEN

Zwei Exemplare der gleichen Datenbank und Sie wissen nicht, welche die aktuellere ist? Kein Problem: Wir bieten die Lösung für einen schnellen Vergleich der Änderungsdaten der enthaltenen Objekte (ab Seite 66).

In diesem Heft:

UNGEBUNDENE FORMULARE MIT DATEN

Manchmal ist eine direkte Datenbindung nicht sinnvoll. Wir zeigen, wie Tabellendaten ohne Bindung in Formularen landen.

SEITE 29

BELEGE AN LEXOFFICE ÜBERTRAGEN

Sie haben hunderte Belege samt Rechnungsdaten, die nach lexoffice sollen? Erfahren Sie, wie Sie das automatieren können.

SEITE 53

BACKUP MIT DEM SQL SERVER

SQL Server-Datenbanken lassen sich nicht einfach kopieren. Lesen Sie, wie Sie Ihre Daten dennoch als Backup sichern.

SEITE 44

Versionen einfach abgleichen

Der Vergleich zweier Versionen der gleichen Datenbank kann viel Aufwand bedeuten. Vor allem, wenn man es wirklich genau wissen will. In den meisten Fällen ist das jedoch gar nicht nötig – zumindest dann nicht, wenn man nur wissen möchte, welche die aktuellere von zwei Versionen ist. Dann reicht es meist schon, einfach die letzten Änderungsdaten der Datenbankobjekte zu vergleichen – und ob neue Elemente hinzugekommen sind oder vorhandene entfernt wurden. Wir liefern eine Lösung, die das automatisch erledigt.

Dieses praktische Tool stellen wir im Beitrag **Aktuelle Datenbankversion ermitteln** ab Seite 66 vor. Sie brauchen nur die beiden zu vergleichenden Dateien auszuwählen, den Rest erledigt das Tool für Sie in Form einer übersichtlichen Auswertung.

Wen der Navigationsbereich in einer Anwendung stört, möchte diesen vielleicht automatisch per VBA ausblenden. Die notwendige Technik vermitteln wir im Beitrag **Navigationsbereich per VBA ein- und ausblenden** ab Seite 6.

Viele Leser haben gefragt, wie man das Backend einer Datenbank automatisch komprimieren kann – beispielsweise vor dem Öffnen oder nach dem Schließen der Anwendung. Wie das gelingt, zeigen wir im Beitrag **Datenbanken automatisch komprimieren** ab Seite 8.

Einen großen Block haben wir dem Thema »Ungebundene Formulare« gewidmet. Manchmal ist es nicht möglich oder nicht sinnvoll, ein Formular direkt an eine Tabelle oder eine Abfrage zu binden. Dann stellt sich die Frage: Wie kommen die Daten sonst in die Steuerelemente des Formulars und wie können wir neue Datensätze anlegen oder vorhandene Datensätze bearbeiten? Im Beitrag **Daten in ungebundenen Formularen bearbeiten** (ab Seite 29) zeigen wir zunächst, wie die Daten einer Tabelle oder Abfrage ohne Datenbindung im Formular angezeigt und dort bearbeitet werden können. Wir zeigen dort auch, wie Sie einen neuen, leeren Datensatz anlegen und diesen in der zugrunde liegenden Tabelle speichern können.

Damit Sie auch die Übersicht über die dort zu bearbeitenden Datensätze haben, zeigen wir im Beitrag **Ungebundene Daten in Übersicht und Detailansicht** ab Seite 38, wie Sie diese in einem Listenfeld darstellen und den dort gewählten Datensatz im Detailformular anzeigen können.

Das dortige Listenfeld soll allerdings wiederum nicht direkt an eine Datenherkunft gebunden sein, weshalb wir es anderweitig füllen müssen. Dafür gibt es zwei Methoden. Die erste gelingt durch das Zusammenstellen einer sogenannten Wertliste – beschrieben im Beitrag **Ungebundene Listen und Kombis mit Daten füllen** ab Seite 14.

Eine alternative Methode ist der Einsatz einer Callback-Funktion als Datensatzquelle. Diese ist allerdings nicht so leicht definiert, weshalb wir dies detailliert im Beitrag **Ungebundene List- und ComboBox per Callback** ab Seite 23 beschrieben haben.

Wer seine SQL Server-Datenbank in Sicherheit wissen möchte, legt regelmäßig Backups an. Eine praktische Anleitung hierzu finden Sie unter dem Titel **SQL Server: Vollsicherung und Wiederherstellung** ab Seite 44.

Und für alle Leser, die selbst mit lexoffice arbeiten oder Kunden haben, die das tun, zeigen wir im Beitrag **Belege und Belegdaten nach lexoffice hochladen** ab Seite 53, wie man fertige Belege in **lexoffice** verfügbar macht.

Viel Spaß beim Erkunden der neuen Ausgabe!

Ihr André Minhorst

Zertifikate und Kennwörter mit KeePass speichern

In der heutigen Zeit ist Datensicherheit wichtiger denn je. Und da wir in einem anderen Beitrag mit dem Titel »SQL Server: Verschlüsselte Backups erstellen« (www.access-im-unternehmen.de/1450) ein Zertifikat erstellen, das wir irgendwo sicher beherbergen müssen, stellen wir im vorliegenden Beitrag einmal eine Methode vor, wie wir Zertifikate und Kennwörter sicher speichern können. KeePass ist ein Open Source-Password-Safe. Das heißt, dass darin Kennwörter und Zertifikate abgelegt werden können, an die man nur herankommt, wenn man das Kennwort zu diesem Password-Safe kennt. Der Password-Safe liegt auf der lokalen Festplatte. Wie wir KeePass herunterladen, installieren und nutzen, beschreiben wir in diesem Beitrag.

Sind Kennwörter noch nötig?

Die erste Frage ist, ob man überhaupt noch Kennwörter benötigt. Es gibt mittlerweile verschiedene alternative Möglichkeiten, sich zu authentifizieren – beispielsweise beim Onlinebanking über das Bestätigen des Zugriffs über eine App oder bei Webseiten oder Apps wie Facebook et cetera, wo man sich ein Kennwort über eine sogenannte Authenticator-App generieren lässt. Es gibt allerdings noch sehr viele Anwendungen, wo echte Kennwörter benötigt werden – und andere Elemente zur Gewährleistung der Sicherheit wie beispielsweise Zertifikate. Deshalb schauen wir uns in diesem Beitrag die Möglichkeiten zum Speichern von Kennwörtern an.

Eines für alle – alle für eines

Kennwörter sind eine sensible Angelegenheit. Es gibt verschiedene Herangehensweisen, die sich zunächst in der Anzahl der verschiedenen verwendeten Kennwörter unterscheiden:

- Man verwendet für alles das gleiche Kennwort (sehr unsicher – wenn jemand ein Kennwort herausfindet, kann er auf alle anderen kennwortgeschützten Bereiche zugreifen).
- Man verwendet unterschiedliche Kennwörter (sicherer als die erste Variante, da die anderen Bereiche immer noch geschützt sind, wenn jemand ein Kennwort herausfindet).

Von leicht bis schwer

Dann gibt es noch Unterschiede bezüglich der Ermittlung der Kennwörter:

- Das Kennwort wird selbst bestimmt.
- Das Kennwort wird durch einen Zufalls-Algorithmus ermittelt.

Letztere Variante ist tendenziell sicherer als das selbstständige Festlegen von Kennwörtern. Das hängt allerdings davon ab, wie die selbst definierten Kennwörter gestaltet sind. Wenn wir diese auf persönlichen Daten wie Initialen, Geburtsdaten et cetera aufbauen, sind die Kennwörter leichter zu erraten als solche, die keine persönlichen Daten enthalten.

Am besten fährt man allerdings mit komplett automatisch generierten Kennwörtern. Der Nachteil: Man kann sich diese in der Regel nicht merken, da diese aus einer ellenlangen Reihe von Zahlen, Buchstaben und Sonderzeichen bestehen. Und hier kommen Tools zum Speichern von Kennwörtern und anderen Zugangsdaten ins Spiel wie zum Beispiel **KeePass**.

KeePass herunterladen und installieren

Den kostenlosen Download von KeePass finden wir unter folgendem Link:

https://KeePass.info/

Nach dem Herunterladen installieren wir die Datei durch einen Doppelklick auf die **.exe**-Datei. Auch wenn dort die Installationssprache auf Deutsch eingestellt werden kann, landet nicht automatisch auch die deutsche Version auf dem Rechner – stattdessen erhalten wir die englischsprachige Version aus Bild 1. Um die deutsche Version zu installieren, besuchen wir die folgende Webseite und wählen die gewünschte Sprache und Version aus:

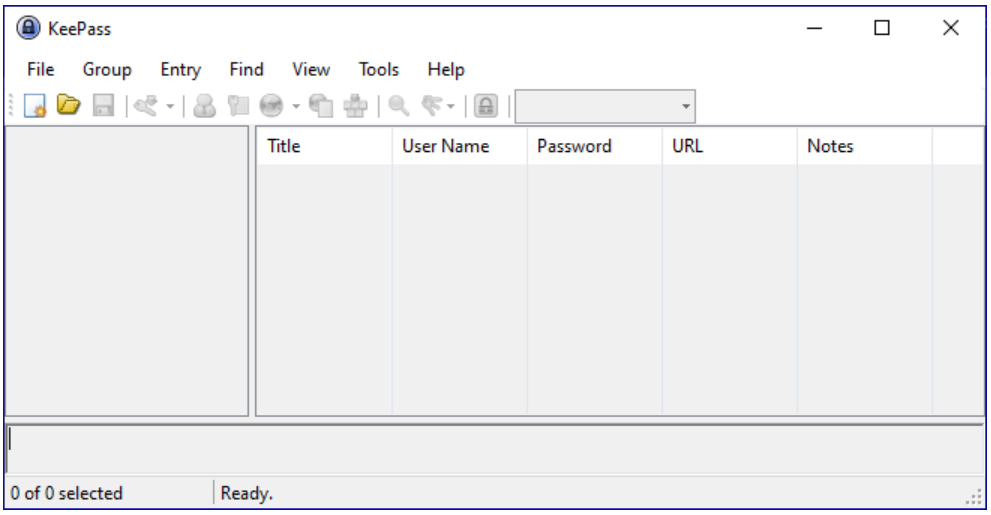


Bild 1: Die englischsprachige Benutzeroberfläche von KeePass

https://KeePass.info/translations.html

Dies speichert eine Datei beispielsweise namens **KeePass-2.54-German.zip** auf unserer Festplatte, welche die Sprachdatei namens **German.lngx** enthält. Diese Datei kopieren wir in das Verzeichnis **Languages** unterhalb des Installationsverzeichnisses von KeePass, zum Beispiel hier:

C:\Program Files\KeePass Password Safe 2\Languages

Danach betätigen wir in KeePass den Befehl **View|Change Language...** (siehe Bild 2), was einen Dialog zur Auswahl der verfügbaren Sprachdateien öffnet. Hier wählen wir den Eintrag **German (Deutsch)** aus.

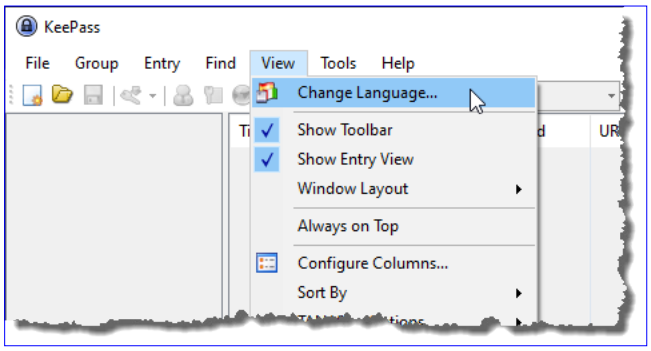


Bild 2: Umstellen der Sprache

Neue Datenbank anlegen

Nun legen wir eine neue Datenbank zum Speichern unserer Kennwörter an. Dazu betätigen wir die Schaltfläche **Neue Datenbank**. Nach einem Hinweis erscheint ein Dialog, der den Speicherort für die Datenbank abfragt.

Speicherort abhängig von Sicherungsstrategie

Dies ist eine wichtige Entscheidung, die auch mit der Strategie zusammenhängt, die für die Erstellung von Backups dieser Datenbank verwendet wird. Eines steht fest: Sie sollten diese Datenbank nicht auf dem lokalen Rechner anlegen und gleichzeitig keine Sicherungen davon an einer externen Stelle anlegen.

In diesem Fall würde ein Crash des Rechners bedeuten, dass Sie nicht mehr an Ihre Kennwörter herankommen.

Also sollten Sie, auch wenn Sie sonst keine Sicherung des Rechners anlegen, zumindest für regelmäßige Sicherungen dieser Datei sorgen. Dafür gibt es wieder verschiedene Möglichkeiten, zum Beispiel:

Navigationsbereich per VBA ein- und ausblenden

Im Normalbetrieb kann man den Navigationsbereich schnell durch einen Klick auf die Schaltfläche zum Öffnen/Schließen der Verkleinerungsleiste oben in diesem Bereich minimieren und anschließend wieder maximieren. Das gelingt auch mit der Taste F11. Was aber, wenn man den Bereich komplett verschwinden lassen möchte – beispielsweise direkt nach dem Öffnen einer Anwendung, die man einem Kunden oder anderen Nutzern bereitstellt, die den Navigationsbereich nicht sehen sollen? Dann kommt die in diesem Beitrag vorgestellte VBA-Prozedur zum Einsatz, mit der wir den Navigationsbereich mit wenigen Zeilen aus- und wieder einblenden.

Drei mögliche Zustände des Navigationsbereichs sehen wir in Bild 1. Links sehen wir den Normalzustand, in dem der Bereich sichtbar ist und die enthaltenen Elemente anzeigt. In der Mitte haben wir den Bereich durch einen Klick auf die Schaltfläche zum Öffnen/Schließen der Verkleinerungsleiste minimiert. Das gelingt auch durch Betätigen der **F11**-Taste. Rechts ist der gewünschte Zustand, den wir per VBA herbeiführen wollen. Um den Navigationsbereich komplett auszublenden, verwenden wir die folgende Prozedur:

```
Public Sub HideNavigationPane()
    On Error Resume Next
    DoCmd.SelectObject acTable, , True
    DoCmd.SelectObject acQuery, , True
    DoCmd.SelectObject acForm, , True
    DoCmd.SelectObject acReport, , True
    DoCmd.SelectObject acMacro, , True
    DoCmd.SelectObject acModule, , True
    DoCmd.RunCommand acCmdWindowHide
End Sub
```

Hier verwenden wir mehrfach die **DoCmd.SelectObject**-Methode, um ein Objekt des angegebenen Typs im Navigationsbereich auszuwählen. Da es auch sein kann, dass diese Prozedur aufgerufen wird, während die Datenbank noch gar kein Element enthält oder das im zweiten Parameter angegebene Element nicht vorhanden ist, geben wir gar nicht erst einen Objektnamen für den zweiten Parameter an, sondern versuchen, nacheinander irgendein

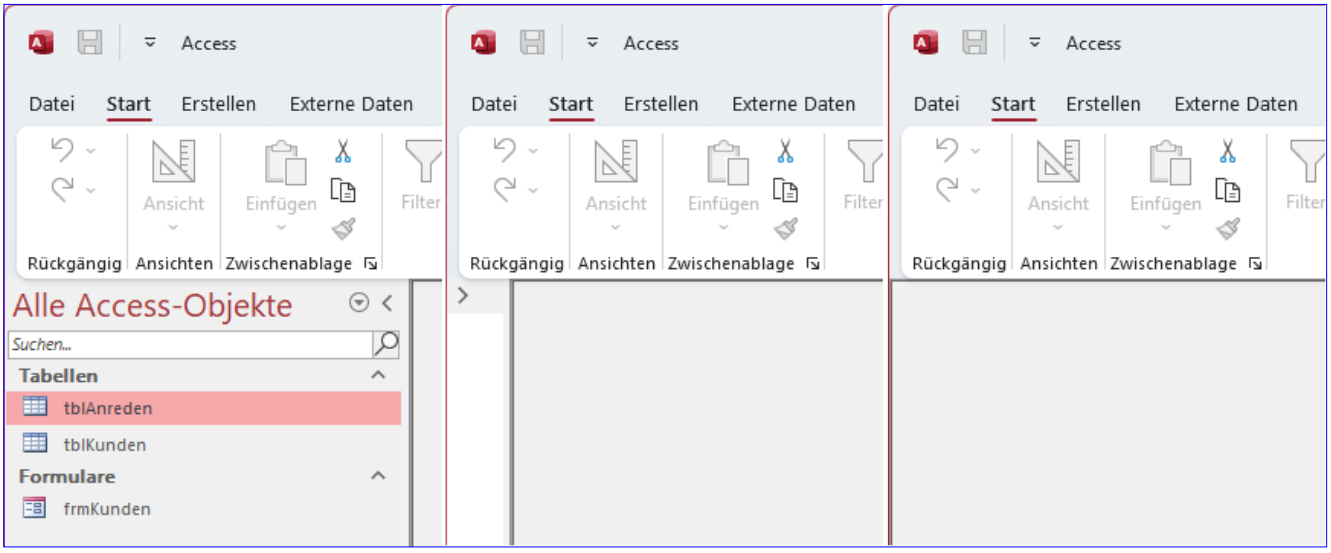


Bild 1: Verschiedene Status des Navigationsbereichs: Eingeblendet, minimiert und ausgeblendet

Datenbanken automatisch komprimieren

In vielen Datenbankanwendungen fallen temporäre Daten an, also Daten, die in Tabellen geschrieben und in der gleichen Session wieder gelöscht werden. Das kann sowohl in nicht aufgeteilten Datenbanken als auch in aufgeteilten Datenbanken mit Frontend und Backend geschehen. Man könnte denken, die Größe der Datenbankdatei würde nach dem Anfügen und Löschen von Daten einigermaßen konstant sein. Das ist jedoch nicht der Fall: Gelöschte Daten sind zwar nicht mehr in den Tabellen zu finden, allerdings benötigt die Datenbank anschließend ungefähr genauso viel Speicherplatz wie vor dem Löschen. Wie das zu erklären ist und wie wir durch die Komprimierung einer Datenbank dennoch dafür sorgen können, dass eine Datenbank sich durch die Verwendung temporärer Daten nicht übermäßig aufbläht, beschreiben wir in diesem Beitrag.

Beispieldatenbank erzeugen

Wenn wir das Verhalten von Access-Datenbanken beim Anlegen und Löschen von Daten und beim Komprimieren der Datenbank untersuchen wollen, brauchen wir die Möglichkeit, schnell Daten hinzuzufügen und zu entfernen – und natürlich eine Tabelle, in der wir die Daten speichern.

Die Tabelle soll **tblTexte** heißen und neben dem Primärschlüsselfeld **ID** ein Textfeld namens **Beispieltext** mit einer Feldgröße von 255 Zeichen enthalten (siehe Bild 1).

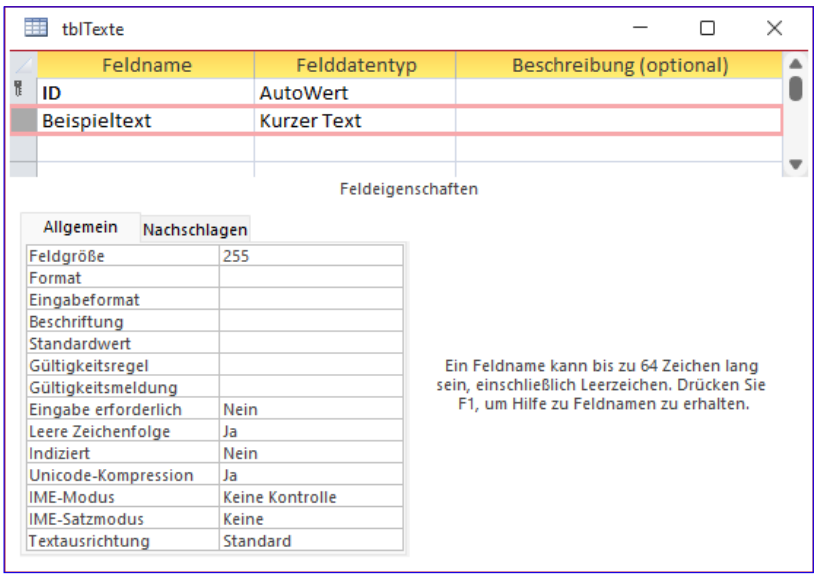


Bild 1: Tabelle für Beispieldaten

Zum Anlegen einer relevanten Menge von Daten verwenden wir die folgende Prozedur:

```
Public Sub TabelleFuellen()
    Dim db As DAO.Database
    Dim i As Long
    Set db = CurrentDb
    For i = 1 To 10000
        db.Execute "INSERT INTO tblTexte(Beispieltext) ⤷
            VALUES('1234567890...12345')", dbFailOnError
    Next i
End Sub
```

Diese legt 10.000 Datensätze mit jeweils 255 Zeichen an. Um die Datensätze schnell wieder zu löschen, verwenden wir die folgende Prozedur:

```
Public Sub TabelleLeeren()
    Dim db As DAO.Database
    Set db = CurrentDb
    db.Execute "DELETE FROM tblTexte", ⤷
                                    dbFailOnError
End Sub
```

Führen wir die erste Prozedur für eine bis dahin leere Datenbank aus, erhalten wir eine Datenbankgröße von ungefähr 3.264 KB.

Untersuchen der Datenbankgröße nach dem Löschen der Daten

Anschließend führen wir die zweite Prozedur aus und löschen damit alle Datensätze der Tabelle **tblTexte**. Schließen wir danach die Datenbank, bleibt die Dateigröße erhalten – obwohl die Tabelle keine Daten mehr enthält.

Platz für gelöschte Daten wird erst beim Komprimieren freigegeben

Der Grund für dieses Verhalten ist, dass der Speicherplatz erst nach dem Komprimieren einer Datenbank freigegeben wird. Also öffnen wir die Datenbank mit der leeren Tabelle erneut und rufen für diese den Befehl **Datenbank komprimieren und reparieren** auf (unter Access 365 beispielsweise unter **DateiInformationen** zu finden).

Betrachten wir anschließend erneut die Dateigröße, weist diese wieder um die 400 KB aus. Daraus können wir ableiten: Der Speicherplatz für gelöschte Datensätze wird erst nach dem Komprimieren der Datenbank freigegeben.

Automatische Komprimierung aktivieren

Für monolithische Access-Anwendungen, also solche, die nur aus einer einzigen Datenbankdatei und nicht aus Frontend und Backend bestehen, können wir in den Access-Optionen die Option **Beim Schließen komprimieren** aktivieren (siehe Bild 2).

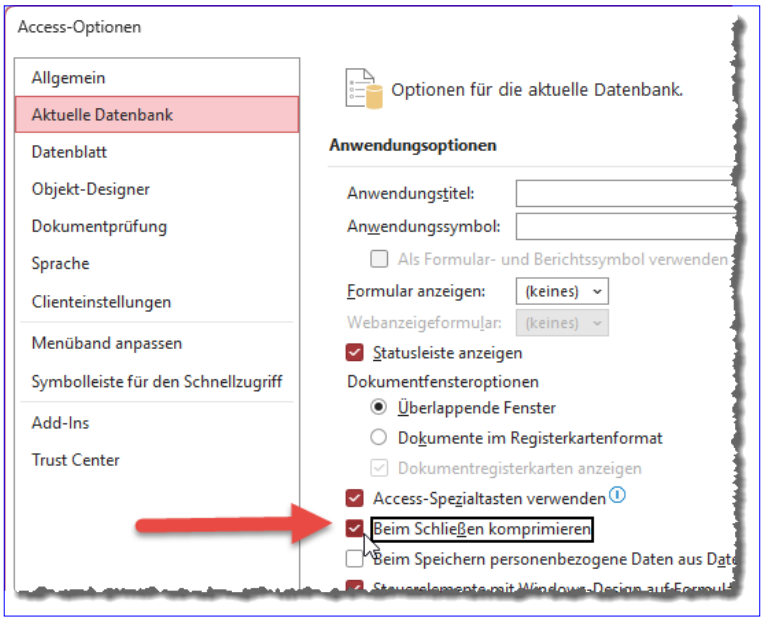


Bild 2: Aktivieren der automatischen Komprimierung

Füllen wir die Tabelle wieder mit Daten und löschen diese erneut, finden wir gleich nach dem nächsten Schließen wieder die geringe Dateigröße von rund 400 KB vor.

Komprimierung bei aufgeteilten Datenbanken

Nun teilen wir die Beispieldatenbank in Frontend und Backend auf, was am schnellsten mit dem Menübefehl **DatenbanktoolslDaten verschiebenlAccess-Datenbank** gelingt. Hier brauchen wir nur den Pfad zur neuen Backenddatenbank anzugeben. Das Ergebnis ist eine neue Backenddatenbank, in welche die einzige Tabelle der bisher verwendeten Datenbank verschoben wurde, sowie eine Verknüpfung zu dieser Tabelle in der aktuellen Datenbankdatei.

Erstere nennen wir nun **BackendKomprimieren_FE.accdb**, letztere **BackendKomprimieren_BE.accdb**.

Anlegen und Löschen in Frontend und Backend

Die Frontenddatenbank mit der Tabellenverknüpfung und das Backend mit der leeren Tabelle haben wir nun zunächst komprimiert. Die Größen lauteten danach:

- Frontend: 432 KB
- Backend: 348 KB

Anschließend rufen wir die Prozedur zum Anlegen der 10.000 Datensätze in der Tabelle **tblTexte** auf. Dabei fällt zuerst auf, dass der Vorgang viel länger dauert als in der monolithischen **.accdb**-Datei. Noch viel interessanter ist

die Beobachtung des benötigten Speicherplatzes für die beiden Dateien:

- Frontend: 40.412 KB
- Backend: 3.208 KB

Der Teil mit der Tabelle ist also ähnlich gewachsen wie zuvor, aber das Frontend ist massiv größer geworden – es wächst um mehr als den zehnfachen Speicherplatz der eigentlich erzeugten Daten. Obwohl es noch nicht einmal eine von uns angelegte Tabelle mit Daten enthält. Die Größe ändert sich auch nach dem Schließen nicht.

Anschließend haben wir die Prozedur zum Schreiben der Daten erneut aufgerufen, um zu prüfen, ob wir das Frontend so bis zur Grenze von 2 GB aufblähen können. Das ist jedoch nicht der Fall: Beim zweiten Durchlauf wird die Frontenddatenbank nur noch unwesentlich vergrößert, während das Backend seine Größe wie erwartet um ca. 2.800 KB ändert.

Wachstum des Frontends

Zum Wachstum der Größe des Frontends noch folgende Anmerkung: Dies geschieht nur, wenn wir die Daten per VBA in die Tabelle **tblTexte** des Backends einfügen. Wir haben in einem zweiten Versuch manuell mehrere tausend Datensätze dort eingefügt und in diesem Fall wurde zwar das Backend vergrößert, nicht jedoch das Frontend.

Komprimieren des Frontends

Komprimieren wir nun das Frontend, wird dieses wieder auf die ursprüngliche Größe geschrumpft. An der Größe des Backends ändert sich zunächst nichts, was auch nicht zu erwarten war, da wir die enthaltenen Daten nicht gelöscht haben. Also öffnen wir das Frontend erneut und löschen die Daten im Backend mit der Prozedur **Tabelle-Leeren**.

Die Größe des Backends bleibt erhalten. Nun rufen wir in der Frontenddatenbank den Befehl **Komprimieren und Reparieren** auf. Obwohl wir die Daten im Backend gelöscht haben, ändert sich dessen Speicherbedarf nicht.

Wenn wir das Frontend einer Kombination aus Frontend und Backend komprimieren, wird also nur der Speicherplatz im Frontend freigegeben, der für das Schreiben von Datensätzen im Backend angefallene Daten verwendet wurde.

Backend komprimieren

Wie also können wir nun dafür sorgen, dass der Speicherplatz für im laufenden Betrieb angelegte und wieder gelöschte Daten im Backend wieder freigegeben wird? Immerhin darf das Backend nicht größer als 2 GB werden, was bei solchen Operationen schnell geschehen kann.

Öffnen wir das Backend ohne das Frontend und komprimieren es nach dem Löschen der Daten mit dem dafür vorgesehen Befehl **Komprimieren und Reparieren**, erhält es wieder seine ursprüngliche Dateigröße.

Backend beim Schließen komprimieren

Die Vermutung liegt nahe, dass wir nur für das Backend die Option **Beim Schließen komprimieren** einstellen müssen, damit dieses beim Schließen des Frontends automatisch komprimiert wird.

Dies führt allerdings nicht zum Erfolg. Nach dem Einstellen dieser Option und dem Schreiben und Löschen der Daten über das Frontend erhält das Backend nicht wieder die Größe der Datenbank mit der geleerten Tabelle.

Woran liegt das? Offensichtlich findet beim Zugriff auf das Backend über die Tabellenverknüpfungen kein Öffnen und Schließen der Backenddatenbank in dem Sinne statt, dass dabei die für das Schließen aktivierte Komprimierung ausgelöst wird.

Wir müssen also einen anderen Weg finden, um das Backend zu komprimieren.

Ungebundene Listen und Kombis mit Daten füllen

Im Beitrag »Daten in ungebundenen Formularen bearbeiten« zeigen wir, wie man die Daten einer Tabelle in einem ungebundenen Formular darstellt und dieses zum Bearbeiten und Anlegen von Datensätzen verwendet. Wenn wir dies konsequent umsetzen wollen, benötigen wir auch eine Möglichkeit, um Daten in Listenform anzuzeigen. Die Datenblatt- oder Endlosansicht fallen aus, also müssen wir uns um Alternativen kümmern. Für die mehrspaltige Listenansicht bietet Access das Listenfeld. Und für die Auswahl von Daten aus Lookup-Tabellen steht das Kombinationsfeld zur Verfügung. Beide sind jedoch, wie auch Formulare, für die Bindung an Tabellen oder Abfragen optimiert. Ungebundene Daten müssen wir dort erst einmal einpflegen. Auf welche Arten das gelingt, zeigt dieser Beitrag.

Listen- oder Kombinationsfeld

Die folgenden Informationen beziehen sich jeweils auf Listen- und Kombinationsfelder, wenn nicht anders angegeben.

Techniken zum Füllen von Kombinations- und Listenfeldern ohne Datenbindung

Wenn wir Kombinations- oder Listenfelder an Tabellen oder Abfragen binden, brauchen wir nur noch festzulegen, aus welchen Feldern der Datensatzherkunft die anzuzeigenden Daten stammen und welches Feld als gebundene Spalte genutzt werden soll. Der Rest funktioniert fast automatisch.

Wenn wir jedoch nicht auf die Datenbindung setzen können, brauchen wir alternative Techniken. Davon gibt es gleich drei, von denen zwei offensichtlich sind und eine nicht. Und von den beiden offensichtlichen können wir auch nur eine für unsere Zwecke nutzen. Bei den offensichtlichen Möglichkeiten reden wir von denen, die neben dem Eintrag **Tabelle/Abfrage** für die Eigenschaft **Herkunftstyp** zur Verfügung stehen (siehe Bild 1).

Dabei handelt es sich um die Folgenden:

- **Wertliste**: Ist dieser Eintrag aktiviert, können wir für die Eigenschaft **Datensatzherkunft** eine durch Semikola separierte Liste von Einträgen angeben. Standardmäßig

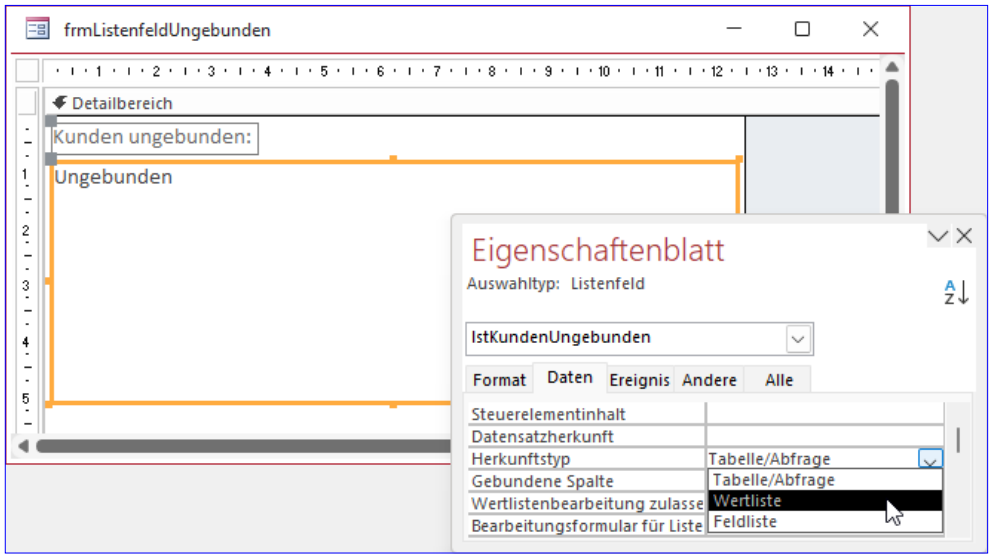


Bild 1: Einstellungen für die Eigenschaft **Herkunftstyp**

landet jeder Eintrag in einer eigenen Zeile. Wenn wir die Eigenschaft **Spaltenanzahl** auf einen anderen Wert als **1** einstellen, werden entsprechend viele Einträge in einer Zeile dargestellt – mehr dazu weiter unten in einem Beispiel.

- **Feldliste**: Wenn wir den Wert **Feldliste** auswählen, müssen wir für die Eigenschaft **Datensatzherkunft** den Namen einer Tabelle oder Abfrage hinterlegen. Das Listenfeld zeigt dann die Feldnamen der angegebenen Tabelle oder Abfrage an. Diese Variante ist für uns in diesem Fall uninteressant.

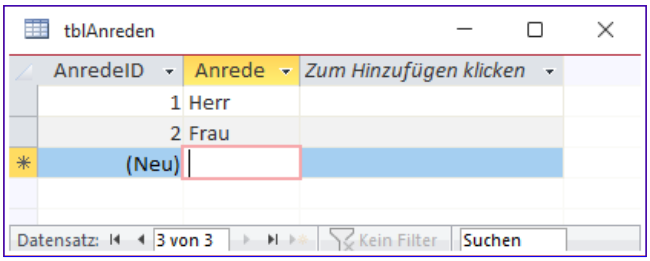


Bild 2: Beispieltabelle zum Füllen eines Kombinationsfeldes

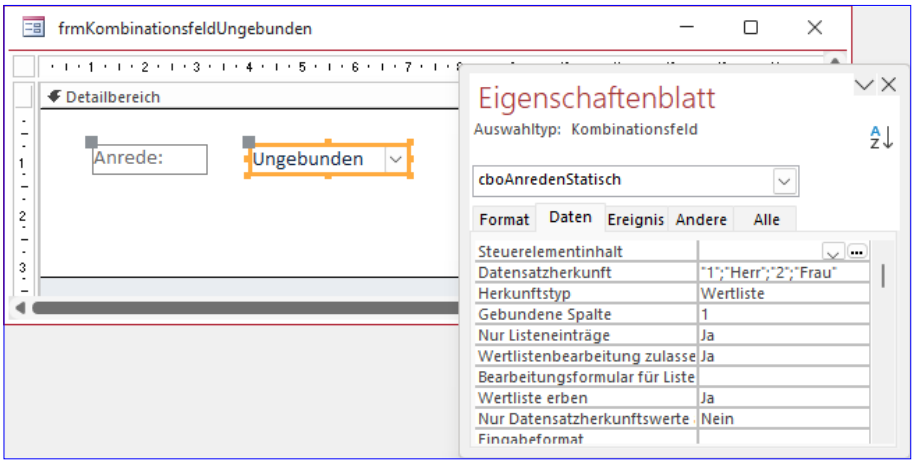


Bild 3: Einstellungen für ein statisch gefülltes Kombinationsfeld

Weitere Variante: Callback-Funktion

Und es gibt noch eine Möglichkeit, die wir später betrachten werden: Dabei handelt es sich um die sogenannte Callback-Funktion. Dies ist eine Funktion, mit der wir definieren, welche Daten im Kombinations- oder Listenfeld enthalten sein sollen.

Der wesentliche Unterschied ist, dass wir mit dieser Methode mehr Einträge zu einem Kombinations- oder Listenfeld hinzufügen können als mit einer Wertliste. Diese Technik schauen wir uns in einem weiteren Beitrag namens **Ungebundene List- und ComboBox per Callback** (**www.access-im-unternehmen.de/1443**).

Kombinationsfeld per Wertliste füllen

Im ersten Beispiel schauen wir uns an, wie wir ein Kombinationsfeld über eine Wertliste mit den Daten einer Tabelle füllen. Wir wollen dabei extra eine Tabelle mit wenigen Datensätzen nutzen wie die Tabelle **tblAnreden** (siehe Bild 2).

Wenn wir nur so wenige Datensätze haben und es sich auch noch um solche Datensätze handelt, die sich eigentlich nie ändern, könnte man versucht sein, diese einfach statisch für die Eigenschaft **Datensatzherkunft** einzustellen. Wenn man gleichzeitig die Eigenschaft **Herkunftstyp** auf **Wertliste** einstellt und die beiden Eigenschaften **Spaltenanzahl** auf **2** und **Spaltenbreiten** auf **0cm**, werden die Daten genau angezeigt wie in einem an die Tabelle **tblAnreden** gebundenen Kombinationsfeld. Die Eigenschaft **Datensatzherkunft** füllen wir wie folgt (siehe Bild 3):

```
"1";"Herr";"2";"Frau"
```

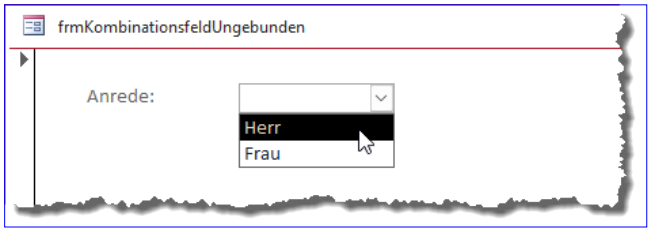


Bild 4: Ungebundenes Kombinationsfeld mit Daten

Durch die Einstellung von **Spaltenanzahl** auf **2** werden jeweils zwei Elemente in einer Zeile angezeigt. Durch die Angabe des Wertes **0cm** für die Eigenschaft **Spaltenbreiten** wird die erste Spalte dabei mit der Breite **0cm** angezeigt und die zweite nimmt den Rest der Breite ein. Das Ergebnis sehen wir in Bild 4 – es sieht aus wie ein übliches gebundenes Kombinationsfeld.

Kombinationsfeld aus Recordset füllen

Wir wollen das Kombinationsfeld jedoch nicht statisch füllen, sondern auch für den Fall vorbereitet sein, dass die zugrunde liegende Tabelle nicht immer die gleichen Datensätze enthält.

Also fügen wir dem Formular ein weiteres Kombinationsfeld hinzu, dem wir diesmal den Namen **cboAnredeRecordset** geben.

Dabei behalten wir die Einstellung **Wertliste** für die Eigenschaft **Herkunftsart** bei, und auch die beiden Eigenschaften **Spaltenanzahl** und **Spaltenbreiten** belassen wir bei den Werten **2** und **0cm**. Die Eigenschaft **Datensatzherkunft** lassen wir jedoch zunächst leer. Um diese zu füllen, legen wir eine VBA-Prozedur an, die wie in Listing 1aussieht.

Sie wird wie folgt durch das Ereignis **Bei Laden** des Formulars ausgelöst:

```
Private Sub Form_Load()
    cboAnredeRecordsetFuellen
End Sub
```

Diese Prozedur erstellt ein Recordset auf Basis der Tabelle **tblAnreden** und durchläuft in einer **Do While**-Schleife alle Datensätze dieser Tabelle. Dabei fügt sie bei jedem Durchlauf der Schleife die Daten eines Datensatzes zu einer Variablen namens **strWertliste** hinzu. Nach dem Hinzufügen der Daten für den ersten Datensatz sieht der Inhalt von **strWertliste** beispielsweise wie folgt aus:

```
1;'Herr';
```

Nach dem Durchlaufen aller beiden Datensätze erhalten wir fast den gleichen Inhalt wie beim vorherigen Beispiel – mit Ausnahme des abschließenden Semikolons:

```
1;'Herr';2;'Frau';
```

Dieses wirkt sich jedoch nicht auf die Darstellung aus, weshalb wir es nicht nachträglich entfernen.

```
Private Sub cboAnredeRecordsetFuellen()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim strWertliste As String
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT * FROM tblAnreden", dbOpenDynaset)
    Do While Not rst.EOF
        strWertliste = strWertliste & rst!AnredeID & ";'" & rst!Anrede & "';"
        rst.MoveNext
    Loop
    Me!cboAnredenRecordset.RowSource = strWertliste
    rst.Close
    Set rst = Nothing
    Set db = Nothing
End Sub
```

Listing 1: Füllen der Wertliste beim Laden des Formulars

```
Private Sub cboAnredeRecordsetAddItemFuellen()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim strWertliste As String
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT * FROM tblAnreden", dbOpenDynaset)
    Do While Not rst.EOF
        Me!cboAnredenRecordsetAddItem.AddItem rst!AnredeID & ";'" & rst!Anrede & "'"
        rst.MoveNext
    Loop
    rst.Close
    Set rst = Nothing
    Set db = Nothing
End Sub
```

Listing 2: Füllen der Wertliste beim Laden des Formulars per **AddItem**

Auch mit dieser Vorgehensweise sehen wir nach dem Öffnen des Formulars die beiden Einträge **Herr** und **Frau** im Kombinationsfeld **cboAnredenRecordset**.

Alternative Variante für das Hinzufügen per Recordset

Eine weitere Möglichkeit erlaubt das direkte Hinzufügen von Elementen zum Kombinationsfeld. Dabei verwenden wir die **AddItem**-Methode des Kombinationsfeldes. In einer weiteren Prozedur, die wir ebenfalls von der Ereignisprozedur **Form_Load** aus aufrufen, durchlaufen wir dabei ebenfalls eine **Do...While**-Schleife über alle Datensätze der Tabelle **tblAnreden** (siehe Listing 2). Diesmal stellen wir allerdings keine Liste in der Variablen **strWertliste** zusammen, sondern weisen jedes Element direkt mit der **AddItem**-Methode hinzu.

Wenn wir wie zuvor eine zweispaltige Darstellung erhalten wollen, bei der die erste Spalte mit der Breite **0cm** ausgeblendet wird, müssen wir die Inhalte der beiden Felder **AnredeID** und **Anrede** jeweils mit einem einzigen Aufruf der **AddItem**-Methode hinzufügen, wobei wir beide durch ein Semikolon voneinander trennen und die Zeichenkette in Hochkommata einfassen.

Auch diese Methode liefert das gleiche Ergebnis wie die beiden zuvor beschriebenen. Wenn wir uns den Wert der Eigenschaft **Datensatzherkunft** per VBA im Direktbereich ausgeben lassen, erhalten wir übrigens das folgende Ergebnis:

```
? Screen.ActiveControl.RowSource
1;'Herr';2;'Frau'
```

Die **AddItem**-Methode macht also nichts anderes als die Wertliste zu einer langen Zeichenkette zusammenzusetzen. Sie ermöglicht lediglich noch, über den zweiten Parameter den Index für die Position anzugeben, an welcher der neue Datensatz angelegt werden soll.

Hinweis zur Angabe der Spaltenzahl

Es gibt einen wichtigen Unterschied bei der Zuweisung der anzuzeigenden Daten für Listen, die entweder direkt durch die Zuweisung einer durch Semikola separierten Wertliste oder durch die Verwendung der **AddItem**-Methode gefüllt werden. Das lässt sich an einem Beispiel am besten verdeutlichen:

```
lst.RowSource = "1;'Wert 1';2;'Wert 2"
```

Wenn wir hier die Eigenschaft **Spaltenanzahl** auf den Wert **1** einstellen, interpretiert Access jedes durch ein Semikolon getrenntes Element als Inhalt für eine Spalte. Das sieht dann so aus:

Ungebundene List- und ComboBox per Callback

Im Beitrag »Ungebundene Listen und Kombis mit Daten füllen« (www.access-im-unternehmen.de/1440) haben wir uns angesehen, wir man Kombinations- und Listenfelder über das Zuweisen einer Wertliste oder mit der AddItem-Methode füllen kann. Es gibt noch eine interessante Alternative, von der wir hoffen, dass wir damit sogar noch einige Einträge mehr zu den Listen-Steuerelementen hinzufügen können als mit den oben genannten Methoden. Bei dieser Alternative handelt es sich um eine Technik, die eher stiefmütterlich behandelt wird: Die Callbackfunktion. Wie man diese nutzt und ob man tatsächlich mehr Daten als mit einer Wertliste in ungebundenen Listen-Steuerelementen anzeigen kann, untersuchen wir in diesem Beitrag.

Einfaches Beispiel für das Füllen per Callbackfunktion

Bevor wir uns daran begeben, die Daten umfangreicher Tabellen in ein Listenfeld zu füllen, schauen wir uns die grundlegende Vorgehensweise beim Verwenden von Callback-Funktionen zum Füllen von Kombinations- und Listenfeldern an. Bei Verwendung einer Callbackfunktion nutzen wir keine der drei bekannten Möglichkeiten für die Eigenschaft **Herkunftsart**, sondern wir geben dort den Namen einer Funktion an.

Diese legen wir außerdem im Klassenmodul des Formulars an, in dem sich auch das Kombinations- oder Listenfeld befindet. Die Callbackfunktion muss einen speziellen Aufbau haben. Zumindest die Parameter müssen nach einem festen Schema angegeben werden und auch innerhalb der Funktion sind einige Regeln zu beachten, damit diese wie gewünscht arbeitet. Für ein einfaches Beispiel fügen wir einem Formular ein Listenfeld hinzu, für dessen Eigenschaft **Herkunftstyp** wir den Namen der Callbackfunktion angeben, in diesem Fall **Beispielcallback** (siehe Bild 1).

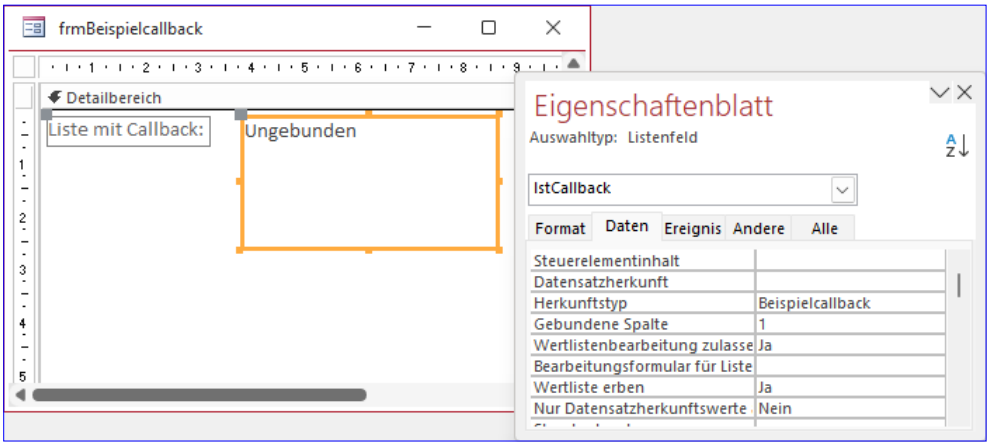


Bild 1: Listenfeld mit Callback-Funktion

Aufbau und Parameter der Callbackfunktion

Die Callbackfunktion erwartet bestimmte Parameter, die beim Aufruf automatisch von dem Steuerelement, dem sie zugeordnet ist, übergeben werden. Der Kopf der Prozedur sieht so aus:

```
Function Beispielcallback(ctl As Control, lngID As Long, _
        lngRow As Long, lngColumn As Long, _
        intCode As Integer) As Variant
```

Die hier verwendeten Parameter liefern die folgenden Werte:

- **ctl**: Verweis auf das Steuerelement, für welches die Callbackfunktion aufgerufen wurde. Dies ist sinnvoll,

falls es mehr als ein Steuerelement gibt, das die gleiche Callbackfunktion aufruft – was aber eher selten der Fall sein dürfte.

- **lngID**: Liefert eine eindeutige ID für das aufrufende Steuerelement.
- **lngRow**: Gibt an, für welche Zeile ein Wert abgefragt wird.
- **lngColumn**: Gibt an, für welche Spalte ein Wert abgefragt wird.
- **intCode**: Gibt die aktuelle Funktion des Aufrufs an.

Der Rückgabewert der Funktion mit dem Datentyp **Variant** transportiert die jeweils angeforderten Informationen an das Steuerelement.

Aufbau der Callbackfunktion

Die Callbackfunktion muss, damit sie korrekt funktioniert und das Steuerelement wie gewünscht gefüllt wird, die mit dem Parameter **intCode** angefragten Informationen zusammenstellen und zurückliefern. intCode kann verschiedene Werte annehmen, für die jeweils Konstanten definiert sind. Diese lauten:

- **acLBInitialize** (**0**): Dieser Wert für **intCode** wird beim ersten Aufruf der Callback-Funktion übergeben. Hier wird geprüft, ob das Kombinations- oder Listenfeld überhaupt mit dieser Funktion gefüllt werden soll. Falls ja, muss der Wert **True** als Rückgabewert der Funktion festgelegt werden.
- **acLBOpen** (**1**): Beim Aufruf der Funktion mit dem Wert **acLBOpen** für den Parameter **intCode** erwartet Access eine Zahl als Rückgabewert, die das Steuerelement, von dem aus die Callback-Funktion aufgerufen wurde, eindeutig identifiziert. Hintergrund ist, dass man theoretisch mehrere Steuerelemente mit der gleichen Callbackfunktion ausstatten kann. Wenn sichergestellt ist, dass nur ein Steuerelement diese Funktion als Wert für die Eigenschaft **Herkunftsart** verwendet, können Sie hier einfach den Wert **1** als Funktionswert angeben. Anderenfalls geben Sie das Ergebnis der Funktion **Timer** zurück, welche die Zeit in Millisekunden berechnet – dies geht nur schief, wenn die Funktion zweimal in der gleichen Millisekunde aufgerufen wird, was unwahrscheinlich ist.
- **acLBGetRowCount** (**3**): Dieser Aufruf der Funktion soll die Anzahl der zu füllenden Zeilen ermitteln. Dies ist der geeignete Ort, um die anzuzeigenden Daten zusammenzustellen.
- **acLBGetColumnCount** (**4**): Dieser Funktionsaufruf erwartet die Übergabe der Anzahl der zu füllenden Spalten.
- **acLBGetColumnWidth** (**5**): Dieser Funktionsaufruf erwartet die Übergabe der Breiten der zu füllenden Spalten.
- **acLBGetValue** (**6**): Die Callback-Funktion wird für jedes anzuzeigende Element einmal mit dem Wert **acLBGetValue** für den Parameter **intCode** aufgerufen. Die Anzahl der Elemente ergibt sich aus dem Produkt aus Zeilen- und Spaltenzahl. Welche Zeile und Spalte gerade abgefragt wird, können Sie den weiteren Parametern **lngRow** und **lngColumn** entnehmen.
- **acLBGetFormat** (**7**): Auch der Aufruf mit diesem Parameterwert erfolgt für jedes anzuzeigende Element einmal. Hier können Sie die Formatierung der Listeneinträge vornehmen (standardmäßig **-1**).
- **acLBClose** (**8**), **acLBEnd** (**9**): Diese beiden Parameter werden in weiteren Aufrufen der Callback-Funktion nach dem Füllen des Steuerelements übergeben. Damit können Sie beispielsweise den Zeitpunkt erfassen, zu dem das Steuerelement komplett gefüllt ist.

Beispiel für eine Callbackfunktion

Nachfolgend haben wir das denkbar einfachste Beispiel für eine Callbackfunktion zusammengestellt:

```
Function Beispielcallback(ctl As Control, _
        lngID As Long, lngRow As Long, _
        lngColumn As Long, intCode As Integer) _
        As Variant
    Select Case intCode
        Case acLBInitialize
            Beispielcallback = True
        Case acLBOpen
            Beispielcallback = 1
        Case acLBGetRowCount
            Beispielcallback = 1
        Case acLBGetColumnCount
            Beispielcallback = 1
        Case acLBGetColumnWidth
            Beispielcallback = -1
        Case acLBGetValue
            Beispielcallback = "Test"
        Case acLBGetFormat
            Beispielcallback = -1
        Case Else
            Debug.Print intCode
    End Select
End Function
```

Dabei werden in diesem Fall alle Codes für den Parameter **intCode** wie oben angegeben nacheinander durchlaufen. Wichtig dabei ist: Der Aufruf von **acLBGetColumnCount** erfolgt einmal. Je nachdem, wie viele Spalten dort zurückgegeben werden, erfolgt die entsprechende Anzahl von Aufrufen mit dem Parameter **acLBGetColumnWidth**. Dabei wird mit dem Parameter **lngColumn** der 0-basierte Index der Spalte übergeben, für welche die Breite abgefragt werden soll. Hier liefern wir immer den Wert **-1** zurück, wenn wir keine explizite Spaltenbreite definieren wollen. Danach wird die Anzahl der Zeilen abgefragt (**acLBGetRowCount**).

Schließlich erfolgen abwechselnde Aufrufe mit den Werten **acLBGetValue** und **acLBGetFormat** für den Parameter **intCode** mit einer Anzahl Wiederholungen, die dem Produkt aus Spalten und Zeilen entspricht.

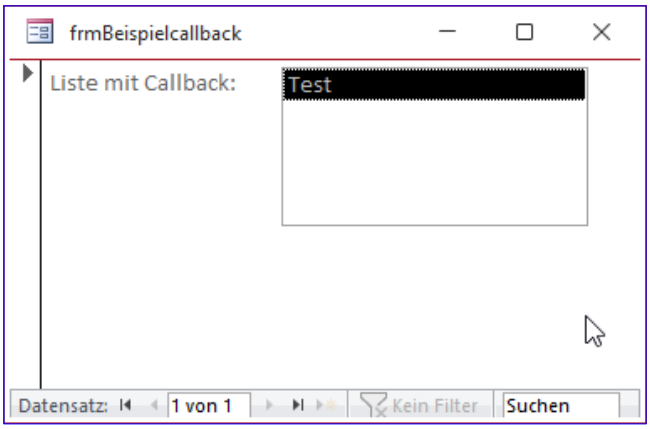


Bild 2: Listenfeld mit einem Beispielwert

In dem Beispiel oben wollen wir nur eine Zeile und eine Spalte füllen, also insgesamt nur einen Wert im Listenfeld anlegen. Das führt zu dem Ergebnis aus Bild 2.

Daten aus einer Tabelle per Callbackfunktion laden

Damit wenden wir uns nun einem echten Anwendungsfall zu – dem Füllen eines Listenfeldes mit acht Spalten und vielen hundert Zeilen. Dabei wollen wir ein Listenfeld namens **lstKunden** mit den Daten der Tabelle **tblKunden** füllen, die mehr als 3.000 Datensätze aufweist. Dies war für das Füllen eines Listenfeldes per Wertliste oder mit der **AddItem**-Methode deutlich zu viel. Wir werden sehen, ob wir mit der Callbackfunktion mehr Daten in das Listenfeld einlesen können. Das Listenfeld bereiten wir wie in Bild 3 vor, indem wir die Eigenschaften **Spaltenanzahl** und die **Spaltenbreiten** einstellen. Außerdem legen wir für das Listenfeld die Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** auf **Beide** fest. Als Herkunftstyp legen wir die Callbackfunktion **Kundenliste** fest. Im Kopf des Klassenmoduls des Formulars deklarieren wir die folgenden beiden Variablen:

```
Dim db As DAO.Database
Dim rst As DAO.Recordset
```

Die Kopfzeile der Callbackfunktion unterscheidet sich vom vorigen Beispiel durch den Namen:

Daten in ungebundenen Formularen bearbeiten

Der übliche Weg, um Daten aus Tabellen in Access-Formularen anzuzeigen, ist die Angabe einer Tabelle oder Abfrage als Datenquelle und das Binden der Steuerelemente an die Felder dieser Quelle. Es gibt jedoch Anwendungsfälle, in denen diese Vorgehensweise nicht das gewünschte Ergebnis liefert. Dann kann man einen alternativen Weg gehen, auch wenn man damit viele Vorteile aufgibt und eine Menge zusätzlicher Aufwand entsteht. In diesem Beitrag erläutern wir, wie man die Daten aus Tabellen oder Abfragen auch ohne direkte Bindung an eine Tabelle oder Abfrage in einem Formular anzeigen, bearbeiten und wieder speichern kann und wie sogar das Anlegen neuer Datensätze möglich ist.

Anwendungsfälle für ungebundene Formulare

Access bietet mit der Bindung von Formularen und Steuerelementen an die Felder von Tabellen oder Abfragen eine sehr einfache Möglichkeit, Daten anzuzeigen, zu bearbeiten und diese wieder zu speichern.

Man kann allerdings auch den gewünschten Datensatz per VBA öffnen und die Daten auslesen und in die Steuerelemente eines Formulars schreiben. Wenn der Benutzer die gewünschten Änderungen durchgeführt hat, lassen sich diese Daten per VBA wieder zurück in die Tabelle schreiben. Damit ist allerdings ein stark erhöhter Programmieraufwand verbunden. Außerdem kann man damit nur jeweils einen Datensatz in einem Formular anzeigen. Die Datenblattansicht oder die Endlosansicht können wir so nicht mit Daten füllen. Die einzige Möglichkeit, mehrere Datensätze anzuzeigen, wäre ein Listenfeld, dem wir die Daten als Wertliste zuweisen.

Also fragt man sich: Aus welchem Grund sollte man sich diese Arbeit machen, wenn Access doch alle notwendigen Funktionen durch die Datenbindung anbietet? Uns fallen die folgenden Szenarien ein:

- Datensicherheit: Beim Zugriff auf geschützte Backenddatenbanken liegt das Kennwort unverschlüsselt beispielsweise in der Eigenschaft **Datensatzquelle** vor. Bei ungebundenen Formularen können wir das Kennwort im Code verstecken.
- Kontrolle: Gebundene Formulare und Steuerelemente übernehmen viele Aufgaben automatisch, aber das kann auch ein Nachteil sein. Wenn es zum Beispiel notwendig ist, die eingebaute Prüfung auf Gültigkeitsregeln oder andere Restriktionen zu umgehen, kann ein ungebundenes Formular helfen.
- Datensatzsperrungen: Im Mehrbenutzerbetrieb können Datensatzsperrungen oder gleichzeitiges Bearbeiten zu Verzögerungen oder Datenverlust führen. Bei ungebundenen Formularen treten Sperrungen maximal beim Lesen oder Schreiben auf, nicht während der gesamten Bearbeitung.

In den folgenden Abschnitten erläutern wir diese Gründe etwas ausführlicher.

Ungebundene Formulare für mehr Datensicherheit

Ein Grund für den Einsatz von ungebundenen Formularen ist die Datensicherheit. Angenommen, wir arbeiten mit einem Frontend und einem Backend, wobei die Daten im Backend durch ein Kennwort geschützt sind. Dann gibt es keine wirklich sichere Möglichkeit, per Datenbindung auf

die Daten im Backend zuzugreifen, ohne dass das Kennwort abgegriffen werden kann.

Dieses wird nämlich in der Verbindungszeichenfolge gespeichert – und das unverschlüsselt.

Mehr eigene Kontrolle durch ungebundene Formulare

In der Regel profitiert man davon, gebundene Formulare zu nutzen, denn damit werden gleich bei der Dateneingabe verschiedene Regeln wie Gültigkeitsregeln, Restriktionen durch Verknüpfungen oder Datentypen berücksichtigt. Manchmal ist das aber vielleicht nicht gewünscht.

Bei ungebundenen Formularen werden die im Datenmodell definierten Regeln gar nicht berücksichtigt – dies geschieht erst beim Speichern eines geänderten oder neuen Datensatzes.

Datensatzsperrungen und Mehrbenutzerbetrieb

Wollen zwei Benutzer auf den gleichen Datensatz zugreifen, kann das je nach festgelegtem Mechanismus für Datensatzsperrungen entweder gar nicht funktionieren oder zu Konflikten führen. Wenn man dies verhindern möchte und es okay ist, wenn die Änderungen des zuletzt auf den Datensatz zugreifenden Benutzers übernommen werden sollen, egal, ob zwischenzeitlich andere Änderungen vorgenommen wurden, kann der Einsatz eines ungebundenen Formulars sinnvoll sein. Dieses sperrt den Datensatz je nach Einstellungen maximal beim Zurückschreiben der geänderten Daten.

Ungebundenes Formular mit Daten füllen

Auch wenn es auf den ersten Blick merkwürdig aussieht: Der schnellste Weg zu einem Formular, das die Daten eines Datensatzes anzeigt, ohne an diesen gebunden zu sein, führt über das Erstellen eines herkömmlichen gebundenen Formulars.

Der Grund ist einfach: Nicht nur erleichtert die Datenbindung unter Access das Anzeigen, Bearbeiten und Speichern von Daten, sondern Access ist auch optimal darauf vorbereitet, gebundene Formulare zu erstellen. Aber das wissen Sie ja bereits.

Dennoch können Sie eine der beiden folgenden Alternativen nutzen:

- Sie legen ein Formular an, legen eine Tabelle oder Abfrage als Datensatzquelle fest, ziehen die gewünschten Felder aus der Feldliste in den Formularentwurf und leeren dann die Datensatzquelle und die Eigenschaft **Steuerelementinhalt** der hinzugefügten Steuerelemente wieder. Dann liegen bereits alle Steuerelemente mit korrekt beschriftetem Bezeichnungsfeld vor und haben außerdem einen Namen, der dem Feld entspricht, dessen Inhalt sie anzeigen sollen.

- Oder Sie legen ein Formular an und fügen für jedes Feld der Tabelle oder Abfrage ein Steuerelement hinzu. Für diese passen Sie dann zuerst die Beschriftungsfelder an und dann die Namen der Steuerelemente, damit wir diese später für das Schreiben und Lesen der enthaltenen Daten referenzieren können.

Wir denken, dass die erste Methode schneller funktioniert und wir damit außerdem die Steuerelemente sinnvoll anordnen können.

Beispieltabelle für ein ungebundenes Formular

Als Beispieltabelle verwenden wir eine sehr einfach gehaltene Kundentabelle namens **tblKunden**, die lediglich das Primärschlüsselfeld **KundeID** sowie die Textfelder **Vorname**, **Nachname**, **Anrede**, **Strasse**, **PLZ**, **Ort** und **Land** enthält. Diese sieht mit einigen Beispieldatensätzen wie in Bild 1 aus.

Schritt 1: Gebundenes Formular erstellen

Damit erstellen wir nun ein neues gebundenes Formular. Dazu gehen wir wie folgt vor:

- Legen Sie ein neues Formular an.

- Wählen Sie für die Eigenschaft **Datensatzquelle** die Tabelle **tblKunden** aus.

- Ziehen Sie alle Felder aus der Feldliste in den Detailbereich des Formularentwurfs.

- Speichern Sie das Formular unter dem Namen **frmKundenUngebunden** und schließen Sie es wieder.

tblKunden

KundeID	Vorname	Nachname	Anrede	Strasse	PLZ	Ort	Land
1	Adi	Stratmann	Herr	Kremser Straße 54	10589	Berlin	Deutschland
2	Heidi	Eich	Frau	Moosstraße 30	42289	Wuppertal	Deutschland
3	Wernfried	Birk	Herr	Wiener Straße 78	22297	Hamburg	Deutschland
4	Vitus	Krauße	Herr	Burgenlandstraße 77	20355	Hamburg	Deutschland
5	Jadwiga	Oehme	Frau	Peter-Rosegger-Straße 72	65187	Wiesbaden	Deutschland
6	Niko	Michel	Herr	Kindergartenstraße 42	12051	Berlin	Deutschland
7	Siegert	Loos	Herr	Lenaustraße 80	66115	Saarbrücken	Deutschland
8	Michl	Schroth	Herr	Dr. Karl Renner-Straße 97	01129	Dresden	Deutschland
9	Florentius	Wittek	Herr	Industriestraße 7	80638	München	Deutschland
10	Gernulf	Riegel	Herr	Erzherzog-Johann-Straße 37	81545	München	Deutschland
(Neu)							

Datensatz: 1 von 10 Kein Filter Suchen

Bild 1: Tabelle mit Beispieldaten

Das Formular sieht nun wie in Bild 2 aus.

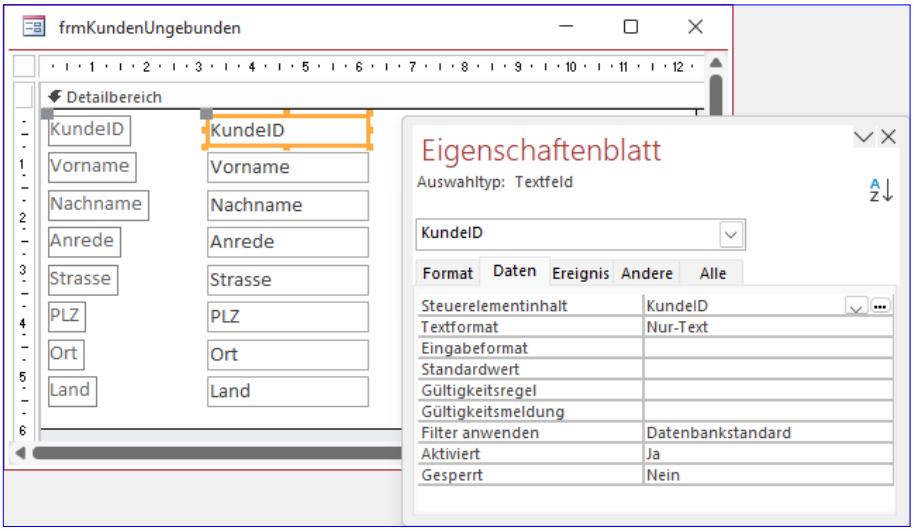


Bild 2: Das gebundene Formular

Schritt 2: Formularbindung aufheben

Nun erledigen wir die folgenden Schritte, um die Bindung von Formular und Steuerelementen aufzuheben:

- Leeren Sie die Eigenschaft **Datensatzquelle** des Formulars.

- Leeren Sie die Eigenschaft **Steuerelementinhalt** aller gebundenen Steuerelemente.

Letzteres geht am schnellsten, indem Sie alle gebundenen Steuerelemente markieren und dann die Eigenschaft **Steuerelementinhalt** bearbeiten. Diese wird nun zunächst leer angezeigt, weil sie für die verschiedenen markierten Steuerelemente unterschiedliche

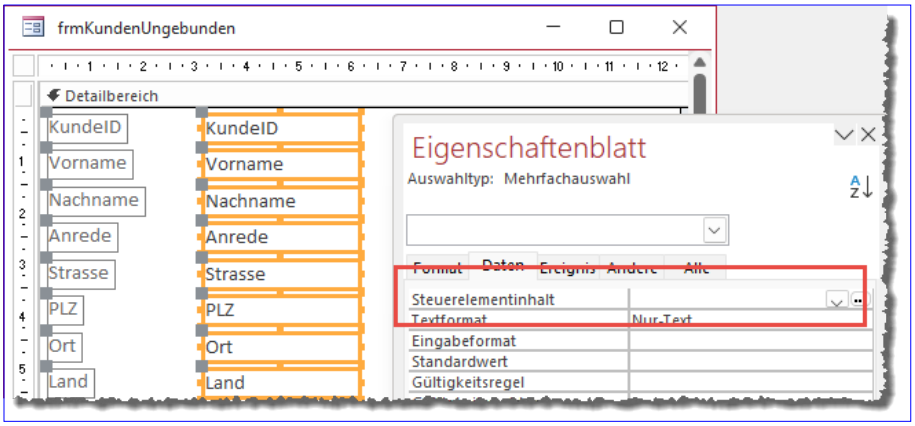


Bild 3: Die Eigenschaft **Steuerelementinhalt** wird leer angezeigt, weil die markierten Steuerelemente unterschiedliche Werte für diese Eigenschaft aufweisen.

Werte enthält (siehe Bild 3). Wie aber sollen wir eine leere Eigenschaft leeren? Dazu geben wir ein beliebiges Zeichen ein und entfernen dieses wieder. Anschließend ist die Eigenschaft wirklich leer, wie das Anklicken der einzelnen Steuerelemente zeigt.

Wechseln wir nun zur Formularansicht, sehen wir ein Formular mit leeren Steuerelementen, die wir nach Bedarf füllen können.

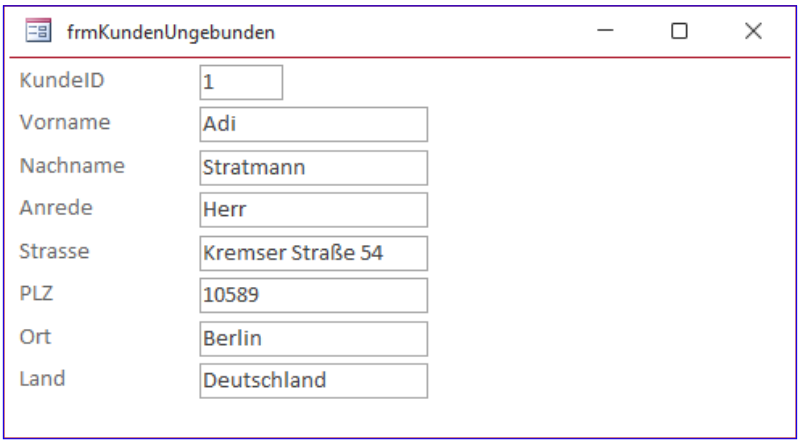


Bild 4: Daten im ungebundenen Formular

Unnötige Elemente entfernen

In einem ungebundenen Formular benötigen wir die Elemente für die Navigation in den Datensätzen nicht.

Daher stellen wir die Eigenschaften **Navigationsschaltflächen** und **Datensatzmarkierer** auf den Wert **Nein** ein.

Ersten Datensatz der Tabelle oder Abfrage anzeigen

Nun wollen wir die Steuerelemente mit den Daten eines Datensatzes füllen. Nur welchen verwenden wir? Wir verwenden zunächst den ersten Datensatz. Später schauen wir uns an, wie wir das Formular für die Anzeige eines speziellen Datensatzes öffnen können.

Um das Formular mit den Daten eines Datensatzes zu versehen, benötigen wir eines der Ereignisse, die direkt beim Öffnen des Formulars ausgelöst werden. In diesem Fall nutzen wir direkt das Ereignis **Beim Öffnen**.

Nachdem wir dieses angelegt haben, ergänzen wir die leere Prozedur um die folgenden Zeilen:

```
Private Sub Form_Open(Cancel As Integer)
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT TOP 1 * " _
        & "FROM tblKunden", dbOpenDynaset)
    Me!KundeID = rst!KundeID
    Me!Vorname = rst!Vorname
    Me!Nachname = rst!Nachname
    Me!Anrede = rst!Anrede
    Me!Strasse = rst!Strasse
    Me!PLZ = rst!PLZ
    Me!Ort = rst!Ort
    Me!Land = rst!Land
    rst.Close
    Set rst = Nothing
    Set db = Nothing
End Sub
```

Diese Prozedur erstellt ein **Database**- und ein **Recordset**-Objekt und liest lediglich den ersten Datensatz der Tabelle **tblKunden** in das Recordset ein (**TOP 1**).

Dann schreibt sie die Werte der einzelnen Felder der Tabelle in die jeweiligen Textfelder.

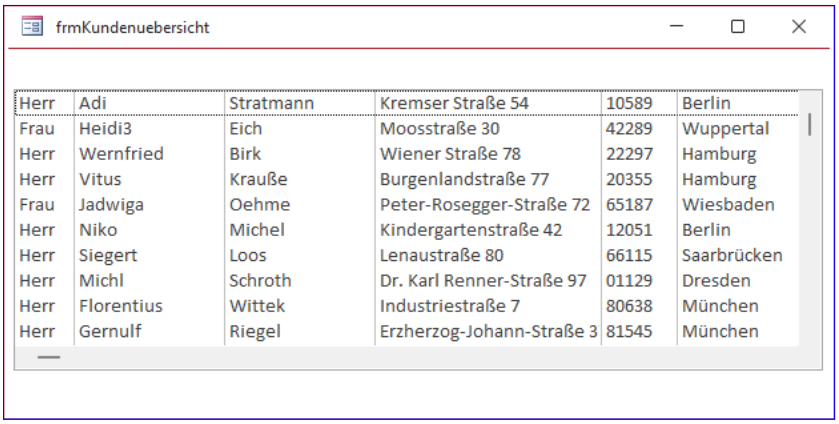
Anschließend schließt sie das Recordset und leert die Variablen **rst** und **db**.

Öffnen wir das Formular nun in der Formularansicht, zeigt es wie in Bild 4 den gewünschten Datensatz an. Wir können die Daten nun ändern, allerdings werden die Änderungen nicht in die zugrunde liegende Tabelle übernommen.

Ungebundene Daten in Übersicht und Detailansicht

In den Beiträgen »Daten in ungebundenen Formularen bearbeiten« (www.access-im-unternehmen.de/1442) und »Ungebundene Listen und Kombis mit Daten füllen« (www.access-im-unternehmen.de/1440) haben wir Möglichkeiten aufgezeigt, um Daten ohne die Bindung an Tabellen oder Abfragen in Formularen und Steuerelementen anzuzeigen. In diesem Beitrag gehen wir einen Schritt weiter und kombinieren die beiden Darstellungen einer Übersicht von Daten und der Detailansicht eines ausgewählten Datensatzes. Dabei zeigen wir, wie wir in der Übersicht einen Datensatz auswählen und diesen zum Bearbeiten öffnen, einen neuen Datensatz anlegen oder Datensätze löschen.

Ausgangspunkt für das Beispiel zu diesem Beitrag sind die Formulare aus den oben genannten Beiträgen. Dabei wollen wir das Formular mit der Übersicht der Kunden in einem Listenfeld, das wir in **frmKundenuebersicht** umbenannt haben, um einige Elemente erweitern (siehe Bild 1).



Herr	Adi	Stratmann	Kremser Straße 54	10589	Berlin
Frau	Heidi3	Eich	Moosstraße 30	42289	Wuppertal
Herr	Wernfried	Birk	Wiener Straße 78	22297	Hamburg
Herr	Vitus	Krauße	Burgenlandstraße 77	20355	Hamburg
Frau	Jadwiga	Oehme	Peter-Rosegger-Straße 72	65187	Wiesbaden
Herr	Niko	Michel	Kindergartenstraße 42	12051	Berlin
Herr	Siegert	Loos	Lenaustraße 80	66115	Saarbrücken
Herr	Michl	Schroth	Dr. Karl Renner-Straße 97	01129	Dresden
Herr	Florentius	Wittek	Industriestraße 7	80638	München
Herr	Gernulf	Riegel	Erzherzog-Johann-Straße 3	81545	München

Bild 1: Ausgangspunkt des Beitrags

Wir haben die folgenden Erweiterungen vorgesehen:

- Hinzufügen einer Schaltfläche zum Anzeigen des aktuell ausgewählten Eintrags im Detailformular **frmKundendetails** (ebenfalls ungebunden)
- Hinzufügen einer Funktion, mit welcher der entsprechende Datensatz per Doppelklick auf den jeweiligen Eintrag im Listenfeld angezeigt werden kann
- Hinzufügen einer Schaltfläche zum Anlegen eines neuen Datensatzes über das Formular **frmKundendetails**
- Hinzufügen einer Schaltfläche zum Löschen des aktuell markierten Datensatzes aus der zugrunde liegenden Tabelle
- Programmieren von Funktionen, die dafür sorgen, dass Änderungen an Datensätzen oder neu hinzugefügte Datensätze nach dem Schließen des Formulars **frmKundendetails** direkt im Listenfeld widergespiegelt werden

Anlegen der Schaltflächen

Bei der Gestaltung der Schaltflächen haben wir neben den Beschriftungen auch noch Icons hinzufügt. Außerdem haben wir die Einstellung **Vertikaler Anker** auf **Unten** eingestellt, damit diese, wenn das Formular und das Listenfeld vergrößert werden, ebenfalls nach unten verschoben werden (für das Listenfeld ist diese Einstellung auf den Wert **Beide** eingestellt). Die Schaltflächen heißen **cmdBearbeiten**, **cmdAnlegen** und **cmdLoeschen** (siehe Bild 2).

Bearbeiten eines Datensatzes per Schaltfläche

Um den aktuell im Listenfeld markierten Datensatz im Formular **frmKundendetails** anzuzeigen, verwenden wir die Schaltfläche **cmdBearbeiten**.

Für diese hinterlegen wir die folgende Prozedur:

```
Private Sub cmdBearbeiten_Click()
    If Not IsNull(Me!lstKundenUngebunden) Then
        DoCmd.OpenForm "frmKundendetails", _
            OpenArgs:=Me!lstKundenUngebunden, _
            WindowMode:=acDialog
    Else
        MsgBox "Bitte markieren Sie einen Datensatz.", _
            vbOKOnly + vbExclamation, _
            "Kein Datensatz markiert"
    End If
End Sub
```

Diese Prozedur prüft, ob im Listenfeld überhaupt ein Eintrag markiert ist. Falls ja, öffnet sie das Formular **frmKundendetails** als modalen Dialog und übergibt den Primärschlüsselwert des markierten Eintrags mit dem Parameter **OpenArgs**.

Ist aktuell kein Eintrag markiert, erscheint eine Meldung mit einem entsprechenden Hinweis.

Im Formular **frmKundendetails** nimmt die Prozedur **Form_Open** das Öffnungsargument über die Eigenschaft **OpenArgs** entgegen und zeigt die Daten des entsprechenden Datensatzes an. Die Details hierzu finden Sie im Beitrag **Daten in ungebundenen Formularen bearbeiten** (**www.access-im-unternehmen.de/1442**).

Nun wird es interessant: In gebundenen Formularen würden wir nun nach dem Bearbeiten des Datensatzes im Detailformular das Formular ausblenden, wodurch der Code im aufrufenden Formular fortgesetzt würde. Hier

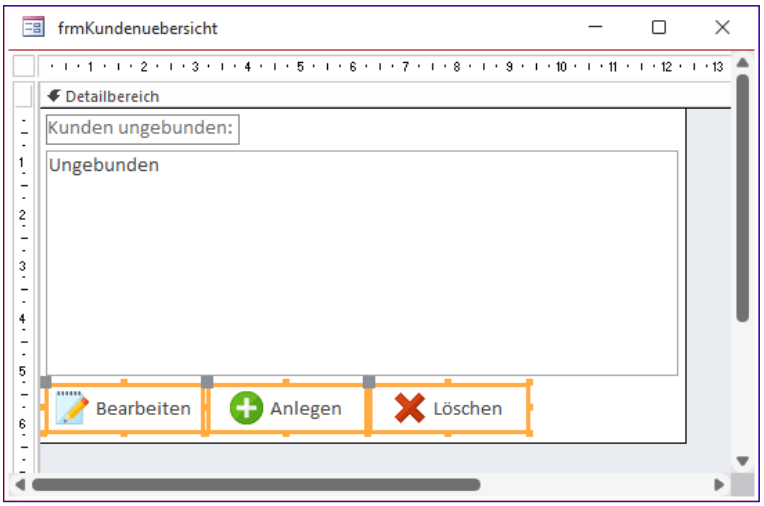


Bild 2: Schaltflächen für die Datenoperationen

würden wir dann abfragen, ob das Detailformular noch geöffnet ist und gegebenenfalls auf die geänderten Daten reagieren, bevor wir die Daten im aufrufenden Formular entsprechend anpassen. Dazu reicht dann meist ein Aufruf der Methode **Requery** der Übersicht.

Hier können wir ähnlich vorgehen. Wir machen das Detailformular beim Anklicken der **OK**-Schaltfläche unsichtbar, damit wir im aufrufenden Formular feststellen können, ob der Benutzer die **OK**- oder die **Abbrechen**-Schaltfläche gedrückt hat (die **Abbrechen**-Schaltfläche würde das Formular direkt schließen).

Den Code der **OK**-Schaltfläche passen wir dazu wie folgt an:

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

Im aufrufenden Formular können wir nun allerdings nicht die **Requery**-Methode des Listenfeldes aufrufen, da das Listenfeld nicht an eine Tabelle oder Abfrage gebunden ist, sondern anderweitig gefüllt wurde. An dieser Stelle haben wir nun zwei Möglichkeiten:

- Wir füllen das Listenfeld erneut oder
- wir lesen nur den geänderten Wert erneut ein.

Die erste Variante ist schnell erledigt: Wir rufen einfach die Prozedur, welche die Daten in das Listenfeld einliest, erneut auf. Die Erweiterung sieht wie folgt aus:

```
...
If Not IsNull(Me!lstKundenUngebunden) Then
    DoCmd.OpenForm "frmKundendetails", _
        OpenArgs:=Me!lstKundenUngebunden, _
        WindowMode:=acDialog
    If IstFormularGeoeffnet("frmKundendetails") Then
        KundenEinlesen
    End If
...
```

Damit ist die Aufgabe prinzipiell erledigt. Wir rufen das Detailformular auf, nehmen eine Änderung vor, schließen das Detailformular und der Inhalt des Listenfeldes wird aktualisiert. In der aktuellen Fassung der Prozedur, welche die Daten der Tabelle **tblKunden** einliest und dem Listenfeld **lstKunden** hinzufügt, wird das Listenfeld allerdings vorher nicht geleert. Das bedeutet, dass die aktualisierten Daten durch die Aufrufe der **AddItem**-Methode an die bestehenden Daten angehängt werden. Um dies zu ändern, müssen wir das Listenfeld an irgendeiner Stelle leeren, was wir mit der folgenden Anweisung erledigen können:

```
Me!lstKundenUngebunden.RowSource = ""
```

Diese können wir beispielsweise in der Prozedur **KundenEinlesen** vor der **Do While**-Schleife einfügen. Allerdings gibt es noch eine Alternative.

Gezieltes Aktualisieren der Daten des Listenfeldes

Da wir ein ungebundenes Listenfeld nutzen und dieses mit der **AddItem**-Methode füllen, können wir uns auch noch eine Alternative zum erneuten Laden aller Daten ansehen – denn je nach der Menge der Datensätze kann dieser Vorgang durchaus zu sichtbaren Verzögerungen führen.

Also erstellen wir eine Prozedur namens **KundeAktualisieren**, die wir von **cmdBearbeiten_Click** aus wie folgt aufrufen.

Dabei übergeben wir den Primärschlüsselwert des geänderten Datensatzes sowie seinen Index im Listenfeld, den wir mit der Eigenschaft **ListIndex** ermitteln:

```
Private Sub cmdBearbeiten_Click()
    ...
        If IstFormularGeoeffnet("frmKundendetails") Then
            KundeAktualisieren Me!lstKundenUngebunden, _
                Me!lstKundenUngebunden.ListIndex
        End If
    ...
End Sub
```

Die Prozedur **KundeAktualisieren** erstellt ein neues Recordset auf Basis der Tabelle **tblKunden** mit dem geänderten Datensatz (siehe Listing 1). Ist dieses Recordset nicht leer, entfernt die Prozedur zuerst den vorhandenen Eintrag mit der **RemoveItem**-Methode. Den zu entfernenden Datensatz identifizieren wir mit dem Index aus **lngIndex**.

Dann fügen wir mit der **AddItem**-Methode den geänderten Eintrag hinzu. Damit dieser nicht einfach hinten angehängt wird, geben wir als zweiten Parameter wieder den Index aus **lngIndex** an.

Damit geschieht das Aktualisieren des geänderten Eintrags im Listenfeld wesentlich schneller.

Bearbeiten eines Datensatzes per Doppelklick

Wesentlich einfacher als das Markieren des zu bearbeitenden Datensatzes und das anschließende Betätigen der **Bearbeiten**-Schaltfläche ist ein Doppelklick auf den Eintrag im Listenfeld.

SQL Server: Vollsicherung und Wiederherstellung

Bernd Jungbluth, Horn und André Minhorst, Duisburg

Wer eine Access-Datenbank zum Speichern seiner Daten verwendet, hat bei der Sicherung leichtes Spiel: Er braucht einfach nur eine Kopie dieser Datenbank anzulegen. Bei SQL Server-Datenbanken ist dies etwas komplizierter. Wir können hier beispielsweise nicht einfach die Datenbankdateien kopieren, da der SQL Server normalerweise ständig darauf zugreift. Welche Schritte zum Anlegen des Backups einer SQL Server-Datenbank nötig sind, zeigen wir in diesem Beitrag. Außerdem schauen wir uns an, wie wir die Datenbank auf Basis der Sicherung wiederherstellen können.

Backup über das SQL Server Management Studio

Zum Erstellen eines Backups einer SQL Server-Datenbank ist das SQL Server Management Studio die erste Adresse. Hier finden wir über die Benutzeroberfläche alle notwendigen Elemente.

Vollsicherung über Nacht

Ziel dieses Beitrags ist es, den SQL Server so zu programmieren, dass er jede Nacht eine Vollsicherung einer Datenbank herstellt. Damit stellen wir sicher, dass die Daten bis zur vorherigen Nacht im Falle eines Datenverlustes wiederherstellbar sind. Es gibt noch andere Sicherungsmodelle, mit denen wir in kleineren Zeitabständen Sicherungen durchführen. Das ist jedoch nicht Thema dieses Beitrags.

Recovery Model einstellen

Wir gehen also davon aus, dass wir mit dem Verlust der Daten des aktuellen Tages leben können. In diesem Kontext stellen wir das Recovery Model für die Datenbank auf einen Wert ein, mit dem die Größe der Datei zum Speichern der Transaktionen klein gehalten wird – damit ist die **.ldf**-Datei der Datenbank im Gegensatz zur **.mdf**-Datei gemeint.

Das Protokollieren von Transaktionen sorgt dafür, dass Transaktionen rückgängig gemacht werden können, solange diese noch nicht abgeschlossen sind. Bei der Einstellung des Recovery Models auf **Simple** entfernt der SQL Server automatisch die bereits abgeschlossenen und als inaktiv gekennzeichneten Transaktionen im Transaktionsprotokoll. Im Gegensatz zum Recovery Model **Full**. Hier löscht der SQL Server die inaktiven Transaktionen nicht automatisch. Diese werden erst bei einer Sicherung des Transaktionsprotokolls entfernt. Das klingt zunächst nach Mehraufwand, bietet aber die Möglichkeit den Datenverlust zu verringern. Ein kurzes Beispiel: Täglich um 22 Uhr wird eine Vollsicherung der Datenbank erstellt. Dazu gibt es im Zeitraum von 6 - 20 Uhr alle 15 Minuten eine Si-

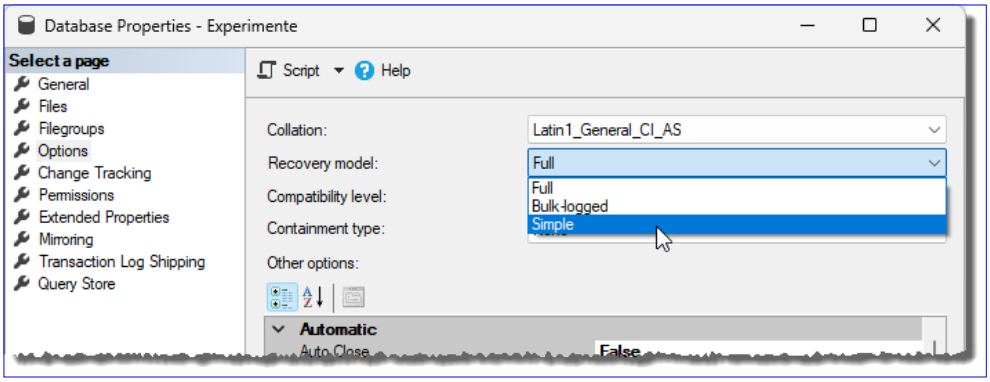


Bild 1: Einstellen des gewünschten **Recovery Models**

cherung des Transaktionsprotokolls. Durch diese Sicherung lässt sich in einem Notfall um 15:05 Uhr die Datenbank bis zum Zeitpunkt 15 Uhr wiederherstellen. Der mögliche Datenverlust liegt bei 5 Minuten. In diesem Beitrag ist unser Ziel, eine Wiederherstellung der Daten bis zur letzten Vollsicherung zu gewährleisten. Hierzu ist das Recovery Model **Simple** ausreichend.

Diese Einstellung nehmen wir in den Optionen der Datenbank vor. Dazu wählen wir aus dem Kontextmenü des jeweilige Datenbankeintrags den Befehl **Properties** aus und wechseln dort zur Seite **Options**. Hier finden wir die Option **Recovery model**, die wir auf **Simple** einstellen (siehe Bild 1).

Backup anlegen

Wenn wir ein Backup von einer bestimmten Datenbank machen wollen, öffnen wir den Objekt-Explorer und navigieren zur entsprechenden Datenbank. Anschließend klicken wir mit der rechten Maustaste auf den Datenbanknamen, hier **Experimente**, und wählen den Eintrag **Tasks|Back Up...** aus (siehe Bild 2).

Damit öffnen wir den Dialog **Back Up Database – [Datenbankname]** (siehe Bild 3). Ein

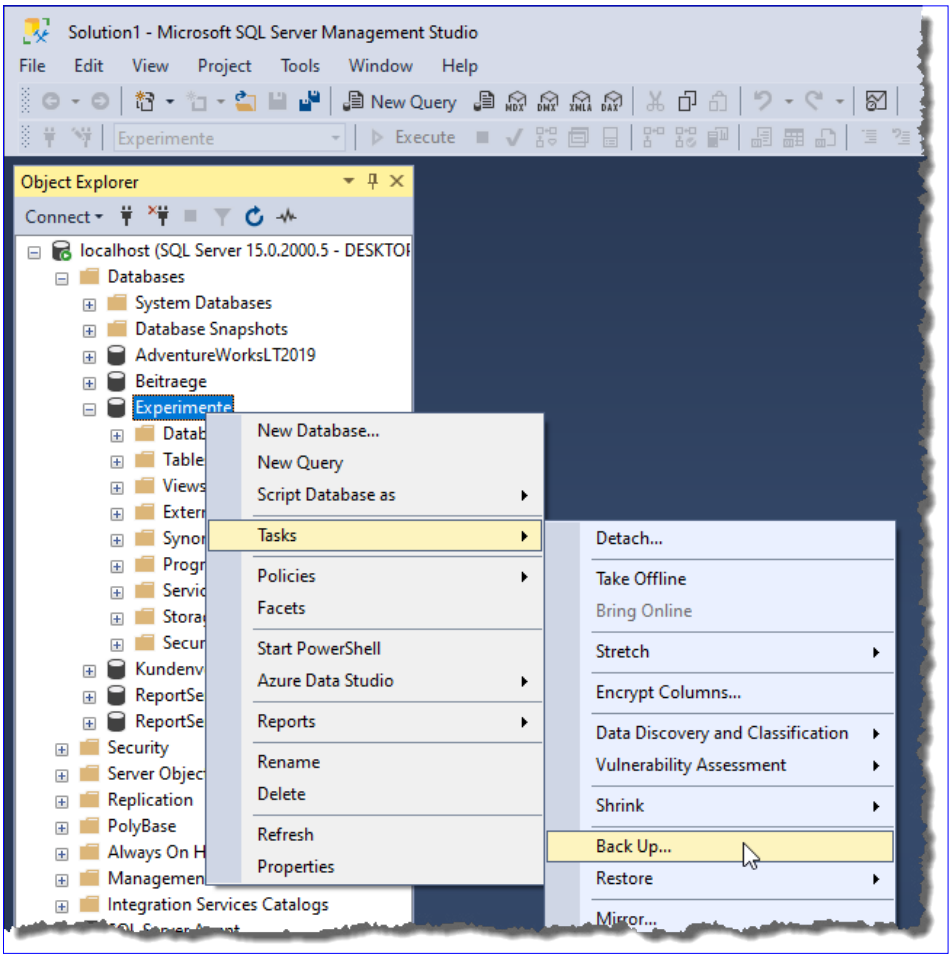


Bild 2: Aufrufen des **Back Up**-Dialogs

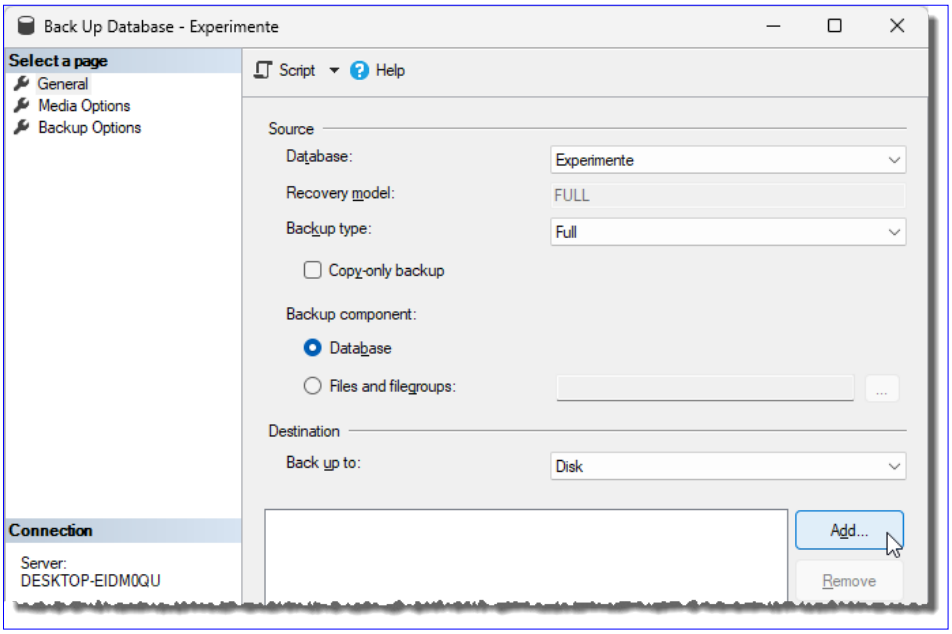


Bild 3: Der **Back Up**-Dialog

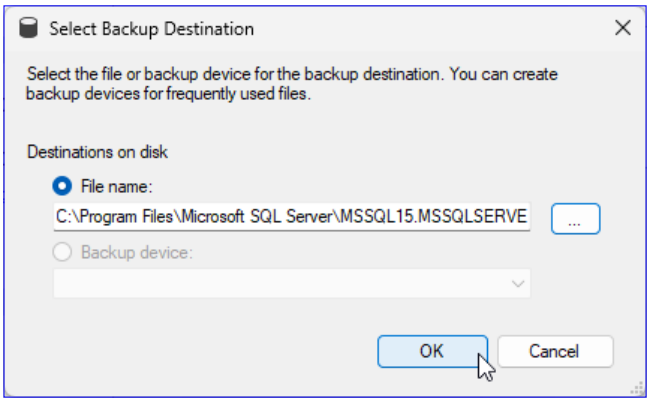


Bild 4: Ziel des Backups eingeben

Klick auf die Schaltfläche **OK** in diesem Dialog führt immer zur Ausführung des Backups. Wenn Sie also aktuell kein Backup erstellen möchten, sondern wie nachfolgend gezeigt nur das T-SQL-Skript für ein solches zusammenstellen wollen, verlassen Sie den Dialog am einfachsten mit der **Cancel**-Schaltfläche. Dieser zeigt im Bereich **General** an, welche Datenbank gesichert werden soll, hier **Experimente**. Wir wollen ein volles Backup erstellen (**Backup type** erhält den Wert **Full**). Und wir wollen das Backup auf der Festplatte anlegen, also behalten wir unter **Destination** den Wert **Disk** für **Back up to** bei (siehe Bild 4).

Darunter klicken wir auf die Schaltfläche **Add**, um den Speicherort für das Backup zu definieren. Es erscheint ein Dialog namens **Select Backup Destination** (siehe Bild 5). Hier wählen wir die Option **File name** aus und klicken auf die Schaltfläche rechts mit den drei Punkten. Es erscheint ein weiterer Dialog, mit dem wir das Zielverzeichnis auswählen und den Dateinamen eingeben können. Als Zielverzeichnis wird standardmäßig der folgende Ordner verwendet:

```
C:\Program Files\Microsoft SQL Server\MSSQL15.MSSQLSERVER\
MSSQL\Backup\
```

Backup regelmäßig ausführen per SQL Server Agent

Damit haben wir den Speicherort für das Backup definiert und können uns darum kümmern, dass dieses regelmäßig

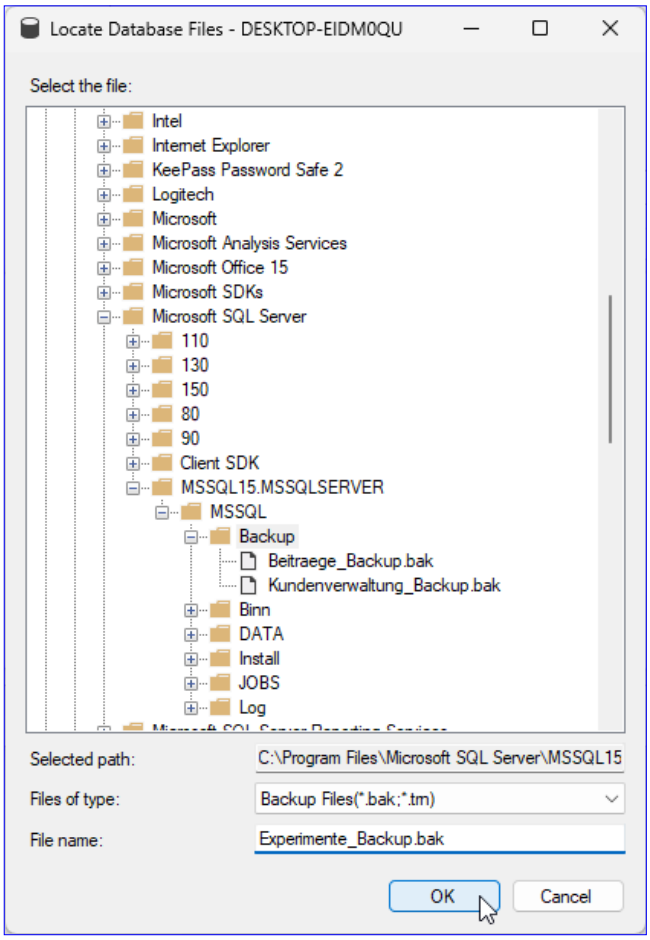


Bild 5: Auswählen des Verzeichnisses und des Dateinamens für das Backup

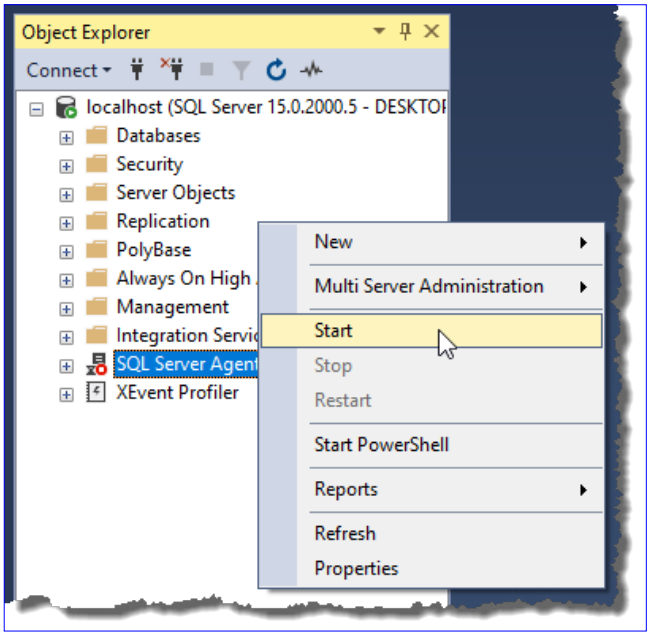


Bild 6: Starten des SQL Server Agents

ausgeführt wird. Dazu nutzen wir den SQL Server Agent, der übrigens in der Express Edition nicht enthalten ist, aber zum Beispiel in der Developer Edition. Dazu schließen wir die geöffneten Dialoge bis auf den Backup-Dialog wieder und schauen uns an, ob der SQL Server Agent bereits läuft. Ist das nicht der Fall, sehen wir ein Icon wie in Bild 6 und starten diesen über den Kontextmenü-Eintrag **Start**. Die anschließende Meldung mit einer Rückfrage, ob der Dienst wirklich gestartet werden soll, beantworten wir mit **Yes**. Wenige Augenblicke später erscheint das Icon des Eintrags **SQL Server Agent** mit einem grünen Symbol, was den gestarteten Zustand zeigt.

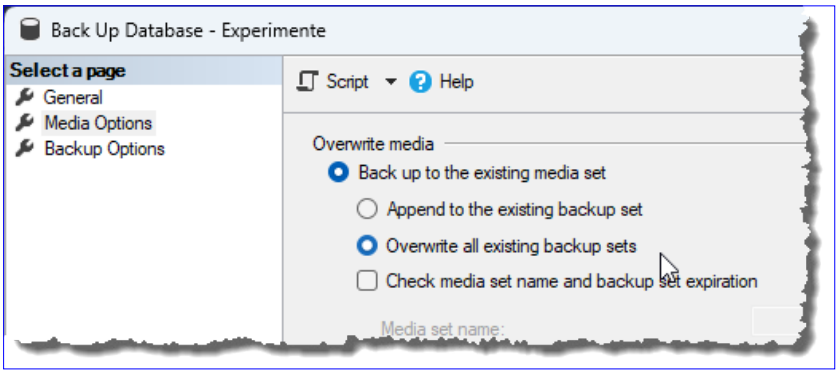


Bild 7: Überschreiben des Backups aktivieren

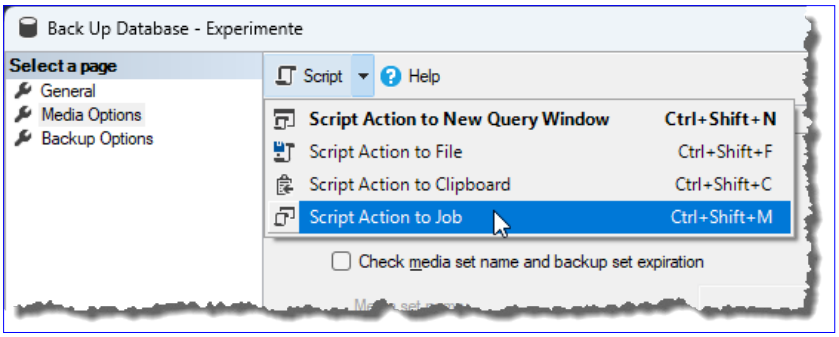


Bild 8: Skript in Job schreiben

Job erstellen

Danach öffnen wir erneut über den Kontextmenü-Eintrag **Tasksl-Backup** den Dialog zum Definieren der Backup-Einstellungen und wechseln zur Seite **Media Options**. Hier stellen wir nun ein, wie mit den vorherigen Backups umgegangen werden soll. Wir möchten das vorherige Backup mit dem neuen Backup überschreiben und stellen dazu die Option **Overwrite media** auf **Overwrite all existing** backup sets ein (siehe Bild 7).

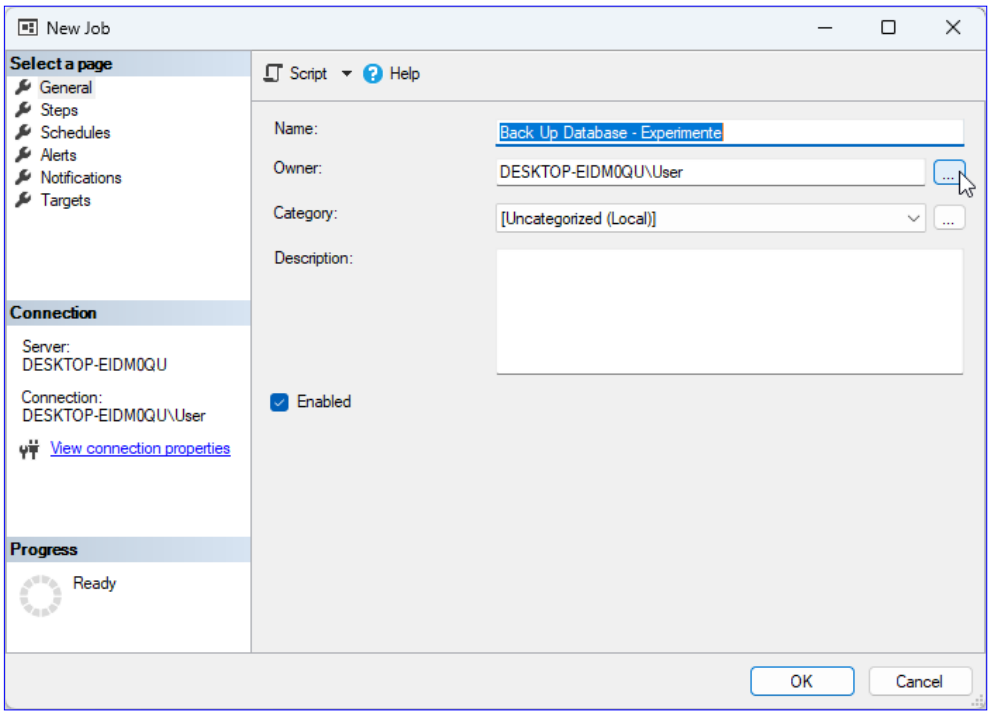


Bild 9: Anlegen eines neuen Jobs

Job mit Backup-Skript erstellen

Wir bleiben in diesem Dialog und klicken oben auf

Belege und Belegdaten nach lexoffice hochladen

Im Beitrag »Zugriff auf lexoffice per REST-API und VBA« (www.access-im-unternehmen.de/1422) haben wir bereits gezeigt, wie wir in lexoffice Rechnungen auf Basis von Rechnungsdaten aus einer Access-Datenbank erstellen können. In manchen Fällen reichen die Möglichkeiten von lexoffice nicht aus, um die gewünschten Rechnungen zu erstellen. Dann kann man lexoffice aber immer noch für die Buchhaltung nutzen. Die Rechnung erstellt man dann in Access statt direkt in lexoffice und überträgt dann ein PDF-Dokument mit der Rechnung sowie die begleitenden Daten wie den Umsatz, die enthaltenen Steuern und weitere Informationen per Rest API nach lexoffice. Dieser Beitrag zeigt, wie wir dies bewerkstelligen können.

Um diesen Vorgang manuell auszuführen, startet man in **lexoffice** das Erfassen eines neuen Belegs (siehe Bild 1).

Hier kann man einen Beleg hochladen. lexoffice versucht dann, die benötigten Informationen für den rechten Bereich automatisch zu ermitteln. Gelingt dies nicht oder nur teilweise, arbeiten wir von Hand nach.

Mit der nachfolgend vorgestellten Lösung wollen wir solche Unwägbarkeiten verhindern und gleichzeitig die benötigte Zeit minimieren. Statt das PDF manuell in das Formular zu ziehen und abzuwarten, bis lexoffice das Dokument analysiert hat, übergeben wir das Dokument samt den benötigten Daten per Webservice. Wie man diesen bedient, haben wir schon ausführlich in dem oben genannten Beitrag beschrieben.

Wir brauchen also nur auf den dort erwähnten Techniken aufzubauen und diese an die neue Aufgabe anzupassen. Wer schon einmal in der Dokumentation der Rest API

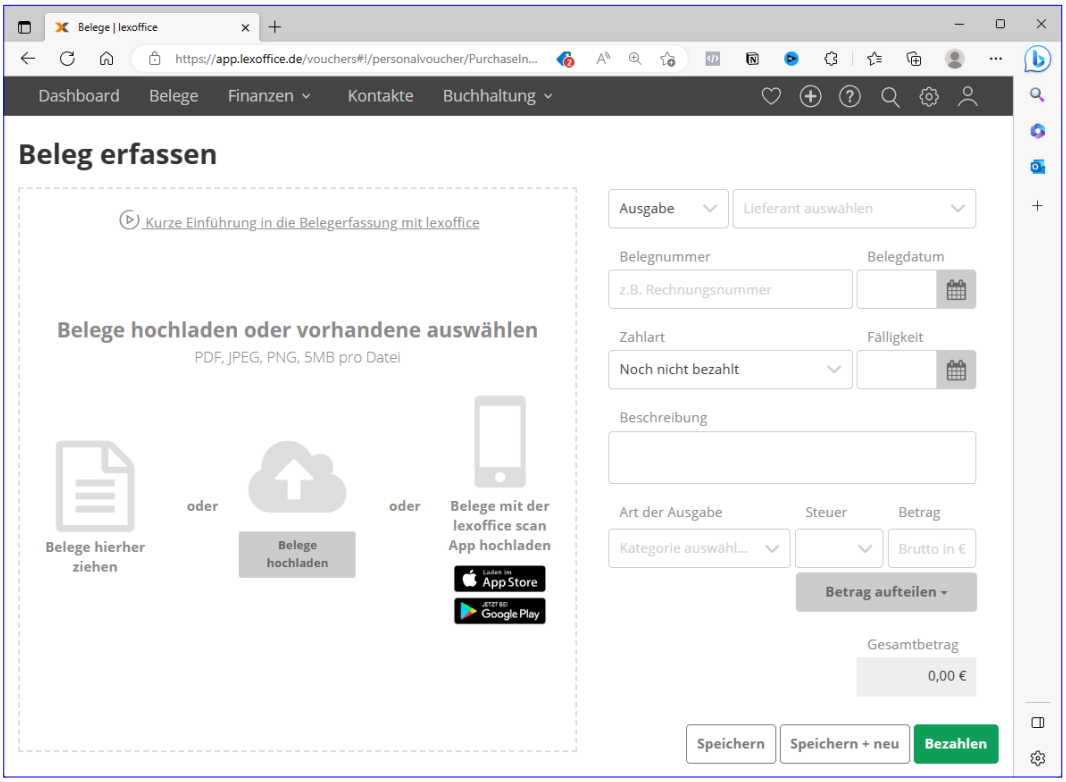


Bild 1: Erfassen eines Belegs in lexoffice

stöbern möchte: Während wir im ersten Beitrag Elemente des Typs **Invoice** erstellt haben (Rechnung), geht es nun um das Hochladen von Elementen des Typs **Voucher** (Beleg). Beide verwenden ein **Contact**-Element, das je nach Anwendungszweck als Debitor oder Kreditor verwendet wird. Da wir im ersten Teil bereits die Techniken zum Hochladen von Kundendaten in Form eines **Contact**-Elements erledigt haben, brauchen wir uns auch darauf nicht mehr zu konzentrieren, sondern können uns dem Upload von Elementen des Typs **Voucher** kümmern.

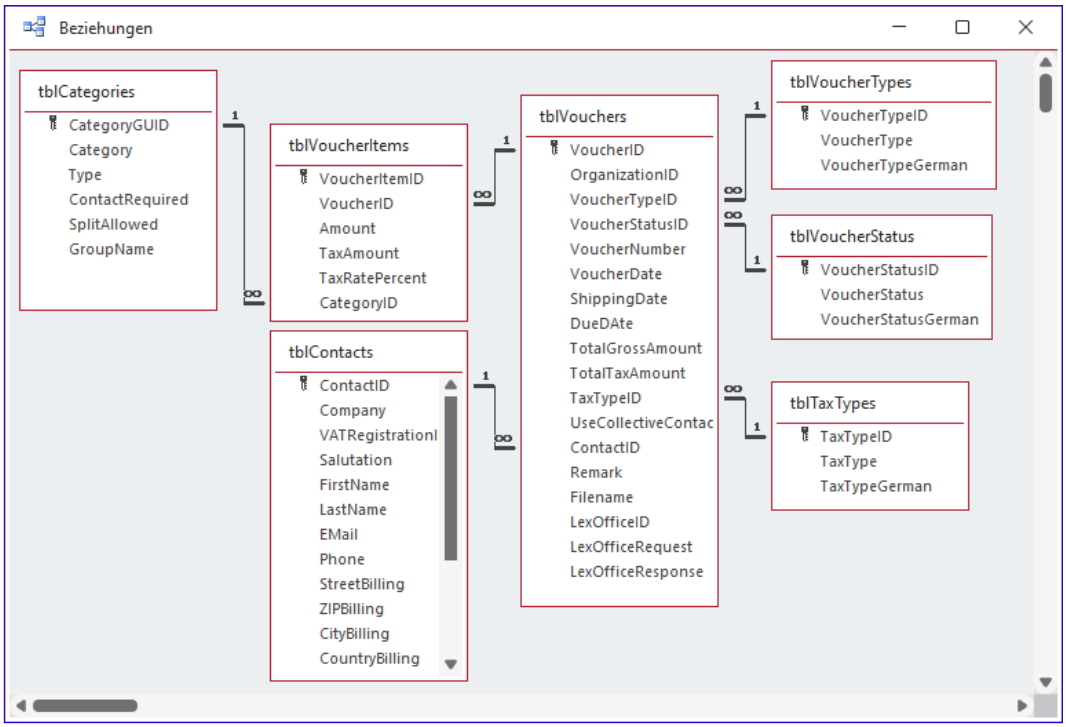


Bild 2: Datenmodell der Beispieldatenbank

Datenmodell für Belege

Für das Erfassen eines Belegs benötigen wir mehr Daten als für das Erstellen von Rechnungen. In Bild 2 sehen wir alle benötigten Tabellen. Die Tabelle **tblContacts** kennen wir bereits aus dem oben genannten Beitrag. Jedem Voucher aus der Tabelle **tblVouchers** weisen wir genau einen Kontakt zu, der je nach der Art der Positionen des Belegs Kreditor oder Debitor ist. Damit kommen wir gleich zur Tabelle **tblVouchers**, welche die grundlegenden Informationen zum Beleg enthält.

Belegtyp

Dazu gehört der Typ des Belegs. Diese haben wir in einer eigenen Lookuptabelle namens **tblVoucherTypes** abgelegt (siehe Bild 3). Diese enthält neben dem Primärschlüsselfeld ein Feld namens **VoucherType**, das die Beschreibung enthält, die wir in die JSON-Datei mit den Informationen zum Erfassen des Belegs schreiben. Ein weiteres Feld namens **VoucherTypeGerman** enthält den Wert, den wir im Formular zur Auswahl des Belegtyps anbieten wollen.

Belegstatus

Ein weiteres Feld namens **VoucherStatusID** dient dazu, den Status des Belegs zu übermitteln. Auch dafür haben wir eine Lookuptabelle angelegt. Diese heißt **tblVoucherStatus** und enthält ebenfalls ein Feld für die Angabe des Wertes für die JSON-Datei und eines für die deutsche

tblVoucherTypes

Voucher	VoucherType	VoucherTypeGerman
1	salesinvoice	Einnahme
2	salescreditnode	Einn-Minderung
3	purchaseinvoice	Ausgabe
4	purchasecreditnote	Ausg.-Minderung
(Neu)		

Datensatz: 1 von 4 Kein Filter Suchen

Bild 3: Belegtypen

Version des Wertes. Diese Tabelle enthält die Werte aus Bild 4.

Steuertyp

Die nächste Lookuptabelle namens **tblTaxTypes** liefert die Werte für das Feld **TaxTypeID**. Hier legen wir fest, ob wir in den Belegpositionen die Netto- oder die Bruttowerte angeben (siehe Bild 5).

Fremdschlüsselfelder der Tabelle tblVoucher

Die Felder **VoucherTypeID**, **VoucherStatusID** und **TaxTypeID** der Tabelle **tblVoucher** richten wir jeweils als Nachschlagefelder mit den zuvor vorgestellten Tabellen als Datensatzherkunft ein. Dabei geben wir als anzuzeigendes Feld jeweils das Feld mit der deutschen Bezeichnung der jeweiligen Werte an.

Weitere Felder der Tabelle tblVoucher

Außerdem enthält die Tabelle noch Felder zum Speichern der Rechnungsnummer (**VoucherID**), das Rechnungsdatum (**VoucherDate**), das Versanddatum (**ShippingDate**) und das Fälligkeitsdatum (**DueDate**). Außerdem enthält die Tabelle die Bruttosumme (**TotalGrossAmount**) und die Summe der Mehrwertsteuer (**TotalTaxAmount**).

Die nächsten beiden Felder **UseCollectiveContact** und **ContactID** geben an, ob wir als Lieferant oder Kunde den Wert **Sammelkunde** angeben wollen oder einen bereits gespeicherten Kontakt. Mit **Remark** geben wir einen Bemerkungstext an und mit **Filename** den Namen des Belegs, beispielsweise in Form einer PDF-Datei.

Die übrigen Felder haben folgenden Nutzen:

- **LexOfficeID**: Speichert die ID, unter welcher der Beleg in lexoffice angelegt wurde.
- **LexOfficeRequest**: Speichert den Request, den wir zum Anlegen des Belegs an lexoffice geschickt haben.

tblVoucherStatus

VoucherStatusID	VoucherStatus	VoucherStatusGerman
1	open	Offen
2	paid	Bezahlt
3	paidoff	paidoff
4	voided	voided
5	transferred	transferred
6	sepadebit	sepadebit
(Neu)		

Bild 4: Tabelle mit den verschiedenen Belegstatus

tblTaxTypes

TaxTypeID	TaxType	TaxTypeGerman	Zum Hinzufügen klicke
1	net	Netto	
2	gross	Brutto	
(Neu)			

Datensatz: 3 von 3 Kein Filter Suchen

Bild 5: Tabelle mit den Steuertypen

- **LexOfficeResponse**: Speichert die Antwort von lexoffice auf diese Anfrage.

Belegpositionen

Im Gegensatz zu einer Rechnung, die verschiedene Positionen mit den zu bezahlenden Produkten oder Leistungen enthält, tragen wir für einen Beleg Zusammenfassungen von Beträgen mit bestimmten Mehrwertsteuersätzen und Buchungskategorien ein. Diese werden jedoch ähnlich wie die Rechnungspositionen in einer eigenen Tabelle gespeichert, wobei jede Belegposition ebenfalls mit einem Datensatz aus der Belegtabelle, hier **tblVouchers**, verknüpft ist.

Die Tabelle zum Speichern dieser Daten heißt **tblVoucherItems** und sieht mit Beispieldaten wie in Bild 6 aus. Neben

tblVoucherItems

VoucherItemID	VoucherID	Amount	TaxAmount	TaxRatePercen	CategoryID
2	12345	119,00 €	19,00 €	19,00%	Einnahmen
4	23456	99,00 €	15,81 €	19,00%	Einnahmen
5	23456	138,00 €	9,03 €	7,00%	Einnahmen
6	34567	100,00 €	19,00 €	19,00%	Einnahmen
(Neu)		0,00 €	0,00 €	0,00%	

Datensatz: 5 von 5 Kein Filter Suchen

Bild 6: Tabelle mit den Positionen eines Belegs

tblCategories

CategoryGUID	Category	Type	ContactR	SplitAllo	GroupName
00d8f5f6-1d8a-40da-96ee-c53b25f37503	Werbung §13b	outgo	☐	☐	Werbung
01d94be1-ba15-4dd8-a8e2-e1d51eaab8f9	Repräsentationskosten	outgo	☐	☑	Werbung
097c0c6f-8fb0-4277-9d4e-f36bb91a0737	Tilgung Gesellschafter	income	☐	☑	Darlehen
09b906eb-8e05-4ee7-88a5-8e49e6c4db72	Geschenke	outgo	☐	☑	Beschränkt abziehbare Betri
09e90a1b-1aad-43b8-9803-b6fadffaf193	Fortbildung §13b	outgo	☐	☐	Fortbildung
113476af-ac83-4bf2-941d-0daa37022945	Fremdleistungen §13b	outgo	☐	☐	Fremdleistungen
121188e5-61ec-4734-bcc2-282e4340bb33	Werkzeuge	outgo	☐	☐	Anlagevermögen
13efe6dc-c3a3-4d6b-a793-81a1c0d78ece	LKW	income	☐	☑	Anlagevermögen
16d04a20-fd91-11e1-a21f-0800200c9a66	Anschaffungen	outgo	☐	☑	Sonstige Ausgaben
16d04a21-fd91-11e1-a21f-0800200c9a66	Bürobedarf	outgo	☐	☑	Sonstige Ausgaben
16d04a22-fd91-11e1-a21f-0800200c9a66	Girokontozinsen	outgo	☐	☑	Zinsen/Gebühren
16d04a23-fd91-11e1-a21f-0800200c9a66	Kontoführung/Kartengebühr	outgo	☐	☑	Zinsen/Gebühren
16d04a24-fd91-11e1-a21f-0800200c9a66	Porto	outgo	☐	☑	Sonstige Ausgaben
16d04a25-fd91-11e1-a21f-0800200c9a66	Privatentnahmen	outgo	☐	☑	Privat
16d04a26-fd91-11e1-a21f-0800200c9a66	Reinigung/Reinigungsmittel	outgo	☐	☑	Sonstige Ausgaben
16d04a27-fd91-11e1-a21f-0800200c9a66	Seminar/Weiterbildung	outgo	☐	☑	Fortbildung

Datensatz: 1 von 229 Kein Filter Suchen

Bild 7: Einträge der Tabelle **tblCategories**

dem Primärschlüsselfeld enthält sie ein Fremdschlüsselfeld, über das die Positionen mit einem der Belege aus der Tabelle **tblVouchers** verknüpft werden. In dieser Tabelle speichern wir den Betrag der Position (**Amount**), die Mehrwertsteuer zu dieser Position (**TaxAmount**) und den Steuersatz (**TaxRatePercentage**).

Hier ist anzumerken, dass das Feld **Amount** sowohl den Netto- als auch den Bruttobetrag enthalten kann. Welcher eingetragen werden muss, hängt davon ab, welchen Wert das Feld **TaxTypeID** der Tabelle **tblVouchers** für diesen Beleg aufweist – mehr dazu weiter unten in der Beschreibung der Formulare.

Belegkategorien

lexoffice bietet eine ganze Reihe von Kategorien an, welche im Grunde Vereinfachungen für die üblicherweise verwendeten Buchungskonten sind (zum Beispiel 8300 für Erlöse mit 7% Mehrwertsteuer). Diese weisen wir einer Belegposition über das Fremdschlüsselfeld **CategoryID** zu, das wir als Nachschlagefeld auslegen. Dieses Feld ist mit der Tabelle **tblCategories** verknüpft. Diese Tabelle sieht wie in Bild 7 aus.

Diese Tabelle haben wir nicht von Hand gefüllt, sondern wir haben die Daten per Rest API von lexoffice eingelesen. Und das wird auch unsere Programmierübung zum Aufwärmen sein.

Kategorien von lexoffice einlesen

Die Kategorien können wir automatisiert von lexoffice einlesen. Dazu nutzen wir die Prozedur aus Listing 1. Die Prozedur heißt **GetCategories** und sie verwendet den folgenden Endpunkt der Rest API von lexoffice:

```
https://api.lexoffice.io/v1/posting-categories
```

Dies liefert ohne weitere Angaben alle Kategorien zurück. Dazu rufen wir die Funktion **Request** auf, die wir bereits im oben erwähnten Beitrag ausführlich beschrieben haben Diese Funktion führt den eigentliche Request aus und liefert mit dem Parameter **strResponse** das Ergebnis zurück. Mit der Funktion **ParseJson**, die wir im Beitrag **JSON-Daten auslesen** (**www.access-im-unternehmen.de/1403**) beschrieben haben, übertragen wir die Inhalte des JSON-Dokuments in eine für uns besser verarbeitbare Form.

Bevor wir diese verarbeiten, löschen wir alle vorhandenen Einträge der Tabelle **tblCategories**. Anschließend durchlaufen wir bereits in einer **For...Next**-Schleife alle Elemente des Objekts **objJSON**. Darin legen wir jeweils einen

```
Public Sub GetCategories()
    Dim db As DAO.Database
    Dim strRequest As String
    Dim strURL As String
    Dim strResponse As String
    Dim objJSON As Object
    Dim i As Integer
    Dim strCategoryGUID As String
    Dim strCategory As String
    Dim strType As String
    Dim strContactRequired As String
    Dim strSplitAllowed As String
    Dim strGroupName As String
    Set db = CurrentDb
    strURL = "https://api.lexoffice.io/v1/posting-categories"
    If Request(strURL, "GET", strRequest, strResponse, "", "", 0) = True Then
        Set objJSON = ParseJson(strResponse)
        db.Execute "DELETE FROM tblCategories", dbFailOnError
        For i = 1 To objJSON.Count
            strCategoryGUID = objJSON.Item(i).Item("id")
            strCategory = objJSON.Item(i).Item("name")
            strType = objJSON.Item(i).Item("type")
            strContactRequired = IIf(objJSON.Item(i).Item("contactRequired") = "Wahr", -1, 0)
            strSplitAllowed = IIf(objJSON.Item(i).Item("splitAllowed") = "Wahr", -1, 0)
            strGroupName = objJSON.Item(i).Item("groupName")
            db.Execute "INSERT INTO tblCategories(CategoryGUID, Category, Type, ContactRequired, SplitAllowed, " _
                & "GroupName) VALUES('" & strCategoryGUID & "', '" & strCategory & "', '" & strType & "', " _
                & strContactRequired & ", " & strSplitAllowed & ", '" & strGroupName & "')", dbFailOnError
        Next i
    End If
End Sub
```

Listing 1: Diese Prozedur liest die Kategorien von lexoffice ein.

neuen Datensatz in der Tabelle **tblCategories** an, wobei wir die Informationen zuvor über das in **objJSON** geschaffene Objektmodell auslesen.

Die Grundlagen dafür beschreiben wir ebenfalls im Beitrag **JSON-Daten auslesen** (**www.access-im-unternehmen.de/1403**).

Formulare zur Eingabe der Belege und Belegpositionen

Bevor wir die Prozeduren zum Übertragen von Belegen an lexoffice erstellen, wollen wir ein Formular schaffen, mit dem wir die zu übertragenden Daten komfortabel eingeben können. Dazu erstellen wir zunächst ein Unterformular namens **sfmVouchers**. Diesem weisen wir über

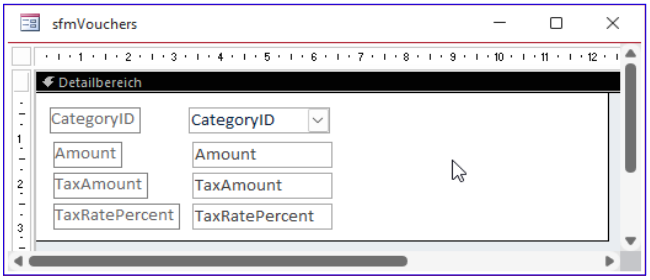


Bild 8: Das Unterformular **sfmVouchers**

Aktuelle Datenbankversion ermitteln

Es gibt verschiedene Gründe, um Kopien einer Datenbank anzulegen. Das Herstellen einer Sicherungskopie ist wohl der am meisten verbreitete Grund. Das ist sinnvoll, aber es kann dabei zu Problemen kommen, wenn man nicht achtsam ist: Dann arbeitet man auf einmal in der Sicherheitskopie weiter und wundert sich, wenn man anschließend die Originaldatenbank öffnet und die Funktionen, die man neu hinzuprogrammiert hat, nicht mehr findet. Oder man testet mit einer Datenbankdatei auf einer virtuellen Maschine und fügt dort Anpassungen hinzu. Auch hier kann es zu einem ähnlichen Durcheinander kommen. Um in einem solchen Fall die aktuelle Version zu finden, reicht es nicht, sich das Änderungsdatum der Datei anzusehen. Warum das nicht reicht und wie wir die aktuellere Datenbank zuverlässig finden, zeigen wir in diesem Beitrag.

Die Ausgangssituation lässt sich auch wie folgt beschreiben: Uns liegen zwei Datenbankdateien gleichen Namens vor, von denen wir nicht mehr wissen, welche den aktuellsten Bearbeitungsstand aufweist. Und leider haben wir auch noch soeben beide Versionen geöffnet, um zu schauen, ob wir so herausfinden können, welche die aktuelle Version ist. Damit haben wir einen wichtigen Marker zerstört – nämlich das Änderungsdatum der Datei.

Daran hätten wir zumindest erkennen können, welche Version wir zuletzt geöffnet hatten. Allerdings wird bereits beim Öffnen einer Access-Anwendung das Änderungsdatum auf das aktuelle Datum eingestellt – wir brauchen dazu gar keine Änderungen am Entwurf oder an den Daten vorzunehmen.

Nun haben wir also zwei verschiedene Versionen einer Datenbank vorliegen und benötigen Kriterien, um herauszufinden, an welcher wir zuletzt gearbeitet haben.

Diese lauten beispielsweise wie folgt:

- Gibt es in einer der beiden Datenbanken neue Objekte, die es in der anderen Datenbank nicht gibt?

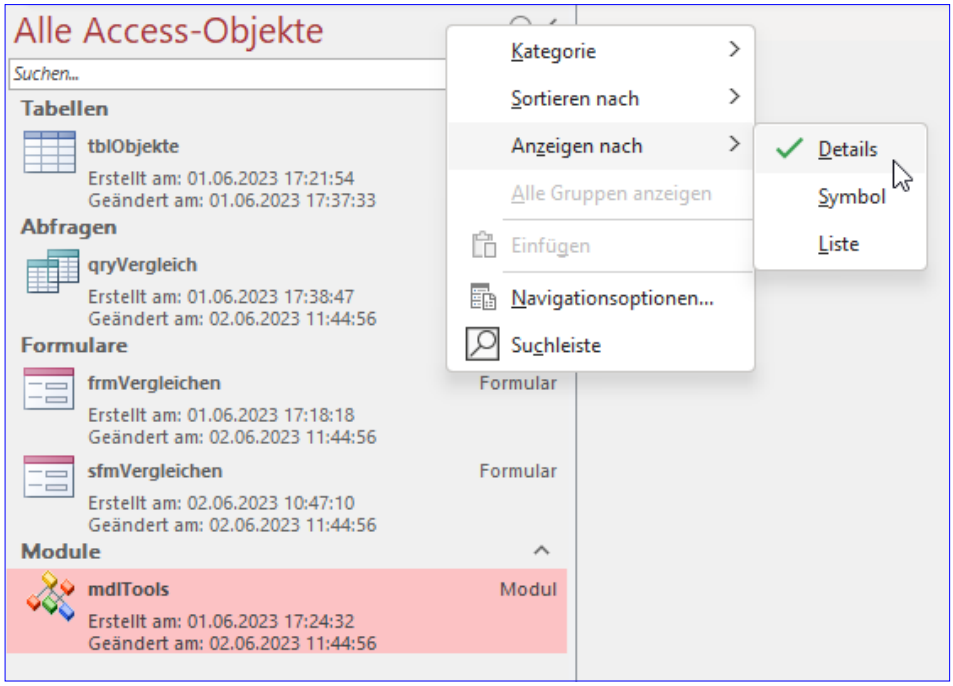


Bild 1: Änderungszeitpunkt von Objekten ermitteln

- Wurden aus einer der Datenbanken vielleicht sogar Objekte gelöscht?
- Wie lautet das letzte Bearbeitungsdatum der Objekte der Datenbank?

Wenn wir diese Kriterien untersuchen, sollten wir annähernd herausfinden können, welche der beiden Datenbankdateien den aktuelleren Stand aufweist. Das ist zumindest dann der Fall, wenn wir nur an einer der beiden Änderungen seit einem bestimmten Zeitpunkt vorgenommen haben.

Wenn wir die Datenbanken abwechselnd geöffnet und Änderungen an verschiedenen (oder sogar den gleichen) Stellen vorgenommen haben, ist das zwar unangenehm, weil wir die Änderungen dann zusammenführen müssen. Wir wissen dann aber zumindest, warum bestimmte Änderungen, von denen wir sicher waren, dass wir sie durchgeführt haben, nicht mehr vorhanden sind.

Änderungszeitpunkt von Objekten ermitteln

Den Änderungszeitpunkt eines Objekts können wir neben dem Erstellungszeitpunkt über die Benutzeroberfläche von Access ermitteln. Dazu klicken wir mit der rechten Maustaste in die Titelleiste des Navigationsbereichs von Access und wählen dort den Eintrag **Anzeigen nachlDetails** aus. Danach sehen wir die gewünschten Informationen zu jedem Eintrag (siehe Bild 1).

Wenn wir Datenbanken mit so wenigen Objekten wie die aus dem Screenshot vergleichen wollen, brauchen wir dafür keine eigene Anwendung zu entwickeln. Aber in der Regel ist die Anzahl der enthaltenen Objekte größer und ein manueller Abgleich macht schnell keinen Spaß mehr.

Also machen wir uns auf die Suche nach dem Speicherort dieser Informationen und stoßen dabei schnell auf die Systemtabelle **MSysObjects**. Systemtabellen wie diese sind standardmäßig ausgeblendet. Wir holen diese hervor, indem wir wieder mit der rechten Maustaste auf die Titelleiste des Navigationsbereichs klicken und nun den Eintrag **Navigationsoptionen...** auswählen. Es erscheint der Dialog **Navigationsoptionen**, wo wir im Bereich **Anzeigeoptionen** die Optionen **Ausgeblendete Objekte anzeigen** und **Systemobjekte anzeigen** aktivieren.

Diese Tabelle liefert für alle Datenbankobjekte wie Tabellen, Abfragen, Formulare, Berichte, Makros und VBA-Module jeweils einen Eintrag (und für einige weitere Objekttypen, die hier nicht von Interesse sind). Dieser enthält neben dem Objekttyp und dem Namen auch zwei Felder namens **DateCreated** und **DateUpdated**. Damit haben wir bereits alle Informationen, die wir benötigen. Wir müssen sie nur noch zusammenführen und abgleichen.

Objektdaten in neue Datenbank importieren

Dazu erstellen wir eine neue Datenbank namens **AktuellereDatenbankFinden.accdb**. In diese wollen wir zuerst die Tabelle **MSysObjects** der beiden Versionen der Datenbank importieren. Dazu benötigen wir bereits ein Formular, denn wir wollen die Auswahl der Datenbankdateien so komfortabel wie möglich gestalten – in diesem Fall mit entsprechenden Dateiauswahl-Dialogen.

Nach der Auswahl der abzugleichenden Datenbanken wollen wir mit einer Schaltfläche einen Vorgang starten, bei dem die relevanten Daten für alle in den beiden Datenbanken enthaltenen Objekte in eine zusammenfassende Tabelle geschrieben werden.

Diese heißt **tblObjekte** und ihr Entwurf sieht wie in Bild 2 aus. Neben dem Objektnamen speichern wir darin den Objekttyp sowie das Änderungsdatum für das Objekt in der ersten und in der zweiten Datenbank. Wenn das Objekt nur in einer der beiden Datenbanken vorhanden ist, schreiben wir das Datum nur in das entsprechende Feld.

Formular zum Anzeigen der Änderungsdaten

Das Formular zur Auswahl der zu untersuchenden Dateien und zum Anzeigen der relevanten Daten legen wir unter dem Namen **frmVergleichen** an. Diesem fügen wir zu-

nächst zwei Textfelder namens **txtVersion1** und **txtVersion2** hinzu. Daneben platzieren wir zwei Schaltflächen.

Diese statten wir jeweils mit einem Ordner-Icon aus und stellen die Eigenschaften **Hintergrundart** und **Rahmenart** auf **Transparent** ein. Das Formular sieht anschließend wie in Bild 3 aus.

Für die Ereignisprozeduren der beiden Schaltflächen, die durch das Ereignis **Beim Klicken** ausgelöst werden, hinterlegen wir die folgenden Prozeduren:

```
Private Sub cmdDateiOeffnen1_Click()
    Me!txtVersion1 = DateiOeffnen
End Sub

Private Sub cmdDateiOeffnen2_Click()
    Me!txtVersion2 = DateiOeffnen
End Sub
```

Beide verwenden die gleiche Funktion, um den Pfad zu der zu untersuchenden Datei zu ermitteln:

```
Private Function DateiOeffnen() As String
    Dim objFiledialog As Office.FileDialog
    Dim varFilename As Variant
    Set objFiledialog = _
        Application.FileDialog(msoFileDialogFilePicker)
    objFiledialog.InitialFileName = CurrentProject.Path
    If objFiledialog.Show = True Then
        DateiOeffnen = objFiledialog.SelectedItems(1)
    End If
End Function
```

Um das **FileDialog**-Objekt zu nutzen, benötigen wir einen Verweis auf die Bibliothek **Microsoft Office 16.0 Object Library**, den wir mit dem **Verweise**-Dialog aus Bild 4 hinzufügen (VBA-Editor, Menüeintrag **Extras|Verweise**). Für die beiden Textfelder stellen wir die Eigenschaft

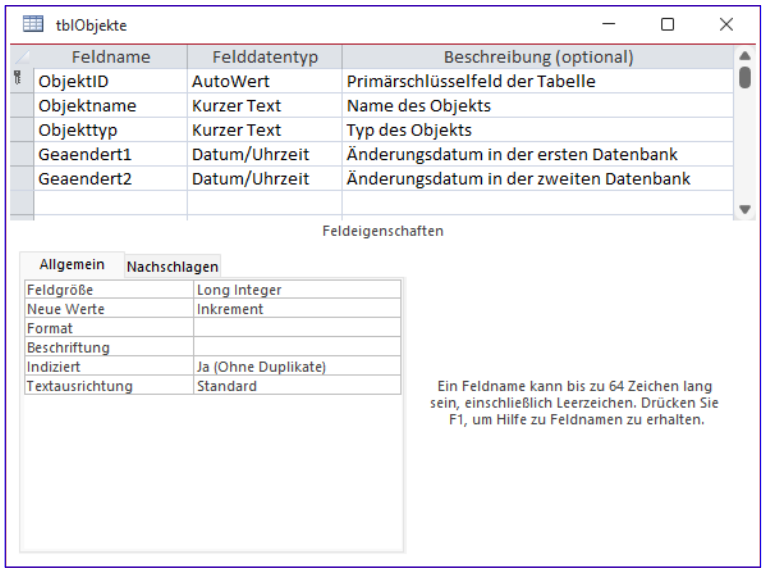


Bild 2: Entwurf der Tabelle zum Speichern der Änderungsdaten der Objekte

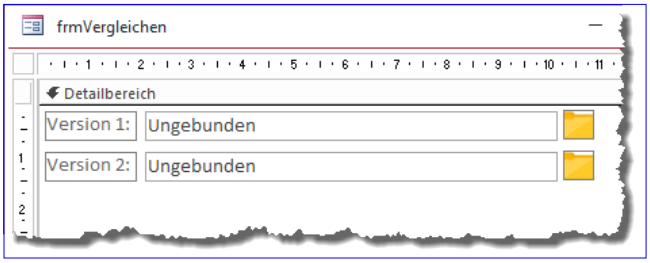


Bild 3: Felder zum Auswählen der zu vergleichenden Dateien

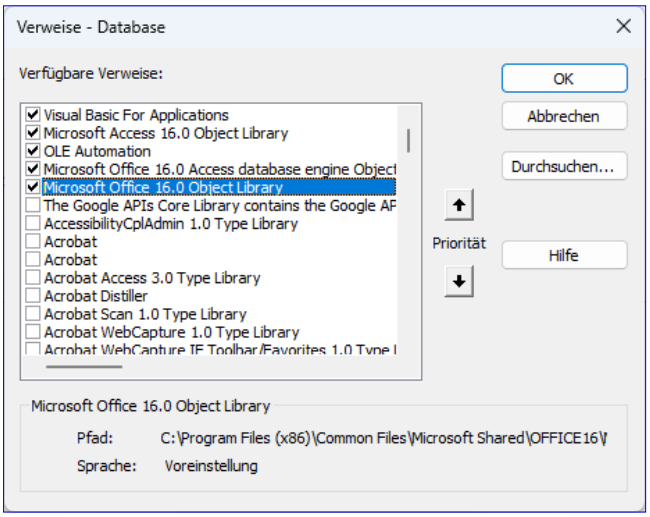


Bild 4: Hinzufügen eines Verweises auf die Office-Bibliothek

Horizontaler Anker auf **Beide** ein, damit sich diese an die Formularbreite anpassen.

Unterformular zum Anpassen der Ergebnisse

Bevor wir die Funktion zum Abgleichen der Unterschiede anlegen, fügen wir dem Formular ein Unterformular zum Anzeigen der Ergebnisse hinzu. Das Unterformular heißt **sfmVergleichen** und verwendet die Tabelle **tblObjekte** als Datensatzquelle. Wir fügen alle Felder dieser Tabelle zum Formular hinzu und stellen seine Eigenschaft **Standardansicht** auf **Datenblatt** ein (siehe Bild 5).

Danach fügen wir das Formular samt einer weiteren Schaltfläche namens **cmdVergleichen** zum Hauptformular hinzu (siehe Bild 6). Für das hinzugefügte Unterformular stellen wir die beiden Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** auf **Beide** ein.

Einlesen der Objektdaten der Datenbanken

Im nächsten Schritt fügen wir der Schaltfläche **cmdVergleichen** eine Prozedur für das Ereignis **Beim Klicken** hinzu (siehe Listing 1).

Hier führen wir zuerst einige Aufräumarbeiten durch. Wir leeren die Tabelle **tblObjekte**, damit keine Daten aus vorherigen Untersuchungen mehr in der Tabelle sind. Außerdem löscht die Prozedur zwei Tabellen namens **tblObjects1** und **tblObjects2**, die wir bisher noch gar nicht angelegt haben.

Genau aus diesem Grund erledigen wir diese Aufgabe auch bei deaktivierter eingebauter Fehlerbehandlung – der Versuch, eine nicht vorhandene Tabelle zu löschen, würde sonst einen Fehler auslösen.

Was aber sind das überhaupt für Tabellen? In diese kopieren wir die Daten der Systemtabellen namens **MSysObjects** der zu untersuchenden Datenbanken. Das erledigen wir mit der Methode **TransferDatabase** der **DoCmd**-Klasse. Damit diese einen Import durchführt, übergeben wir als ersten Parameter den Wert **acImport**. Der zweite

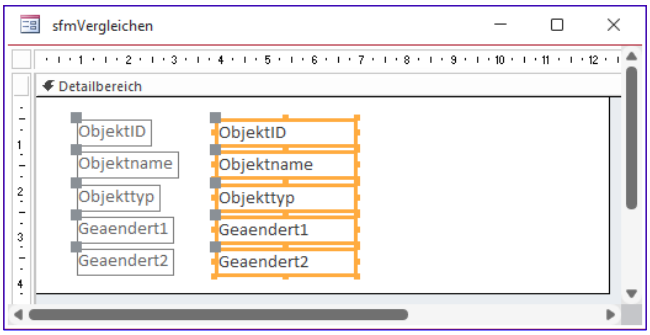


Bild 5: Unterformular der Anwendung

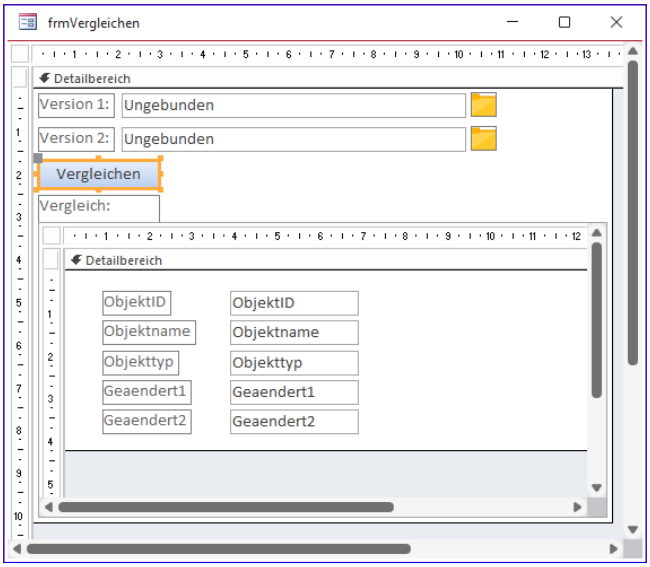


Bild 6: Das Hauptformular der Anwendung in der Entwurfsansicht

Parameter gibt die Herkunftsanwendung der Daten an, der dritte den Typ des einzulesenden Objekts. Die letzten beiden Parameter legen fest, dass die Daten der Tabelle **MSysObjects** in die Tabelle **tblObjects1** beziehungsweise **tblObjects2** eingelesen werden sollen. Dieser Befehl erstellt die Tabellen auf Basis der Quelltabelle neu.

Abgleich der Änderungsdaten

Die folgenden Anweisungen legen die für den Abgleich der Änderungsdaten notwendigen Daten in der Tabelle **tblObjekte** an. Dabei wollen wir nur die Datensätze der Tabelle **MSysObjects** berücksichtigen, die eines der zu untersuchenden Objekte betreffen. Außerdem wollen wir Systemtabellen und temporäre Objekte oder im Hintergrund verwendete Objekte nicht berücksichtigen.