AUSGABE 06/2024

VISUAL BASIC ENTWICKLER

MAGAZIN FÜR DIE PROGRAMMIERUNG VON MICROSOFT OFFICE UND ANDEREN ANWENDUNGEN MIT VB.NET, VBA UND TWINBASIC

IN DIESEM HEFT:

André Minhorst Verlag

Blick über den Tellerrand

In Zeiten von Homeoffice und in denen immer mehr Menschen ihre Freiheit genieße und von dort arbeiten, wo auch immer sie sich gerade befinden, erweisen sich Anwendungen, die nicht diese Flexibilität erlauben, als hinderlich. Auch solche Anwendungen, die an ein bestimmtes Betriebssystem gebunden sind, wie beispielsweise Microsoft Access, haben es nicht leicht in diesen Zeiten. Denn es gibt immer mehr Anwendungen, sogenannte SaaS-Tools (Software as a Service), die in der Cloud liegen und bequem über den Webbrowser bedient werden können.

Und diese Anwendungen bieten noch weitere Vorteile: Die meisten sind nämlich mit einer umfangreichen Rest-API-Schnittstelle ausgestattet. Das erlaubt uns zwei Dinge:

- Wir können auf diese per VBA zugreifen und benötigen dazu nur wenig mehr als die Bordmittel.
- Wir können zusätzliche Tools nutzen, um diese Anwendungen komplett unabhängig zu automatisieren.

Damit lassen sich Anwendungen wie ein CRM-System beispielsweise automatisch mit den Daten füttern, die der Benutzer in ein Kontaktformular eingetragen hat. Oder wir erstellen automatisch auf Basis eingehender E-Mails Aufgaben in Microsoft To Do.

Deshalb liefern wir mit dem Artikel **Automation mit Zapier, Make und Co.** ab Seite 27 eine kleine Einführung in das Thema, das wir in den folgenden Ausgaben vertiefen werden.

Wer herausfinden möchte, ob er gerade die 32- oder 64-Bit-Version von Office nutzt, was beispielsweise für die Installation von Tools wichtig ist, die von der Bitness abhängen, findet unter dem Titel **Ist die 32- oder 64-Bit-Version von Office installiert?** ab Seite 4 hilfreiche Informationen.

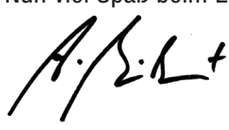
Immer wieder wird man mit dem Zugriff auf Dateien und Verzeichnisse konfrontiert. Deshalb haben wir in dieser Ausgabe gleich zwei Artikel zu diesem Thema. Unter **Dateien mit VBA-Bordmitteln verwalten** zeigen wir ab Seite 7, welche Möglichkeiten bereits die Grundausstattung von VBA liefern.

Und wenn wir schon bei Dateien sind, gehen wir gleich weiter zum Thema Textdateien: Unter **VBA und Textdateien: Alles über Open, Close und Co.** zeigen wir ab Seite 19, wie wir Textdateien mit VBA lesen und schreiben können.

Wer per VBA auf den SQL Server zugreift, findet in eventuellen Fehlermeldungen möglicherweise nur wenig hilfreiche Informationen. Wie wir mehr als die Fehlermeldung **ODBC-Aufruf fehlgeschlagen** herausholen, zeigen wir ab Seite 34 in **SQL Server: Fehlerbehandlung in DAO und ADODB**.

Schließlich steigen wir in die Programmierung des Datenzugriffs mit ADODB ein, und zwar in den Artikeln **ADODB: Connections und Connectionstrings** (ab Seite 44), **ADODB: SQL-Befehle schnell ausführen mit Execute** (ab Seite 54) und in **ADODB: Transaktionen im SQL Server** (ab Seite 59). Und das ist nur der Anfang – in den nächsten Ausgaben vertiefen wir das Thema, bis Du ein ADODB-Profi bist.

Nun viel Spaß beim Lesen!



Dein André Minhorst

Ist die 32- oder 64-Bit-Version von Office installiert?

Diese Frage, ob Office in der 32-Bit- oder in der 64-Bit-Version vorliegt, ist für viele Aufgaben interessant. Seit Access 2019 wird Office standardmäßig in der 64-Bit-Version installiert. Es gibt jedoch auch immer alternativ die 32-Bit-Version. Vor Access 2019 war die 32-Bit-Installation Standard. Wozu aber benötigen wir diese Information überhaupt? Wenn wir das VBA-Projekt einer Access-, Excel-, Word- oder PowerPoint-Datei programmieren oder das Outlook-Objektmodell und dabei weder ActiveX-Steuerelemente noch Integrationen wie COM-DLLs oder COM-Add-Ins oder API-Funktionen nutzen, spielt es keine Rolle, ob wir mit 32-Bit- oder 64-Bit-Office arbeiten. Sobald jedoch eines der genannten Elemente auftaucht, müssen wir genau prüfen, ob dieses unter beiden Versionen arbeitet. Wir schauen uns kurz an, wo besonderes Augenmerk gefragt ist und wie wir es dem Benutzer mitteilen können, wenn seine Office-Version und unsere Erweiterungen nicht kompatibel sind.

Wie finde ich heraus, ob die 32- oder 64-Bit-Version von Access/Office installiert ist? Die Information, ob auf einem Rechner die 32-Bit- oder 64-Bit-Version von Microsoft Access oder Office installiert ist, ist in verschiedenen Szenarien wichtig. Beispielsweise spielt dies bei der Entwicklung von Access-Anwendungen oder bei der Verwendung von VBA (Visual Basic for Applications) eine Rolle, da die verwendeten Bibliotheken und API-Deklarationen an die Bit-Version angepasst werden müssen.

Das ist vor allem bei den folgenden Situationen wichtig:

- Wenn wir API-Funktionen verwenden. Diese müssen unter 64-Bit anders deklariert werden als unter 32-Bit. In den meisten Fällen erhalten wir bereits beim Kompilieren eines VBA-Projekts in der 64-Bit-Version Fehlermeldungen, wenn die Deklaration der API-Funktionen nicht kompatibel ist. Der Teufel steckt aber im Detail, denn es können auch Korrekturen der Datentypen etwa von Parametern erforderlich sein, da es sonst zu Laufzeitfehlern kommt.
- Wenn wir ActiveX-Steuerelemente von Drittherstellern nutzen, sind diese oft nur für die 32-Bit-Versionen von Office verfügbar. Gerade bei der Verwendung älterer Steuerelemente, die nicht mehr weiterentwickelt werden, ist oft das Ersetzen des vollständigen Steuerelements durch eine Alternative notwendig.
- Wenn wir COM-DLLs oder COM-Add-Ins nutzen, gelten die gleichen Regeln. Diese sind entweder für 32-Bit oder für 64-Bit entwickelt worden und funktionieren nur mit der jeweiligen Office-Version zusammen. Wer selbst solche Elemente entwickelt, beispielsweise mit twinBASIC, VB6 oder Visual Studio .NET, hat immerhin die Möglichkeit, die COM-DLLs oder COM-Add-Ins für die entsprechende Version zu kompilieren.

Möglichkeiten zur Prüfung auf 32-Bit oder 64-Bit

In den folgenden Abschnitten schauen wir uns verschiedene Möglichkeiten an, um herauszufinden, ob Office in der 32-Bit- oder in der 64-Bit-Version installiert ist:

- Über die Office-Anwendung selbst herausfinden
- Bit-Version mit VBA ermitteln

32-/64-Bit über die Benutzeroberfläche identifizieren

Alle Office-Anwendungen stellen die Information bereit, ob es sich dabei um die 32-Bit- oder 64-Bit-Version handelt. Diese finden wir aber nicht direkt, sondern wir müssen ein wenig graben. Dazu klicken wir zunächst im Ribbon auf **Datei**. Dies öffnet das Datei-Menü, wo wir beispielsweise unter **Microsoft Access** den Eintrag **Konto** aufklappen (unter Outlook heißt der Eintrag **Office-Konto**). Hier finden wir nun eine Schaltfläche namens **Info zu Access** (siehe Bild 1).

Klicken wir diese an, erscheint ein weiterer Dialog mit einigen zusätzlichen Informationen, zum Beispiel der genauen Versionsangabe. Dahinter finden wir schließlich die Angabe, dass wir es hier mit der 32-Bit-Version von Access zu tun haben (siehe Bild 2).

32-/64-Bit per VBA ermitteln

Die zweite Version erfordert keine Suche in der Benutzeroberfläche der jeweiligen Office-Anwendung. Stattdessen verwenden wir VBA, um die gewünschte Information zu finden.

Allerdings gibt es auch hier keine einfache, direkte Möglichkeit, um herauszufinden, ob die Office-Anwendung in der 32-Bit- oder 64-Bit-Version vorliegt.

Es gibt also keine Eigenschaft wie die **Version**-Eigenschaft des **Application**-Objekts, die uns für Office 365 beispielsweise den Wert **16.0** zurückliefert:

```
? Application.Version
16.0
```

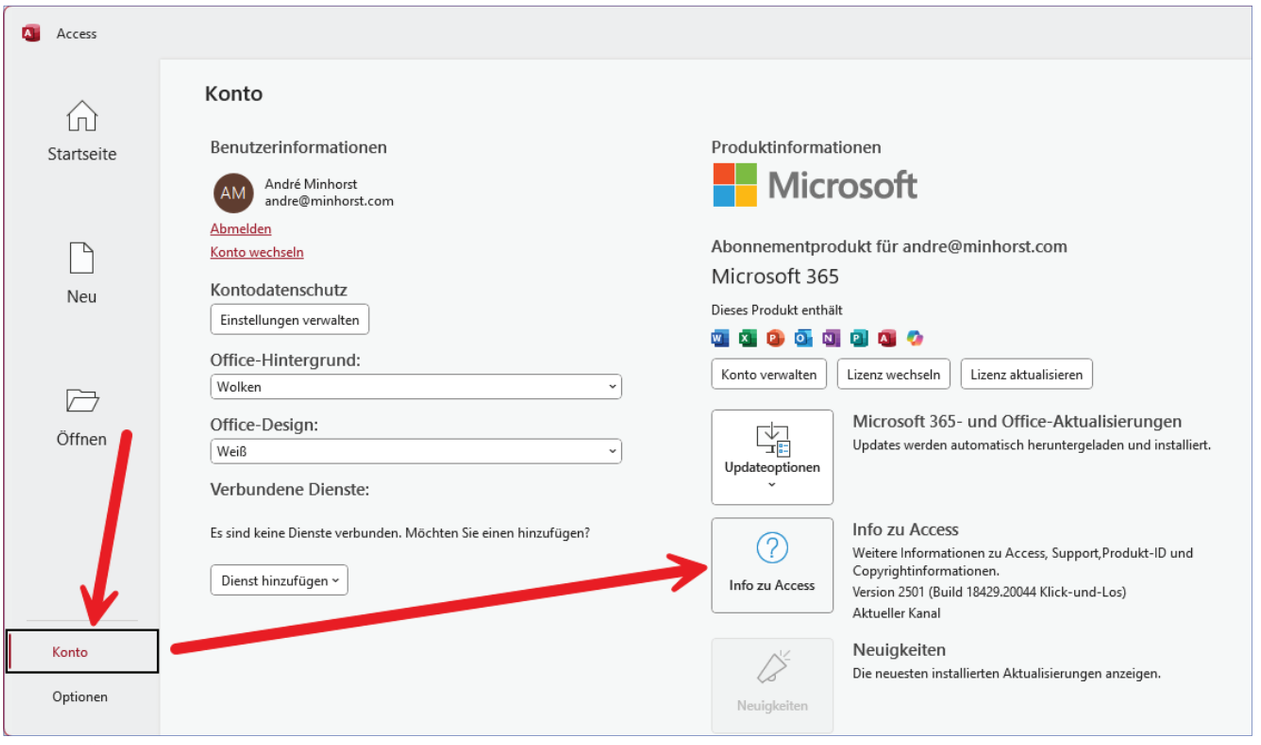


Bild 1: Öffnen der Infos zur jeweiligen Office-Anwendung, hier für Access

Dateien mit VBA-Bordmitteln verwalten

Die Verwaltung von Dateien per Code ist öfter gefragt, als man denkt. Damit lassen sich Verzeichnisse erstellen, auslesen und entfernen, Dateien erstellen, löschen und bearbeiten und vieles mehr. Leider sind die Bordmitteln von VBA hier teilweise ein wenig sperrig. Allerdings sind sie für einfache Aufgaben durchaus ausreichend und erfordern nicht den Einsatz einer zusätzlichen Bibliothek wie der Scripting Runtime, wie es beispielsweise beim FileSystemObject der Fall ist. Also schauen wir uns in diesem Artikel einmal die Elemente von VBA an, mit denen wir auf die Elemente des Dateisystems zugreifen können und zeigen, was sich damit alles anstellen lässt. Schließlich schauen wir aber auch noch kritisch auf mögliche Nachteile.

Definitionen

In diesem Artikel werden wir die verschiedenen Elemente des Dateisystems benennen. Dabei wollen wir folgende Elemente definieren: Ein Pfad ist der gesamte Pfad von der Wurzel bis zum Dateinamen inklusive Dateiendung, also zum Beispiel **c:\Verzeichnis1\Datei1.txt**, oder bis zu einem Verzeichnis (**c:\Verzeichnis1**). Ein Verzeichnis ist das darin enthaltene **Verzeichnis1**. Eine Datei oder Dateiname ist **Datei1.txt**.

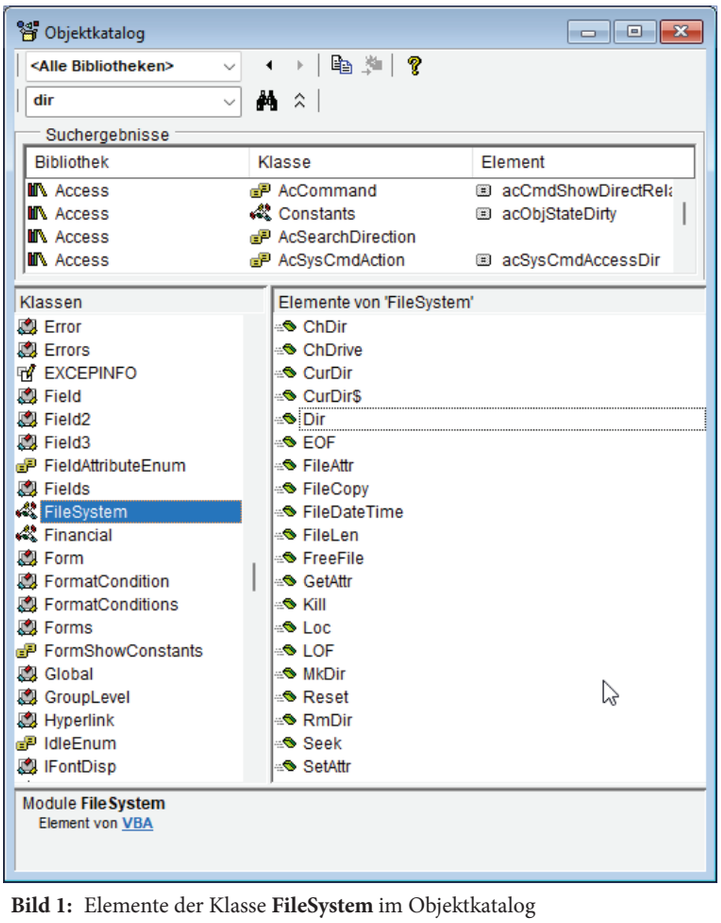


Bild 1: Elemente der Klasse **FileSystem** im Objektkatalog

Die FileSystem-Klasse und Alternativen

Eine der bekanntesten Anweisungen zum Arbeiten mit dem Dateisystem ist vermutlich die **Dir()**-Funktion. Suchen wir im Objektkatalog nach dieser, finden wir schnell heraus, dass sie zu einer Klasse namens **FileSystem** gehört (siehe Bild 1).

Diese enthält noch einige weitere Elemente, von denen man vermutlich nicht alle auf dem Schirm hat.

In diesem Artikel werden wir uns diese im Detail ansehen und verschiedene praktische Lösungen zeigen, die sich damit realisieren lassen – ganz ohne weitere Bibliotheken.

Allerdings wollen wir alternative Möglichkeiten, die überdies in vielen Fällen leichter

handzuhaben sind, nicht außen vor lassen. Deshalb gehen wir in einem weiteren Artikel beispielsweise auf das **FileSystemObject** ein, einer Klasse der Bibliothek **Dateimanagement mit dem FileSystemObject verwalten** (**www.vbentwickler.de/455**).

Übersicht über die Klasse FileSystem

Nun allerdings schauen wir uns erst einmal die grundlegenden Funktionen der verfügbaren Elemente zum Arbeiten mit Dateien und Verzeichnissen an.

Hier finden wir die folgenden Elemente:

- **ChDir**: Ändert das aktuelle Arbeitsverzeichnis. Erwartet den einzustellenden Pfad als Parameter, zum Beispiel **"C:\"**.
- **ChDrive**: Ändert das aktuelle Laufwerk. Erwartet das Ziellaufwerk als Parameter, zum Beispiel **"C"**.
- **CurDir**: Gibt das aktuelle Arbeitsverzeichnis aus. Bei Office 365 wird hier standardmäßig das Verzeichnis **C:\Users\[Benutzername]\AppData\Local\Temp** ausgegeben.
- **CurDir$**: Arbeitet wir **CurDir**, liefert jedoch das Ergebnis direkt als String zurück statt als Variant. Dies ist gegebenenfalls performanter, wenn das Ergebnis einer **String**-Variablen zugewiesen werden soll.
- **Dir**: Sucht Dateien oder Verzeichnisse und gibt ihren Namen zurück. Der erste Parameter erwartet das zu untersuchende Verzeichnis oder die Datei und der zweite kann den Dateityp einschränken. **Dir("c:\")** liefert beispielsweise die erste Datei des Verzeichnisses **c:**.
- **EOF**: Abkürzung für **End Of File**. Erwartet die Angabe einer Dateinummer einer mit **Open** geöffneten Datei. Für diese Datei wird angegeben, ob beim Lesen das Ende der Datei erreicht wurde.
- **FileAttr**: Gibt den Dateiattributwert für eine mit Open geöffnete Datei zurück. Erwartet die Dateinummer als ersten Parameter und den Zugriffsmodus als zweiten Parameter. Hier kommen die Werte **1** (Zugriffsmodus) und **2** (Dateizugriffsnummer) vor. **1** liefert also den Zugriffsmodus zurück, **2** die Dateizugriffsnummer.
- **FileCopy**: Kopiert eine Datei von einem Ort zum anderen. Dabei wird als erster Parameter der Pfad zur Quelldatei angegeben und als zweiter Parameter der Pfad der zu erstellenden Datei.
- **FileDateTime**: Gibt das Datum und die Uhrzeit der letzten Änderung einer Datei oder eines Verzeichnisses zurück. Als Parameter erwartet die Funktion den Pfad zu der zu untersuchenden Datei.
- **FileLen**: Gibt die Größe einer Datei oder eines Verzeichnisses (immer 0) in Bytes zurück und erwartet den Pfad zu der zu untersuchenden Datei als Parameter.
- **FreeFile**: Gibt eine verfügbare Dateinummer zurück, die wir mit der **Open**-Anweisung nutzen können, um eine Datei zu öffnen. Keine Parameter.
- **GetAttr**: Liest die Attribute einer Datei oder eines Verzeichnisses aus.
- **Kill**: Löscht die mit dem Parameter angegebene Datei.
- **Loc**: Gibt die Position innerhalb einer mit **Open** geöffneten Datei aus.
- **LOF**: Abkürzung für **Length Of File**. Liefert die Länge einer geöffneten Datei in Bytes. Erwartet die Dateinummer als Parameter.
- **MkDir**: Erstellt ein Verzeichnis. Das Verzeichnis muss als Parameter angegeben werden. Das übergeordnete Verzeichnis muss bereits existieren.

- **Reset**: Schließt alle mit Open geöffneten Dateien.
- **RmDir**: Löscht ein Verzeichnis, sofern dieses leer ist. Erwartet den Pfad des zu löschenden Verzeichnisses.
- **Seek**: Setzt die aktuelle Schreibposition innerhalt einer Datei oder gibt diese zurück. Der erste Parameter erwartet die Dateinummer. Wenn der zweite Parameter gesetzt ist, wird die Schreibposition an die angegebenen Position gesetzt. Ist der zweite Parameter leer, wird die Schreibposition zurückgegeben.
- **SetAttr**: Setzt Attribute für eine Datei oder ein Verzeichnis. Der erste Parameter nimmt den Pfad der Datei oder des Verzeichnisses entgegen, der zweite den zu setzenden Wert für die Attribute. Es können nur die Attribute **vbNormal**, **vbHidden** oder **vbReadOnly** hinzugefügt oder entfernt werden.
- **Name**: Ist keine Methode der Klasse **FileSystem**, aber wir gehen doch in diesem Artikel auf diese Anweisung ein, mit der man Dateien oder Verzeichnisse umbenennen kann.

Arbeitsverzeichnis wechseln

Mit der Funktion **ChDir** können wir das Arbeitsverzeichnis wechseln. Mit **CurDir** ermitteln wir das aktuelle Arbeitsverzeichnis.

Mit dem folgenden Beispiel lassen wir uns das Arbeitsverzeichnis vor der Änderung ausgeben, ändern es dann und geben dann das geänderte Arbeitsverzeichnis aus:

```
Public Sub Beispiel_ChDir()
    Debug.Print "Verzeichnis vorher: " & CurDir
    ChDir CurrentProject.Path
    Debug.Print "Verzeichnis nachner: " & CurDir
End Sub
```

Performanceunterschied CurDir und CurDir$

Um zu prüfen, wie sich die Performance bei den beiden Funktionen **CurDir** und **CurDir$** verhält, haben wir die folgenden beiden Funktionen jeweils 10.000 Mal durchlaufen lassen:

```
Public Function CurDirVariant()
    Dim str As String
    str = CurDir
End Function

Public Function CurDirString()
    Dim str As String
    str = CurDir$
End Function
```

CurDir$ war dabei im Schnitt ca. 5% schneller als **CurDir**.

Zugriffsmodus einer Datei ermitteln

Wenn wir eine Datei mit Open geöffnet haben, können wir anschließend mit **FileAttr** prüfen, in welchem Zugriffsmodus sich die Datei befindet.

Im folgenden Beispiel öffnen wir eine bereits vorhandene Textdatei und lesen mit **FileAttr(intFileNum, 1)** den Zugriffsmodus aus:

```
Public Sub Dateizugriffsmodus()
    Dim intFileNum As Integer
    Dim intMode As Integer
    Dim strMode As String

    intFileNum = FreeFile
    Open CurrentProject.Path & "\Test.txt" _
        For Input As #intFileNum

    intMode = FileAttr(intFileNum, 1)
    Select Case intMode
        Case 1
```

```
            strMode = "Input (Lesemodus)"
        Case 2
            strMode = "Output (Schreibmodus)"
        Case 4
            strMode = "Random (Zufälliger Zugriff)"
        Case 8
            strMode = "Append (Anfügemodus)"
        Case 32
            strMode = "Binary (Binärmodus)"
    End Select
    Debug.Print "Datei ist im folgenden Modus geöffnet: " _
        & strMode
    Close #intFileNum
End Sub
```

Kopieren von Dateien

Um eine Datei von einem Ort zum anderen zu kopieren, verwenden wir die Anweisung **FileCopy**. Diese erwartet die beiden Parameter Quellpfad und Zielpfad.

Ein einfaches Beispiel kopiert die Datei **Test.txt** in eine neue Datei namens **Text_Kopie.txt**:

```
Public Sub DateienKopieren()
    Dim strQuelle As String
    Dim strZiel As String
    strQuelle = CurrentProject.Path & "\Test.txt"
    strZiel = CurrentProject.Path & "\Test_Kopie.txt"
    FileCopy strQuelle, strZiel
End Sub
```

Datum und Zeit der letzten Änderung einer Datei ermitteln

Das Datum der letzten Änderung können wir mit der Funktion **FileDateTime** ermitteln:

```
Public Sub Dateidatum()
    Dim strPfad As String
    strPfad = CurrentProject.Path & "\Test.txt"
    Debug.Print FileDateTime(strPfad)
End Sub
```

Das Ergebnis lautet beispielsweise:

```
08.01.2025 19:53:24
```

Größe einer Datei ermitteln

Mit der **FileLen**-Funktion ermitteln wir die Größe einer Datei in Byte:

```
Public Sub Dateigroesse()
    Dim strPfad As String
    strPfad = CurrentProject.Path & "\Test.txt"
    Debug.Print FileLen(strPfad)
End Sub
```

Dateinummer zum Öffnen einer Datei ermitteln

Mit der Funktion **FreeFile** erhalten wir eine Nummer zurück, die aktuell nicht in Zusammenhang mit einer geöffneten Datei in Verwendung ist, um eine aktuell geöffnete Datei zu referenzieren.

Im folgenden Beispiel schreiben wir die nächste freie Nummer in die Variable **intFileNum**, öffnen damit eine Datei, ermitteln dann mit **FreeFile** die nächste freie Nummer, schließen die Datei wieder und geben erneut das Ergebnis von **FreeFile** aus. Hier ist die Prozedur dazu:

```
Public Sub Dateinummer()
    Dim intFileNum As Integer

    intFileNum = FreeFile
    Debug.Print "Aktuelle Dateinummer: " & intFileNum

    Open CurrentProject.Path & "\Test.txt" _
        For Input As #intFileNum

    Debug.Print "Nächste Dateinummer: " & FreeFile
    Close #intFileNum
```

```
    Debug.Print "Nächste Dateinummer nach dem Schließen: _
        " & FreeFile
End Sub
```

Das Ergebnis sah so aus:

```
Aktuelle Dateinummer: 1
Nächste Dateinummer: 2
Nächste Dateinummer nach dem Schließen: 1
```

Ausführliche Beispiele für den Umgang mit Anweisungen wie Open, Close, Print et cetera beschreiben wir in einem eigenen Artikel namens **VBA und Textdateien: Alles über Open, Close, Print und Co.** (**www.vbentwickler.de/454**).

Alle Dateiattribute ausgeben

Mit der folgenden Prozedur nutzen wir die Funktion **GetAttr**, um den Zahlenwert auszugeben, der die Dateiattribute des angegebenen Pfades repräsentiert. Hier geben wir zunächst einmal eine Zeile im Direktbereich aus für jedes Attribut, das zutreffend ist.

Dazu vergleichen wir den jeweiligen Wert von lngAttribut über eine bitweise **And**-Operation mit dem jeweiligen Zahlenwert und geben einen Text aus, wenn das jeweilige Attribut zutreffend ist:

```
Public Sub AlleDateiattribute()
    Dim strPfad As String
    Dim lngAttribute As Long
    strPfad = CurrentProject.Path & "\Test.txt"
    lngAttribute = GetAttr(strPfad)
    If lngAttribute = 0 Then
        Debug.Print "vbNormal"
    End If
    If (lngAttribute And 1) = 1 Then
        Debug.Print "vbReadOnly"
    End If
    If (lngAttribute And 2) = 2 Then
        Debug.Print "vbHidden"
    End If
    If (lngAttribute And 4) = 4 Then
        Debug.Print "vbSystem"
    End If
    If (lngAttribute And 16) = 16 Then
        Debug.Print "vbDirectory"
    End If
    If (lngAttribute And 32) = 32 Then
        Debug.Print "vbArchive"
    End If
End Sub
```

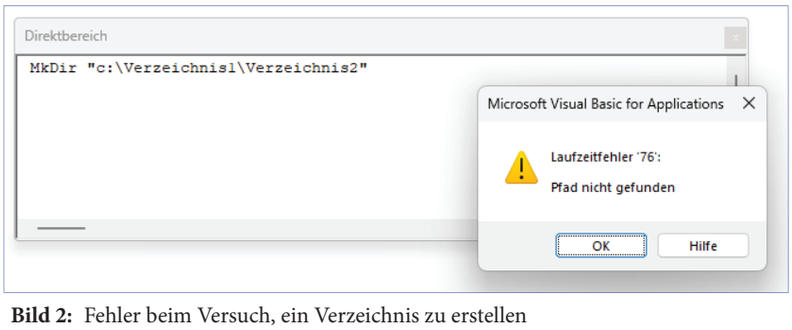


Bild 2: Fehler beim Versuch, ein Verzeichnis zu erstellen

Ermitteln, ob ein Pfad eine Datei oder ein Verzeichnis ist

Damit lässt sich dann beispielsweise herausfinden, ob ein Pfad eine Datei oder ein Verzeichnis ist.

```
Public Function IstDatei(strPfad As String) As Boolean
    Dim lngAttribute As Long
    lngAttribute = GetAttr(strPfad)
        If lngAttribute And vbDirectory = _
            vbDirectory Then
            IstDatei = True
        End If
End Function
```

Verzeichnis erstellen mit MkDir

Verzeichnisse erstellen wir mit der **MkDir**-Anweisung. Diese erwartet

VBA und Textdateien: Alles über Open, Close und Co.

VBA bietet einige Möglichkeiten für das Erstellen, Füllen, Ändern und Auslesen von Textdateien. Mit diesen kann man zwar nicht alle Aufgaben bewältigen, aber für einfache Zwecke lohnt sich ein Blick auf diese Sammlung. Wir meinen damit Anweisungen wie Open, Close, Print, Input und einige weitere, die spannenderweise im Objektkatalog noch nicht einmal erwähnt werden. Wir werden uns in diesem Artikel eingehend mit diesen Anweisungen beschäftigen und in verschiedenen Beispielen zeigen, wie sich damit immer wiederkehrende Aufgaben wie das Schreiben oder Lesen von Textdateien leicht bewältigen lässt.

Im Artikel **Dateien mit VBA-Bordmitteln verwalten** (**www.vbentwickler.de/453**) haben wir uns angesehen, welche Elemente die Klasse **FileSystem** für uns bereithält.

Hier finden wir einige Anweisungen, mit denen wir Verzeichnisse erstellen und löschen, Dateien und Verzeichnisse durchlaufen und weitere Dinge erledigen können.

Die Klasse **FileSystem** enthält jedoch auch einige Elemente, die wir in diesem Zusammenhang gar nicht benötigen. Sie sind vielmehr für den Einsatz mit einigen Anweisungen vorgesehen, die wir nicht in der Klasse **FileSystem** und sogar noch nicht einmal überhaupt im Objektkatalog des VBA-Editors finden.

Dabei geht es um solche Anweisungen wie Open, Close, Write et cetera:

- **Close**: Schließt eine oder mehrere zuvor mit Open geöffnete Dateien.
- **Get**: Dient dem Binärzugriff auf Dateien und wird in diesem Artikel nicht erläutert.
- **Input #**: Erlaubt das Einlesen der
- **Line Input #**: Erlaubt das zeilenweise Einlesen von Inhalten einer Textdatei.
- **Open**: Öffnet eine Datei für den lesenden oder schreibenden Zugriff oder erstellt eine neue Datei für den schreibenden Zugriff.
- **Print #**: Schreibt Freitext in Dateien.
- **Put**: Dient dem Binärzugriff auf Dateien und wird in diesem Artikel nicht erläutert.
- **Seek**: Ruft die aktuelle Position beim Binärzugriff ab oder setzt diese. Wird in diesem Artikel nicht erläutert.
- **Width**: Erlaubt die Begrenzung der Zeilenlänge beim Schreiben mit **Print**.
- **Write #**: Schreibt strukturierte Daten in Dateien.

Neben diesen Anweisungen erläutern wir in diesem Artikel auch noch diese Elemente, die Teil der Klasse **FileSystem** sind:

- **EOF**: Abkürzung für **End Of File**. Erwartet die Angabe einer Dateinummer einer mit **Open** geöffneten Datei. Für diese Datei wird angegeben, ob beim Lesen das Ende der Datei erreicht wurde.
- **FileLen**: Gibt die Größe einer Datei oder eines Verzeichnisses (immer 0) in Bytes zurück und erwartet den Pfad zu des Elements als Parameter.

- **FreeFile**: Gibt eine verfügbare Dateinummer zurück, die wir mit der **Open**-Anweisung nutzen können, um eine Datei zu öffnen. Keine Parameter.
- **Loc**: Gibt die Position innerhalb einer mit **Open** geöffneten Datei aus.
- **LOF**: Abkürzung für **Length Of File**. Liefert die Länge einer geöffneten Datei in Bytes. Erwartet die Dateinummer als Parameter.
- **Reset**: Schließt alle mit Open geöffneten Dateien.
- **Seek**: Setzt die aktuelle Schreibposition innerhalt einer Datei oder gibt diese zurück. Der erste Parameter erwartet die Dateinummer. Wenn der zweite Parameter gesetzt ist, wird die Schreibposition an die angegebenen Position gesetzt. Ist der zweite Parameter leer, wird die Schreibposition zurückgegeben.

Öffnen oder Erstellen einer Datei mit Open

Wenn wir bestehende Datei öffnen oder eine neue Datei erstellen wollen, benötigen wir die **Open**-Anweisung.

Die **Open**-Anweisung ist nicht wie herkömmliche VBA-Anweisungen aufgebaut, die einfach aufgerufen werden und gegebenenfalls noch einen oder mehrere Parameter entgegennehmen.

Die **Open**-Anweisung besteht vielmehr aus mehreren Teilen, die nacheinander notiert werden müssen. Hier ist die grundlegende Syntax:

```
Open pathname For mode [ Access access ] [ lock ] As [ # ] filenumber [ Len = reclength ]
```

Wir finden hier verschiedene Bestandteile, die wir wie Pflichtparameter oder optionale Parameter angeben können:

- **pathname**: Pfad zu der zu erstellenden oder zu öffnenden Datei
- **mode**: Modus, in dem die Datei geöffnet werden soll. Hier stehen die Werte **Append**, **Binary**, **Input**, **Output** oder **Random** zur Verfügung.
- **access**: Wird beginnend mit dem Schlüsselwort **Access** mit einem der Werte **Read**, **Write** oder **Read Write** kombiniert.
- **lock**: Gibt an, ob und wie andere Prozesse noch auf die geöffnete Datei zugreifen können sollen. Hier gibt es die Werte **Shared**, **Lock Read**, **Lock Write** und **Lock Read Write**.
- **filenumber**: Enthält eine Nummer für diesen Dateizugriff, der auch von folgenden Anweisungen zum Lesen oder Schreiben der Datei verwendet werden muss. Der Wert kann von **1** bis **511** reichen. Mit der Funktion **FreeFile** können wir die nächste Nummer ermitteln, die in der aktuellen Session noch nicht für den Zugriff auf eine Datei verwendet wird.
- **reclength**: Eine Zahl, mit der die Datensatzgröße bei der Zugriffsart **Random** festgelegt wird.

Für einfache lesende oder schreibende Zugriff benötigen wir nicht unbedingt alle dieser Parameter.

Schließen einer Datei mit Close

Eine geöffnete oder neu erstellte Datei kann je nach Modus lesend und/oder schreibend verwendet werden.

Danach sollte sie jedoch auch wieder geschlossen werden. Das erledigt die Anweisung **Close**.

Diese Anweisung erwartet keinen, einen oder mehrere Parameter. Ohne Parameter schließt die **Close**-Anweisung alle aktuell geöffneten Dateien wieder.

Dateien werden mit der **Open**-Anweisung unter Angabe einer Dateinummer geöffnet. Diese können wir auch angeben, wenn wir diese Datei gezielt wieder schließen möchten.

Wenn wir mehrere Dateien geöffnet haben, können wir diese entweder durch mehrfaches Aufrufen der **Close**-Methode unter Angabe der jeweiligen Dateinummer wieder schließen. Wir können die Dateinummern aber auch als kommaseparierte Liste übergeben.

Hier ein Beispiel, das eine einfache Datei zum Schreiben anlegt oder öffnet:

```
Dim intFile As Integer
intFile = FreeFile
Open CurrentProject.Path & "\Test.txt" For Output As
#intFile
Close #intFile
```

Wenn wir sicher sind, dass in der aktuellen Session nur eine Datei geöffnet wird, können wir auch einen festen Zahlenwert nutzen:

```
Open CurrentProject.Path & "\Test.txt" For Output As #1
Close #1
```

Wenn wir zwei Dateien gleichzeitig benötigen, können wir zwei Mal die **FreeFile**-Methode nutzen und die Dateien nacheinander öffnen und wieder schließen:

```
Dim intFile1 As Integer
Dim intFile2 As Integer
intFile1 = FreeFile
Open CurrentProject.Path & "\Test1.txt" For Output As
#intFile1
intFile2 = FreeFile
Open CurrentProject.Path & "\Test2.txt" For Output As
#intFile2
Close #intFile1
Close #intFile2
```

Wichtig ist, dass wir den zweiten Aufruf von **FreeFile** erst starten, wenn die erste Datei bereits geöffnet ist. Ansonsten würde **FreeFile** wieder den gleichen Wert wie beim ersten Aufruf liefern, da dieser noch nicht verwendet wurde.

Wir können die beiden letzten Anweisungen auch zusammenfassen:

```
Close #intFile1, #intFile2
```

Die vorherigen Beispiele legen immer leere Textdateien an. Da wir diese nicht mit Inhalt füllen, haben diese immer eine Größe von 0 Bytes.

Alle Dateien mit Reset schließen

Die Alternative zum Schließen aller geöffneten Dateien mit der **Close**-Anweisung ist die **Reset**-Anweisung.

Einfache Textdateien schreiben

Die folgende Prozedur erstellt eine Textdatei und füllt diese mit fünf Zeilen Text.

Dabei schreiben wir zuerst den Pfad zu der zu erstellenden Datei in die Variable **strDateipfad**. Wir ermitteln mit der Funktion **FreeFile** die nächste freie Dateinummer und schreiben diese in die Variable **intDateinummer**. Diese verwenden wir in der **Open**-Anweisung, der wir als Erstes mit **strDateipfad** den Namen der zu erstellenden Datei übergeben.

Danach folgt der Teil **For Output**, der angibt, das wir in die Datei schreiben wollen. Schließlich leitet das Schlüsselwort **As** die Angabe der Dateinummer ein. Die Angabe des Raute-Zeichens (#) ist hier übrigens nicht verpflichtend – im Gegensatz zu den späteren Vorkommen in Zusammenhang mit den **Print**- und **Close**-Anweisungen. Der Übersichtlichkeit halber wollen wir das Raute-Zeichen jedoch konsequent überall benutzen, wo die Dateinummer zum Einsatz kommt.

Automation mit Zapier, Make und Co.

Es gibt immer mehr Möglichkeiten, sogenannte SaaS (Software as a Service) durch Automationen zu verknüpfen. SaaS sind zum Beispiel Calendly zur Terminplanung, Online-Buchhaltungsprogramme wie lexoffice, Projektmanagement-Tools wie Trello, E-Mail-Marketing-Tools, E-Commerce-Anwendungen, CRM-Systeme, Business Intelligence Tools wie Power BI oder täglich verwendete Tools wie Microsoft Teams, Zoom oder die Google Suite. Diese Tools haben wichtige Gemeinsamkeiten: Sie sind allesamt online verfügbar und somit über den Browser steuerbar. Und sie bieten Automations-Schnittstellen an, mit denen sie einerseits Trigger auslösen können und damit andererseits verschiedene Aktionen anstoßen. Da Du als Leser dieses Magazins zweifelsohne früher oder später mit Tools wie diesen in Verbindung kommen wirst, wollen wir Dir zeigen, wie Du Automationen zwischen diesen Tools erstellen kannst. Dabei wollen wir in diesem Einführungsartikel zwei der bekanntesten Tools ansehen, und zwar Zapier und Make.com. Microsoft Power Automate haben wir bereits früher einmal in diesem Magazin vorgestellt.

Wozu überhaupt Automation?

Diese Frage wirst Du Dir vermutlich nicht stellen, denn mit zum Beispiel mit VBA-Code automatisierst Du ja selbst auch bereits Deine Access-Anwendungen. Du definierst Ereignisprozeduren, die Du für Ereigniseigenschaften von Formularen oder Steuerelementen hinterlegst und führst bestimmte Aktionen aus, wenn der Benutzer diese betätigt.

Vielleicht verwendest Du sogar anwendungsübergreifende Automationen, wo Du beispielsweise von Access aus andere Anwendungen wie Excel, Word oder Outlook so programmierst, dass Du von Access aus auf Aktionen in diesen Anwendungen reagieren kannst.

Software as a Service (SaaS)

Seit einigen Jahren gibt es immer mehr Anwendungen, die nicht wie Microsoft oder andere Desktop-Anwendungen komplett auf Deinem Rechner ablaufen, sondern die auf einem Server liegen und die Du über den Internetbrowser bedienen kannst.

Dazu gehören Anwendungen aus den folgenden Bereichen:

- Buchhaltung und Finanzen (Lexware Office, Datev, Easybill, ...)
- Vertrieb und Kundenmanagement (CRM-Systeme)
- Projektmanagement (Trello, Asana, Notion, ...)
- Schulungsplattformen/E-Learning
- E-Mail-Marketing-Tools zum Versenden von Newslettern
- Online-Shops (Shopware, Shopify, Magenta, ...)
- Business Intelligence (Power BI)
- Kundenbetreuung (Zendesk, Intercom)
- Kommunikation und Videokonferenzen (Microsoft Teams, Slack, ...)

Vielleicht bist Du schon seit Ewigkeiten Access- oder VB-Entwickler und stehst auf dem Standpunkt, dass Du bis zu Deinem Ruhestand auch noch mit den al-

ten Tools auskommen wirst. Vielleicht steckst Du aber auch schon mittendrin und nutzt schon Tools für den einen oder anderen Zweck.

Dann hast Du vermutlich auch schon einmal in irgendeinem meiner Artikel ein Beispiel dafür gesehen, dass solche Online-Softwareprodukte Schnittstellen anbieten – und damit meinen wir nicht solche, wo Du Daten im CSV-, Excel- oder auch XML-Format hochladen oder exportieren konntest, um diese dann weiterzuverarbeiten.

Wir meinen eher sogenannte Rest-APIs. Dabei handelte es sich um programmierbare Endpunkte dieser Anwendungen, die wir alternativ zu ihrer Benutzeroberfläche nutzen können, um Daten auszutauschen, Prozesse anzustoßen oder auch um auf Ereignisse zu reagieren.

Wenn Du mir (André Minhorst) schon länger folgst, hast Du sicher in dem einen oder anderen Magazin schon einmal einen Artikel gelesen, in dem ich zeige, wir man solche Rest-APIs auch von Access auf per VBA ansprechen kann, um Daten hochzuladen, abzurufen oder auch Aktionen anzustoßen. Dabei haben wir uns beispielsweise mit dem Anlegen von Rechnungen in Lexware Office, dem Abfragen oder Erstellen von Auktionen bei eBay, dem Einlesen von Buchinformationen von Amazon und weiteren Beispielen beschäftigt.

Zapier, Make.com und Co.

Das erste Automationstool, das größere Verbreitung erlangte, war IFTTT (If This Than That). Schon damit konnte man zum Beispiel Automationen schreiben, die durch Trigger ausgelöst wurden und bestimmte Aktionen ausgelöst haben.

Aktuell gibt es verschiedene Anbieter, die alle Vor- und Nachteile haben. Die bekanntesten sind wohl Zapier und Make.com (früher Integromat) sowie Power Automate von Microsoft. Es gibt aber auch noch Alternativen.

Hier ist zu prüfen, welche Funktionen Dir wichtiger sind:

Bekanntere Tools haben meist eine größere Anzahl von Integrationen in die SaaS-Anwendungen. Ihre Nutzung hat einen entsprechenden Preis, wobei es verschiedene Grundtarife gibt und Zusatzkosten, wenn mehr als die im Grundtarif enthaltenen Aufrufe anfallen.

Weniger bekannte Tools haben vielleicht weniger Integrationen, glänzen aber durch spezielle Funktionen. Allerdings haben wir uns im Rahmen dieses Artikels nur die beiden Lösungen von Zapier und Make.com angesehen.

Rest-API für Interaktion zwischen Internetanwendungen

Nun ist es sicher praktisch, wenn wir von unserer heimischen Access-Datenbank Daten mit all diesen praktischen Softwarelösungen austauschen und Prozesse anstoßen können. Wir sollten uns jedoch allmählich mit dem Gedanken anfreunden, dass es da draußen richtig gute Lösungen gibt.

Man hätte sich vor dem Jahr 2020 noch herausreden können, dass schließlich alle Mitarbeiter, die irgendwie auf Kundendaten et cetera zugreifen müssen, im gleichen Büro sitzen und über das Netzwerk auf die Daten zugreifen können, die sie benötigen. Aber spätestens seit der Corona-bedingten Homeoffice-Offensive macht es Sinn, verschiedene Prozesse über Softwareanwendungen zu erledigen, wie wir sie oben beschrieben haben.

Und damit kommen wir zum eigentlichen Thema dieses Artikels: Der Automation solcher Softwareanwendungen mit Tools wie Zapier.

Rest-APIs bieten grundsätzlich zwei verschiedene Arten von Interaktionsmöglichkeiten:

- solche, bei denen wir den Vorgang über die Rest-API auslösen oder
- das Reagieren auf Trigger, die durch die Software ausgelöst werden.

Zugriff auf die Rest-API durch uns

Weiter oben haben wir beschrieben, dass wir bereits verschiedene Lösungen programmiert haben, wo wir von Access/VBA aus auf die Rest-API einer Software im Web zugegriffen haben.

Dabei konnten wir wiederum verschiedene Aktionen durchführen, die sich im Wesentlichen auf die folgenden Arten aufteilen:

- Ändern von Daten der Software, also Hochladen von Daten, Ändern von Daten oder Löschen von Daten.
- Aufrufen von weiteren Aktionen wie beispielsweise das Versenden einer E-Mail, das Erstellen eines Rechnungsdokuments et cetera.
- Abrufen von Informationen wie zum Beispiel Artikeldaten

Reagieren auf Trigger der Software

Eine weitere Funktion, die SaaS bieten, sind die sogenannten Trigger. Diese können nicht über die Rest-API genutzt werden, da die Rest-API immer nur Aufrufe von außen entgegennimmt und gegebenenfalls noch Informationen zurückliefert.

Deshalb funktionieren Trigger intern andersherum: Wenn wir beispielsweise eine Automation programmieren wollen, bei der wir, wenn ein Kunde einen Termin in Calendly angelegt hat, automatisch einen Eintrag im CRM-Tool für diesen Kunden anlegen wollen, würden wir zwei Schritte definieren:

- ein Trigger, der durch das Anlegen des Termins in Calendly ausgelöst wird und
- eine Aktion, welche die Kundendaten aus diesem Trigger entnimmt und diesen in das CRM einträgt.

Aber wir kann unsere Automation nun überhaupt auf den Trigger reagieren? Dazu gibt es bestimmte Schnittstellen, die wie folgt funktionieren:

- Die erste Software, zum Beispiel Calendly, hat beispielsweise eine Schnittstelle für Zapier. Das bedeutet, dass jeder neue Termin, der mit Calendly erstellt wird, automatisch an Zapier geschickt wird.
- Zapier erhält damit auch Informationen über das Benutzerkonto des Anbieters, der das Anlegen neuer Termine mit Calendly ermöglicht.
- Damit prüft Zapier, wenn ein neuer Termin eingeht, ob es Automationen gibt (im Falle von Zapier »Zaps« genannt), die mit dem Konto des entsprechenden Benutzers verknüpft sind.
- Ist das der Fall, wird der Zap ausgeführt.
- Der Trigger liefert alle wichtigen Informationen über den neuen Calendly-Termin und stellt diese innerhalb des Zaps bereit.
- Die nachfolgenden Aktionen können nun auf die Daten des Termins zugreifen und diese nutzen, um beispielsweise mit einem Vorgang wie »Neuen Benutzer anlegen« den Eintrag für diesen Kunden im CRM anzulegen.

Das wäre die vereinfachte Version. Natürlich kann es auch einmal vorkommen, dass der Kunde bereits im

SQL Server: Fehlerbehandlung in DAO und ADODB

Die Fehlerbehandlung in VBA ist relativ einfach zu handhaben, sobald man einmal die grundlegenden Techniken kennt. Nicht viel anders sollte es eigentlich sein, wenn man die Daten nicht mehr aus Access-Tabellen bezieht, sondern aus SQL Server-Tabellen. Es gibt jedoch einige Unterschiede, die man kennen sollte. In diesem Artikel zeigen wir, welche Besonderheiten auftreten, wenn man nicht mehr auf Access-Tabellen zugreift, sondern auf verknüpfte SQL Server-Tabellen und dabei die Datenzugriffstechniken DAO- und ADODB nutzt. Im letzteren Fall gibt es sogar noch Unterschiede bezüglich der verwendeten Treiber/Provider.

Wenn wir mit VBA programmieren und es wird ein Laufzeitfehler ausgelöst, liefert dieser immer genau ein Fehlerobjekt namens **Err** zurück, das uns mit Eigenschaften wie **Number** oder **Descriptionen** Informationen über den Fehler liefert.

Wenn wir mit DAO auf die Daten eines SQL Servers oder einer anderen per ODBC-Tabellenverknüpfung eingebundenen Datenbanksystems zugreifen, erhalten wir ebenfalls eine Fehlermeldung mit **Err**.

Allerdings lautet diese immer gleich – nämlich den Fehler **ODBC-Aufruf fehlgeschlagen** mit der Nummer **3146**.

Damit kann man eigentlich nichts anfangen – da dieser Fehler immer in Zusammenhang mit dem Datenzugriff auftritt, wissen wir zwar grob, worum es geht, aber nicht genau, was der eigentliche Fehler ist.

Man kann zwar nun die auslösende SQL-Anweisung kopieren, sie im SQL Server Management Studio ausführen und erhält dann dort die eigentliche Fehlermeldung. Das hilft uns allerdings nicht, wenn wir im VBA-Code darauf reagieren wollen.

In diesem Artikel stellen wir Dir eine Möglichkeit vor, wie Du dennoch auf die Fehlerinformationen zugreifen kannst. Dazu gibt es in DAO und ADODB verschiedene Möglichkeiten.

Unterscheidung der verschiedenen Fehler-Klassen

Zuerst jedoch ein kleiner Blick auf das »normale« **Err**-Objekt. Dieses hat den Typ **ErrObject** und bietet die folgenden Eigenschaften und Methoden.

Hier zunächst die Eigenschaften:

- **Description**: Enthält die Fehlerbeschreibung als String.
- **HelpContext**: Gibt die ID des Hilfethemas zurück, das mit dem Fehler verbunden ist.
- **HelpFile**: Gibt den Pfad zur zugehörigen Hilfedatei zurück (falls vorhanden).
- **LastDLLError**: Gibt den letzten Windows-API-Fehlercode zurück (nur bei API-Aufrufen relevant).
- **Number**: Gibt die Fehlernummer zurück. **0** bedeutet, dass kein Fehler aufgetreten ist. Bei eingebauten Elementen sind die Fehlernummern positiv und ein- bis fünfstellig. Es gibt aber auch Fehlermeldungen mit großen negativen Fehlernummern.
- **Source**: Gibt die Quelle des Fehlers zurück (zum Beispiel **VBAProject**).

Und hier sind die beiden Methoden:

- **Clear**: Setzt alle Fehlerinformationen zurück (Number = 0, Description = "", et cetera).

- **Raise**: Erzeugt einen benutzerdefinierten Fehler mit einer bestimmten Fehlernummer.

Unter DAO gibt es dagegen wesentlich weniger Elemente. Die Fehlerklasse heißt hier auch nicht **ErrObject**, sondern **Error**. Hier gibt es nur (Beschreibung wie bei VBA):

- **Description**

- **Number**

- **Source**

Unter ADO finden wir wieder fast die gleichen Eigenschaften wie unter VBA, nämlich **Description**, **HelpContext**, **HelpFile**, **Number** und **Source**. Wir finden aber auch zwei weitere Eigenschaften:

- **NativeError**: Liefert uns die Server-spezifische Fehlernummer. So erhalten wir beispielsweise für den gleichen Fehler im VBA-Error die Nummer **-2147217873** und im ADODB-Error die Nummer **547**.

- **SQLState**: Gibt den SQL-Fehlercode im standardisierten 5-stelligen Zeichenformat zurück. Diese Codes folgen dem SQL-92-Standard und sind von der Datenbank unabhängig.

Fehler beim Zugriff mit DAO auswerten

Wenn beim Zugriff auf eine SQL Server-Datenbank über eine Tabellenverknüpfung oder andere Zugriffe per ODBC Fehler auftreten, können wir diese mit der herkömmlichen **Err**-Klasse nur bedingt analysieren.

Der Grund ist, dass wir mit der **Err**-Klasse nur Informationen zum letzten aufgetretenen Fehler erhalten. Das ist auch so, wenn wir per Tabellenverknüpfung auf eine per ODBC verknüpfte Datenbank zugreifen. Hier können wir dann jedoch die **Errors**-Auflistung von DAO hinzuziehen. Hier finden wir jedoch noch ein weiteres Problem vor, nämlich dass die **Err**-Klasse immer nur eine generische Meldung liefert, die keinen Hinweis auf den tatsächlichen Fehler enthält. Die folgenden Anweisungen sollten beispielsweise einen Fehler auslösen, weil wir dem Feld **AnlageID** keinen Wert zuweisen können:

```
Dim db As DAO.Database
Dim rst As DAO.Recordset
Set db = CurrentDb
db.Execute "INSERT INTO tblAnlagen(AnlageID) VALUES(1)",
dbFailOnError + dbSeeChanges
```

Wir erhalten dafür jedoch lediglich die Fehlermeldung aus Bild 1.

Fehler mit DAO.Errors auswerten

Es gibt jedoch nicht nur das **Err**-Objekt, um Fehler auszuwerten. Wir können auch die **DAO.Errors**-Auflistung bemühen, um den tatsächlich vom SQL Server gelieferten Fehler zu erhalten. Diese liefert uns Elemente des gleichen Typs wie unser bekanntes **Error**-

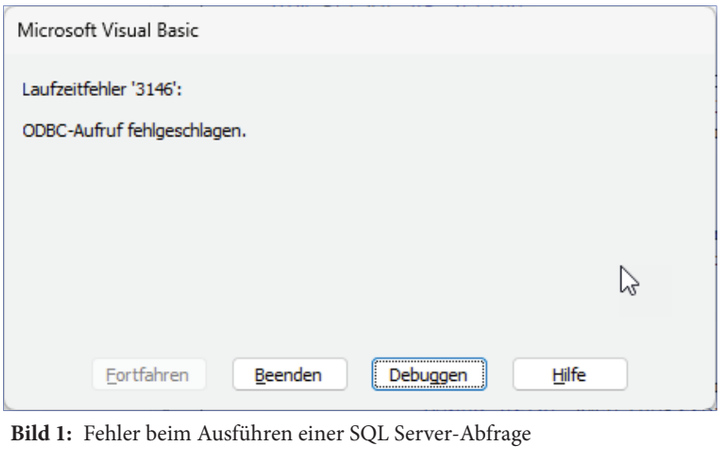


Bild 1: Fehler beim Ausführen einer SQL Server-Abfrage

Objekt. Diese können wir in einer Schleife über die **Errors**-Auflistung der DAO-Klasse durchlaufen:

```
For i = 0 To DAO.Errors.Count
    Debug.Print i, DAO.Errors(i).Description
Next i
```

Wir schauen uns diese Codezeilen einmal im Zusammenhang mit einer Prozedur an, die einen entsprechenden Fehler auslöst (siehe Listing 1). Hier erhalten wir die Fehlermeldungen, wie sie auch im SQL Server angezeigt werden:

```
[Microsoft][ODBC Driver 17 for SQL Server][SQL Server]Verletzung der PRIMARY KEY-Einschränkung "tblAnlagen$PrimaryKey". Ein doppelter Schlüssel kann in das dbo.tblAnlagen-Objekt nicht eingefügt werden. Der doppelte Schlüsselwert ist (1).

[Microsoft][ODBC Driver 17 for SQL Server][SQL Server]Die Anweisung wurde beendet.

ODBC-Aufruf fehlgeschlagen.
```

Fehlerbehandlung unter ADODB

Das ist bei Fehlern bei Datenbankzugriffen mit ADODB bereits etwas besser gelöst – hier erhalten wir eine aussagekräftige Fehlermeldung. Wir lösen in Listing 2 den gleichen Fehler aus wie in der DAO-Version, nur diesmal über die **Execute**-Methode der **Connection**-Klasse.

Bereits für **Err.Number** und **Err.Description** erhalten wir diese Meldung:

```
Err.Number: -2147217873
Err.Description: Ein expliziter Wert für die Identitätsspalte kann nicht in der tblAnlagen-Tabelle eingefügt werden, wenn IDENTITY_INSERT auf OFF festgelegt ist.
```

Die Ausgabe des Fehlers der **Errors**-Auflistung des **Connection**-Objekts liefert folgende Informationen:

```
Number:          -2147217873
Description:     Ein expliziter Wert für die Identitätsspalte kann nicht in der tblAnlagen-Tabelle eingefügt werden, wenn IDENTITY_INSERT auf OFF festgelegt ist.
NativeError:     544
Source:          Microsoft OLE DB Driver for SQL Server
SQLState:        23000
```

Unter **Number** und **Description** erhalten wir die gleichen Informationen wir über das **Err**-Objekt.

```
Public Sub DAOFehlerAnalysieren()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim i As Integer
    Set db = CurrentDb
    On Error Resume Next
    db.Execute "INSERT INTO tblAnlagen(AnlageID) VALUES(1)", dbFailOnError + dbSeeChanges
    Debug.Print db.RecordsAffected, Err.Number
    Debug.Print "DAO.Errors:"
    For i = 0 To DAO.Errors.Count
        Debug.Print i, DAO.Errors(i).Description
    Next i
End Sub
```

Listing 1: Analysieren von Fehlern beim DAO-Zugriff auf ODBC-Verbindungen

```
Public Function FehlerAnalysieren()
    Dim cnn As ADODB.Connection
    Dim strConnection As String
    Dim strSQL As String
    Dim objError As ADODB.Error
    Set cnn = New ADODB.Connection
    strConnection = "Provider=MSOLEDBSQL;Server=amvDesktop2023;Trusted_Connection=yes;Database=Anlagen"
    strSQL = "INSERT INTO tblAnlagen(AnlageID) VALUES(1)"
    cnn.ConnectionString = strConnection
    cnn.Open
    On Error Resume Next
    cnn.Execute strSQL
    Debug.Print "Err.Number: " & Err.Number
    Debug.Print "Err.Description: " & Err.Description
    If Not Err.Number = 0 Then
        For Each objError In cnn.Errors
            With objError
                Debug.Print "Number:          " & .Number
                Debug.Print "Description:     " & .Description
                Debug.Print "NativeError:     " & .NativeError
                Debug.Print "Source:          " & .Source
                Debug.Print "SQLState:        " & .SQLState
            End With
        Next objError
    End If
    cnn.Close
End Function
```

Listing 2: Beispiel für einen SQL Server-Fehler

Mit **NativeError** erhalten wir die SQL Server-interne Fehlernummer. Alle Fehlernummern und Fehlerbeschreibungen des SQL Servers in deutscher Sprache können wir uns übrigens mit der folgenden Anweisung, abgesetzt im SQL Server Management Studio, anzeigen lassen:

```
SELECT * FROM sys.messages WHERE language_id = 1031;
```

Source liefert die Fehlerquelle, in diesem Fall den Treiber, der hinter dem Provider **MSOLEDBSQL** steckt. Und **SQLState** liefert einen Zahlenwert, der wiederum einen ODBC-Fehlercode enthält. Diese Fehlercodes können wir beispielsweise auf der folgende Seite nachlesen:

```
https://learn.microsoft.com/en-us/sql/odbc/reference/appendixes/appendix-a-odbc-error-codes?view=sql-server-ver15
```

Anderenfalls sollten diese unter dem Schlüsselwort **ODBC Error Codes** zu finden sein.

Für den Zahlencode **23000** finden wir hier beispielsweise die Information **Integrity constraint violation**.

Benutzerdefinierte Fehler in gespeicherten Prozeduren

Um das nächste Beispiel sowohl mit DAO als auch mit ADODB nachvollziehen zu können, legen wir eine gespeicherte Prozedur in einer SQL Server-Datenbank an, die wie in Listing 3 aussieht.

ADODB: Connections und Connectionstrings

Wer auf die Inhalte von Datenbanken wie Access, SQL Server und anderen zugreifen möchte, benötigt eine spezielle Datenzugriffstechnologie. Unter VB6, VBA und twinBASIC verwendet man dazu in der Regel die DAO-Bibliothek, in Office »Microsoft Office 16.0 Access database engine Object Library« genannt, oder die ADODB-Bibliothek (»Microsoft ActiveX Data Objects 6.1 Library«). Geschichtlich wurde mal die eine, mal die andere von Microsoft als die zu bevorzugende Datenzugriffstechnik bezeichnet. Derzeit verwendet man meist DAO, vor allem in Verbindung mit Access-Datenbanken, aber beim Zugriff auf SQL Server-Datenbanken bietet ADODB einige Features, die wir mit DAO nicht nutzen können. In diesem Artikel steigen wir in die Programmierung von Datenbankzugriffen mit ADODB ein. Dabei schauen wir uns als Erstes die Connection-Klasse an, mit der erst einmal eine Verbindung zur Datenbank aufgebaut werden kann.

ADODB wurde mit der Version Access 2000 erstmalig eingeführt. Zuvor war DAO die meistverwendete Datenzugriffstechnik (Bibliothek **Microsoft DAO 3.6 Object Library**) in Zusammenhang mit dem Zugriff über die JET-Datenbank-Engine. Im Laufe der Jahre hat Microsoft jedoch entschieden, dass DAO wieder die wichtigste Technik für den Zugriff auf Datenbanken sein soll. Damit einher ging mit Access 2007 die Einführung einer neuen Variante der **JET**-Engine namens **ACE** (Access Database Engine) und damit eine neue Bibliothek mit dem Titel **Microsoft Office 16.0 Access database engine Object Library**.

Wie bereits erwähnt, gibt es jedoch einige Einsatzzwecke, bei denen ADODB Vorteile gegenüber DAO hat, weil es verschiedene Funktionen zur Verfügung stellt, die es in DAO nicht gibt. Dazu gehören die folgenden:

- **Bessere Unterstützung für SQL Server**: ADODB ist für client-server-basierte Datenbanken wie SQL Server optimiert. DAO ist primär für Microsoft Access und Jet-Datenbanken konzipiert und weniger effizient bei der Arbeit mit SQL Server.

- **Unterstützung für OLE DB**: ADODB verwendet OLE DB oder ODBC, um sich mit SQL Server zu verbinden, was eine performante und flexible Verbindung ermöglicht. DAO ist auf die Jet-Engine beschränkt, die für SQL Server nicht optimiert ist. Mit DAO können wir nur in direkt über Tabellenverknüpfungen oder Pass-Through-Abfragen auf SQL Server-Tabellen zugreifen.

- **Bessere Performance bei großen Datenmengen**: ADODB kann große Datenmengen effizient über serverseitige Cursor oder Forward-Only-Recordsets abrufen. DAO lädt oft ganze Datensatzgruppen in den Speicher, was zu Performance-Problemen führt.

- **Mehr Flexibilität bei der Abfrageverarbeitung**: ADODB erlaubt den direkten Aufruf von gespeicherten Prozeduren und somit parametrisierten Abfragen, was die Sicherheit und Performance erhöht. DAO unterstützt keine direkten Prozeduraufrufe in SQL Server.

- **Transaktionsunterstützung für SQL Server**: ADODB bietet eine bessere Transaktionskontrolle mit **BeginTrans**, **CommitTrans** und **RollbackTrans**. DAO bietet zwar Transaktionsunterstützung, aber diese ist für Jet-Datenbanken optimiert und nicht für SQL Server.

Early oder Late binding

Wir können die Elemente der ADODB-Bibliothek auf verschiedene Arten nutzen – durch vorheriges Einbinden der Bibliothek (Early Binding) oder durch Deklarieren der Objektvariablen mit dem Datentyp Object und Einbinden zur Laufzeit (Late Binding).

Wir wollen wegen der damit verbundenen Vorteile zum Beispiel durch den Einsatz von IntelliSense hier mit Early Binding arbeiten.

Dazu benötigen wir einen Verweis auf die Bibliothek **Microsoft ActiveX Data Objects 6.1 Library**.

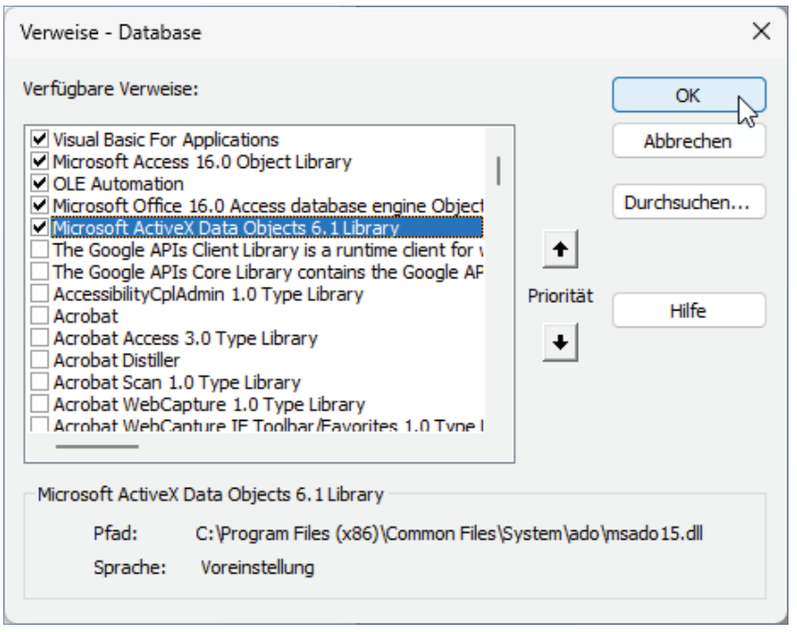


Bild 1: Verweis auf die ADODB-Bibliothek

Diesen fügen wir wie in Bild 1 über den **Verweise**-Dialog hinzu.

Eigenschaften der Connection-Klasse

Die **Connection**-Klasse liefert die folgenden Eigenschaften, von denen wir uns die wichtigsten in den folgenden Abschnitten ansehen:

- **Attributes**: Bestimmt oder setzt verschiedene Verbindungsattribute. Wird selten verwendet.
- **CommandTimeout**: Legt fest, wie lange (in Sekunden) eine Abfrage auf eine Antwort warten soll, bevor ein Timeout-Fehler auftritt.
- **ConnectionString**: Enthält die Verbindungsinformationen zur Datenbank (zum Beispiel Servername, Datenbankname, Benutzer, Passwort).
- **ConnectionTimeout**: Gibt die maximale Zeit (in Sekunden) an, die beim Verbindungsaufbau auf eine Antwort gewartet wird, bevor ein Timeout-Fehler auftritt.
- **CursorLocation**: Bestimmt, ob die Cursor-Verarbeitung clientseitig oder serverseitig erfolgt (**adUseClient** oder **adUseServer**).
- **DefaultDatabase**: Gibt die Standard-Datenbank an, die nach dem Öffnen der Verbindung verwendet wird.
- **Errors**: Liefert eine Sammlung von Fehlern, die während der letzten ADODB-Operation aufgetreten sind.
- **IsolationLevel**: Gibt das Isolationsniveau für Transaktionen an (zum Beispiel **ReadCommitted**, **Serializable**).
- **Mode**: Bestimmt die Art des Datenbankzugriffs (zum Beispiel **Nur-Lesen**, **Schreibgeschützt**, **Exklusiv**).
- **Properties**: Eine Sammlung aller Eigenschaften des **Connection**-Objekts.

- **Provider**: Gibt den verwendeten OLE DB-Provider an (zum Beispiel **SQLOLEDB** für SQL Server).
- **State**: Gibt den aktuellen Status der Verbindung zurück (zum Beispiel **adStateOpen** oder **adStateClosed**).
- **Version**: Gibt die Version von ADO zurück.

Methoden der Connection-Klasse

Außerdem bietet die **Connection**-Klasse die folgenden Methoden:

- **BeginTrans**: Startet eine Transaktion, um mehrere Operationen als Einheit auszuführen.
- **Cancel**: Bricht eine laufende Abfrage oder Operation ab.
- **Close**: Schließt die Verbindung zur Datenbank.
- **CommitTrans**: Bestätigt eine laufende Transaktion und speichert die Änderungen dauerhaft in der Datenbank.
- **Execute**: Führt eine SQL-Abfrage aus und gibt ein Recordset oder die Anzahl der betroffenen Datensätze zurück.
- **Open**: Stellt eine Verbindung zur Datenbank mit den angegebenen Verbindungsparametern her.
- **OpenSchema**: Ruft Metadaten über die Datenbank ab (zum Beispiel Tabellen, Spalten, Indexe).
- **RollbackTrans**: Bricht eine Transaktion ab und stellt die Daten vor der Transaktion wieder her.

Zugriff auf die aktuelle Access-Datenbank

Innerhalb von Access-Datenbanken liefert die ADODB-Bibliothek über die Eigenschaft **Connection** der Klasse **CurrentProject** ein **Connection**-Objekt für den Zugriff auf die aktuelle Datenbank.

Um diese zu nutzen, können wir direkt auf ihre Eigenschaften zugreifen – beispielsweise, indem wir die Verbindungszeichenfolge im Direktbereich ausgeben:

```
? CurrentProject.Connection.ConnectionString
Provider=Microsoft.ACE.OLEDB.12.0;User ID=Admin;Data
Source=C:\...\ADODB_Connections.accdb;Mode=Share Deny
None;Extended Properties="";Jet OLEDB:System database=C:\
Users\User\AppData\Roaming\Microsoft\Access\System.mdw;Jet
OLEDB:Registry Path=Software\Microsoft\Office\16.0\
Access\Access Connectivity Engine;Jet OLEDB:Database
Password="";Jet OLEDB:Engine Type=6;Jet OLEDB:Database
Locking Mode=0;Jet OLEDB:Global Partial Bulk Ops=2;Jet
OLEDB:Global Bulk Transactions=1;Jet OLEDB:New Database
Password="";Jet OLEDB:Create System Database=False;Jet
OLEDB:Encrypt Database=False;Jet OLEDB:Don't Copy Lo-
cale on Compact=False;Jet OLEDB:Compact Without Replica
Repair=False;Jet OLEDB:SFP=False;Jet OLEDB:Support Complex
Data=True;Jet OLEDB:Bypass UserInfo Validation=False;Jet
OLEDB:Limited DB Caching=False;Jet OLEDB:Bypass Choice-
Field Validation=False
```

Hier finden wir eine ganze Reihe Eigenschaften, die jedoch nicht alle immer benötigt werden.

Wir können die Verbindung auch direkt in Form einer **Connection**-Variablen speichern und darüber auf die enthaltenen Eigenschaften zugreifen:

```
Public Sub AktiveDatenbank()
    Dim cnn As ADODB.Connection
    Set cnn = CurrentProject.Connection
    Debug.Print cnn.ConnectionString
End Sub
```

Zu diesem Zeitpunkt haben wir übrigens noch keine Verbindung zur Datenbank hergestellt. Das **Connection**-Objekt kennt erst einmal nur die Verbindungszei-

chenfolge. Wie wir die Verbindung herstellen, schauen wir uns weiter unten an.

Verbindungszeichenfolgen für den Zugriff per ADODB

Dies war der stark vereinfachte Fall für den Zugriff auf die aktuelle Access-Datenbank. ADODB nutzt man jedoch in der Regel eher dazu, auf Datenquellen außerhalb der aktuellen Anwendung zuzugreifen. Wir schauen uns verschiedene Beispiele an, die oft verwendet werden.

Die Seite **https://www.connectionstrings.com/** stellt immer aktuelle Informationen rund um die Verbindungszeichenfolgen für den Zugriff von System A auf Datenbank B bereitstellt.

Solltest Du einmal eine andere Datenquelle benötigen als die hier vorgestellten, findest Du dort alle denkbaren Konstellationen.

Hier siehst Du auch, dass Verbindungszeichenfolgen nicht immer so umfangreich sein müssen wie die soeben für **CurrentProject.Connection** ermittelte.

Verbindungszeichenfolge für SQL Server mit Windows-Authentifizierung

Wenn wir beispielsweise auf eine SQL Server-Datenbank zugreifen wollen, können wir grob Verbindungszeichenfolgen für die beiden Authentifizierungsarten Windows-Authentifizierung und SQL ServerAuthentifizierung zusammenstellen.

Die Verbindungszeichenfolge für die Windows-Authentifizierung sieht beispielsweise wie folgt aus – wobei wir als Servername **amvDesktop2023** und als Datenbankname **Mitarbeiterverwaltung** verwendet haben:

```
Provider=MSOLEDBSQL;Server=amvDesktop2023;Database=Mitar-
beiterverwaltung;Trusted_Connection=yes;
```

Verbindungszeichenfolge für SQL Server mit SQL Server-Authentifizierung

Die Verbindungszeichenfolge für die SQL Server-Authentifizierung sieht so aus. Hier verwenden wir statt dem Parameter **Trusted_Connection=yes** die beiden Parameter **UID** und **PWD** mit den jeweiligen Werten:

```
Provider=MSOLEDBSQL;Server=amvDesktop2023;Database=Mitar-
beiterverwaltung;UID=[Benutzername];PWD=[Kennwort];
```

Verbindungszeichenfolge ohne Angabe der Datenbank

Wir können die beiden zuvor beschriebenen Verbindungszeichenfolgen auch ohne die Angabe einer Datenbank verwenden. Hier wird dann standardmäßig die **master**-Datenbank als Datenbank genutzt. Wir wir herausfinden können, welche Datenbank verwendet wird, zeigen wir weiter unten mit der Eigenschaft **DefaultDatabase**.

Verbindungszeichenfolge für Access-Datenbanken

Und wenn wir die Verbindungszeichenfolge für den Zugriff auf eine andere Access-Datenbank benötigen, stellen wir diese wie folgt zusammen – hier für eine Datenbank namens **ADODB_Test.accdb**:

```
Provider=Microsoft.ACE.OLEDB.12.0;Data Source=C:\...\
ADODB_Test.accdb;Persist Security Info=False;
```

Egal, ob wir eine reine Access-Connection erstellen oder eine für den SQL Server: Die angegebenen, teilweise nur aus vier Parametern bestehenden Verbindungszeichenfolgen werden nach dem Zuweisen automatisch um weitere Parameter ergänzt, die mit Standardwerten gefüllt werden.

Öffnen und Testen der Datenbankverbindung

Ob die Datenbankverbindung für die angegebene Verbindungszeichenfolge funktioniert, erfahren wir nur

ADODB: SQL-Befehle schnell ausführen mit Execute

ADODB bietet verschiedene Techniken, um SQL-Befehle zum Abfragen oder Ändern von Daten auszuführen. Die bekannteste zum Abfragen von Daten dürfte die OpenRecordset-Methode sein. Für das Ausführen von Anweisungen kann man für vollen Komfort am besten die Command-Klasse mit der Execute-Methode verwenden. Aber auch die Connection-Klasse bietet bereite eine Execute-Methode an. Mit dieser lässt sich schnell und flexibel Einiges erledigen. Welche Möglichkeiten sie bietet und wie wir diese für den Datenzugriff nutzen können, zeigen wir in diesem Artikel.

Die **Execute**-Methode des **Connection**-Objekts ist die einfachste Möglichkeit, eine Abfrage zum Ändern der Daten einer SQL Server-Datenbank auszuführen. Damit können wir jedoch nicht nur Daten ändern, in dem wir eine der folgenden Aktualisierungsabfrage nutzen:

- **DELETE**: Löschen von Datensätzen
- **UPDATE**: Aktualisieren von Datensätzen
- **INSERT INTO**: Einfügen von Datensätzen

Wir können durch das Ausführen der **Execute**-Methode auch solche SQL-Anweisungen ausführen, mit denen wir den Entwurf des Datenmodells selbst ändern – zum Anlegen, Ändern oder Löschen von Tabellen, Feldern, Indizes und vielem mehr.

Erstellen einer Tabelle mit Execute

Im ersten Beispiel erstellen wir erst einmal eine Tabelle, mit der wir danach weitere Aktionen durchführen. Dieses Beispiel finden wir in Listing 1. Hier deklarieren wir zunächst ein **Connection**-Objekt sowie eine Zeichenkette zum Speichern der SQL-Anweisung.

```
Public Sub TabelleErstellen()
    Dim cnn As ADODB.Connection
    Dim strSQL As String

    Set cnn = New ADODB.Connection
    cnn.ConnectionString = "Provider=MSOLEDBSQL;Data Source=amvDesktop2023;Initial Catalog=Anlagen;" _
        & "Integrated Security=SSPI;"
    cnn.Open

    strSQL = "CREATE TABLE tblKategorien (" & vbCrLf
    strSQL = strSQL & "    KategorieID INT IDENTITY(1,1) PRIMARY KEY," & vbCrLf
    strSQL = strSQL & "    Kategoriebezeichnung VARCHAR(255) NOT NULL" & vbCrLf
    strSQL = strSQL & ");"

    cnn.Execute strSQL

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

Listing 1: Prozedur zum Erstellen einer Tabelle

Danach erstellen wir das **Connection**-Objekt und speichern es in der Variablen **cnn**. Wir weisen die Verbindungszeichenfolge zu und öffnen die Verbindung mit der **Open**-Anweisung.

Danach stellen wir in der Variablen **strSQL** die SQL-Anweisung zusammen, die wie folgt aussieht:

```
CREATE TABLE tblKategorien (
    KategorieID INT IDENTITY(1,1) PRIMARY
KEY,
    Kategoriebezeichnung VARCHAR(255) NOT
NULL
);
```

Diese brauchen wir nun nur noch mit der **Execute**-Methode des **Connection**-Objekts auszuführen. Damit haben wir bereits eine Tabelle zum Experimentieren angelegt.

Woher wissen wir in diesem Fall, dass das Anlegen erfolgreich war? Ganz einfach dadurch, dass wir keinen Fehler ausgelöst haben.

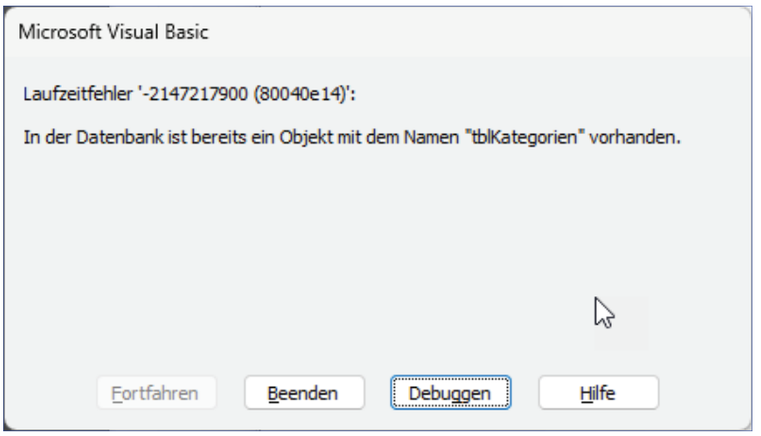


Bild 1: Fehler beim Versuch, eine Tabelle erneut anzulegen

Wir können die Gegenprobe machen, indem wir die Prozedur einfach erneut aufrufen. Damit versuchen wir, eine Tabelle anzulegen, die bereits vorhanden ist. Dies löst einen Fehler aus, der in Access mit der Meldung aus Bild 1 angezeigt wird.

Datensatz anlegen mit Execute

Im zweiten Beispiel wollen wir dieser Tabelle einen Datensatz hinzufügen (siehe Listing 2). Dieser soll im Feld **Kategoriebezeichnung** den Wert **Beispielkategorie** erhalten. Das Feld **KategorieID** wird auto-

```
Public Sub DatensatzHinzufuegen()
    Dim cnn As ADODB.Connection
    Dim strSQL As String

    Set cnn = New ADODB.Connection
    cnn.ConnectionString = "Provider=MSOLEDBSQL;Data Source=amvDesktop2023;Initial Catalog=Anlagen;" _
        & "Integrated Security=SSPI;"
    cnn.Open

    strSQL = "INSERT INTO tblKategorien(Kategoriebezeichnung) VALUES('Beispielkategorie')"
    cnn.Execute strSQL

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

Listing 2: Prozedur zum Hinzufügen eines Datensatzes

ADODB: Transaktionen im SQL Server

Transaktionen im Kontext von Datenbankanwendungen sind mehrere SQL-Anweisungen zum Ändern, Löschen oder Hinzufügen von Daten, die entweder alle oder gar nicht ausgeführt werden. Wir markieren den Start einer solchen Transaktion durch eine bestimmte Methode, führen dann eine oder mehrere SQL-Anweisungen aus und entscheiden dann, ob die Transaktion abgebrochen oder abgeschlossen werden soll. Wenn die Transaktion abgebrochen wird, werden alle seit Beginn der Transaktion in ihrem Kontext durchgeführten Änderungen rückgängig gemacht. In diesem Artikel sehen wir uns an, wie wir solche Transaktionen mit der Execute-Methode von ADODB umsetzen können.

Methoden zum Steuern von Transaktionen

Zum Durchführen von Transaktionen bietet die **Connection**-Klasse der ADODB-Bibliothek die folgenden Methoden an:

- **BeginTrans**: Startet eine Transaktion, um mehrere Operationen als Einheit auszuführen.
- **CommitTrans**: Bestätigt eine laufende Transaktion und speichert die Änderungen dauerhaft in der Datenbank.
- **RollbackTrans**: Bricht eine Transaktion ab und stellt die Daten vor der Transaktion wieder her.

Beispiel für eine Transaktion

In einem einfachen Beispiel haben wir eine Tabelle namens **tblKonten** angelegt, die wir wie in Bild 1 im SQL Server entworfen haben.

Hier finden wir die Felder **KontoID** (Primärschlüsselfeld der Tabelle), **Kontostand** (Datentyp **money**, speichert den aktuellen Kontostand und darf in diesem Fall nicht kleiner als **0** werden) und **Bezeichnung** (**nvarchar(255)**, speichert eine Bezeichnung für das Konto).

Die Tabelle kannst Du mit der folgenden T-SQL-Anweisung in einer SQL Server-Datenbank anlegen:

```
CREATE TABLE dbo.tblKonten (
    KontoID INT IDENTITY(1,1) PRIMARY KEY,
    Kontostand MONEY NOT NULL DEFAULT 0
        CHECK (Kontostand >= 0),
    Bezeichnung NVARCHAR(255) NULL
);
```

Der besseren Übersichtlichkeit halber haben wir der Beispieldatenbank eine Tabellenverknüpfung auf diese Tabelle hinzufügt. Die Transaktion wollen wir jedoch nicht auf Basis der Tabellenverknüpfungen durchführen (was auch möglich wäre), sondern auf Basis der SQL Server-Tabellen über eine entsprechende **Connection**.

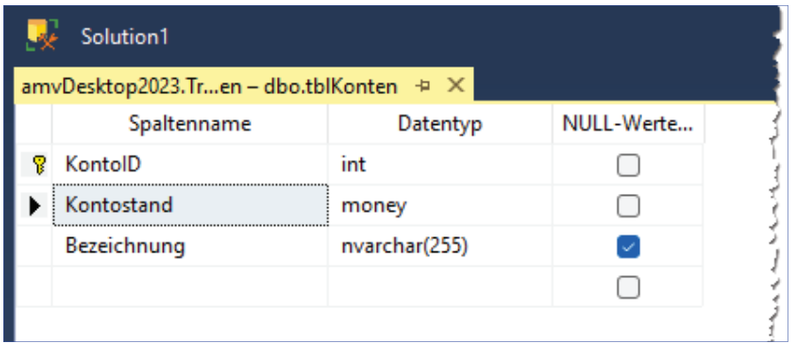


Bild 1: Die Tabelle **tblKonten** im SQL Server

```
Public Sub TransaktionBank()
    Dim cnn As ADODB.Connection
    Set cnn = New ADODB.Connection
    cnn.ConnectionString = "Provider=MSOLEDBSQL;Data Source=amvDesktop2023;" _
        & "Initial Catalog=Transaktionen;Integrated Security=SSPI;"
    cnn.Open

    cnn.BeginTrans

    On Error GoTo Fehler

    '1 Geld von Konto A abbuchen
    cnn.Execute "UPDATE tblKonten SET Kontostand = Kontostand - 100 WHERE KontoID = 1"

    '2 Geld auf Konto B gutschreiben
    cnn.Execute "UPDATE tblKonten SET Kontostand = Kontostand + 100 WHERE KontoID = 2"

    On Error GoTo 0

    cnn.CommitTrans
    MsgBox "Überweisung erfolgreich!", vbInformation
    Exit Sub

Fehler:
    cnn.RollbackTrans
    MsgBox "Fehler aufgetreten - Überweisung wurde storniert!", vbCritical

End Sub
```

Listing 1: Diese Prozedur führt eine einfache Transaktion durch.

Dazu starten wir in der Prozedur aus Listing 1 mit dem Deklarieren eines **Connection**-Objekts namens **cnn**. Danach öffnen wir die Verbindung mit der **Open**-Methode.

Anschließend starten wir bereits die Transaktion mit der Methode **BeginTrans** des **Connection**-Objekts. Diese Methode hat keine Parameter. Die einzige Eigenschaft, die sie hat, ist das Objekt, in dessen Kontext sie aufgerufen wird – sie bezieht sich nun auf die Verbindung aus **cnn**.

Danach wollen wir eine Banktransaktion ausführen, die sich vereinfacht aus dem Abbuchen eines bestimmten Betrags von einem Konto und dem Hinzubuchen des gleichen Betrags zu einem anderen Konto zusammensetzt.

Dazu sind zwei **UPDATE**-Anweisung nötig, mit denen wir die betroffenen Datensätze anpassen.

Dass dabei etwas nicht funktioniert, wollen wir am Auftreten eines Fehlers festmachen.

Dazu deaktivieren wir die eingebaute Fehlerbehandlung und sorgen mit **On Error Resume Fehler** dafür, dass der Code bei einem Fehler an der Sprungmarke **Fehler** weiterläuft.

Anschließend führen wir die erste der beiden **UPDATE**-Anweisungen durch, mit der wir den Kontostand des Kontos mit dem Wert **1** im Feld **KontoID** um **100** verringern.

Die zweite **UPDATE**-Anweisung erhöht den Wert des zweiten Kontos um 100.

Da hier bereits ein entsprechender Fehler aufgetreten sein sollte, können wir die eingebaute Fehlerbehandlung mit **On Error Goto 0** wieder aktivieren.

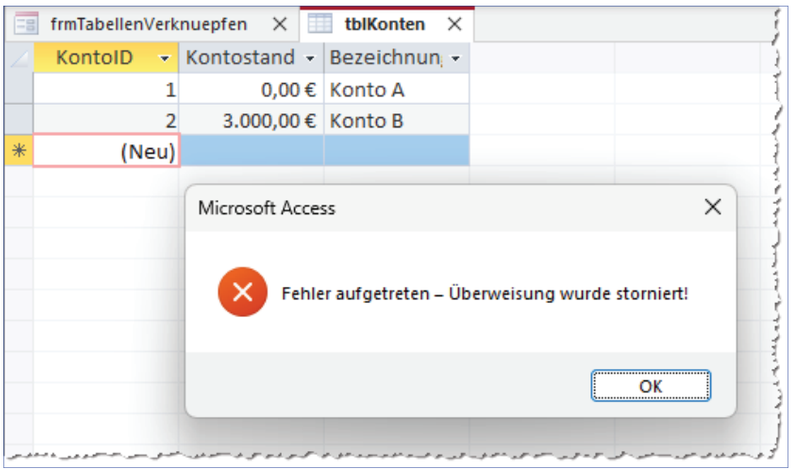


Bild 2: Die Tabelle **tblKonten** nach einem Fehler innerhalb der Transaktion

Schließlich führen wir mit der **CommitTrans**-Methode des **Connection**-Objekts die Transaktion endgültig durch und liefern eine entsprechende Erfolgsmeldung.

Es kann jedoch auch ein Fehler aufgetreten sein. Zu Beispielzwecken haben wir dazu bei der Tabellendefinition angegeben, dass der Wert des Feldes **Kontostand** niemals kleiner als **0** werden darf.

Wenn wir also eine Überweisung durchführen, die mit der ersten **UPDATE**-Anweisung dafür sorgt, dass der Kontostand auf dem ersten Konto kleiner wird als **0**, löst dies einen Fehler aus und wir landen bei der Sprungmarke **Fehler**.

Hier wird die Transaktion mit der Methode **RollbackTrans** komplett rückabgewickelt und es erscheint eine entsprechende Meldung (siehe Bild 2).

Das ACID-Prinzip

Transaktionen sollen das Einhalten des **ACID**-Prinzips sicherstellen. Dieses besteht aus:

- **Atomicity** (Atomarität): Entweder alle Änderungen oder keine – keine halben Updates.
- **Consistency** (Konsistenz): Datenbank bleibt immer in einem gültigen Zustand.
- **Isolation** (Isolation): Eine Transaktion sieht keine halbfertigen Änderungen anderer.
- **Durability** (Dauerhaftigkeit): Nach **CommitTrans** sind Änderungen dauerhaft gespeichert.

Was genau geschieht bei der Transaktion, wenn diese fehlschlägt?

- Die Transaktion wird gestartet.
- Die erste **UPDATE**-Anweisung wird ausgeführt und der Betrag wird vom ersten Konto abgezogen.
- Die Änderung wird nur im Speicher (Buffer Pool) und in der Transaction Log-Datei (LDF) gespeichert – aber noch nicht endgültig in die Datenbank (MDF) geschrieben.
- Die zweite **UPDATE**-Anweisung schlägt fehl (zum Beispiel, weil das Konto nicht existiert).
- Die Prozedur erkennt den Fehler und führt die **RollbackTrans**-Methode aus.