AUSGABE 03/2026

Access, SQL und Cloud AUTOMATION

MAGAZIN FÜR DIE PROGRAMMIERUNG VON MICROSOFT ACCESS, SQL SERVER UND CLOUD-AUTOMATIONEN MIT VBA UND CO.

IN DIESEM HEFT:

André Minhorst Verlag

Nutze die Access-KI!

Es gibt eine spannende Neuerung in unserer Lernplattform. Du findest dort nicht nur alle Artikel inklusive Beispieldownloads und eine sehr gute Suchfunktion sowie ein Forum, wo Du mit uns und den anderen Lesern diskutieren kannst. Außerdem kannst Du ab sofort unsere eigene KI nutzen, die spannende Funktionen bietet!

Wenn Du noch nicht für die Lernplattform registriert bist, kannst Du das hier erledigen:

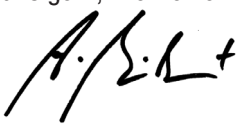
`https://andreminhorst.de/anmeldung-an-learningsuite`

Nach der Anmeldung siehst Du rechts unten in der Lernplattform einen Button mit Frage- und Ausrufezeichen. Klickst Du diesen an, erscheint unser KI-Assistent.

Der Clou: Er beantwortet Dir nicht nur Deine Fragen rund um Access, VBA und SQL Server, sondern verweist Dich auch noch auf die passenden Artikel, wo Du Dein Wissen vertiefen kannst.

Denn bedenke: Du solltest immer noch verstehen können, was Du mit der Unterstützung von KI programmierst oder Dir programmieren lässt. Daher kann Grundlagenwissen nicht schaen, und genau das liefert Dir unser KI-Assistent.

Viel Spaß beim Experimentieren mit unsere KI! Schreib uns gern, wie Deine Erfahrungen damit sind.



André Minhorst

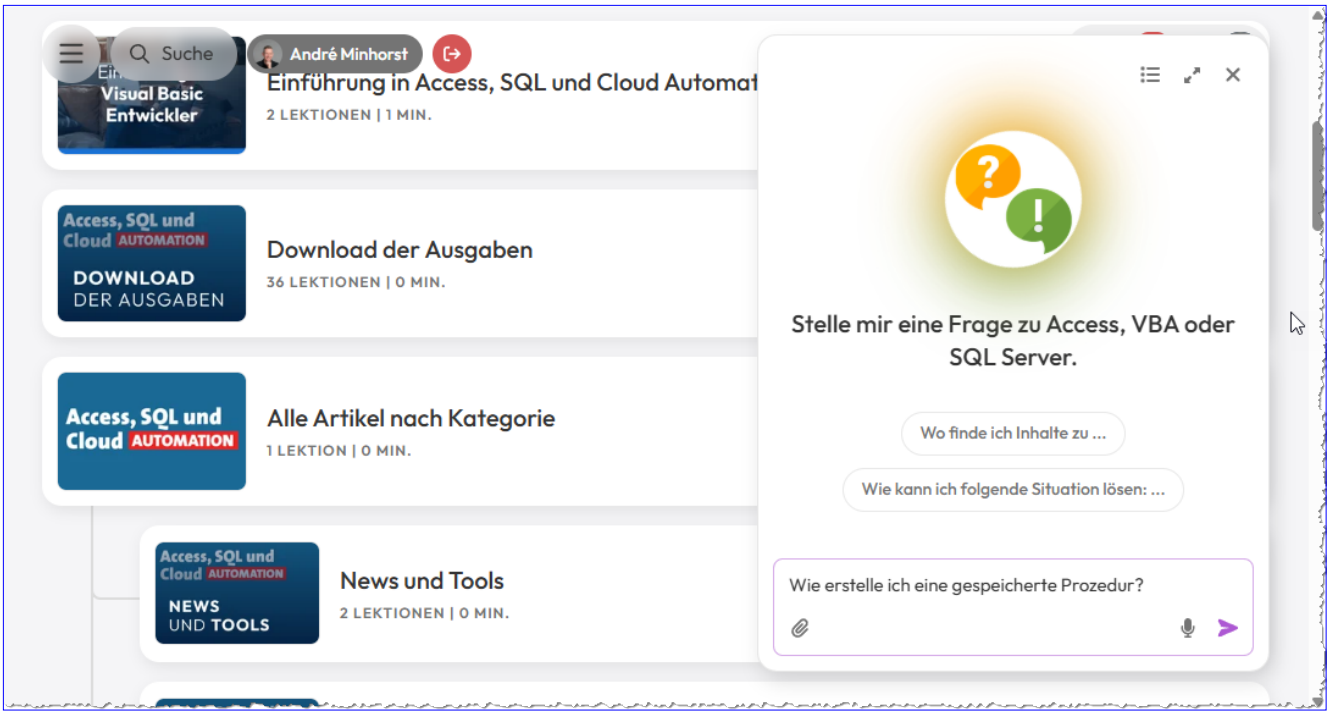


Bild 1: Unser eigener KI-Assistent

Fehlerbehandlung in VBA: Patterns für die Praxis

Eine Prozedur, die monatelang unauffällig ihren Dienst getan hat, bricht plötzlich mit einer kryptischen Meldung ab – und der Anwender steht vor einem Dialog, mit dem er nichts anfangen kann. Der Grund ist fast immer derselbe: Es gibt keine oder nur eine halbherzige Fehlerbehandlung. Dabei ist »On Error« eines der mächtigsten Werkzeuge in VBA, wenn man weiß, wie man es einsetzt. Wir schauen uns die Muster an, die aus einer fragilen Lösung eine robuste machen: wie wir Fehler nicht nur abfangen, sondern strukturiert behandeln, Ressourcen zuverlässig freigeben, Fehler zwischen den Schichten einer Anwendung weiterreichen und sie in einer Tabelle protokollieren. Die Beispiele findest Du im Modul »mdlFehlerbehandlung« der Beispieldatenbank. Wie das im Einzelnen gelingt, zeigt der vorliegende Beitrag.

Was passiert ohne Fehlerbehandlung?

Schauen wir uns zuerst an, was VBA von sich aus tut, wenn ein Laufzeitfehler auftritt.

Die Prozedur **DateiOhneSchutz** aus folgendem Listing 1 öffnet eine Textdatei, die es gar nicht gibt:

```
Public Sub DateiOhneSchutz()
    Dim intFile As Integer
    intFile = FreeFile
    Open "C:\NichtVorhanden.txt" _
        For Input As #intFile
    Close #intFile
End Sub
```

Existiert die Datei nicht, bricht VBA mit dem Laufzeitfehler **53** ab und zeigt den Dialog aus Bild 1 – mit einer Meldung, die dem Anwender nichts sagt, und mit den Schaltflächen **Beenden** und **Debuggen**, die in einer ausgelieferten Anwendung beide nichts zu suchen haben.

Klicken wir auf **Debuggen**, sehen wir die farbig hinterlegte fehlerhafte Anweisung (siehe Bild 2).

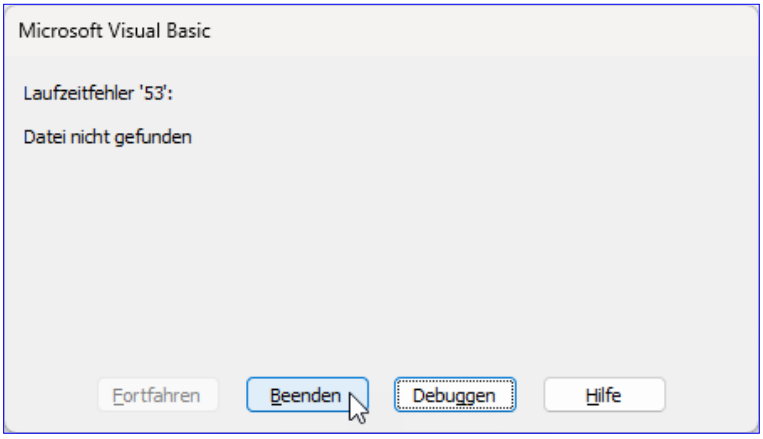


Bild 1: Der Standarddialog von VBA beim Laufzeitfehler 53 – so etwas darf der Anwender nie zu sehen bekommen

Zwei Dinge sind hier schiefgelaufen: Der Anwender bekommt keine verständliche Meldung, und die Prozedur bricht mitten im Ablauf ab. In anderen Prozeduren

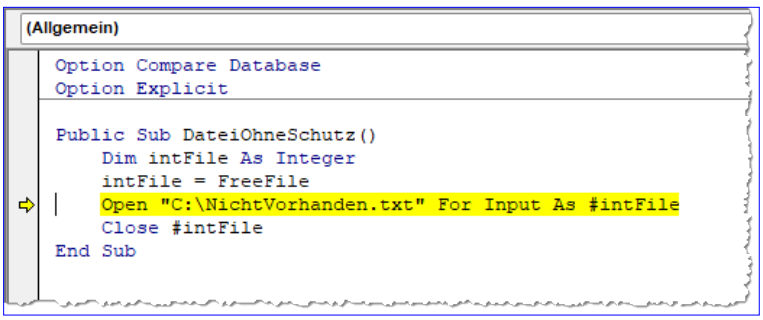


Bild 2: Anschließend wird die fehlerhafte Zeile farbig hinterlegt.

können dadurch bereits geöffnete Dateien, Recordsets oder andere Ressourcen unaufgeräumt zurückbleiben.

Die drei Formen von On Error

VBA kennt genau drei Varianten der **On Error**-Anweisung, und sie bestimmen, was bei einem Laufzeitfehler geschieht:

- **On Error GoTo Marke** springt zur angegebenen Sprungmarke. Das ist die Standardform für eine strukturierte Fehlerbehandlung.
- **On Error Resume Next** ignoriert den Fehler und macht mit der nächsten Anweisung weiter. Das ist nützlich, wenn wir den Fehler unmittelbar danach selbst prüfen wollen.
- **On Error GoTo 0** schaltet jede aktive Fehlerbehandlung in der aktuellen Prozedur ab. Ein Fehler führt danach wieder zum Standarddialog.

Ein Punkt, der später noch wichtig wird: **On Error** gilt immer nur innerhalb der Prozedur, in der es steht. Eine globale Fehlerbehandlung gibt es in VBA nicht – jede Prozedur muss sich selbst schützen.

Das Standard-Pattern: On Error GoTo

Das bewährteste Muster sieht so aus: Am Anfang der Prozedur leiten wir auftretende Fehler auf eine Sprungmarke um, am Ende springen wir über den Fehlerblock hinweg. Bauen wir unser Beispiel entsprechend um – das Ergebnis heißt **DateiMitSchutz** und steht in folgendem Listing:

```
Public Sub DateiMitSchutz()
    On Error GoTo ErrHandler
    Dim intFile As Integer
    intFile = FreeFile
    Open "C:\NichtVorhanden.txt" For Input As #intFile
    ' ... Datei verarbeiten ...
    Close #intFile
ExitHere:
    Exit Sub
ErrHandler:
    MsgBox "Fehler " & Err.Number & ": " _
        & Err.Description, vbCritical, "DateiMitSchutz"
    Resume ExitHere
End Sub
```

Der Ablauf im Einzelnen: **On Error GoTo ErrHandler** aktiviert die Fehlerbehandlung – ab hier springt VBA bei einem Laufzeitfehler zur Marke **ErrHandler**.

Läuft alles glatt, erreichen wir **ExitHere** und verlassen die Prozedur mit **Exit Sub**. Tritt dagegen ein Fehler auf, landen wir im **ErrHandler**, zeigen eine Meldung an und springen mit **Resume ExitHere** zur Austrittsmarke.

Warum eigentlich **Resume ExitHere** und nicht einfach **GoTo ExitHere**? Der Unterschied sieht nach Geschmackssache aus, ist aber entscheidend: **Resume** meldet VBA, dass der Fehler behandelt wurde. Erst danach kann überhaupt wieder ein Fehler abgefangen werden. Mit **GoTo** bleibt die Prozedur dagegen im Fehlerzustand.

Den Fehlerzustand sichtbar machen

Das ist eine Behauptung – schauen wir sie uns an, statt sie zu glauben. Die Prozedur **GoToStattResume** in macht den Unterschied im Direktbereich sichtbar:

```
Public Sub GoToStattResume()
    On Error GoTo ErrHandler
    Err.Raise 53
ExitHere:
    Debug.Print "In ExitHere: Err.Number = " & Err.Number
    Exit Sub
ErrHandler:
    Debug.Print "Im Handler:  Err.Number = " & Err.Number
    GoTo ExitHere
End Sub
```

Rufe die Prozedur im Direktbereich mit **GoToStattResume** auf. Die Ausgabe zeigt in beiden Zeilen die **53** – der Fehler ist in **ExitHere** also immer noch aktiv (siehe Bild 3).

Tauschst Du das **GoTo ExitHere** gegen **Resume ExitHere**, steht in der zweiten Zeile eine **0**: Der Fehler gilt als erledigt.

Und genau daran hängt die Praxisfolge – ein weiterer Fehler in der Aufräumlogik würde in der **GoTo**-Variante nicht mehr abgefangen, sondern führte direkt zum Standarddialog.

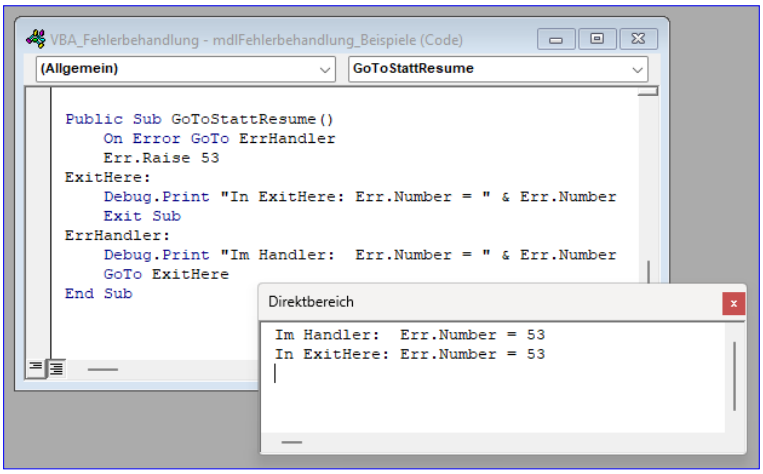


Bild 3: Der Direktbereich zeigt es schwarz auf weiß: Nach **GoTo** steht **Err.Number** immer noch auf **53**

Das Err-Objekt im Detail

Das **Err**-Objekt ist immer verfügbar und beschreibt den zuletzt aufgetretenen Fehler. Wichtig sind drei Eigenschaften:

- **Number** ist die Fehlernummer, wobei **0** für „kein Fehler“ steht. Für eigene Fehler nutzen wir Nummern ab **vbObjectError + 512** – dazu kommen wir gleich.
- **Description** liefert die Fehlerbeschreibung im Klartext.
- **Source** nennt die Quelle: standardmäßig den Namen des VBA-Projekts, bei externen Bibliotheken deren Namen, etwa **DAO.Database** oder **ADODB.Connection**.

Dazu kommen zwei Methoden. **Clear** setzt **Number** auf **0** und leert die übrigen Eigenschaften des **Err**-Objekts. Beim Verlassen einer Prozedur und durch eine **Resume**-Anweisung wird der Fehlerzustand ebenfalls zurückgesetzt. Wenn wir mehrere Operationen unter **On Error Resume Next** prüfen, sollten wir **Err.Clear** dagegen ausdrücklich zwischen den Operationen aufrufen.

Und **Raise** löst einen Laufzeitfehler aus – damit erzeugen wir gleich gezielt eigene Fehler.

Resume – drei Varianten

Aus dem Fehlerblock heraus haben wir drei Möglichkeiten, den Programmfluss fortzusetzen. **Resume** führt die fehlerauslösende Zeile erneut aus – sinnvoll nur dann, wenn wir die Ursache im Fehlerblock tatsächlich beseitigt haben.

Resume Next macht mit der Zeile danach weiter, wenn der Fehler tolerierbar ist. Und **Resume Marke** springt zu einer bestimmten Marke, was die häufigste Variante ist: Genau so kommen wir geordnet zur Austrittsmarke.

Wie Resume gezielt eingesetzt werden kann, zeigt das folgende Listing. Die Prozedur **OrdnerAnlegen** legt einen Ordner an, dessen übergeordneter Ordner womöglich noch fehlt – in dem Fall legen wir diesen an und versuchen es erneut:

```
Public Sub OrdnerAnlegen(strPfad As String)
    On Error GoTo ErrHandler
    MkDir strPfad
ExitHere:
    Exit Sub
```

```
ErrHandler:
    If Err.Number = 76 Then
        'Pfad nicht gefunden:
        'übergeordneten Ordner anlegen
        MkDir Left(strPfad, InStrRev(strPfad, "\") - 1)
        Resume
    Else
        MsgBox "Fehler " & Err.Number & ": " _
            & Err.Description
        Resume ExitHere
    End If
End Sub
```

Aufgerufen wird das etwa so:

```
OrdnerAnlegen "C:\Temp\Export\Kunden"
```

Zwei Stolpersteine stecken hier drin: Erstens legt die Korrektur nur eine Ebene an – fehlen zwei, scheitert schon das **MkDir** im Fehlerblock. Ein weiterer Fehler innerhalb eines bereits aktiven Fehlerhandlers kann von demselben Handler nicht erneut behandelt werden. Gibt es keinen übergeordneten aktiven Handler, erscheint der VBA-Standarddialog. Zweitens droht eine Endlosschleife, wenn die Korrektur das Problem gar nicht löst: **Resume** wiederholt dann immer wieder dieselbe Zeile. Wie wir das mit einem Zähler entschärfen, sehen wir weiter unten.

On Error Resume Next – gezielt und sparsam

On Error Resume Next hat einen schlechten Ruf, und der ist meistens verdient: Wer es als Allzweckwaffe einsetzt, verschluckt Fehler stillschweigend und macht die Anwendung unberechenbar. Es gibt aber Situationen, in denen es genau das richtige Werkzeug ist – nämlich dann, wenn wir den Fehlerstatus einer einzelnen Operation bewusst abfragen wollen.

Ein typischer Fall: Wir wollen wissen, ob ein Eintrag in einer **Collection** existiert. Eine **Exists**-Methode gibt es dort nicht, also nutzen wir den Zugriffsfehler als Prüfmittel. Das erledigt die Funktion **ExistiertInCollection**. Sie erwartet die zu prüfende **Collection** und den Schlüssel und liefert **True** oder **False** zurück:

```
Public Function ExistiertInCollection(col As Collection, _
        strKey As String) As Boolean
    Dim varDummy As Variant
    On Error Resume Next
    varDummy = col(strKey)
    ExistiertInCollection = (Err.Number = 0)
    On Error GoTo 0
End Function
```

Das Muster dahinter ist immer dasselbe: **On Error Resume Next** einschalten, die kritische Anweisung ausführen, **Err.Number** prüfen und sofort danach mit **On Error GoTo 0** zurücksetzen – oder, wenn die Prozedur eine eigene Fehlerbehandlung hat, mit **On Error GoTo ErrHandler** wieder auf die reguläre Marke umleiten. Ausprobieren lässt sich das wie folgt:

```
Dim col As New Collection
col.Add "Wert", "Schluessel"
Debug.Print ExistiertInCollection(col, "Schluessel")
'Liefert True
Debug.Print ExistiertInCollection(col, "Fehlt")
'Liefert False
```

Entscheidend ist die Disziplin: **On Error Resume Next** sollte nie länger als zwei oder drei Zeilen aktiv sein. Alles andere ist ein Wartungsrisiko.

Aufräumen bei Fehlern: das Exit-Pattern

In vielen Prozeduren müssen wir am Ende Ressourcen freigeben – Recordsets schließen, Objektvariablen auf **Nothing** setzen, Dateien schließen. Diese Arbeit darf der Fehlerblock nicht überspringen.

Deshalb bekommt die Prozedur eine gemeinsame Austrittsmarke, die sowohl der reguläre Ablauf als auch

```
Public Sub DatenExportieren(strPfad As String)
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim intFile As Integer
    Dim bolDateiOffen As Boolean
    On Error GoTo ErrHandler
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT KundenName FROM tblKunden", dbOpenSnapshot)

    intFile = FreeFile
    Open strPfad For Output As #intFile
    bolDateiOffen = True

    Do Until rst.EOF
        Print #intFile, rst!KundenName
        rst.MoveNext
    Loop

ExitHere:
    If bolDateiOffen Then
        Close #intFile
        bolDateiOffen = False
    End If

    If Not rst Is Nothing Then
        rst.Close
        Set rst = Nothing
    End If

    Set db = Nothing
    Exit Sub

ErrHandler:
    MsgBox "Fehler in DatenExportieren:" & vbCrLf _
        & Err.Number & " - " & Err.Description, _
        vbCritical
    Resume ExitHere
End Sub
```

Listing 1: Beispiel für das Exit-Pattern

der Fehlerblock anspringt. Wie das aussieht, zeigt die Prozedur **DatenExportieren** (siehe Listing 1). Sie liest die Tabelle **tblKunden** aus und schreibt die Namen in eine Textdatei.

Der Aufruf sieht so aus:

```
DatenExportieren "C:\Temp\Kunden.txt"
```

Beachte die Prüfung **If Not rst Is Nothing**: Der Fehler kann auch schon vor dem **OpenRecordset** auftreten – dann wäre **rst** noch **Nothing**, und ausgerechnet das Aufräumen würde einen neuen Fehler auslösen. Ob die Datei tatsächlich geöffnet wurde, halten wir in der Variablen **bolDateiOffen** fest. Eine von **FreeFile** gelieferte Dateinummer allein beweist noch nicht, dass die anschließende **Open**-Anweisung erfolgreich war.

Fehler weiterreichen mit Err.Raise

Nicht jeder Fehler gehört dort behandelt, wo er auftritt. In einer sauber aufgebauten Anwendung gibt es Schichten: Datenzugriff, Geschäftslogik und Benutzeroberfläche. Eine Datenzugriffsfunktion soll dem Aufrufer melden, dass etwas schiefgelaufen ist – aber keine **MsgBox** anzeigen. Dafür ist **Err.Raise** da.

Die Funktion **KundeNachID** liest zu einer Kundennummer den Namen aus **tblKunden**. Findet sie keinen Datensatz, löst sie selbst einen Fehler aus; tritt ein

VBA Basics: Auflistungen mit Dictionarys

Im Artikel »VBA Basics: Auflistungen mit Collections« (www.vbentwickler.de/514) haben wir die Collection kennengelernt – eine einfache, geordnete Liste ohne Verweis. Ihr grösstes Manko: Es gibt keine Möglichkeit zu prüfen, ob ein Schlüssel existiert. Genau hier setzt das Dictionary aus der Microsoft Scripting Runtime an. Es bietet Exists, erlaubt das direkte Ändern von Werten und liefert Schlüssel und Werte als Array – damit ist es die erste Wahl für Duplikatprüfungen, Häufigkeitszählungen und Lookup-Tabellen im Speicher. Wie wir es einrichten, was es kann und wo Stolpersteine lauern, zeigt dieser Beitrag.

Das Dictionary einrichten

Das **Dictionary** steckt in der Bibliothek **Microsoft Scripting Runtime** (**scrrun.dll**). Bevor wir es per Early Binding nutzen können, müssen wir im VBA-Editor über **Extras|Verweise** den Eintrag **Microsoft Scripting Runtime** aktivieren (siehe Bild 1).

Alternativ funktioniert Late Binding – dann entfällt der Verweis, aber auch IntelliSense:

```
' Early Binding (mit Verweis)
Dim dic As New Scripting.Dictionary

' Late Binding (ohne Verweis)
Dim dic As Object
Set dic = CreateObject("Scripting.Dictionary")
```

Für die Entwicklung empfehlen wir Early Binding wegen IntelliSense und Kompilierungsprüfung.

Die Member des Dictionary

Das **Dictionary** ist reichhaltiger als die **Collection**. Die wichtigsten Member:

- **Add Key, Item:** fügt ein Schlüssel-Wert-Paar hinzu. Beachte: Der Schlüssel steht an erster Stelle – umgekehrt als bei der **Collection**.
- **Item(Key):** der Zugriff über **dic(Key)** liefert den Wert zum Schlüssel. Wird einem nicht vorhandenen Schlüssel auf diese Weise ein Wert zugewiesen, wird der Eintrag automatisch angelegt.
- **Exists(Key):** liefert **True**, wenn der Schlüssel vorhanden ist, sonst **False**.
- **Keys:** liefert ein Array aller Schlüssel.
- **Items:** liefert ein Array aller Werte.
- **Count:** liefert die Anzahl der Einträge.

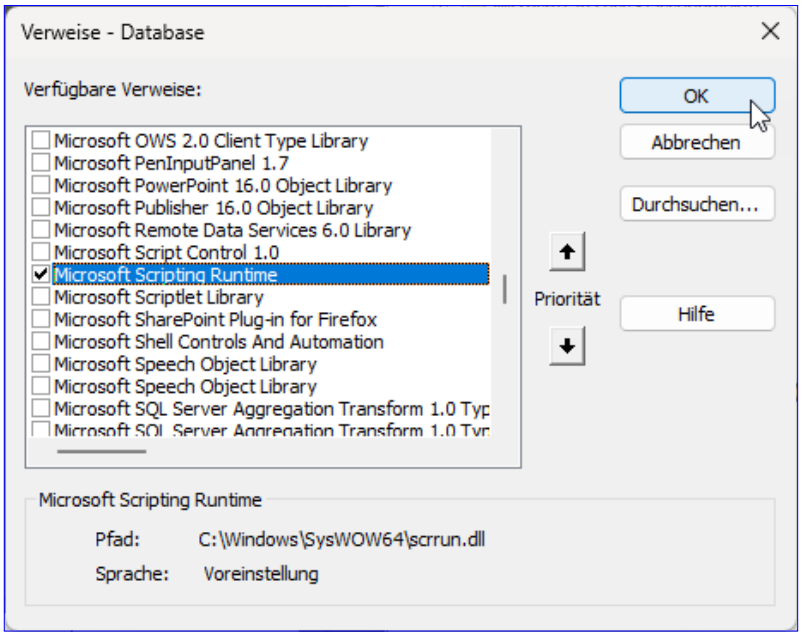


Bild 1: Verweis auf die Bibliothek **Microsoft Scripting Runtime**

- **Remove(Key):** entfernt einen einzelnen Eintrag.
- **RemoveAll:** leert das gesamte Dictionary.
- **CompareMode:** legt fest, ob Schlüssel binär (**vbBinaryCompare**) oder textuell (**vbTextCompare**) verglichen werden. Muss gesetzt werden, solange das Dictionary leer ist.
- **Key(Key):** ändert einen bestehenden Schlüssel, ohne den Wert zu berühren.

Das Dictionary in Aktion

Ein erstes Beispiel – wir speichern Bundesländer-Kürzel und greifen gezielt darauf zu.

```
Public Sub DictionaryDemo()

    Dim dic As New Scripting.Dictionary
    Dim varKey As Variant

    dic.CompareMode = vbTextCompare

    dic.Add "BE", "Berlin"
    dic.Add "HH", "Hamburg"
    dic.Add "BY", "München"

    ' Existenz pruefen
    If dic.Exists("HH") Then
        Debug.Print dic("HH")    ' -> Hamburg
    End If

    ' Wert direkt aendern
    dic("BY") = "Bayern (München)"

    ' Alle Eintraege ausgeben
    For Each varKey In dic.Keys
        Debug.Print varKey & ": " & dic(varKey)
    Next varKey

End Sub
```

Listing 1: Dictionary mit **Exists**, Wertänderung und **Keys**

In Listing 1 sehen wir gleich drei Vorteile gegenüber der **Collection**: **Exists** statt Fehler-Workaround, direktes Ändern eines Werts per Zuweisung und **dic.Keys** als Array für die Schleife.

CompareMode: Gross- und Kleinschreibung

Standardmässig vergleicht das **Dictionary** Schlüssel binär – **"hh"** und **"HH"** wären also zwei verschiedene Einträge. In den meisten Access-Szenarien wollen wir das nicht.

Deshalb setzen wir **CompareMode** auf **vbTextCompare**, damit Schlüssel ohne Berücksichtigung der Gross-/Kleinschreibung verglichen werden.

Wichtig: **CompareMode** lässt sich nur setzen, solange das Dictionary leer ist. Enthält es bereits Einträge, gibt es Laufzeitfehler **5**. Also immer direkt nach dem Erzeugen setzen.

Vorsicht: stille Anlage bei Item-Zugriff

Das **Dictionary** hat ein Verhalten, das regelmässig für Überraschungen sorgt: Greift man per **dic("XY")** auf einen Schlüssel zu, der nicht existiert, gibt es keinen Fehler.

Stattdessen legt das **Dictionary** den Schlüssel stillschweigend an – mit dem Wert **Empty**. Das ist beim Zählen nützlich (dazu gleich mehr), kann aber auch Geistereinträge erzeugen, wenn wir unvorsichtig sind.

Faustregel: Vor dem lesenden Zugriff immer **Exists** prüfen – oder die stille Anlage bewusst nutzen.

Timer() – Performance-Messungen in VBA

Wer VBA-Code optimieren möchte, braucht eine zuverlässige Methode zur Zeitmessung. Die VBA-Funktion Timer() liefert die verstrichenen Sekunden seit Mitternacht – und damit ein einfaches, aber wirkungsvolles Werkzeug, um die Laufzeit von Prozeduren zu messen und verschiedene Implementierungen zu vergleichen. In diesem Artikel zeigen wir, wie Du Timer() gezielt einsetzt und die Ergebnisse sinnvoll auswertest.

Beispieldatenbank

Die Beispieldatenbank zu diesem Artikel trägt den Namen **Timer.accdb**. Sie enthält ein Modul namens **mdlTimer** mit allen Prozeduren und Funktionen aus diesem Artikel sowie eine Tabelle namens **tblTestdaten** mit Testdatensätzen für die Performance-Vergleiche.

Die Timer()-Funktion

Die VBA-Funktion **Timer()** gibt einen **Single**-Wert zurück: die Anzahl der Sekunden seit Mitternacht. Auf Windows-Systemen liegt die Auflösung bei etwa zehn Millisekunden, was für die meisten Messungen ausreicht.

Die einfachste Art, eine Laufzeit zu messen, kommt ohne jede Hilfsfunktion aus. Man speichert den Wert von **Timer()** vor dem Code, führt den Code aus und gibt die Differenz danach direkt aus:

```
Dim sngStart As Single
sngStart = Timer()
'Hier kommt der zu messende Code
Debug.Print "Dauer: " & Timer() - sngStart
```

Das genügt für schnelle Stichproben im Direktbereich, aber auch, wenn man mal eben prüfen will, welche Prozedur im Ablauf etwa beim Öffnen eines Formulars wie viel Zeit in Anspruch nimmt.

Dann könnte man jede betroffene Prozedur mit diesen Anweisungen versehen – das Speichern des Ergebnisses von **Timer()** bei Start landet gleich mit der ersten Anweisung in **sngTimer**, und am Ende der Prozedur lassen wir das Ergebnis ausgeben, dann mit Angabe des Prozedurnamens.

So landen alle Ergebnisse übersichtlich im Direktbereich, wenn alle Prozeduren des Formulars ausgeführt worden sind. Außerdem kann man sich so einen Überblick verschaffen, welche Prozeduren überhaupt an einem solchen Vorgang beteiligt sind.

Der Sonderfall Mitternacht

Timer() setzt seinen Wert um Mitternacht auf null zurück. Läuft eine Messung über diesen Zeitpunkt hinweg, ergibt die Subtraktion einen negativen Wert. In der Praxis betrifft das nur sehr lang laufende Prozeduren. Wer auf Nummer sicher gehen will, prüft das Ergebnis auf ein negatives Vorzeichen und addiert dann **86400** (Sekunden pro Tag). Die Hilfsfunktion in Listing 2 erledigt das:

```
Public Function TimerDauer(sngStart As Single) _
        As Single
    Dim sngDauer As Single
    sngDauer = Timer() - sngStart
    If sngDauer < 0 Then
        sngDauer = sngDauer + 86400
    End If
    TimerDauer = sngDauer
End Function
```

Zwei Varianten vergleichen

Der eigentliche Nutzen von Zeitmessungen liegt im direkten Vergleich zweier Implementierungen. Als Bei-

ADODB: Gespeicherte Prozeduren ausführen

Recordsets sind zentrale Bestandteile beim Zugriff auf Daten mit ADODB. Sie bieten umfangreiche Funktionen zur Navigation, Bearbeitung, Filterung und Analyse von Daten. In diesem Artikel zeigen wir eine vollständige Übersicht aller Eigenschaften und Methoden des Recordset-Objekts und erläutern diese jeweils ausführlich mit praktischen Beispielen. Besonderes Augenmerk legen wir auf die Ereignisse eines Recordsetes. Im Gegensatz zum DAO-Recordset bietet das ADODB-Recordset nämlich die Möglichkeit, auf verschiedene Ereignisse zu reagieren – beispielsweise auf Änderungen im Datensatz.

Beispieldatenbank

Die Beispiele dieses Artikels findest Du in der Beispieldatenbank **ADODB_GespeicherteProzedurenAusfuehren.accdb**.

Anwendungszweck für die vorgestellten Beispiele

Die einfachste Möglichkeit, Daten aus Tabellen einer SQL Server-Datenbank in Access anzuzeigen und zu bearbeiten ist die Verwendung von Tabellenverknüpfungen per ODBC.

Diese kann man einfach als Datensatzquelle von Formularen, Berichten oder Steuerelementen angeben – genau wie bei herkömmlichen Access-Tabellen.

Manchmal ist jedoch eine ungebundene Darstellung gewünscht, beispielsweise weil man aus Sicherheitsgründen keine Tabellenverknüpfungen im Frontend verfügbar machen möchte, weil diese einfach ausgelesen werden können.

Für diesen Fall bietet es sich an, die Daten auf SQL Server-Seite per gespeicherter Prozedur bereitzustellen und diese per ADODB-Recordset in Formularen und Steuerelementen anzuzeigen.

Für die Anzeige etwa der Daten eines Mitarbeiters könnte man ein solches Recordset mit ADODB per Zugriff auf eine entsprechende gespeicherte Prozedur füllen und die Daten der einzelnen Felder in ungebundenen Steuerelementen anzeigen.

Möchte man diese wieder speichern, verwendet man eine entsprechende gespeicherte Prozedur, welche die per Parameter übergebenen Daten über eine **UPDATE**-Anweisung in der Tabelle im SQL Server speichert.

Das Gleiche gilt für neu angelegte Datensätze, wobei man hier eine **INSERT INTO**-Anweisung nutzt. Und mit **DELETE** können wir einen angegebenen Datensatz auch wieder löschen.

Die nachfolgenden Beispiele zeigen, wie die gespeicherten Prozeduren aufgebaut sein müssen und wie wir diese per VBA und ADODB nutzen können.

Beispieldatenbank auf dem SQL Server anlegen

Um die Beispiele dieses Artikels auszuprobieren, benötigst Du eine SQL-Server-Datenbank mit einigen Tabellen und den entsprechenden gespeicherten Prozeduren. Im Modul **mdlSQLServerDB** findest Du den Code, den Du im SQL Server Management Studio ausführen kannst, um die Tabellen mit Beispieldaten und die gespeicherten Prozeduren anlegen kannst.

Dazu gehst Du wie folgt vor:

- Lege im SQL Server eine neue Datenbank namens **Test_GespeicherteProzeduren** an.
- Öffne eine neue Abfrage im Kontext dieser Datenbank.
- Gehe in das Modul **mdlSQLServerDB** und entferne die Kommentarzeichen – am einfachsten durch Markieren des Inhalts und Auswahl des Kontextmenü-Eintrags Auskommentierung des Blocks aufheben (sollte dieser nicht sichtbar sein, blende mit **Ansicht|Symbolleisten|Bearbeiten** die entsprechende Symbolleiste ein).
- Kopiere den Code in das Abfragefenster im SQL Server Management Studio.
- Führe den Code mit **F5** aus. Dies legt alle benötigten Objekte in der Datenbank an (siehe Bild 1).

Das Datenmodell ist schnell erklärt: Die Tabelle **tblMitarbeiter** ist die Haupttabelle.

Sie ist über das Feld **AbteilungID** mit der Tabelle **tblAbteilungen** verknüpft.

Definieren von Konstanten für Server und Datenbank

Da wir in allen VBA-Prozeduren auf den gleichen Server und die gleiche Datenbank zugreifen wollen, definieren wir diese als Konstanten:

```
Public Const cStrServer As String = "amvDesktop2023"
Public Const cStrDatabase As String = "Test_Gespeicherte-
Prozeduren"
```

Diese müssen an die jeweiligen Gegebenheiten angepasst werden.

Aufbau der gespeicherten Prozeduren

Es gibt vier verschiedene Arten von gespeicherten Prozeduren:

- zum Ermitteln der Daten einer Tabelle,
- zum Anlegen eines Datensatzes,

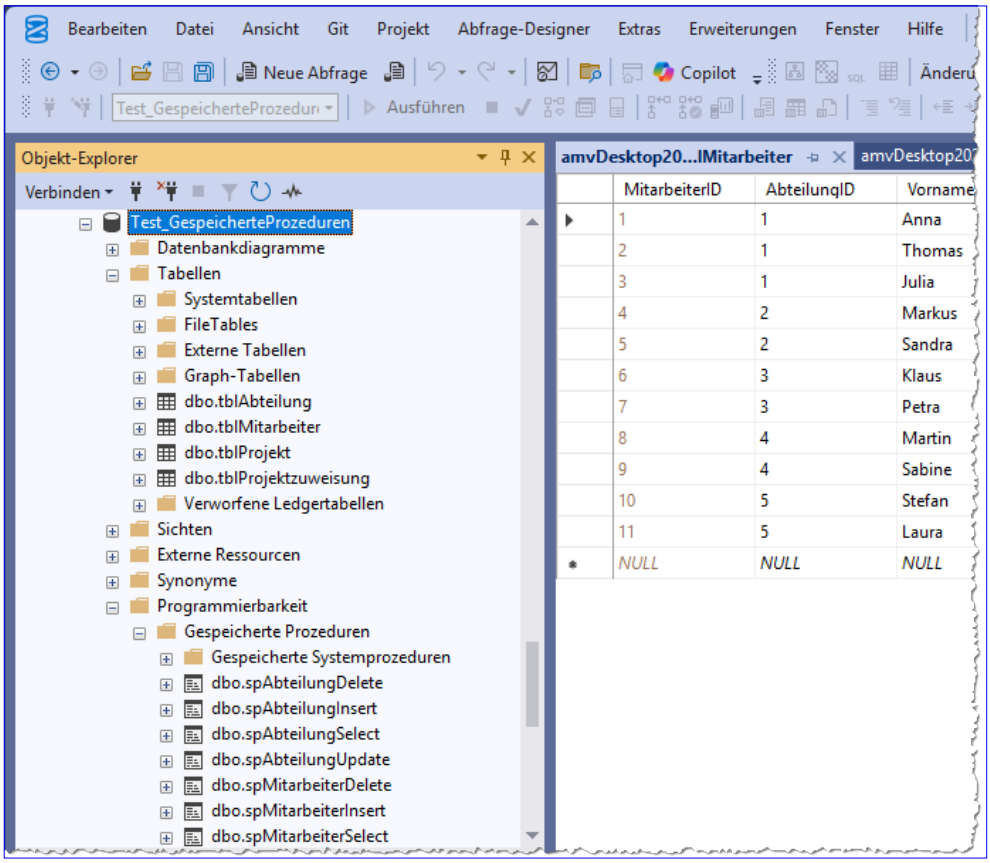


Bild 1: Die SQL Server-Datenbank mit den angelegten Objekten

- zum Bearbeiten eines Datensatzes und
- zum Löschen eines Datensatzes.

Gespeicherte Prozedur zum Ermitteln der Daten einer Tabelle

In diesem Beispiel wollen wir direkt mit einer gespeicherten Prozedur einsteigen, mit der wir einen oder alle Einträge der Tabelle **tblMitarbeiter** ermitteln. Die gespeicherte Prozedur heißt **spMitarbeiterSelectNachMitarbeiterID** und ist in Listing 1 abgebildet.

Sie nimmt den Parameter **@MitarbeiterID** entgegen und wertet diesen in der **WHERE**-Klausel aus.

Hier verwenden wir einen Trick, um sicherzustellen, dass die gespeicherte Prozedur auf zwei Arten arbeitet:

- erstens um alle Datensätze der Tabelle zurückzugeben, wenn kein Parameter übergeben wurde und
- zweitens, um nur den Datensatz mit dem per Parameter übergebenen Wert zu ermitteln.

Wie können wir das in einer gespeicherten Prozedur berücksichtigen? Indem wir den Parameter **@MitarbeiterID** als optionalen Parameter definieren, der als Standardwert **NULL** erhält.

Diesen werten wir in der **WHERE**-Klausel so aus, dass wir prüfen, ob entweder **@MitarbeiterID** gleich **NULL** ist oder der Wert des Feldes **MitarbeiterID** gleich **@MitarbeiterID** ist.

Wird kein Parameter übergeben, trifft für jeden Datensatz die Bedingung **@MitarbeiterID IS NULL** zu und alle Datensätze zurückgegeben. Ist **@MitarbeiterID** nicht **NULL**, wird nur der Datensatz zurückgegeben, dessen Wert im Feld **MitarbeiterID** dem Wert aus **@MitarbeiterID** entspricht.

Aufruf der SELECT-Prozedur per VBA

Um die gespeicherte Prozedur **spMitarbeiterSelectNachMitarbeiterID** per ADODB aufzurufen, sieht der Code wie in Listing 2 aus. Zunächst wird eine Verbindung zur SQL-Server-Datenbank aufgebaut, indem ein neues **ADODB.Connection**-Objekt erstellt und mit dem Verbindungsstring geöffnet wird.

Anschließend wird ein **ADODB.Command**-Objekt erstellt, dem die Verbindung, der Name der gespeicherten Prozedur sowie der Befehlstyp **adCmdStoredProc** zugewiesen werden. Der Befehlstyp teilt ADODB mit, dass es sich um eine gespeicherte Prozedur handelt und nicht etwa um eine direkte SQL-Anweisung.

Danach wird ein **ADODB.Recordset**-Objekt erstellt. Die Eigenschaft **CursorLocation** wird auf **adUseClient** gesetzt, was bedeutet, dass die Daten vollständig auf den Client übertragen werden – dies ist notwen-

```
CREATE PROCEDURE [dbo].[spMitarbeiterSelectNachMitarbeiterID]
    @MitarbeiterID INT = NULL
AS
SET NOCOUNT ON
SELECT  MitarbeiterID,
        Vorname,
        Nachname,
        Email,
        Eintrittsdatum,
        Gehalt,
        ErstelltVon,
        ErstelltAm
FROM    tblMitarbeiter
WHERE   (@MitarbeiterID IS NULL OR MitarbeiterID = @MitarbeiterID)
ORDER BY Nachname, Vorname
```

Listing 1: Gespeicherte Prozedur zum Ermitteln von Mitarbeitern

dig, damit die Eigenschaft **RecordCount** die korrekte Anzahl der Datensätze liefert. Mit **adOpenStatic** und **adLockOptimistic** wird das Recordset schließlich geöffnet.

Anschließend kann die Anzahl der zurückgegebenen Datensätze über **RecordCount** ausgegeben und die Datensätze in einer Schleife durchlaufen werden.

Am Ende werden Recordset und Verbindung sauber geschlossen und die Objektvariablen auf **Nothing** gesetzt, um den Speicher freizugeben.

Aufruf der SELECT-Prozedur mit Parameter

Die Prozedur **Beispiel_Select_NachMitarbeiterID** in Listing 3 zeigt, wie der Aufruf mit einem konkreten Parameter funktioniert.

Der Aufbau ist identisch mit dem vorherigen Beispiel – Verbindung öffnen, **Command**-Objekt konfigurieren – mit einem wesentlichen Unterschied:

Vor dem Öffnen des Recordsets wird der Parameter **@MitarbeiterID** mit dem gewünschten Wert übergeben, um nach diesem zu filtern.

In diesem Beispiel ist das der Wert **1**, der in der Variablen **lngMitarbeiterID** gespeichert ist.

Um einen Parameter zu übergeben, deklarieren wir zuvor eine Variable namens **prm** mit dem Datentyp **Parameter**. Den Parameter erstellen wir mit der Funktion **CreateParameter**. Dieser übergeben wir die folgenden Parameter:

```
Public Sub Beispiel_Select()
    Dim cnn As ADODB.Connection
    Dim cmd As ADODB.Command
    Dim rst As ADODB.Recordset
    Dim strConnection As String

    Set cnn = New ADODB.Connection
    strConnection = "Provider=SQLOLEDB;Data Source=" & cStrServer _
        & ";Initial Catalog=" & cStrDatabase & ";Integrated Security=SSPI;"

    cnn.Open strConnection

    Set cmd = New ADODB.Command
    cmd.ActiveConnection = cnn
    cmd.CommandText = "spMitarbeiterSelectNachMitarbeiterID"
    cmd.CommandType = adCmdStoredProc

    Set rst = New ADODB.Recordset
    rst.CursorLocation = adUseClient
    rst.Open cmd, , adOpenStatic, adLockOptimistic

    Debug.Print "Datensätze: " & rst.RecordCount

    Do While Not rst.EOF
        Debug.Print rst!Nachname
        rst.MoveNext
    Loop

    rst.Close
    cnn.Close
    Set rst = Nothing
    Set cmd = Nothing
    Set cnn = Nothing
End Sub
```

Listing 2: Aufruf der gespeicherten Prozedur zum Ermitteln aller Mitarbeiter

- Name des Parameters, wie er in der gespeicherten Prozedur benannt wurde, hier **@MitarbeiterID**
- Datentyp des Parameters wie er in der gespeicherten Prozeduren verwendet wird, hier **adInteger**

```
Public Sub Beispiel_Select_NachMitarbeiterID()
    Dim cnn As ADODB.Connection
    Dim cmd As ADODB.Command
    Dim rst As ADODB.Recordset
    Dim prm As ADODB.Parameter
    Dim strConnection As String
    Dim lngMitarbeiterID As Long

    lngMitarbeiterID = 1

    Set cnn = New ADODB.Connection
    strConnection = "Provider=SQLOLEDB;Data Source=" & cStrServer & ";Initial Catalog=" & cStrDatabase _
        & ";Integrated Security=SSPI;"

    cnn.Open strConnection

    Set cmd = New ADODB.Command
    cmd.ActiveConnection = cnn
    cmd.CommandText = "spMitarbeiterSelectNachMitarbeiterID"
    cmd.CommandType = adCmdStoredProc

    Set prm = cmd.CreateParameter("@MitarbeiterID", adInteger, adParamInput, _
        4, lngMitarbeiterID)
    cmd.Parameters.Append prm

    Set rst = New ADODB.Recordset
    rst.CursorLocation = adUseClient
    rst.Open cmd, , adOpenStatic, adLockOptimistic

    Debug.Print "Datensätze: " & rst.RecordCount
    Do While Not rst.EOF
        Debug.Print rst!Nachname
        rst.MoveNext
    Loop

    rst.Close
    cnn.Close
    Set rst = Nothing
    Set cmd = Nothing
    Set cnn = Nothing
End Sub
```

Listing 3: Aufruf der gespeicherten Prozedur zum Ermitteln eines Mitarbeiters

ADODB: Access-Wrapper für gespeicherte Prozeduren

Wer Daten zwischen einer Access-Anwendung und einem SQL Server austauscht, landet früher oder später bei gespeicherten Prozeduren. Sie kapseln die Datenbanklogik serverseitig, liefern bessere Performance als dynamisch zusammengebautes SQL und lassen sich sauber rechtebasiert absichern. Der Aufruf per ADODB erfordert allerdings jedes Mal eine ganze Reihe gleicher Schritte: Verbindung öffnen, Command-Objekt erzeugen, CommandType setzen, Parameter definieren, Werte zuweisen, Prozedur ausführen, Ergebnis auslesen. Schreibt man das für jede Prozedur neu, wächst der Code schnell zu unübersichtlichen Ausmaßen an. In diesem Artikel bauen wir ein kleines Wrapper-Modul, das diese Arbeit einmalig erledigt. Anschließend kannst Du jede gespeicherte Prozedur mit einer einzigen Zeile Code aufrufen – inklusive Parameterübergabe und Rückgabewert. Als praktisches Beispiel binden wir den Wrapper an ein vollständiges Mitarbeiter-Formular mit allen vier CRUD-Operationen an.

Beispieldatenbank

Die Beispiele dieses Artikels findest Du in der Beispieldatenbank **ADODB_GespeicherteProzeduren.accdb**. Die zugehörige SQL Server-Datenbank heißt **Test_GespeicherteProzeduren** und enthält die beiden Tabellen **tblAbteilung** und **tblMitarbeiter**. Das Skript zum Anlegen der Tabellen, der Testdaten und der gespeicherten Prozeduren findest Du im Modul **mdlSQLServerDB** als auskommentierten T-SQL-Code, den Du direkt in ein Abfragefenster des SQL Server Management Studio kopieren kannst.

Die vier gespeicherten Prozeduren

Für die Tabelle **tblMitarbeiter** stehen vier gespeicherte Prozeduren bereit, die sich an den vier klassischen CRUD-Operationen orientieren: **SELECT**, **INSERT**, **UPDATE** und **DELETE**. Schauen wir uns diese kurz der Reihe nach an, damit wir wissen, welche Parameter unser Wrapper später bedienen muss.

Die Prozedur **spMitarbeiterSelectNachMitarbeiterID** liefert entweder alle Mitarbeiter zurück oder nur den Mitarbeiter mit einer bestimmten ID, abhängig davon, ob der optionale Parameter **@MitarbeiterID** übergeben wird (siehe Listing 1).

Die Prozedur **spMitarbeiterInsert** legt einen neuen Mitarbeiter an und liefert die neu vergebene **MitarbeiterID** über den Output-Parameter **@IDNEW** zurück. Wir verwenden dafür die Funktion **SCOPE_**

```
CREATE PROCEDURE [dbo].[spMitarbeiterSelectNachMitarbeiterID]
    @MitarbeiterID INT = NULL
AS
SET NOCOUNT ON
SELECT  MitarbeiterID, AbteilungID, Vorname, Nachname, Email, Eintrittsdatum, Gehalt, ErstelltVon, ErstelltAm
FROM tblMitarbeiter
WHERE   (@MitarbeiterID IS NULL
         OR MitarbeiterID = @MitarbeiterID)
ORDER BY Nachname, Vorname
```

Listing 1: Gespeicherte Prozedur zum Auslesen der Mitarbeiter

IDENTITY, die den zuletzt in der aktuellen Sitzung und im aktuellen Gültigkeitsbereich erzeugten Autowert liefert (siehe Listing 2).

Die Prozedur **spMitarbeiterUpdate** aktualisiert einen bestehenden Datensatz und liefert über **@RowsAffected** zurück, wie viele Datensätze betroffen waren (siehe Listing 3).

Die Prozedur **spMitarbeiterDelete** schließlich löscht einen Datensatz und liefert ebenfalls **@RowsAffected** zurück. Den Wert für **@@ROWCOUNT** greifen wir in beiden Fällen direkt im Anschluss an die **UPDATE**- beziehungsweise **DELETE**-Anweisung ab, bevor eine andere Anweisung diesen Wert überschreibt (siehe Listing 4).

```
CREATE PROCEDURE [dbo].[spMitarbeiterInsert]
    @AbteilungID    INT,
    @Vorname        NVARCHAR(100),
    @Nachname       NVARCHAR(100),
    @Email          NVARCHAR(255) = NULL,
    @Eintrittsdatum DATE          = NULL,
    @Gehalt         DECIMAL(10,2) = NULL,
    @IDNEW          INT           = -1 OUTPUT
AS
SET NOCOUNT ON
INSERT INTO tblMitarbeiter (AbteilungID,
    Vorname, Nachname, Email,
    Eintrittsdatum, Gehalt)
VALUES (@AbteilungID, @Vorname, @Nachname,
    @Email, @Eintrittsdatum, @Gehalt)
SET @IDNEW = SCOPE_IDENTITY()
```

Listing 2: Gespeicherte Prozedur zum Anlegen eines Mitarbeiters

Warum ein Wrapper?

Der direkte Aufruf einer gespeicherten Prozedur per ADODB sieht üblicherweise so aus: Du erzeugst ein **Connection**-Objekt, öffnest es mit einer Verbindungszeichenfolge, erzeugst ein **Command**-Objekt, weist diesem die Verbindung zu, setzt den Namen der Prozedur als **CommandText** und den **CommandType** auf **adCmdStoredProc**. Danach legst Du per **CreateParameter** jeden einzelnen Parameter an, weist ihm einen Wert zu und hängst ihn an die **Parameters**-Auflistung des **Command**-Objekts an. Erst dann kannst Du **Execute** aufrufen.

Wie das im Einzelnen aussieht, haben wir bereits im Artikel **ADODB: Gespeicherte Prozeduren ausführen**(**www.vbentwickler.de/503**) beschrieben.

Macht man das für vier Prozeduren, vervierfacht sich dieser Code. Macht man es für 40 Prozeduren, hat man sehr viel Schreibarbeit und sehr viele Stellen, an denen Tippfehler passieren können. Ein Wrapper-Modul erledigt diese immer gleiche Arbeit genau einmal und stellt den Aufruf als kurze Funktion zur Verfügung. Der Aufruf einer Prozedur sieht damit so aus:

```
CREATE PROCEDURE [dbo].[spMitarbeiterUpdate]
    @MitarbeiterID  INT,
    @AbteilungID    INT,
    @Vorname        NVARCHAR(100),
    @Nachname       NVARCHAR(100),
    @Email          NVARCHAR(255)   = NULL,
    @Eintrittsdatum DATE            = NULL,
    @Gehalt         DECIMAL(10,2)   = NULL,
    @RowsAffected   INT             = 0 OUTPUT
AS
SET NOCOUNT ON
Update tblMitarbeiter
SET     AbteilungID    = @AbteilungID,
        Vorname        = @Vorname,
        Nachname       = @Nachname,
        Email          = @Email,
        Eintrittsdatum = @Eintrittsdatum,
        Gehalt         = @Gehalt
WHERE   MitarbeiterID  = @MitarbeiterID
SET @RowsAffected = @@ROWCOUNT
```

Listing 3: Gespeicherte Prozedur zum Aktualsiieren eines Mitarbeiters

```
Set rst = ADODB_GetRecordset( _
    "spMitarbeiterSelectNachMitarbeiterID", 5)
```

Eine Zeile – und Du erhältst das Recordset für den Mitarbeiter mit der ID **5**. Das ist das Ziel.

Die Verbindungsfunktion

Den ganzen Code unseres Wrappers packen wir in das Modul **mdlADODB_Parameter**. Als Erstes definieren wir oben im Modul einige Konstanten mit den Verbindungsdaten. Diese legen wir als **Private Const** an, damit sie nur innerhalb des Moduls sichtbar sind. In einer produktiven Anwendung würdest Du diese Werte vermutlich aus einer Konfigurationstabelle oder aus der Registry lesen, aber für unser Beispiel reicht die Lösung als Konstanten.

```
Option Compare Database
Option Explicit

Private Const strServer As String = "Dein Server"
Private Const strDatabase As String = "Deine Datenbank"
Private Const blnUseSSPI As Boolean = True
Private Const strUsr As String = ""
Private Const strPwd As String = ""
```

```
CREATE PROCEDURE [dbo].[spMitarbeiterDelete]
    @MitarbeiterID  INT,
    @RowsAffected   INT = 0 OUTPUT
AS
SET NOCOUNT ON
DELETE FROM tblMitarbeiter
WHERE MitarbeiterID = @MitarbeiterID
SET @RowsAffected = @@ROWCOUNT
```

Listing 4: Gespeicherte Prozedur zum Löschen eines Mitarbeiters

Die Funktion **ADODB_GetConnection** baut aus diesen Konstanten eine Verbindungszeichenfolge zusammen und öffnet eine neue **ADODB.Connection** (siehe Listing 5).

Über die Konstante **blnUseSSPI** steuern wir, ob die Verbindung per Windows-Authentifizierung (**Integrated Security=SSPI**) oder per SQL Server-Authentifizierung mit Benutzername und Kennwort aufgebaut wird:

```
Public Function ADODB_GetConnection() As ADODB.Connection
    Dim strConnection As String
    Dim cnn As ADODB.Connection
    On Error GoTo ErrHandler
    Set cnn = New ADODB.Connection
    If blnUseSSPI Then
        strConnection = "Provider=MSOLEDBSQL;Data Source=" & strServer & ";Initial Catalog=" _
            & strDatabase & ";Integrated Security=SSPI;"
    Else
        strConnection = "Provider=MSOLEDBSQL;Data Source=" & strServer & ";Initial Catalog=" _
            & strDatabase & ";User ID=" & strUsr & ";Password=" & strPwd & ";"
    End If
    cnn.Open strConnection
    Set ADODB_GetConnection = cnn
    Exit Function
ErrHandler:
    MsgBox "Fehler in ADODB_Connection:" & vbCrLf & Err.Description, vbCritical, "DB Connection"
    Set ADODB_GetConnection = Nothing
End Function
```

Listing 5: Verbindung zum SQL Server herstellen

Als Provider verwenden wir **MSOLEDBSQL**, den aktuellen OLE DB-Treiber für den SQL Server. Im Fehlerfall zeigen wir eine Meldung an und liefern **Nothing** zurück, damit der aufrufende Code prüfen kann, ob die Verbindung erfolgreich war.

Die Helferfunktion ADODB_BuildCommand

Der Kern des Wrappers steckt in der Funktion **ADODB_BuildCommand** (siehe Listing 6). Sie ist als **Private** deklariert, weil sie nur intern von den öffentlichen Funktionen aufgerufen wird. Ihre Aufgabe ist es, ein **Command**-Objekt zu erzeugen, ihm die Verbindung und den Prozedurnamen zuzuweisen und anschließend die Werte aus einem Parameter-Array automatisch in die Parameter der gespeicherten Prozedur zu kopieren.

Damit das funktioniert, müssen wir die Parameter-Definitionen der Prozedur kennen. Statt sie mühsam

```
Private Function ADODB_BuildCommand(ByVal strProcedure As String, ByRef varParameters As Variant) As ADODB.Command
    Dim cmd As ADODB.Command
    Dim intParameter As Integer
    Dim lngCurrentParameter As Long
    Dim lngMaxIndex As Long
    Set cmd = New ADODB.Command
    cmd.ActiveConnection = ADODB_GetConnection
    cmd.CommandText = strProcedure
    cmd.CommandType = adCmdStoredProc
    cmd.Parameters.Refresh
    On Error Resume Next
    lngMaxIndex = UBound(varParameters)
    If Err.Number <> 0 Then
        lngMaxIndex = -1
    End If
    On Error GoTo 0
    lngCurrentParameter = 0
    For intParameter = 1 To cmd.Parameters.Count - 1
        If cmd.Parameters(intParameter).Direction = adParamInput _
                Or cmd.Parameters(intParameter).Direction = adParamInputOutput Then
            If lngCurrentParameter <= lngMaxIndex Then
                If IsNull(varParameters(lngCurrentParameter)) Then
                    cmd.Parameters(intParameter).Value = Null
                Else
                    cmd.Parameters(intParameter).Value = varParameters(lngCurrentParameter)
                End If
                lngCurrentParameter = lngCurrentParameter + 1
            End If
        End If
    Next intParameter
    Set ADODB_BuildCommand = cmd
End Function
```

Listing 6: **Command**-Objekt bauen und Parameter füllen

ADODB: Formulare mit gespeicherten Prozeduren

Access-Formulare leben traditionell von ihrer Datensatzquelle. Du weist dem Formular eine Tabelle oder eine Abfrage zu, und Access kümmert sich um alles Weitere – Navigieren, Bearbeiten, Speichern, Löschen. Das funktioniert hervorragend, solange die Daten in einer Access-Datenbank oder in verknüpften Tabellen liegen. Arbeitest Du jedoch mit einem SQL Server und möchtest den Datenzugriff ausschließlich über gespeicherte Prozeduren abwickeln – etwa, weil Du die Datenbanklogik serverseitig kapseln, Berechtigungen feingranular steuern oder die Performance optimieren willst –, stößt dieses Modell an seine Grenzen. Gespeicherte Prozeduren lassen sich nicht als Datensatzquelle eines Formulars zuweisen, und die klassischen Access-Mechanismen zum Speichern greifen nicht. In diesem Artikel zeigen wir Dir, wie Du ein vollständiges CRUD-Formular mit Übersicht und Detailansicht baust, das komplett ohne Datensatzquelle auskommt. Stattdessen füllen wir die Unterformular-Recordsets und die Detailfelder zur Laufzeit mit dem Ergebnis gespeicherter Prozeduren, die wir über den Wrapper aus dem Artikel »ADODB: Gespeicherte Prozeduren einfach aufrufen« (www.vbentwickler.de/505) aufrufen. Das Ergebnis ist eine Formularoberfläche, die sich vom Benutzer aus anfühlt wie ein klassisches Access-Formular, intern aber ausschließlich mit serverseitigen Prozeduren kommuniziert.

Beispieldatenbank

Die Beispiele dieses Artikels findest Du in der Beispieldatenbank **ADODB_GespeicherteProzeduren.accdb**. Die zugehörige SQL Server-Datenbank heißt **Test_GespeicherteProzeduren** und enthält die beiden Tabellen **tblAbteilung** und **tblMitarbeiter**.

Das Skript zum Anlegen der Tabellen, der Testdaten und der gespeicherten Prozeduren findest Du im Modul **mdlSQLServerDB** als auskommentierten T-SQL-Code, den Du direkt in ein Abfragefenster des SQL Server Management Studio kopieren kannst.

Praxiseinsatz im Übersichtsformular

Damit der Wrapper mehr sein kann als ein theoretisches Konstrukt, verbinden wir ihn nun mit einem vollständigen Access-Formular. Das Formular

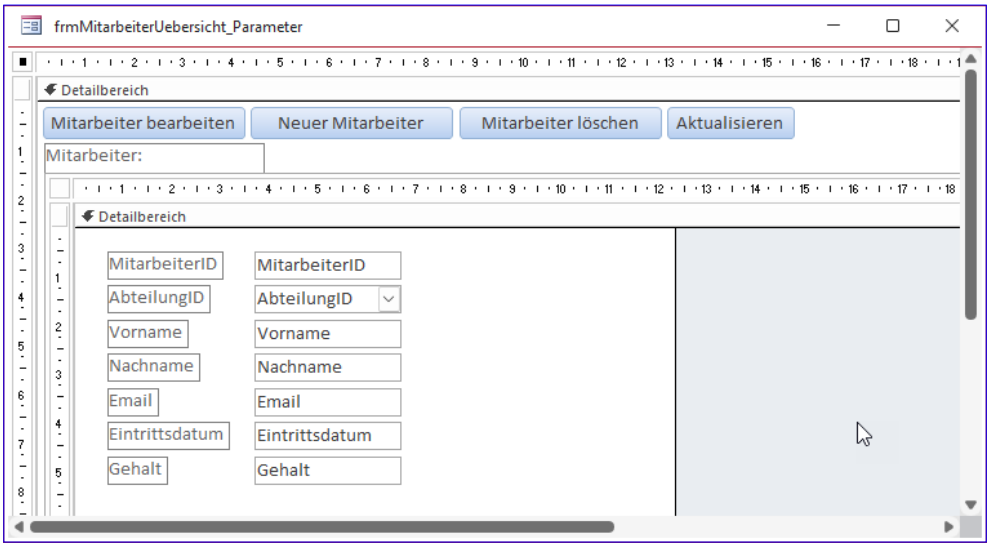


Bild 1: Formular zur Anzeige einer Mitarbeiter-Übersicht

```
Private Sub MitarbeiterAktualisieren()
    With Me.sfmMitarbeiterUebersicht.Form
        Set .cboAbteilungID.Recordset = ADODB_GetRecordset("spAbteilungenSelect")
        Set .Recordset = ADODB_GetRecordset("spMitarbeiterSelectNachMitarbeiterID")
    End With
End Sub
```

Listing 1: Recordset des Unterformulars setzen

frmMitarbeiterUebersicht_Parameter enthält ein Unterformular in Datenblattansicht, das alle Mitarbeiter auflistet, sowie vier Schaltflächen zum Neuanlegen, Bearbeiten, Löschen und Aktualisieren (siehe Bild 1).

Das Übersichtsformular selbst hat keine Datensatzquelle. Stattdessen bekommt das Unterformular **sfmMitarbeiterUebersicht_Parameter** das Recordset erst zur Laufzeit zugewiesen – nämlich aus dem Rückgabewert unserer Funktion **ADODB_GetRecordset**.

Diese Zuweisung passiert in der Prozedur **MitarbeiterAktualisieren** (siehe Listing 1).

Zwei Zuweisungen – mehr ist nicht nötig. Die erste Zeile weist einem Kombinationsfeld **cboAbteilungID** im Datenblatt ein Recordset der Abteilungen zu, damit dort statt der nackten **AbteilungID** die entsprechende Abteilungsbezeichnung angezeigt werden kann.

Die zweite Zeile weist dem Unterformular das Recordset der Mitarbeiter zu. Weil wir **ADODB_GetRecordset** ohne Parameter aufrufen, werden alle Mitarbeiter zurückgeliefert.

An einer Stelle müssen wir vorsichtig sein: Wir können die Prozedur **MitarbeiterAktualisieren** nicht direkt im **Form_Load**- oder **Form_Open**-Ereignis aufrufen, weil das Unterformular zu diesem Zeitpunkt noch nicht vollständig initialisiert ist.

Stattdessen setzen wir im **Form_Load**-Ereignis das **TimerInterval** auf **100** Millisekunden und rufen die Aktualisierung dann im **Form_Timer**-Ereignis auf, wobei wir den Timer gleich wieder mit dem Wert **100** deaktivieren:

```
Private Sub Form_Load()
    Me.TimerInterval = 100
End Sub
```

```
Private Sub cmdNeu_Click()
    Dim strForm As String
    Dim lngMitarbeiterID As Long
    strForm = "frmMitarbeiterDetail_Parameter"
    DoCmd.OpenForm strForm, WindowMode:=acDialog
    If IstFormularGeoeffnet(strForm) Then
        lngMitarbeiterID = Nz(Forms(strForm).Form.MitarbeiterID, 0)
        DoCmd.Close acForm, strForm
        Call MitarbeiterAktualisieren
        Me.sfmMitarbeiterUebersicht.Form .Recordset.Find "MitarbeiterID = " & lngMitarbeiterID
    End If
End Sub
```

Listing 2: Neuen Mitarbeiter anlegen

```
Private Sub cmdBearbeiten_Click()
    Dim strForm As String
    Dim lngMitarbeiterID As Long
    lngMitarbeiterID = Me.sfmMitarbeiterUebersicht.Form.MitarbeiterID
    strForm = "frmMitarbeiterDetail_Parameter"
    DoCmd.OpenForm strForm, WindowMode:=acDialog, OpenArgs:=lngMitarbeiterID
    If IstFormularGeoeffnet(strForm) Then
        DoCmd.Close acForm, strForm
        Call MitarbeiterAktualisieren
        Me.sfmMitarbeiterUebersicht.Form.Recordset.Find "MitarbeiterID = " & lngMitarbeiterID
    End If
End Sub
```

Listing 3: Mitarbeiter bearbeiten

```
Private Sub Form_Timer()
    Me.TimerInterval = 0
    Call MitarbeiterAktualisieren
End Sub
```

Die Schaltfläche **cmdAktualisieren** ruft einfach wieder **MitarbeiterAktualisieren** auf und lädt so die Daten neu vom SQL Server.

Neuen Datensatz anlegen

Die Schaltfläche **cmdNeu** öffnet das Detailformular **frmMitarbeiterDetail_Parameter** als Dialog. Nach dem Schließen des Dialogs liest sie die ID des neu angelegten Datensatzes aus, aktualisiert die Liste und positioniert den Datensatzmarkierer auf dem neuen Eintrag (siehe Listing 2).

Das **WindowMode:=acDialog** sorgt dafür, dass das Detailformular modal geöffnet wird und der Code an dieser Stelle stehenbleibt, bis das Formular entweder geschlossen oder unsichtbar gemacht wird. Die Hilfsfunktion **IstFormularGeoeffnet**, die im Modul **mdlTools** liegt, prüft über **SysCmd**, ob das Detailformular noch geöffnet ist – dann hat der Benutzer gespeichert (das Detailformular setzt in diesem Fall nur **Me.Visible = False**) und wir lesen die vergebene ID aus.

Bearbeiten und Löschen

Die Schaltfläche **cmdBearbeiten** arbeitet nach demselben Prinzip, übergibt dem Detailformular aber zusätzlich die zu bearbeitende ID als **OpenArgs** (siehe Listing 3).

Die Schaltfläche **cmdLoeschen** schließlich ist die kürzeste von allen, weil wir hier direkt unsere Wrapper-Funktion **ADODB_ExecDelete** einsetzen können (siehe Listing 4).

```
Private Sub cmdLoeschen_Click()
    Dim lngMitarbeiterID As Long
    lngMitarbeiterID = Me.sfmMitarbeiterUebersicht.Form.MitarbeiterID
    If ADODB_ExecDelete("spMitarbeiterDelete", lngMitarbeiterID) > 0 Then
        MsgBox "Mitarbeiter gelöscht.", vbOKOnly + vbInformation, "Mitarbeiter gelöscht"
        Call MitarbeiterAktualisieren
    End If
End Sub
```

Listing 4: Mitarbeiter löschen

Datenbank-Zeitreisen: Temporal Tables im SQL Server

Wer hat den Preis geändert, und wie hoch war er eigentlich vor drei Wochen? Wann ist dieser Datensatz gelöscht worden, und wie sah er kurz davor aus? Sobald eine Datenbank produktiv läuft, tauchen solche Fragen auf – und oft steht man mit leeren Händen da, weil die alten Werte längst überschrieben sind. Der klassische Ausweg ist eine selbstgebaute Historientabelle samt Triggern, die bei jedem INSERT, UPDATE und DELETE die alte Zeile wegschreiben. Das funktioniert, ist aber Handarbeit. Der SQL Server bringt seit der Version 2016 eine eingebaute Lösung mit: System-Versioned Temporal Tables. Die Datenbank führt die komplette Änderungshistorie vollautomatisch mit, und Abfragen zu einem beliebigen Zeitpunkt der Vergangenheit sind eine Zeile Zusatz-SQL. Wie das gelingt, wie und wo die alten Daten landen und wie Du eine bestehende Tabelle nachträglich versionierst, liest Du in diesem Beitrag – und auch, was Temporal Tables nicht können.

Was eine Temporal Table eigentlich ist

Eine systemversionierte temporale Tabelle besteht immer aus einem Paar: der aktuellen Tabelle, die – wie gewohnt – den jeweils gültigen Stand jeder Zeile enthält, und einer zugehörigen Historientabelle, in der jede frühere Version einer Zeile aufbewahrt wird. Beide zusammen ergeben eine lückenlose Zeitleiste.

Damit der SQL Server weiß, ab wann und bis wann eine bestimmte Version einer Zeile gültig war, braucht die Tabelle zwei zusätzliche Spalten vom Typ **DATETIME2**, die sogenannten Periodenspalten. Die eine hält den Startzeitpunkt (im Beispiel gleich **GueltigVon**), die andere den Endzeitpunkt (**GueltigBis**).

Solange eine Zeile aktuell ist, steht in **GueltigBis** der Maximalwert **9999-12-31 23:59:59.9999999** – der SQL Server liest das als „gilt bis auf Weiteres".

Ändern wir die Zeile, passiert im Hintergrund Folgendes: Die bisherige Version wandert mit ihrem echten Endzeitpunkt in die Historientabelle, und in der aktuellen Tabelle steht die neue Version mit frischem Startzeitpunkt. Wir müssen dafür nichts programmieren – kein Trigger, keine Prozedur.

Ein Punkt vorweg, der später noch wichtig wird: Der SQL Server speichert diese Zeitstempel in **UTC**, nicht in lokaler Zeit.

Für uns in Deutschland heißt das ein bis zwei Stunden Versatz zur Uhr an der Wand. Darauf kommen wir bei den Abfragen zurück.

Was Temporal Tables nicht können

Temporal Tables erstellen nur eine Kopie eines geänderten oder gelöschten Datensatzes und tragen ein, von wann bis wann dieser Datensatz gültig war. Wir können damit nicht automatisch eine Information hinterlegen, wer die neue Version des Datensatzes angelegt hat.

Es wird für eine historisierte Version eines Datensatze auch nicht eingetragen, ob dieser Datensatz durch einen geänderten Datensatz ersetzt oder ob dieser gelöscht wurde – es wird lediglich jede geänderte oder gelöschte Version eines Datensatzes gespeichert.

Die Beispieldatenbank anlegen

Bauen wir zunächst eine ganz normale Tabelle ohne jede Versionierung auf und füllen sie mit ein paar

```
CREATE DATABASE TemporalDemo;
GO

USE TemporalDemo;
GO

CREATE TABLE dbo.tblProdukte
(
    ProduktID   INT IDENTITY(1,1) NOT NULL CONSTRAINT PK_Produkt PRIMARY KEY CLUSTERED,
    Bezeichnung NVARCHAR(100) NOT NULL,
    Preis       DECIMAL(10,2) NOT NULL
);
GO

INSERT INTO dbo.tblProdukte (Bezeichnung, Preis)
VALUES (N'Kaffeebohnen Brasil',   12.90),
       (N'Teekanne Gusseisen',    34.50),
       (N'Espressotassen-Set',    19.99);
GO
```

Listing 1: Beispieldatenbank und Tabelle **tblProdukte** anlegen und mit Daten füllen

Das erledigt Listing 1. Es legt die Datenbank **TemporalDemo** an, darin die Tabelle **tblProdukte** mit den Feldern **ProduktID**, **Bezeichnung** und **Preis** – und füllt drei Zeilen ein.

Ein Detail ist keine Kür, sondern Voraussetzung für alles Weitere: Die Tabelle **tblProdukte** hat einen Primärschlüssel (**PK_Produkt** auf **ProduktID**). Ohne Primärschlüssel lässt sich später keine Systemversionierung einschalten – der SQL Server braucht ihn, um eine Zeile über ihre Versionen hinweg eindeutig zu verketten. Wer hier eine Tabelle ohne Schlüssel hat, muss also zuerst einen nachrüsten.

Datensätzen – so, wie eine bestehende Anwendung sie mitbringen könnte.

Erst im nächsten Schritt rüsten wir die Historie nach. Das entspricht dem häufigsten Fall aus der Praxis: Die Tabelle ist längst da, die Historie soll dazu.

```
USE TemporalDemo;
GO

ALTER TABLE dbo.tblProdukte ADD
    GueltigVon DATETIME2(7) GENERATED ALWAYS AS ROW START NOT NULL
        CONSTRAINT DF_Produkt_GueltigVon DEFAULT SYSUTCDATETIME(),
    GueltigBis DATETIME2(7) GENERATED ALWAYS AS ROW END NOT NULL
        CONSTRAINT DF_Produkt_GueltigBis
        DEFAULT CONVERT(DATETIME2(7), '9999-12-31 23:59:59.9999999'),
    PERIOD FOR SYSTEM_TIME (GueltigVon, GueltigBis);
GO

ALTER TABLE dbo.tblProdukte
    SET (SYSTEM_VERSIONING = ON (HISTORY_TABLE = dbo.tblProdukteHistory));
GO
```

Listing 2: Die bestehende Tabelle **tblProdukte** um Periodenspalten erweitern und die Systemversionierung einschalten

Versionierung an einer bestehenden Tabelle nachrüsten

Jetzt kommt der eigentliche Schritt: Wir erweitern die bestehende Tabelle **tblProdukte** um die beiden Periodenspalten und schalten die Systemversionierung ein.

Das geschieht in Listing 2 in zwei getrennten **ALTER TABLE**-Anweisungen – erst die Spalten und die Periode, dann die Versionierung selbst.

Das ist zugleich das allgemeine Rezept für jede Tabelle, die es in Deiner Datenbank schon gibt – einzige harte Voraussetzung bleibt der Primärschlüssel aus dem vorigen Abschnitt.

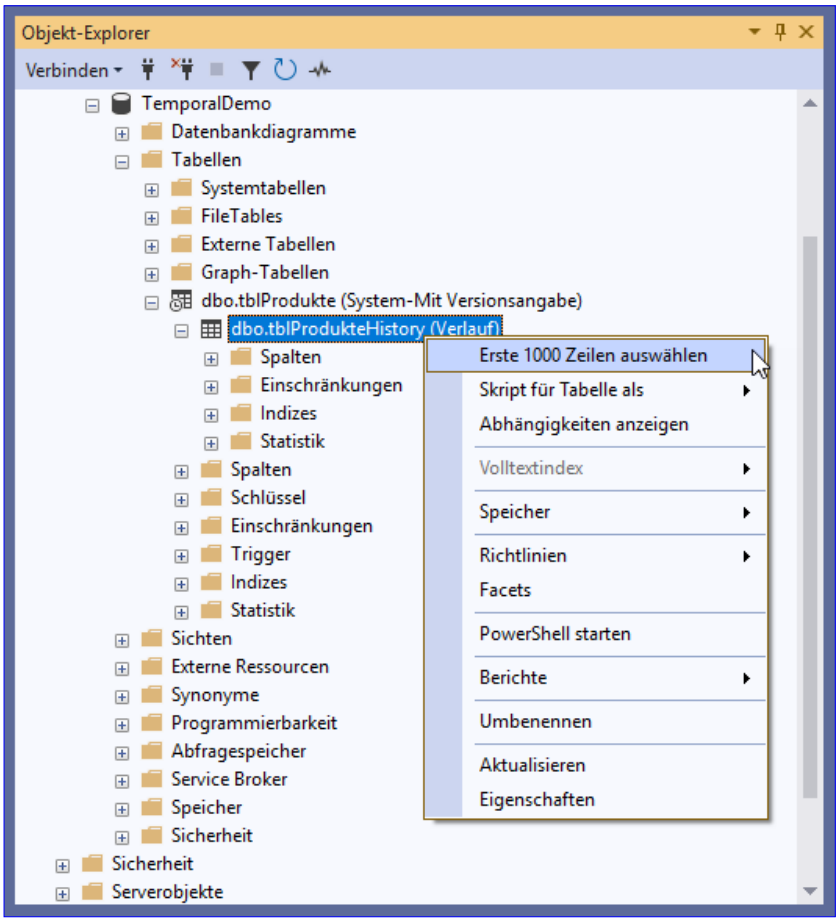


Bild 1: Der Objekt-Explorer zeigt die versionierte Tabelle **Produkt** mit Uhr-Symbol und der eingerückten Historientabelle **ProduktHistory**

Gehen wir das durch, denn hier lauern gleich mehrere Stolpersteine. Die beiden neuen Spalten sind mit **GENERATED ALWAYS AS ROW START** bzw. **ROW END** gekennzeichnet – das sagt dem SQL Server, dass er diese Werte setzt und wir sie niemals von Hand befüllen.

Genau daraus ergibt sich das erste Problem: Die Spalten sind **NOT NULL**, aber unsere Tabelle enthält bereits Daten. Woher soll der SQL Server die Zeitstempel für die drei vorhandenen Zeilen nehmen? Wir würden die Werte nun gern selbst eintragen – aber das ist bei **GENERATED ALWAYS**-Spalten nicht möglich.

Die Lösung sind die beiden **DEFAULT**-Constraints: Sie liefern die Startwerte für die schon vorhandenen Zeilen. Ohne diese Defaults bricht das **ALTER TABLE** bei einer gefüllten Tabelle mit einer Fehlermeldung ab.

Der zweite Stolperstein steckt in der Wahl dieser Defaults. Für **GueltigVon** nehmen wir **SYSUTCDATETIME()**, also den aktuellen Zeitpunkt in **UTC**. Für **GueltigBis** muss es zwingend der Maximalwert **9999-12-31 23:59:59.9999999** sein – nicht etwa ebenfalls die aktuelle Zeit.

Der Endzeitpunkt markiert bei einer aktuellen Zeile das „gilt bis auf Weiteres“, und er muss größer sein als der Startzeitpunkt. Wer hier versehentlich auch **SYSUTCDATETIME()** einsetzt, erntet einen Fehler.

Und der dritte Punkt ist weniger ein Fehler als eine ehrliche Einordnung: Die Historie beginnt in dem Moment, in dem wir die Versionierung einschalten.

Unsere drei Bestandszeilen bekommen alle denselben Startzeitpunkt – den der Umstellung –, nicht ihr echtes Anlagedatum. Rückwirkend entsteht keine Historie, die vorher nie aufgezeichnet wurde. Das ist keine Schwäche des Verfahrens, aber gut zu wissen, bevor man sich über gleiche Zeitstempel wundert.

Die zweite Anweisung schaltet mit **SET (SYSTEM_VERSIONING = ON …)** die Versionierung scharf und legt dabei die Historientabelle **dbo.tblProdukteHistory** an. Den Namen geben wir bewusst selbst an. Lässt man ihn weg, erzeugt der SQL Server eine automatisch benannte Tabelle nach dem Muster **MSSQL_TemporalHistoryFor_…** samt Objekt-ID – funktioniert, ist aber unhandlich.

Im Objekt-Explorer des SQL Server Management Studios erkennst Du das Ergebnis sofort: Die Tabelle **tblProdukte** trägt jetzt ein kleines Uhr-Symbol, und darunter hängt als Unterknoten die zugehörige **tblProdukteHistory** (siehe Bild 1). Im Kontextmenü sehen wir auch gleich, dass wir die Datensätze dieser Tabelle nicht bearbeiten können – es fehlt der Eintrag **Oberste 200 Einträge bearbeiten**.

Das war der Weg für eine Tabelle, die es schon gibt – der häufigste Fall in der Praxis. Steht die Tabelle dagegen noch gar nicht, gibt es einen kürzeren Weg, den wir uns weiter unten im Abschnitt zum direkten Anlegen ansehen.

Wo die alten Daten landen

Werfen wir einen Blick auf die Verteilung, bevor überhaupt etwas geändert wurde. Die aktuelle Tabelle **tblProdukte** enthält unsere drei Zeilen, die Historientabelle **tblProdukteHistory** ist noch leer – logisch, denn es gibt ja noch keine früheren Versionen. Erst eine Änderung füllt die Historie.

Wichtig ist die Rollenverteilung: In der aktuellen Tabelle steht immer nur der jeweils gültige Stand. Jede überschriebene oder gelöschte Version zieht in die Historientabelle um.

Diese ist eine ganz normale Tabelle, die Du direkt abfragen kannst – nur ändern darfst Du sie nicht, solange die Versionierung eingeschaltet ist. Der SQL Server verwaltet ihren Inhalt allein.

Und noch einmal der Hinweis auf die Zeitzone, weil er gleich praktisch wird: Die Werte in **GueltigVon** und **GueltigBis** stehen in **UTC**. Ein Datensatz, der um 14:00 Uhr deutscher Sommerzeit geändert wird, trägt in der Historie den Zeitstempel 12:00 Uhr.

```
USE TemporalDemo;
GO

UPDATE dbo.tblProdukte SET Preis = 13.90 WHERE Bezeichnung = N'Kaffeebohnen Brasil';

WAITFOR DELAY '00:00:03';

UPDATE dbo.tblProdukte SET Preis = 29.90 WHERE Bezeichnung = N'Teekanne Gusseisen';

WAITFOR DELAY '00:00:03';

DELETE FROM dbo.tblProdukte WHERE Bezeichnung = N'Espressotassen-Set';
GO
```

Listing 3: Datensätze ändern und löschen, damit die Historie sich füllt

Backstage-Bereich von Access erweitern, Teil 2

Im ersten Teil dieser Artikelserie haben wir den Backstage-Bereich von Access mit eigenen Befehlen und einer Registerkarte ausgestattet, die Steuerelemente wie editBox, checkBox, dropDown, comboBox und radioGroup enthält. In diesem zweiten Teil gehen wir einen Schritt weiter: Wir lernen die Layout-Elemente groupBox, layoutContainer und imageControl kennen und bauen mit taskFormGroup und taskGroup komplexere Aufgabenbereiche auf.

Beispieldatenbank

Die Beispiele zu diesem Artikel findest Du in der Datenbank **BackstageBereichProgrammieren.accdb**. Die Tabelle **USysRibbons** enthält unter dem Eintrag **BackstageRef** bereits alle drei Registerkarten.

Die Callback-Prozeduren aus dem Modul **mdlBackstage** funktionieren auch für die neuen Elemente ohne Änderungen, da wir **control.Id** als universellen Schlüssel verwenden.

Übersicht der drei Registerkarten

Die Backstage-Definition aus dem ersten Teil enthält bisher eine Registerkarte **Steuerelemente (tab)** mit den grundlegenden Eingabe- und Auswahl-Steuerelementen.

In diesem Artikel ergänzen wir zwei weitere Registerkarten für unterschiedliche Anwendungszwecke: **Layout (tab)** demonstriert die visuelle Anordnung von Steuerelementen mit **groupBox** und **layoutContainer**. **Aufgaben (tab)** zeigt die Verwen-

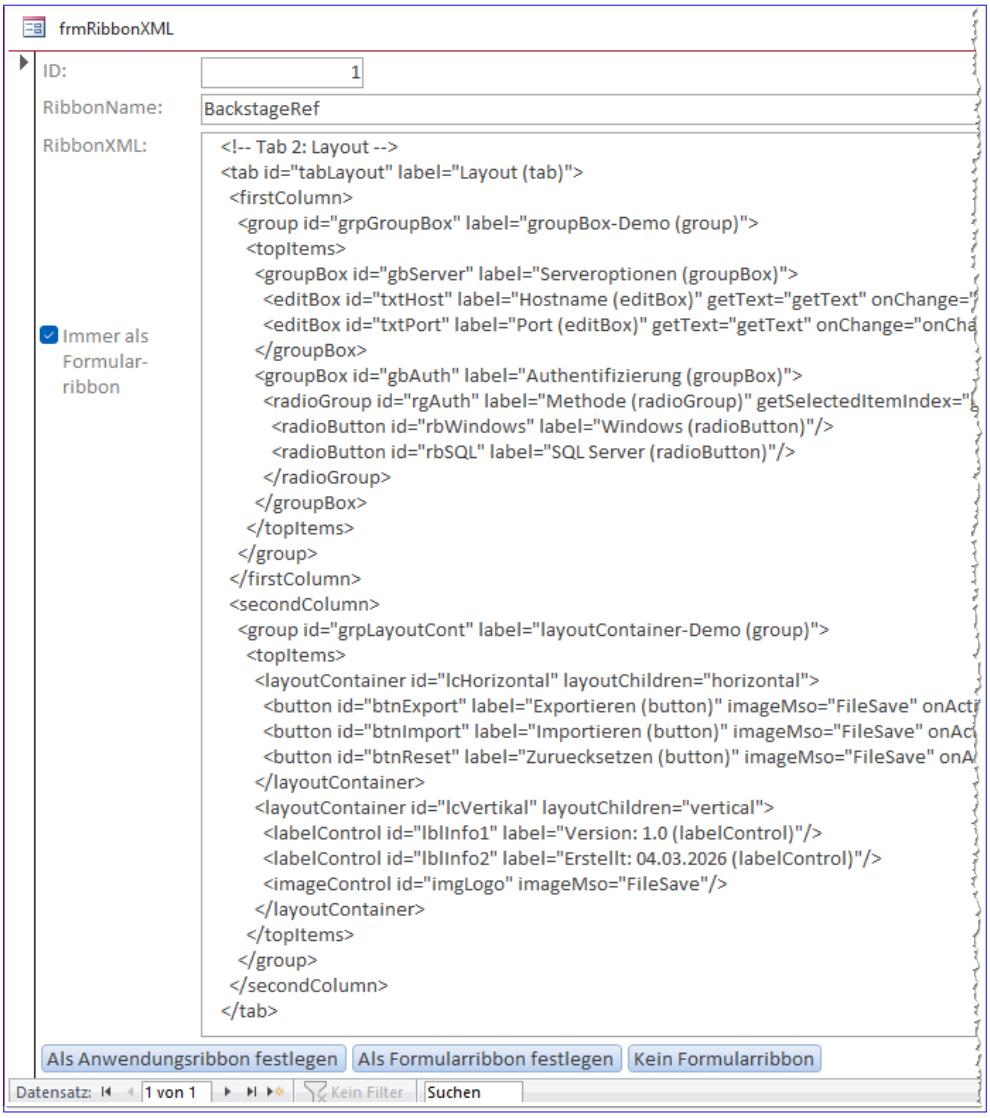


Bild 1: Definition des zweiten Backstage-Tabs

dung von **taskFormGroup** und **taskGroup** für aufgabenorientierte Benutzeroberflächen.

Die Registerkarte »Layout«

Die zweite Registerkarte widmet sich den Layout-Elementen. Während die Steuerelemente aus dem ersten Teil ihren Platz automatisch untereinander erhalten, bieten **groupBox** und **layoutContainer** zusätzliche Möglichkeiten, die Anordnung gezielt zu steuern.

Den vollständigen XML-Code findest Du in der Tabelle **USysRibbons** unter **BackstageRef** (siehe Bild 1). Wir erläutern ihn im Folgenden schrittweise.

Die Registerkarte selbst legen wir wie gewohnt als **tab**-Element mit **firstColumn** und **secondColumn** an. Du findest diesen Bereich gemeinsam mit den anderen neuen Tabs in der Tabelle **USysRibbons** für den Eintrag mit dem Wert **BackstageRef** im Feld **RibbonName**.

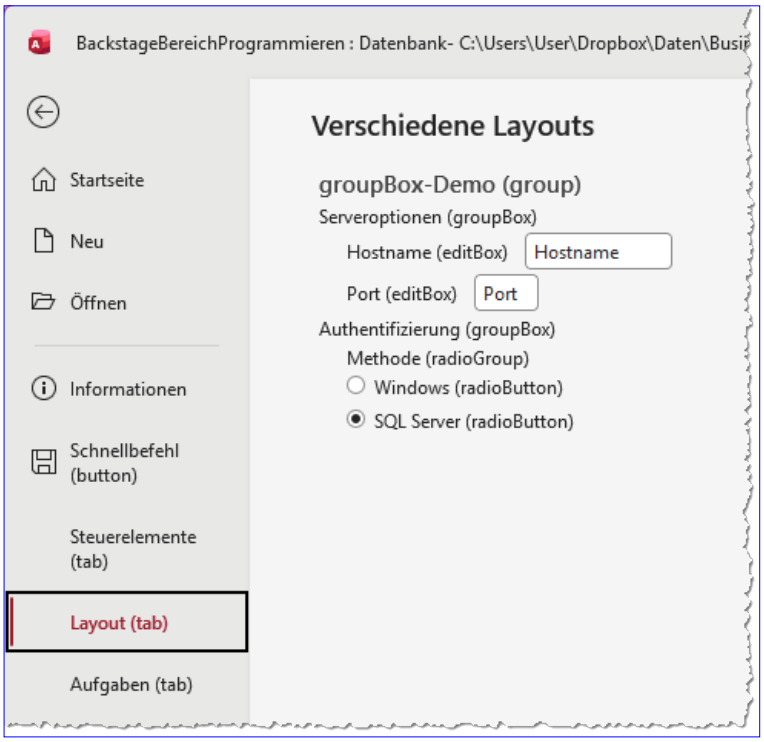


Bild 2: Eine Gruppe in der ersten Spalte eines **tab**-Elements

Diesmal fügen wir das Tab vor dem eingebauten Element **FileSave** ein:

```
<tab id="tabLayout" label="Layout (tab)"
    insertBeforeMso="FileSave"
    title="Verschiedene Layouts">
  <firstColumn>
    ...
  </firstColumn>
  <secondColumn>
    ...
  </secondColumn>
</tab>
```

Das groupBox-Element

Mit **groupBox** fassen wir mehrere Steuerelemente in einem beschrifteten Rahmen zusammen. Das Element besitzt ein **label**-Attribut, das als Überschrift des Rahmens angezeigt wird.

Im Inneren können alle Steuerelemente platziert werden, die auch in **topItems** und **bottomItems** erlaubt sind – also **editBox**, **checkBox**, **radioGroup** und alle weiteren aus der Gruppe **EG_GroupControls**.

In der **firstColumn** der Layout-Registerkarte platzieren wir eine Gruppe mit zwei **groupBox**-Elementen:

```
<group id="grpGroupBox" label="groupBox-Demo (group)">
  <topItems>
    <groupBox id="gbServer"
        label="Serveroptionen (groupBox)">
      <editBox id="txtHost" label="Hostname (editBox)"
        getText="getText" onChange="onChange"
        sizeString="yyyyyyyyyyyyy"/>
      <editBox id="txtPort" label="Port (editBox)"
        getText="getText" onChange="onChange"
        sizeString="yyyyy"/>
```

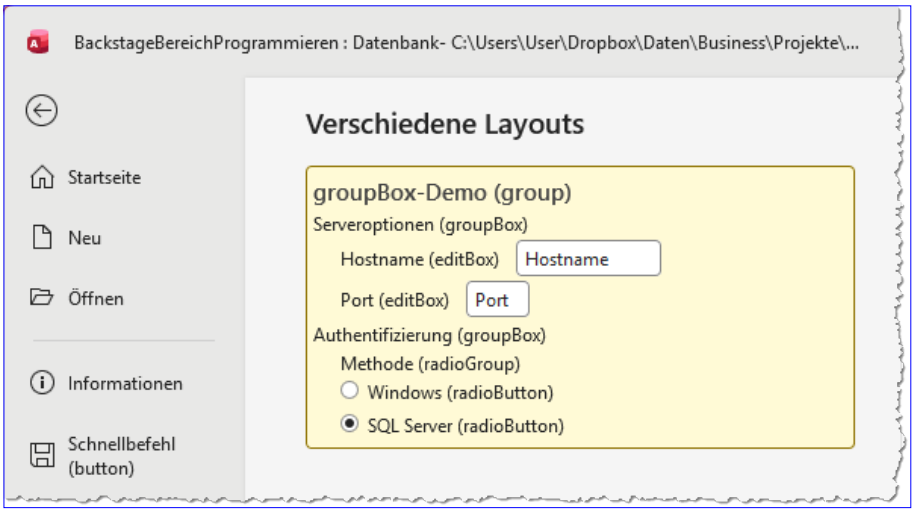


Bild 3: Farbig hinterlegtes **group**-Element

```
    </groupBox>
    <groupBox id="gbAuth"
        label="Authentifizierung (groupBox)">
      <radioGroup id="rgAuth" label="Methode (radioGroup)"
          getSelectedItemIndex="getSelectedItemIndex"
          onAction="onActionRadio">
        <radioButton id="rbWindows"
          label="Windows (radioButton)"/>
        <radioButton id="rbSQL"
          label="SQL Server (radioButton)"/>
      </radioGroup>
    </groupBox>
  </topItems>
</group>
```

Die erste **groupBox** mit der Beschriftung »Serveroptionen« enthält zwei **editBox**-Elemente für Hostname und Port.

Die zweite **groupBox** »Authentifizierung« enthält eine **radioGroup** mit zwei Optionen.

Beide verwenden die Callback-Funktionen aus dem ersten Teil ohne Änderungen.

Laut XSD-Schema unterstützt **groupBox** neben **id** und **label** auch das Attribut **expand**. Damit steuern wir, ob sich der Rahmen horizontal, vertikal oder in beide Richtungen ausdehnt. Die möglichen Werte sind **horizontal**, **vertical**, **both** und **neither**.

Das Ergebnis erscheint wie in Bild 2.

group-Element farbig hinterlegen

Das **group**-Element können wir mit dem Attribut **style** und den Werten **warning** oder **error** auch farbig hinterlegen. Dazu ändern wir es beispielsweise wie folgt:

```
<group id="grpGroupBox"
  label="groupBox-Demo (group)"
  style="warning">
```

Dies sieht wie in Bild 3 aus.

Das layoutContainer-Element

Während **groupBox** einen sichtbaren Rahmen erzeugt, dient **layoutContainer** ausschließlich der Anordnung.

Es erzeugt keinen sichtbaren Rahmen, sondern bestimmt über das Attribut **layoutChildren**, ob die enthaltenen Steuerelemente horizontal nebeneinander oder vertikal untereinander angeordnet werden. Die beiden möglichen Werte sind **horizontal** und **vertical**.

In der **secondColumn** der Layout-Registerkarte demonstrieren wir beide Varianten:

```
<group id="grpLayoutCont"
    label="layoutContainer-Demo (group)">
  <topItems>
    <layoutContainer id="lcHorizontal"
```

```
        layoutChildren="horizontal">
      <button id="btnExport" label="Exportieren (button)"
        imageMso="FileSave" onAction="onAction"
        style="normal"/>
      <button id="btnImport" label="Importieren (button)"
        imageMso="FileSave" onAction="onAction"
        style="normal"/>
      <button id="btnReset" label="Zuruecksetzen (button)"
        imageMso="FileSave" onAction="onAction"
        style="borderless"/>
    </layoutContainer>
    <layoutContainer id="lcVertikal"
        layoutChildren="vertical">
      <labelControl id="lblInfo1"
        label="Version: 1.0 (labelControl)"/>
      <labelControl id="lblInfo2"
        label="Erstellt: 04.03.2026 (labelControl)"/>
      <imageControl id="imgLogo" imageMso="FileSave"/>
    </layoutContainer>
  </topItems>
</group>
```

Der erste **layoutContainer** mit **layoutChildren="horizontal"** ordnet drei Buttons nebeneinander an.

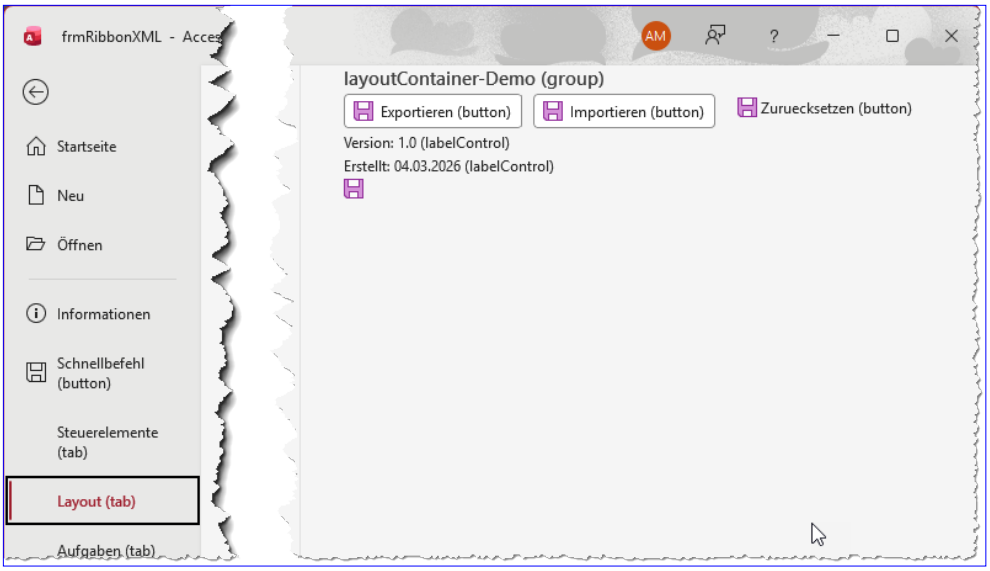


Bild 4: Die **layoutContainer**-Elemente in der **secondColumn** der Layout-Registerkarte

Hier fällt das Attribut **style** der Buttons auf: Im Backstage-Bereich unterstützen Buttons die Werte **normal** (Standarddarstellung mit Rahmen), **borderless** (ohne sichtbaren Rahmen) und **large** (große Darstellung mit Beschriftung unterhalb des Icons).

Der zweite **layoutContainer** mit **layoutChildren="vertical"** zeigt zwei **labelControl**-Elemente und ein **imageControl** untereinander. Ohne den **layoutContainer** wäre das Ergebnis in diesem Fall identisch, da Steuerelemente standardmäßig vertikal angeordnet werden. Der Unterschied wird deutlich, wenn man den Wert auf **horizontal** ändert.

Neben **layoutChildren** unterstützt **layoutContainer** laut XSD die Attribute **align** und **expand**. Mit **align** bestimmen wir die Ausrichtung des Containers innerhalb seines übergeordneten Elements.

Die Werte entsprechen denen von **alignLabel**: **topLeft**, **top**, **topRight**, **left**, **center**, **right**, **bottomLeft**, **bottom** und **bottomRight** (siehe Bild 4).

Das imageControl-Element

Das **imageControl**-Element zeigt ein Bild an, ohne dass der Benutzer damit interagieren kann.

In unserem Beispiel verwenden wir das Attribut **imageMso**, um eines der eingebauten Office-Icons anzuzeigen:

```
<imageControl id="imgLogo"
imageMso="FileSave"/>
```

Alternativ kann man über das Attribut **image** ein eigenes Bild referenzieren oder mit dem Callback