AUSGABE 02/2026

# Access, SQL und Cloud AUTOMATION

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

## IN DIESEM HEFT:


### REGULÄRE AUSDRÜCKE MIT VBA

Lerne die neue Klasse der VBA-Bibliothek für reguläre Ausdrücke kennen.

SEITE 18

### REST-APIS MIT VBA PROGRAMMIEREN

Erfahre, wie Du Rest-APIs mit VBA abfragen kannst.

SEITE 36

### CONNECTIONSTRINGS IM BACKSTAGE VERWALTEN

Erweitere den Backstage-Bereich um Befehle zum Zusammenstellen von Verbindungszeichenfolgen.

SEITE 50


André Minhorst Verlag

# Lerne unsere Access-KI kennen!

Falls Du Dir noch nicht Deinen Zugang zu unserer Lernplattform geholt hast, liefern wir Dir noch einen guten Grund, das jetzt zu tun: Ab sofort haben wir unsere Access-KI freigeschaltet, die Dir Antworten auf Deine Fragen auf Basis all unserer Artikel liefert! Du bekommst also nicht nur eine für Deine Frage perfekte Antwort, sondern wirst auch noch zu den relevanten Artikeln verlinkt, damit Du alles in Ruhe nachlesen kannst.

Um Dich für die Lernplattform zu registrieren, rufe einfach den folgenden Link auf:

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

Hier findest Du das Formular aus der Abbildung. Gib bitte Deinen Vor- und Nachnamen sowie Deine E-Mail-Adresse ein. Wenn Du bereits bei der LearningSuite registriert bist, verwende bitte die entsprechende E-Mail-Adresse. Der Kurs wird dann zu Deinen anderen Kursen hinzugefügt.

Unter Benutzername und Kennwort gibst Du einfach die Login-Daten ein, die Du auf Seite 2 findest – in diesem Fall **reg** als Benutzername und **exp** als Kennwort.

Du bekommst dann eine E-Mail mit einem Link zur Lernplattform und brauchst nur noch Dein Kennwort einzustellen. Anschließend kannst Du direkt loslegen.

Die Access-KI findest Du, wenn Du unten rechts auf das Icon klickst – siehe Abbildung. Hier kannst Du Deine Fragen rund um Access, VBA, SQL Server und Automation eingeben.

Viel Spaß beim Erkunden der neuen Funktion!

André Minhorst

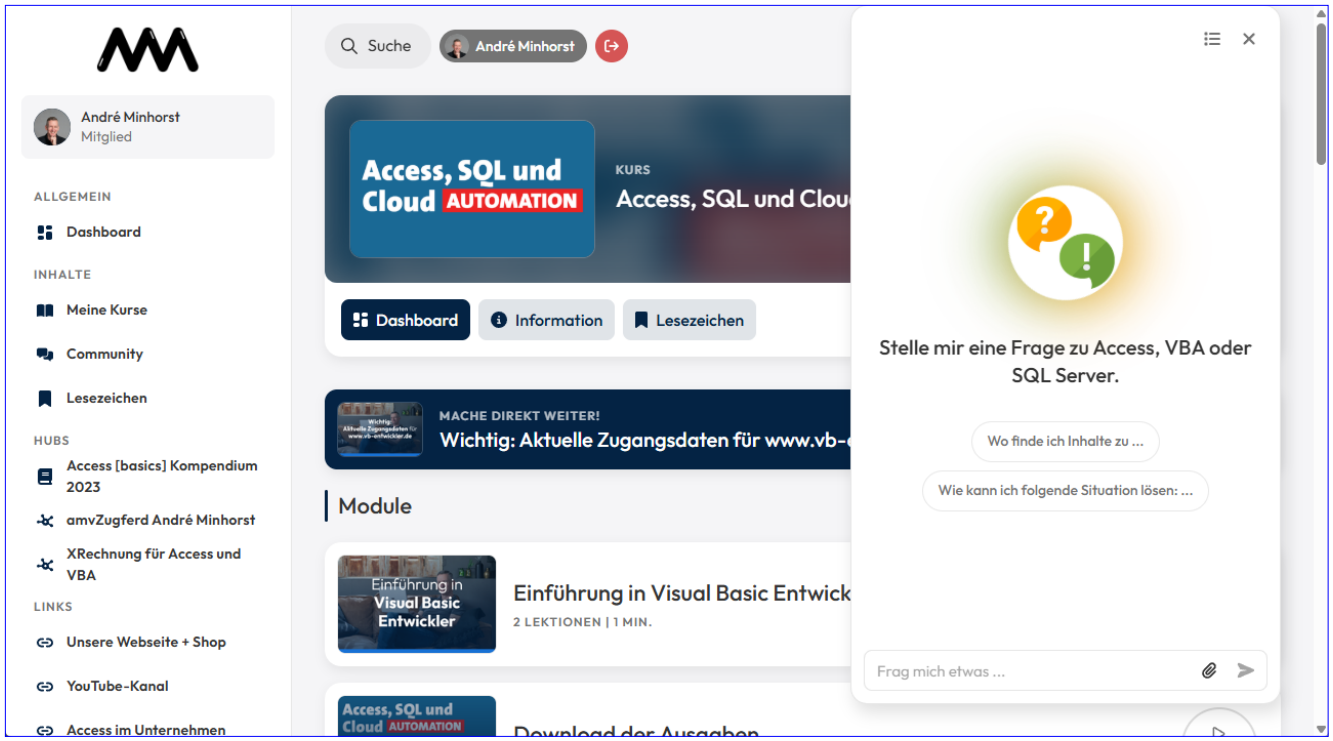


**Bild 1:** Unsere Access-KI

# Backstage-Bereich von Access erweitern, Teil 1

**Klickt man in Access auf den Registerreiter Datei, öffnet sich der sogenannte Backstage-Bereich. Hier findet man Befehle wie Speichern, Drucken oder Optionen sowie verschiedene Informationsbereiche. Was viele nicht wissen: Dieser Bereich lässt sich per XML-Definition umfangreich anpassen. Wir können eigene Befehle und Registerkarten hinzufügen, eingebaute Bereiche ausblenden und sogar vollständige Optionsseiten mit Textfeldern, Kontrollkästchen und Auswahllisten gestalten. In diesem Artikel zeigen wir Schritt für Schritt, wie das in Access funktioniert – direkt über die Systemtabelle USysRibbons und VBA-Callback-Prozeduren.**

## Beispieldatenbank

Die Beispiele zu diesem Artikel findest Du in der Beispieldatenbank. Diese enthält ein Modul **mdlBackstageReferenz**, das die Tabelle **USysRibbons** anlegt und mit der XML-Definition befüllt. Die Callback-Prozeduren befinden sich im Modul **mdlBackstageCallbacks**. Führe im Direktbereich die Prozedur **BackstageReferenz** aus, um die Tabelle zu erstellen.

Danach trägst Du unter **Datei|Optionen|Aktuelle Datenbank** den Ribbon-Namen **BackstageRef** ein und startest Access neu (siehe Bild 1).

## Hilfsformular zum Bearbeiten der Ribbon-Definitionen

In der Beispieldatenbank findest Du ein Formular namens **frmRibbonXML**, mit dem Du die Ribbon-Definitionen einfacher bearbeiten kannst, als direkt in der Tabelle (siehe Bild 2).

## Voraussetzungen

Die Backstage-Definition verwendet den Namespace **http://schemas.microsoft.com/office/2009/07/customui**. Dieser wurde mit Office 2010 eingeführt und ist die Voraussetzung dafür, dass das **backstage**-Element überhaupt zur Verfügung steht. Der ältere Namespace **2006/01** kennt kein **backstage**-Element.

Jede Backstage-Definition beginnt daher mit dem folgenden Grundgerüst:

```
<customUI xmlns=
    "http://schemas.microsoft.com/office/2009/07/customui"
      onLoad="onLoad">
  <backstage>
    ...
  </backstage>
</customUI>
```

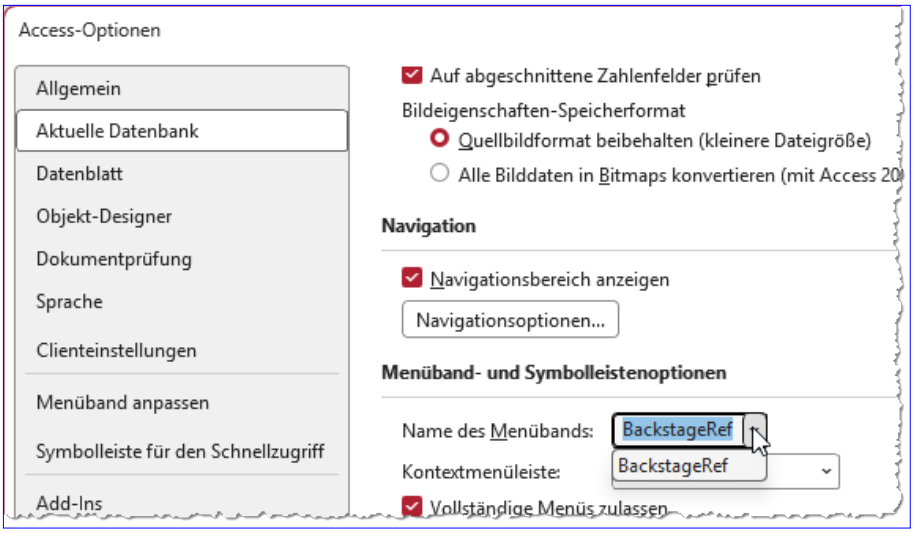


**Bild 1:** Auswählen der aktuellen Ribbon-Definition

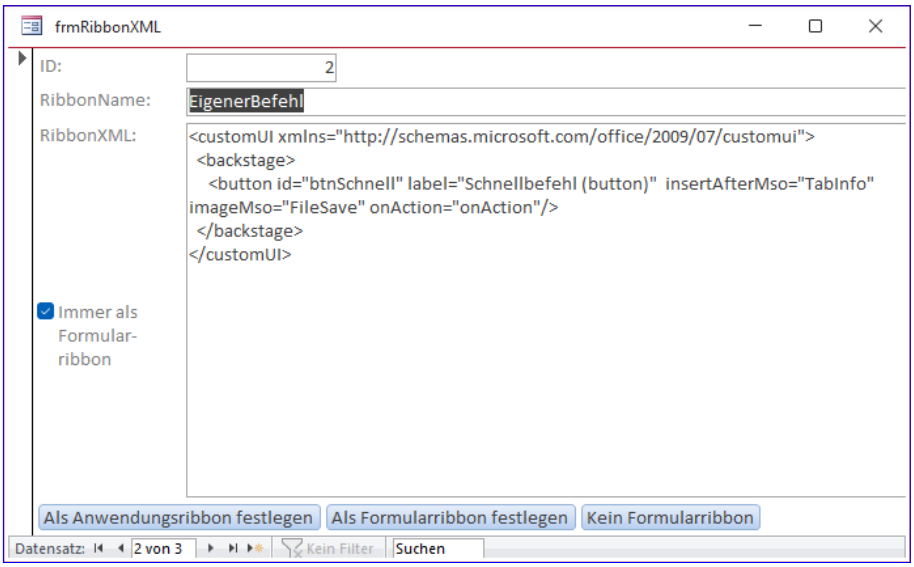


**Bild 2:** Formular zum Bearbeiten der Ribbon-Definitionen

Das Attribut **onLoad** im **customUI**-Element verweist auf eine VBA-Callback-Prozedur, die beim Laden der Ribbon-Definition aufgerufen wird. Sie übergibt ein **IRibbonUI**-Objekt, das wir in einer Modulvariablen speichern. Darüber können wir später die Anzeige aller Steuerelemente aktualisieren. Dazu legen wir im Modul mdlBackstage die folgenden Elemente an:

```
Dim m_objRibbon As IRibbonUI

Public Sub onLoad(ribbon As IRibbonUI)
    Set m_objRibbon = ribbon
End Sub
```

## Die Tabelle USysRibbons

In Access speichern wir Ribbon- und Backstage-Definitionen in der Systemtabelle **USysRibbons**. Diese Tabelle enthält mindestens die Spalten **RibbonName** (Textfeld) und **RibbonXml** (Memofeld) – siehe Bild 3.

Den Namen aus der Spalte **RibbonName** tragen wir unter **Datei|Optionen|Aktuelle Datenbank** im Feld **Name des Menübands** ein. Beim nächsten Start der Datenbank liest Access die zugehörige XML-Definition ein und wendet sie an. Die Tabelle ist als Systemtabelle nur sichtbar, wenn Du im Navigationsbereich unter **Navigationsoptionen** die Option **Systemobjekte anzeigen** aktivierst.

## Die eingebauten Backstage-Elemente

Bevor wir eigene Elemente hinzufügen, werfen wir einen Blick auf die eingebauten Elemente des Backstage-Bereichs. Jedes Element besitzt einen eindeutigen **idMso**-Wert, über den wir es referenzieren. In der Reihenfolge ihres Erscheinens sind das: **PlaceTabHome** (Startseite), **TabOfficeStart** (Office-Start), **TabRecent** (Zuletzt verwendet), **TabInfo** (Informationen),

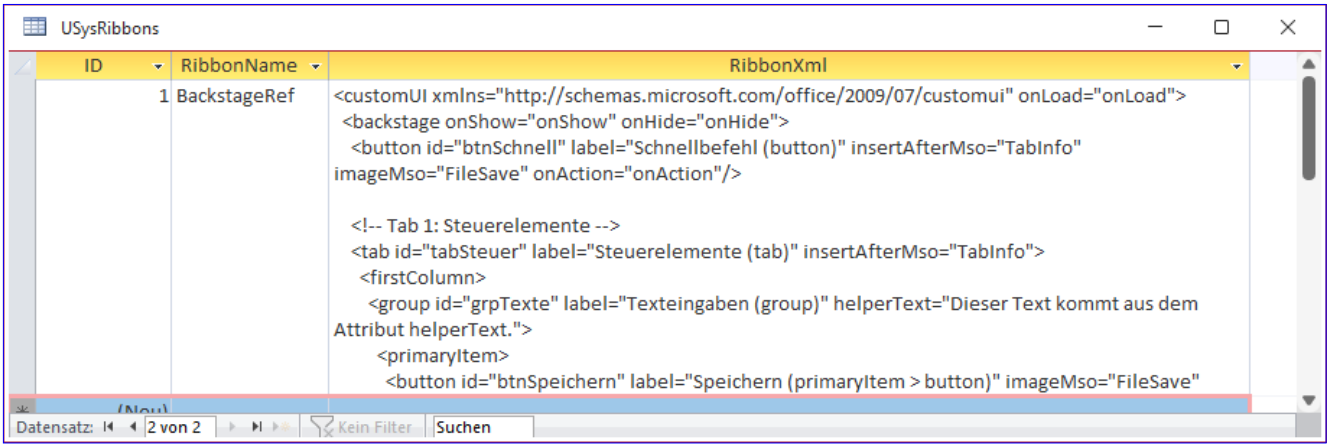


**Bild 3:** Die Tabelle **USysRibbons**

**FileSave** (Speichern-Befehl), **TabSave** (Speichern unter), **TabPrint** (Drucken), **FileCloseDatabase** (Schließen-Befehl), **TabHelp** (Hilfe), **TabOfficeFeedback** (Feedback) und **ApplicationOptionsDialog** (Optionen-Befehl).

Es gibt also zwei Typen:

**tab**-Elemente öffnen beim Anklicken einen Inhaltsbereich auf der rechten Seite, **button**-Elemente führen direkt eine Aktion aus (siehe Bild 4).

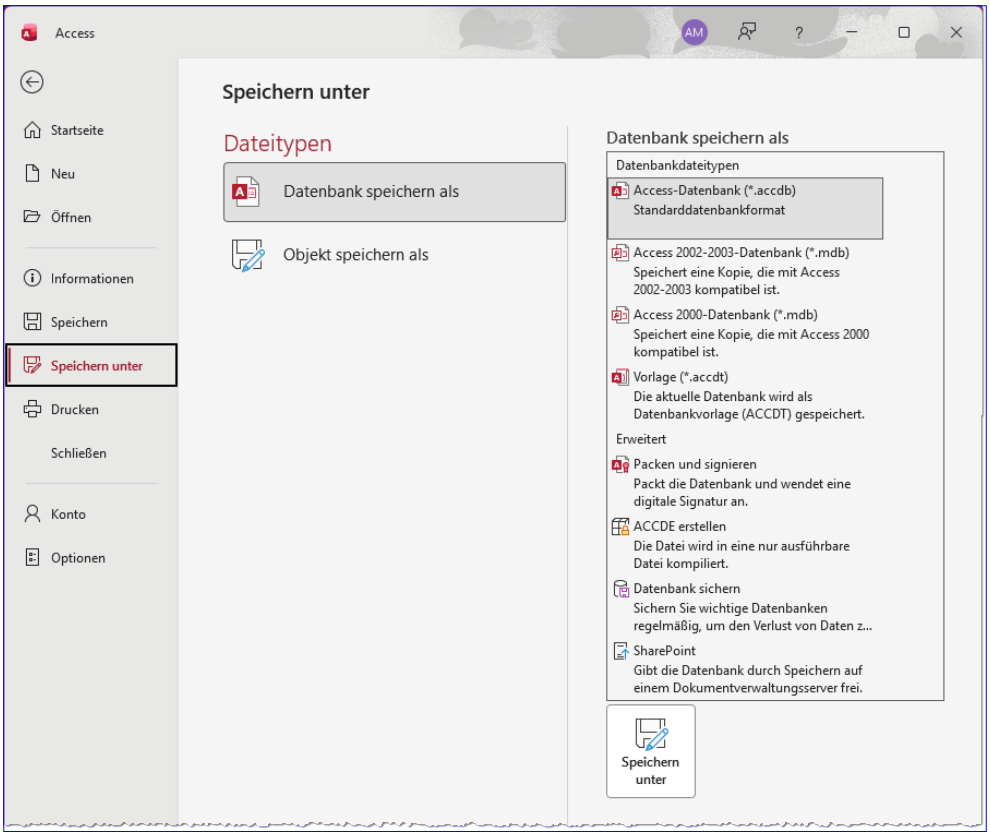


**Bild 4:** Der Standard-Backstage-Bereich von Access mit allen eingebauten Elementen

## Ein eigener Befehl im Backstage-Bereich

Wir starten mit einem einfachen Befehl: einem **button**-Element direkt unterhalb von **backstage**.

Den folgenden Code fügen wir wie in Bild 5 in die Tabelle **USysRibbons** ein und öffnen die Anwendung danach neu. Du findest die Ribbon-Definition in der Tabelle **USysRibbons** unter **EigenerBefehl**. Dann stellen wir in den Optionen die Eigenschaft **Menüband** auf **EigenerBefehl** ein, also auf den Wert der Spalte **RibbonName**:

```
<customUI
    xmlns="http://schemas.microsoft.com/office/2009/07/
        customui" onLoad="onLoad">
  <backstage>
```

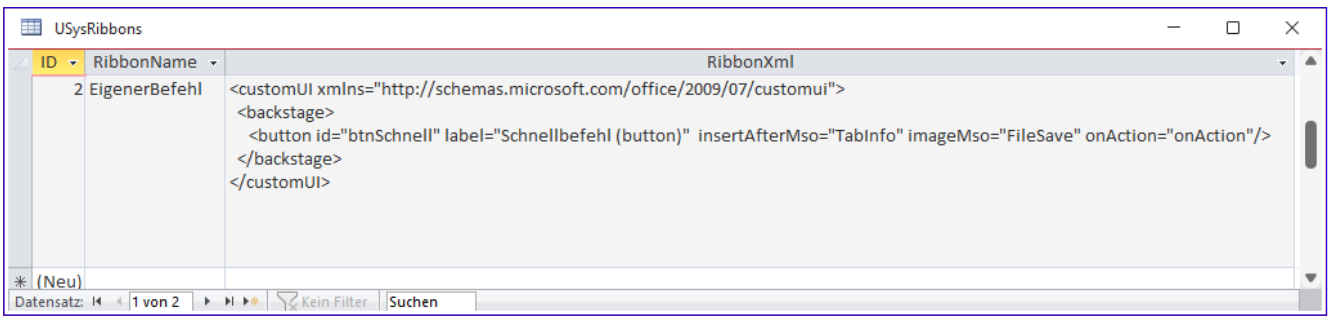


**Bild 5:** Ribbon-Definition für eine neue Schaltfläche im Backstage-Bereich

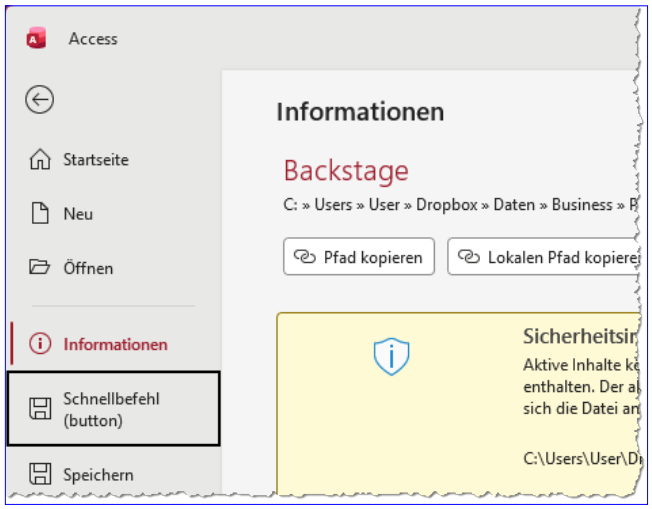


**Bild 6:** Neuer Befehl im Backstage-Bereich

```
    <button id="btnSchnell"
      label="Schnellbefehl (button)"
      insertAfterMso="TabInfo"
      imageMso="FileSave"
      onAction="onAction"/>
  </backstage>
</customUI>
```

Dieses Element erscheint in der linken Befehlsleiste – genau wie die eingebauten Befehle **Speichern** oder **Schließen** (siehe Bild 6).

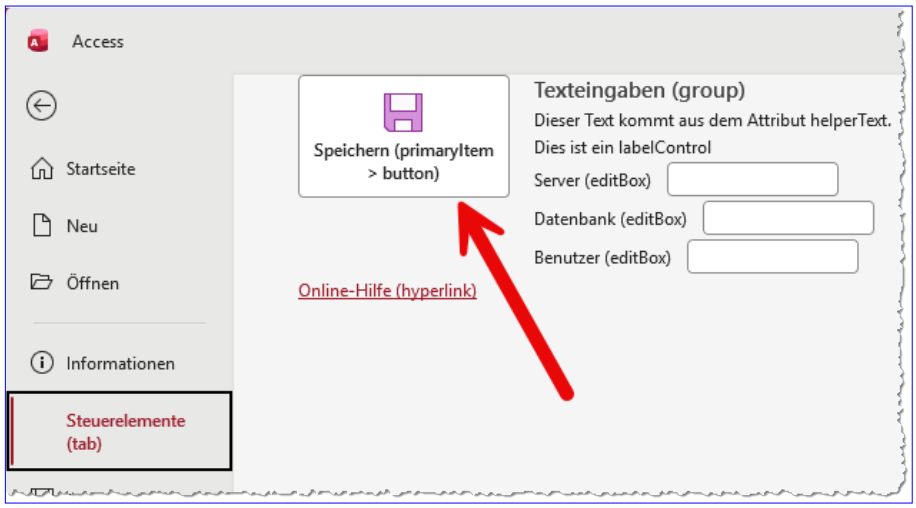


**Bild 7:** Ein Element im Bereich **primaryItem**

Das Attribut **insertAfterMso="TabInfo"** sorgt dafür, dass der Befehl nach dem Informationen-Bereich eingefügt wird. Mit **imageMso** weisen wir eines der eingebauten Office-Icons zu. Der Callback **onAction** verweist auf die VBA-Prozedur, die beim Klick aufgerufen wird. Die Signatur dieser Prozedur sieht so aus:

```
Public Sub onAction(control As IRibbonControl)
    MsgBox "onAction: " & control.Id
End Sub
```

Die Prozedur empfängt ein **IRibbonControl**-Objekt. Über dessen Eigenschaft **Id** lässt sich ermitteln, welches Steuerelement den Aufruf ausgelöst hat. So können wir mit einer einzigen Prozedur die Klicks mehrerer Backstage-Elemente verarbeiten.

## Eine eigene Registerkarte anlegen

Über das **tab**-Element legen wir eine eigene Registerkarte an. Diese erzeugt einen Eintrag in der linken Befehlsleiste, der beim Anklicken einen Inhaltsbereich auf der rechten Seite öffnet. Das **tab**-Element enthält bis zu zwei Spalten: **firstColumn** und optional **secondColumn**:

```
<tab id="tabSteuer"
     label="Steuerelemente (tab)"
     insertAfterMso="TabInfo">
  <firstColumn>
    ...
  </firstColumn>
  <secondColumn>
    ...
  </secondColumn>
</tab>
```

Innerhalb der Spalten platzieren wir **group**-Elemente, die Steuerelemente thematisch zusammenfassen. Die Hierarchie lautet: **tab** enthält **firstColumn**/

# Eigene Icons in Ribbon und Backstage anzeigen

**Durch Anpassung des Ribbons oder des Backstage-Bereichs kann man seinen Anwendungen praktische Elemente zum Aufrufen von Funktionen, Formularen oder Berichten hinzufügen. Noch individueller werden benutzerdefinierte Ribbons und der Backstage-Bereich, wenn man den Steuerelementen auch noch eigene Icons hinzufügt. Das ist jedoch mit Bordmitteln nicht so einfach möglich. Früher hat man diese Aufgabe unter Verwendung zahlreicher API-Funktionen aus der GDI-Bibliothek erledigt. Mittlerweile haben wir wesentlich einfachere Routinen dafür entwickelt. In diesem Artikel stellen wir diese vor und zeigen, wie Du Ribbon- und Backstage-Steuerelemente einfach mit solchen Icons ausstatten kannst.**

## Voraussetzung

Wie immer, wenn wir das Ribbon für eine Datenbank-Anwendung individuell anpassen wollen, benötigen wir die Ribbon-Definition selbst. Diese speichern wir in der Tabelle **USysRibbons** (siehe Bild 1).

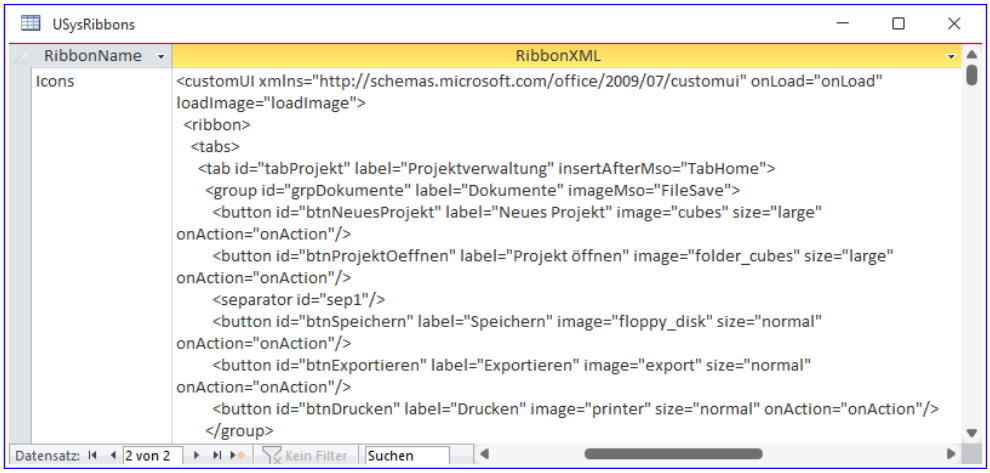


**Bild 1:** Ribbon-Definition in der Tabelle **USysRibbons**

## Bilder im Ribbon

Der erste Bereich, den wir mit eigenen Bildern ausstatten wollen, ist ein **tab**-Element im Ribbon. Dafür haben wir ein neues Tab namens Projektverwaltung angelegt, das im Endergebnis wie in Bild 2 aussieht.

Den XML-Code für diesen Bereich sehen wir in Listing 1. Hier haben wir an einigen Stellen für das Attribut **image** den Namen des jeweiligen Bildes angegeben.

## Bilder im Backstage-Bereich

Außerdem wollen wir den Backstage-Bereich an einigen Stellen mit eigenen Bildern versehen. Dazu haben wir der Ribbon-Definition im Bereich **backstage** die Elemente wie in Listing 2 hinzugefügt.

Diese Erweiterungen spiegeln sich im Backstage-Bereich in einem neuen **tab**-Element wieder, das nach dem Anklicken den Bereich aus Bild 3 liefert.

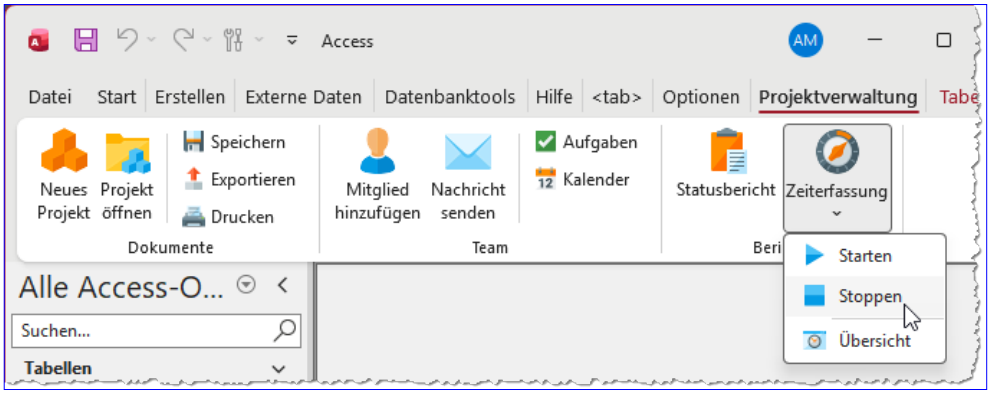


**Bild 2:** Ribbon-Steuerelemente mit Bildern

```
Bild 1<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui" onLoad="onLoad" loadImage="loadImage">
  <ribbon>
    <tabs>
      <tab id="tabProjekt" label="Projektverwaltung" insertAfterMso="TabHome">
        <group id="grpDokumente" label="Dokumente" imageMso="FileSave">
          <button id="btnNeuesProjekt" label="Neues Projekt" image="cubes" size="large" onAction="onAction"/>
          <button id="btnProjektOeffnen" label="Projekt öffnen" image="folder_cubes" size="large" onAction="onAction"/>
          <separator id="sep1"/>
          <button id="btnSpeichern" label="Speichern" image="floppy_disk" size="normal" onAction="onAction"/>
          <button id="btnExportieren" label="Exportieren" image="export" size="normal" onAction="onAction"/>
          <button id="btnDrucken" label="Drucken" image="printer" size="normal" onAction="onAction"/>
        </group>
        <group id="grpTeam" label="Team">
          <button id="btnMitgliedHinzu" label="Mitglied hinzufügen" image="user" size="large" onAction="onAction"/>
          <button id="btnNachricht" label="Nachricht senden" image="mail" size="large" onAction="onAction"/>
          <separator id="sep2"/>
          <button id="btnAufgaben" label="Aufgaben" image="checkbox" size="normal" onAction="onAction"/>
          <button id="btnKalender" label="Kalender" image="calendar" size="normal" onAction="onAction"/>
        </group>
        <group id="grpBerichte" label="Berichte">
          <button id="btnStatusbericht" label="Statusbericht" image="clipboard_paste" size="large" onAction="onAction"/>
          <splitButton id="sbZeiterfassung" size="large">
            <button id="btnZeiterfassung" label="Zeiterfassung" image="timer" onAction="onAction"/>
            <menu id="mnuZeiterfassung" label="Zeiterfassung">
              <button id="btnZeitStarten" label="Starten" image="media_play" onAction="onAction"/>
              <button id="btnZeitStoppen" label="Stoppen" image="media_stop" onAction="onAction"/>
              <menuSeparator id="msep1"/>
              <button id="btnZeitUebersicht" label="Übersicht" image="window_time" onAction="onAction"/>
            </menu>
          </splitButton>
        </group>
      </tab>
    </tabs>
  </ribbon>
  ...
```

**Listing 1:** Erster Teil der Ribbon-Definition

## Verschieden Bildgrößen

Wir können in den meisten Steuerelementen im Ribbon zwei Bildgrößen verwenden.

Wenn wir die Eigenschaft **size** für das Element auf **normal** einstellen oder diese weglassen, werden die Bilder mit 16 x 16 Pixeln angezeigt. Geben wir large an, erscheinen die Bilder mit 32 x 32 Pixeln. Optimalerweise liegen die Bilddateien in der passenden Größe vor, also 16 x 16 Pixel für kleine und 32 x 32 Pixel für große Bilder.

Die Bilddateien sollten außerdem in einem Format vorliegen, das Transparenz unterstützt. Optimal sind **.png**-Bilder geeignet, aber es funktioniert auch mit **.ico**- und **.bmp**-Dateien.

```
...
<backstage>
    <tab id="tabProjektInfo" label="Projektinfo" insertAfterMso="TabInfo" title="Projektverwaltung">
      <firstColumn>
        <group id="grpProjektdaten" label="Projektdaten">
          <primaryItem>
            <button id="btnProjektSpeichern" label="Projekt speichern" _
                    image="clipboard_paste" onAction="onAction"/>
          </primaryItem>
          <topItems>
            ...
          </topItems>
          <bottomItems>
            <hyperlink id="lnkDoku" label="Projektdokumentation" image="upload" _
                    target="https://www.example.com"/>
          </bottomItems>
        </group>
      </firstColumn>
      ...
    </tab>
  </backstage>
</customUI>
```

**Listing 2:** Zweiter Teil der Ribbon-Definition, hier für den Backstage-Bereich

## Weitere Schritte

Mit der Angabe von Bildnamen für die **image**-Elemente haben wir bereits den ersten Teil geschafft – wir haben festgelegt, welches Bild für welches Steuerelement angezeigt werden soll. Es fehlen noch zwei weitere Schritte:

- Wir müssen für das Element **customUI** im Attribut **loadImage** angeben, welche VBA-Prozedur dafür sorgt, dass die im **image**-Attribut angegebenen Bilddateien geladen werden.
- Außerdem müssen wir die Bilddateien noch verfügbar machen. Diese hinterlegt man dazu am einfachsten in der Tabelle **MSysResources**, in der auch die Icons für Schaltflächen oder Bild-Steuerelemente in Formularen hinterlegt werden.

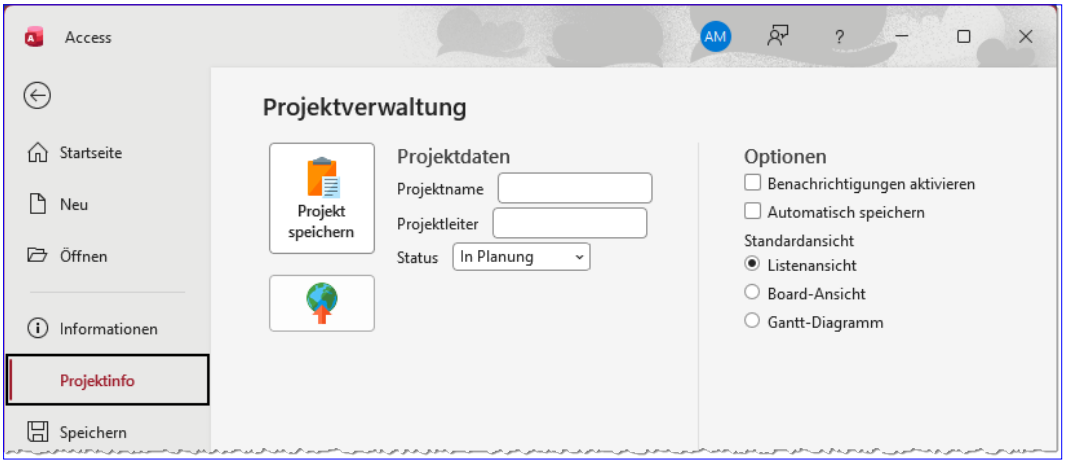


**Bild 3:** Backstage-Bereich mit einigen Bildern

# Reguläre Ausdrücke in VBA: Die neue RegExp-Klasse

**Seit Office Version 2508 sind reguläre Ausdrücke direkt in die VBA-Objektbibliothek integriert. Damit entfällt der bisherige Umweg über einen externen Verweis auf die VBScript-Bibliothek. Dieser Artikel erklärt die Hintergründe dieser Änderung, stellt alle Klassen und ihre Mitglieder vor und zeigt anhand zahlreicher Beispiele, wie Du reguläre Ausdrücke in Deinen VBA-Projekten einsetzen kannst.**

## Beispieldatenbank

Die Beispiele in diesem Artikel sind nicht an eine Datenbank gebunden. Du kannst alle Prozeduren und Funktionen in einem beliebigen Standardmodul einer Access-, Excel- oder Word-Datei anlegen und im Direktbereich aufrufen.

## Warum jetzt diese Änderung?

Im Mai 2024 kündigte Microsoft die geplante Abschaffung von VBScript als Windows-Komponente an. Diese Nachricht löste unter VBA-Entwicklern erhebliche Unruhe aus, denn die bis dahin genutzte Klasse **RegExp** war Bestandteil der Datei **vbscript.dll**, die im Zuge der Abschaffung entfernt werden soll.

Microsoft reagierte auf die Bedenken der Community: Mit Office Version 2508 (Build 19127.20154), veröffentlicht im September 2025, wurden die Klassen **RegExp**, **Match**, **MatchCollection** und **SubMatches** direkt in die VBA-Bibliothek aufgenommen.

Es handelt sich um eine der ersten größeren Erweiterungen der VBA-Objektbibliothek seit vielen Jahren.

## Welche Office-Versionen sind betroffen?

Die neuen Klassen stehen nicht nur in Microsoft 365 zur Verfügung, sondern generell in Click-to-Run-Installationen von Office, die auf Version 2508 oder höher aktualisiert wurden. Dazu zählen auch viele Einzelplatzinstallationen von Office 2016, 2019, 2021 und 2024. Ausgenommen sind die LTSC-Versionen (Long Term Service Channel), da diese keine Feature-Updates erhalten.

Du kannst im Objektkatalog des VBA-Editors überprüfen, ob die Klassen verfügbar sind: Öffne den Objektkatalog mit **F2** und wähle in der Bibliotheksliste den Eintrag **VBA**. Dort sollten die vier neuen Klassen **RegExp**, **Match**, **MatchCollection** und **SubMatches** erscheinen (siehe Bild 1).

Durch die Integration in die VBA-Bibliothek kannst Du nun mit Early Binding arbeiten, ohne einen externen Verweis hinzuzufügen. Die Deklaration **Dim rx**

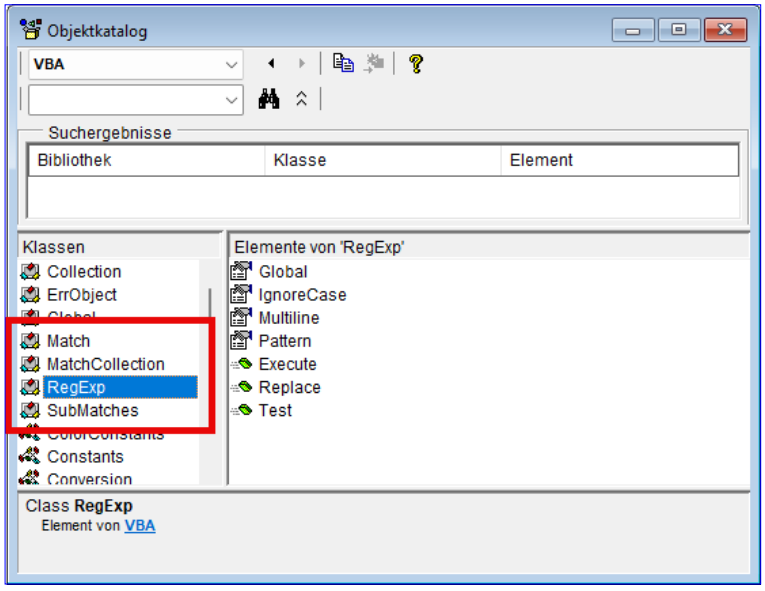


**Bild 1:** Die neuen Elemente der VBA-Bibliothek

**As RegExp** funktioniert direkt, und Du erhältst IntelliSense-Unterstützung im VBA-Editor.

## Vorsicht in Kundenumgebungen

Du darfst nicht davon ausgehen, dass diese Klasse bereits überall verfügbar ist. Eventuell aktualisiert ein Kunde seine Office-Installation nicht regelmäßig und arbeitet noch mit einer Version, in der die RegExp-Klasse nicht in der VBA-Bibliothek enthalten ist – oder er verwendet eine ältere Runtime.

## Die Klasse RegExp

Die Klasse **RegExp** ist das zentrale Objekt für die Arbeit mit regulären Ausdrücken.

Über ihre Eigenschaften konfigurierst Du das Suchverhalten, und über ihre Methoden führst Du die eigentliche Suche durch.

Eigenschaften der Klasse **RegExp**:

- **Pattern** (**String**, Lesen/Schreiben): Das Suchmuster als regulärer Ausdruck. Diese Eigenschaft muss vor dem Aufruf einer Methode gesetzt werden.

- **Global** (**Boolean**, Lesen/Schreiben): Bestimmt, ob alle Treffer oder nur der erste Treffer gesucht werden. Der Standardwert ist **False**, es wird also nur der erste Treffer ermittelt.

- **IgnoreCase** (**Boolean**, Lesen/Schreiben): Steuert, ob die Suche zwischen Groß- und Kleinschreibung unterscheidet. Der Standardwert ist **False**, die Suche ist also standardmäßig empfindlich gegenüber der Schreibweise.

- **Multiline** (**Boolean**, Lesen/Schreiben): Legt fest, ob die Anker **^** und **$** nur am Anfang und Ende der gesamten Zeichenkette wirken oder auch am Anfang und Ende jeder einzelnen Zeile. Der Standardwert ist **False**.

Methoden der Klasse **RegExp**:

- **Test(Zeichenkette)**: Gibt **True** zurück, wenn das Muster in der übergebenen Zeichenkette gefunden wird, andernfalls **False**. Die Eigenschaft **Global** hat auf das Ergebnis keinen Einfluss.

- **Execute(Zeichenkette)**: Durchsucht die übergebene Zeichenkette und gibt ein **MatchCollection**-Objekt zurück, das alle gefundenen Treffer enthält (beziehungsweise nur den ersten, wenn **Global** auf **False** steht).

- **Replace(Zeichenkette, Ersetzung)**: Ersetzt die gefundenen Treffer in der Zeichenkette durch den angegebenen Ersetzungstext und gibt das Ergebnis als neue Zeichenkette zurück. In der Ersetzung kannst Du mit **$1**, **$2** und so weiter auf eingefangene Gruppen zugreifen. Mit **$&** fügst Du den gesamten Treffer ein.

## Die Klasse MatchCollection

Ein **MatchCollection**-Objekt wird ausschließlich von der Methode **Execute** der Klasse **RegExp** erzeugt.

Es handelt sich um eine schreibgeschützte Auflistung von **Match**-Objekten:

- **Count** (**Long**, nur Lesen): Die Anzahl der gefundenen Treffer. Wenn kein Treffer gefunden wurde, ist der Wert **0**.

- **Item(Index)** (**Match**, nur Lesen): Gibt den Treffer an der angegebenen Position zurück. Der Index beginnt bei **0** und reicht bis **Count - 1**. Da **Item** die Standardeigenschaft ist, kannst Du auch die Kurzform **MatchCollection(0)** verwenden.

Am häufigsten wirst Du eine **MatchCollection** mit **For Each** durchlaufen. Alternativ ist auch eine Schleife mit **For...Next** über den Index möglich.

## Die Klasse Match

Jedes **Match**-Objekt repräsentiert einen einzelnen Treffer innerhalb der durchsuchten Zeichenkette. Alle Eigenschaften sind schreibgeschützt:

- **Value** (**String**, nur Lesen): Der Text, der durch das Muster gefunden wurde. **Value** ist die Standardeigenschaft der Klasse.
- **FirstIndex** (**Long**, nur Lesen): Die Position des ersten Zeichens des Treffers innerhalb der durchsuchten Zeichenkette. Achtung: Der Index ist nullbasiert. Das erste Zeichen der Zeichenkette hat den Index **0**. Dies unterscheidet sich von den meisten anderen VBA-Funktionen wie **Mid** oder **InStr**, die mit **1** beginnen.
- **Length** (**Long**, nur Lesen): Die Länge des gefundenen Treffers in Zeichen.
- **SubMatches** (**SubMatches**, nur Lesen): Eine Auflistung der durch einfangende Gruppen (Klammern im Muster) gefundenen Teilübereinstimmungen. Wenn das Muster keine Klammern enthält, ist die Auflistung leer.

## Die Klasse SubMatches

Die Klasse **SubMatches** ist eine schreibgeschützte Auflistung von Zeichenketten.

Jeder Eintrag entspricht dem Text, der durch eine einfangende Gruppe im Suchmuster gefunden wurde.

- **Count** (**Long**, nur Lesen): Die Anzahl der Teilübereinstimmungen.
- **Item(Index)** (**String**, nur Lesen): Gibt die Teilübereinstimmung an der angegebenen Position zurück. Der Index beginnt bei **0**. Beachte, dass **SubMatches(0)** nicht den gesamten Treffer enthält, sondern den Text der ersten einfangenden Gruppe. Dies unterscheidet sich von vielen anderen Programmiersprachen, in denen Index **0** den Gesamttreffer liefert.

## Kurzübersicht: Muster-Syntax

Bevor wir zu den Beispielen kommen, hier eine kompakte Übersicht der wichtigsten Elemente der Muster-Syntax. Die Zeichen in einem regulären Ausdruck lassen sich in drei Kategorien einteilen: literale Zeichen, Metazeichen und Quantifizierer. Wir listen diese in den folgenden Abschnitten auf.

**Zeichenklassen und Metazeichen:**

- **.** (Punkt): Steht für ein beliebiges Zeichen außer dem Zeilenumbruch.
- **\d**: Eine Ziffer (entspricht **[0-9]**).
- **\D**: Ein Zeichen, das keine Ziffer ist.
- **\w**: Ein Wortzeichen (Buchstabe, Ziffer oder Unterstrich).
- **\W**: Ein Zeichen, das kein Wortzeichen ist.
- **\s**: Ein Whitespace-Zeichen (Leerzeichen, Tabulator, Zeilenumbruch).
- **\S**: Ein Zeichen, das kein Whitespace ist.
- **[abc]**: Eines der Zeichen **a**, **b** oder **c**.
- **[^abc]**: Kein Zeichen aus **a**, **b** oder **c**.
- **[a-z]**: Ein Zeichen im Bereich von **a** bis **z**.

**Quantifizierer:**

- *: Null oder mehr Wiederholungen.
- +: Eine oder mehr Wiederholungen.

- **?**: Null oder eine Wiederholung.
- **{n}**: Genau **n** Wiederholungen.
- **{n,}**: Mindestens **n** Wiederholungen.
- **{n,m}**: Mindestens **n**, höchstens **m** Wiederholungen.

**Anker und Gruppen:**

- ^: Anfang der Zeichenkette (beziehungsweise der Zeile bei **Multiline** = **True**).
- **$**: Ende der Zeichenkette (beziehungsweise der Zeile bei **Multiline** = **True**).
- **\b**: Wortgrenze.
- **(Ausdruck)**: Einfangende Gruppe. Der Treffer der Gruppe wird in **SubMatches** gespeichert.
- **|**: Alternation (»oder«).

## Grundlegende Beispiele

Wir beginnen mit den grundlegenden Methoden der **RegExp**-Klasse. Alle Prozeduren verwenden Early Binding, das heißt, Du benötigst mindestens Office Version 2508.

## Einfacher Test auf ein Muster

Die Methode Test gibt **True** zurück, sobald das Muster irgendwo in der Zeichenkette gefunden wird.

In diesem Fall sucht das Muster **\d+** nach einer oder mehreren aufeinanderfolgenden Ziffern.

```
Public Sub TestAufMuster()
    Dim rx As RegExp
    Set rx = New RegExp
    rx.Pattern = "\d+"
    'Prüft, ob Ziffern enthalten sind
    Debug.Print rx.Test("Hallo 42")
    'Ausgabe: True
    Debug.Print rx.Test("Hallo Welt")
    'Ausgabe: False
End Sub
```

## Alle Treffer mit Execute ermitteln

Beachte, dass **Global** auf **True** gesetzt werden muss, damit alle Treffer und nicht nur der erste ermittelt werden.

Die Eigenschaft **FirstIndex** ist nullbasiert, Position **5** entspricht also dem sechsten Zeichen:

```
Public Sub AlleTreffer()
    Dim rx As RegExp
    Dim colMatches As MatchCollection
    Dim objMatch As Match

    Set rx = New RegExp
    rx.Pattern = "\d+"
    rx.Global = True

    Set colMatches = rx.Execute("Art. 12, Art. 345")

    Debug.Print "Treffer: " & colMatches.Count
    'Ausgabe: Treffer: 2

    For Each objMatch In colMatches
        Debug.Print objMatch.Value & " an Position " _
            & objMatch.FirstIndex
    Next objMatch
    'Ausgabe:
    '12 an Position 5
    '345 an Position 14
End Sub
```

## Einfaches Ersetzen mit Replace

Im folgenden Beispiel wollen wir alle Auftreten von Zahlenwerten durch **XXX** ersetzen. Dazu stellen wir

# Textdateien und Stream mit dem FileSystemObject

**Im Artikel Dateimanagement mit dem FileSystemObject (www.vbentwickler.de/478) haben wir die Klasse FileSystemObject kennengelernt und damit Laufwerke, Verzeichnisse und Dateien verwaltet. Dabei haben wir drei Methoden und Funktionen erwähnt, die wir in einem separaten Artikel behandeln wollten: CreateTextFile, OpenTextFile und OpenAsTextStream. Diese drei Elemente sind das Tor zur Klasse TextStream, mit der wir Textdateien erstellen, beschreiben und auslesen können. In diesem Artikel schauen wir uns zuerst an, wie wir TextStream-Objekte erzeugen, und gehen dann alle Eigenschaften und Methoden der Klasse durch. Abschließend bauen wir ein praxisnahes Beispiel, in dem wir eine CSV-Datei erzeugen und wieder einlesen.**

## Beispieldatenbank

Die Beispiele dieses Artikels findest Du in der Beispieldatenbank zum Artikel. Alle Prozeduren befinden sich im Modul **mdlTextStream**. Um die Funktionen des **FileSystemObject** nutzen zu können, benötigst Du einen Verweis auf die Bibliothek **Microsoft Scripting Runtime**. Diesen richtest Du über den Menüpunkt **Extras|Verweise** im VBA-Editor ein.

## Drei Wege zum TextStream

Die Klasse **TextStream** lässt sich nicht direkt instanzieren. Wir können also kein Objekt mit **New TextStream** erzeugen. Stattdessen liefern uns drei Funktionen beziehungsweise Methoden ein solches Objekt zurück:

- **CreateTextFile**: Erstellt eine neue Textdatei und gibt ein **TextStream**-Objekt zum Beschreiben zurück.
- **OpenTextFile**: Öffnet eine vorhandene Textdatei und gibt ein **TextStream**-Objekt zum Lesen, Schreiben oder Anhängen zurück.
- **OpenAsTextStream**: Methode der **File**-Klasse, die ein **TextStream**-Objekt auf Basis eines bereits referenzierten **File**-Objekts öffnet.

Alle drei Varianten liefern am Ende ein Objekt vom Typ **TextStream**, das über dieselben Eigenschaften und Methoden verfügt. Der Unterschied liegt nur darin, wie wir an dieses Objekt gelangen.

## Textdatei erstellen mit CreateTextFile

Die Funktion **CreateTextFile** ist eine Methode des **FileSystemObject**. Sie erwartet als ersten Parameter den Pfad zu der zu erstellenden Datei. Optional können wir mit dem zweiten Parameter (**Overwrite**) angeben, ob eine bereits vorhandene Datei überschrieben werden soll. Der Standardwert ist **True**. Ein dritter optionaler Parameter (**Unicode**) legt fest, ob die Datei im Unicode-Format erstellt werden soll. Der Standardwert ist **False**, was eine ANSI-Datei erzeugt.

Die folgende Prozedur erstellt eine neue Textdatei im aktuellen Datenbankverzeichnis und schreibt drei Zeilen hinein:

```
Public Sub TextdateiErstellen()
    Dim objFSO As Scripting.FileSystemObject
    Dim objTextstream As Scripting.TextStream
    Dim strPfad As String

    strPfad = CurrentProject.Path & "\Beispiel.txt"
    Set objFSO = New Scripting.FileSystemObject
    Set objTextstream = objFSO.CreateTextFile(strPfad, True)
    objTextstream.WriteLine "Erste Zeile"
    objTextstream.WriteLine "Zweite Zeile"
    objTextstream.WriteLine "Dritte Zeile"
```

```
    objTextstream.Close
    Set objTextstream = Nothing
    Set objFSO = Nothing
End Sub
```

Wir erzeugen ein **FileSystemObject** und rufen dessen **CreateTextFile**-Methode auf. Das Ergebnis ist ein **TextStream**-Objekt, das bereits zum Schreiben geöffnet ist. Mit **WriteLine** schreiben wir drei Zeilen in die Datei und schließen den Stream anschließend mit **Close**.

Übrigens: **CreateTextFile** gibt es nicht nur am **FileSystemObject**, sondern auch an der **Folder**-Klasse. Dort übergeben wir lediglich den Dateinamen ohne Verzeichnisangabe, weil das Verzeichnis bereits durch das **Folder**-Objekt festgelegt ist.

## Textdatei öffnen mit OpenTextFile

Um eine vorhandene Textdatei zu öffnen, verwenden wir die Funktion **OpenTextFile** des **FileSystemObject**. Diese erwartet als ersten Parameter den Pfad zur Datei.

Der zweite Parameter (**IOMode**) gibt den Zugriffsmodus an. Hier stehen drei Konstanten zur Verfügung:

- **ForReading** (**1**): Öffnet die Datei nur zum Lesen.
- **ForWriting** (**2**): Öffnet die Datei zum Schreiben. Dabei wird der vorhandene Inhalt überschrieben.
- **ForAppending** (**8**): Öffnet die Datei zum Anhängen. Neuer Inhalt wird am Ende der Datei angefügt.

Optional kann ein dritter Parameter (**Create**) angegeben werden, der festlegt, ob die Datei erstellt werden soll, wenn sie noch nicht existiert. Der Standardwert ist **False**. Ein vierter Parameter (**Format**) steuert die Zeichenkodierung: **TristateFalse** (**0**) für ANSI, **TristateTrue** (**-1**) für Unicode und **TristateUseDefault** (**-2**) für die Systemvorgabe.

Das folgende Beispiel öffnet die zuvor erstellte Datei zum Lesen und gibt den gesamten Inhalt im Direktbereich aus:

```
Public Sub TextdateiLesen()
    Dim objFSO As Scripting.FileSystemObject
    Dim objTextstream As Scripting.TextStream
    Dim strPfad As String

    strPfad = CurrentProject.Path & "\Beispiel.txt"
    Set objFSO = New Scripting.FileSystemObject
    Set objTextstream = objFSO.OpenTextFile(strPfad, _
        ForReading)
    Debug.Print objTextstream.ReadAll
    objTextstream.Close
    Set objTextstream = Nothing
    Set objFSO = Nothing
End Sub
```

Hier nutzen wir die Methode **ReadAll**, um den gesamten Inhalt der Datei auf einmal einzulesen. Die Details zu **ReadAll** und den übrigen Methoden der **TextStream**-Klasse folgen weiter unten.

## TextStream über ein File-Objekt öffnen mit OpenAsTextStream

Der dritte Weg führt über die **File**-Klasse. Wenn wir bereits ein **File**-Objekt referenziert haben, können wir dessen Methode **OpenAsTextStream** aufrufen. Die Parameter sind identisch mit denen von **OpenTextFile**, allerdings ohne den Dateipfad, da dieser bereits durch das **File**-Objekt feststeht.

Das folgende Beispiel zeigt, wie wir zunächst ein **File**-Objekt holen und darauf einen **TextStream** öffnen:

```
Public Sub TextdateiPerFileObjekt()
    Dim objFSO As Scripting.FileSystemObject
    Dim objFile As Scripting.File
    Dim objTextstream As Scripting.TextStream
    Dim strPfad As String
```

```
    strPfad = CurrentProject.Path _
        & "\Beispiel.txt"
    Set objFSO = New Scripting.FileSystemObject
    Set objFile = objFSO.GetFile(strPfad)
    Set objTextstream = objFile.OpenAsTextStream( _
        ForReading)
    Debug.Print objTextstream.ReadAll
    objTextstream.Close
    Set objTextstream = Nothing
    Set objFile = Nothing
    Set objFSO = Nothing
End Sub
```

Wir holen zunächst mit **GetFile** ein **File**-Objekt und rufen dann dessen **OpenAsTextStream**-Methode auf. Das Ergebnis ist wiederum ein **TextStream**-Objekt, das wir wie gewohnt verwenden können.

Dieser Weg bietet sich vor allem an, wenn wir ohnehin bereits mit **File**-Objekten arbeiten, zum Beispiel beim Durchlaufen der **Files**-Auflistung eines Verzeichnisses.

## Die Eigenschaften der TextStream-Klasse

Die **TextStream**-Klasse bietet vier Eigenschaften, mit denen wir Informationen über die aktuelle Position innerhalb des Streams abfragen können. Alle vier sind schreibgeschützt.

- **AtEndOfLine** gibt **True** zurück, wenn sich die Leseposition am Ende einer Zeile befindet, also direkt vor dem Zeilenumbruch. Diese Eigenschaft steht nur bei Streams zur Verfügung, die zum Lesen geöffnet wurden. Versuchen wir, sie bei einem zum Schreiben geöffneten Stream abzufragen, erhalten wir einen Laufzeitfehler.

- **AtEndOfStream** gibt **True** zurück, wenn die Leseposition das Ende der Datei erreicht hat. Auch diese Eigenschaft ist nur bei lesend geöffneten Streams verfügbar. Sie entspricht in etwa der **EOF**-Funktion, die wir bei den klassischen VBA-Dateioperationen mit **Open** und **Close** verwenden.

- **Column** liefert die aktuelle Spaltenposition innerhalb der Zeile. Die Zählung beginnt bei 1. Nach dem Öffnen einer Datei zum Lesen steht **Column** auf 1. Nach dem Lesen des ersten Zeichens steht der Wert auf 2 und so weiter.

- **Line** gibt die aktuelle Zeilennummer zurück. Auch hier beginnt die Zählung bei 1. Jedes Mal, wenn ein Zeilenumbruch überschritten wird, erhöht sich der Wert um eins.

Mit **Column** und **Line** zusammen können wir jederzeit feststellen, an welcher Stelle in der Datei wir uns beim Lesen befinden. Das kann zum Beispiel bei der Fehleranalyse hilfreich sein, wenn beim Einlesen einer Datei ein unerwarteter Wert auftritt und wir die genaue Position im Fehlerprotokoll festhalten möchten.

## Methoden zum Lesen: Read, ReadLine und ReadAll

Die **TextStream**-Klasse bietet drei Methoden zum Lesen von Inhalten. Alle drei setzen voraus, dass der Stream zum Lesen geöffnet wurde (Modus **ForReading**).

**Read** erwartet als Parameter die Anzahl der zu lesenden Zeichen und gibt diese als Zeichenkette zurück. Nach dem Aufruf wird die Leseposition um die entsprechende Anzahl Zeichen weitergerückt. Das folgende Beispiel liest die ersten zehn Zeichen einer Datei:

```
Public Sub ErsteZeichenLesen()
    ...
    Dim strPfad As String
    Dim strErgebnis As String

    strPfad = CurrentProject.Path & "\Beispiel.txt"
    Set objFSO = New Scripting.FileSystemObject
    Set objTextstream = _
```

```
        objFSO.OpenTextFile(strPfad, ForReading)
    strErgebnis = objTextstream.Read(10)
    Debug.Print strErgebnis
    ...
End Sub
```

**ReadLine** liest eine komplette Zeile bis zum nächsten Zeilenumbruch und gibt diese ohne den Zeilenumbruch zurück. Die Leseposition wird in die nächste Zeile verschoben. In Kombination mit **AtEndOfStream** können wir damit alle Zeilen einer Datei durchlaufen:

```
Public Sub AlleZeilenLesen()
    ...
    strPfad = CurrentProject.Path & "\Beispiel.txt"
    Set objFSO = New Scripting.FileSystemObject
    Set objTextstream = _
        objFSO.OpenTextFile(strPfad, ForReading)
    Do While Not objTextstream.AtEndOfStream
        Debug.Print objTextstream.ReadLine
    Loop
    ...
End Sub
```

Dies ist das Gegenstück zu der aus den klassischen VBA-Dateioperationen bekannten Kombination aus **Line Input #** und **EOF**.

**ReadAll** liest den gesamten verbleibenden Inhalt der Datei in eine einzige Zeichenkette. Das ist praktisch, wenn wir den vollständigen Dateiinhalt auf einmal verarbeiten möchten. Bei sehr großen Dateien sollte man diese Methode allerdings mit Bedacht einsetzen, da der gesamte Inhalt im Arbeitsspeicher gehalten wird.

## Methoden zum Schreiben: Write, WriteLine und WriteBlankLines

Für das Schreiben stehen ebenfalls drei Methoden bereit. Diese setzen voraus, dass der Stream zum Schreiben (**ForWriting**) oder zum Anhängen (**ForAppending**) geöffnet wurde.

**Write** schreibt eine Zeichenkette in den Stream, ohne einen Zeilenumbruch anzuhängen. Mehrere aufeinanderfolgende Aufrufe von **Write** erzeugen damit zusammenhängenden Text in derselben Zeile:

```
Public Sub TextOhneUmbruch()
    ...
    strPfad = CurrentProject.Path & "\WriteTest.txt"
    Set objFSO = New Scripting.FileSystemObject
    Set objTextstream = _
        objFSO.CreateTextFile(strPfad, True)
    objTextstream.Write "Vorname"
    objTextstream.Write ";"
    objTextstream.Write "Nachname"
    ...
End Sub
```

Das Ergebnis in der Datei wäre eine einzelne Zeile mit dem Inhalt **Vorname;Nachname**.

**WriteLine** funktioniert wie **Write**, hängt aber automatisch einen Zeilenumbruch an. Jeder Aufruf von **WriteLine** erzeugt also eine eigene Zeile in der Datei. Rufen wir **WriteLine** ohne Parameter auf, wird lediglich eine Leerzeile geschrieben.

**WriteBlankLines** erwartet als Parameter die Anzahl der einzufügenden Leerzeilen. Der folgende Aufruf fügt drei leere Zeilen in die Datei ein:

```
Public Sub LeerzeilenEinfuegen()
    ...
    strPfad = CurrentProject.Path & "\Leerzeilen.txt"
    Set objFSO = New Scripting.FileSystemObject
    Set objTextstream = objFSO.CreateTextFile( _
        strPfad, True)
    objTextstream.WriteLine "Abschnitt 1"
    objTextstream.WriteBlankLines 3
```

# Rest-APIs mit VBA programmieren

**Wer mit VBA arbeitet, kommt heute kaum noch daran vorbei: Fast jeder interessante Online-Dienst – von Wetter- und Geo-Daten über Projektmanagement-Tools bis hin zu KI-Diensten – bietet eine Rest-API an. Dabei ist das Prinzip immer dasselbe: Du schickst eine HTTP-Anfrage an eine bestimmte URL, und der Dienst antwortet mit strukturierten Daten – in der Regel im JSON-Format. Ob Du Kundenadressen mit einem CRM abgleichen, Versandetiketten bei DHL anfordern, Aufgaben in Trello anlegen oder Texte mit DeepL übersetzen willst – hinter all diesen Integrationen steckt dieselbe Technik. VBA bringt dafür alle nötigen Werkzeuge von Haus aus mit: Ein einziger Verweis auf eine Windows-Systembibliothek genügt, um HTTP-Anfragen abzusetzen und die Antworten auszuwerten. Dieser Artikel erklärt, was eine Rest-API überhaupt ist, welche Grundbegriffe Du kennen musst, um jede beliebige API-Dokumentation selbstständig zu lesen, und wie Du eine wiederverwendbare Funktion baust, die Du als solide Basis für jeden weiteren API-Aufruf einsetzen kannst.**

## Was ist eine Rest-API?

**REST** steht für Representational State Transfer und bezeichnet einen Architekturstil für verteilte Systeme, der auf dem HTTP-Protokoll basiert. Eine Rest-API (Application Programming Interface) stellt Ressourcen – zum Beispiel Kundendatensätze, Wetterdaten oder Projektaufgaben – über eindeutige URLs bereit. Der Zugriff darauf erfolgt mit den bekannten HTTP-Methoden:

- **GET**: Liest eine Ressource, ohne sie zu verändern.
- **POST**: Legt eine neue Ressource an.
- **PUT**: Ersetzt eine vorhandene Ressource vollständig.
- **PATCH**: Aktualisiert einzelne Felder einer Ressource.
- **DELETE**: Löscht eine Ressource.

Im Unterschied zu älteren SOAP-Webservices – die einen aufwendigen XML-Umschlag benötigen – verwendet **REST** meist das leichtgewichtige JSON-Format für Anfragen und Antworten. Das macht Rest-APIs deutlich einfacher konsumierbar.

## Aufbau einer Rest-API-URL

Jede Rest-API-URL besteht aus mehreren Teilen, die wir kennen müssen, bevor wir den ersten VBA-Aufruf absetzen. Nehmen wir als Beispiel eine fiktive Aufgaben-API:

```
https://api.beispiel.de/v1/aufgaben?status=offen&limit=10
```

Der erste Teil, **https://api.beispiel.de**, ist die Basis-URL des Dienstes. Das Segment **/v1/** bezeichnet die API-Version – viele Anbieter versionieren ihre API, damit bestehender Code nicht bricht, wenn neue Funktionen hinzukommen. Der Pfad **/aufgaben** benennt die Ressource. Ab dem Fragezeichen folgen optionale URL-Parameter (auch »Query-String« genannt), die mit dem **&**-Zeichen voneinander getrennt werden.

Manche APIs ergänzen die URL um eine Ressourcen-ID, wenn ein einzelner Datensatz angesprochen wird:

```
https://api.beispiel.de/v1/aufgaben/42
```

Das Prinzip ist immer gleich – und genau das macht Rest-APIs so gut erlernbar.

## HTTP-Header

Neben der URL trägt jede HTTP-Anfrage einen Satz von Headern, also Name-Wert-Paaren, die Metainformationen über den Aufruf übermitteln. Die häufigsten Header, die wir in VBA selbst setzen, sind:

- **Content-Type**: Gibt an, in welchem Format wir Daten senden – in der Regel **application/json**.
- **Accept**: Teilt dem Server mit, welches Antwortformat wir erwarten – ebenfalls meist **application/json**.
- **Authorization**: Enthält das Authentifizierungs-Token oder den API-Key, mit dem wir uns beim Dienst ausweisen.

Wie genau das Authentifizierungs-Token aussieht und wie man es beschafft, ist von Dienst zu Dienst unterschiedlich und Gegenstand eigener Artikel.

## HTTP-Statuscodes

Die Antwort des Servers enthält immer einen dreistelligen Statuscode, der uns sofort sagt, ob der Aufruf erfolgreich war.

Folgende Auflistung zeigt die wichtigsten Codes im Überblick:

- **200 – OK**: Der Aufruf war erfolgreich. Die Antwort enthält die angeforderten Daten.
- **201 – Created**: Eine neue Ressource wurde erfolgreich angelegt (typisch bei **POST**).
- **204 – No Content**: Der Aufruf war erfolgreich, aber der Server liefert keinen Antwortbody (typisch bei **DELETE**).
- **400 – Bad Request**: Die Anfrage ist fehlerhaft, zum Beispiel wegen eines ungültigen JSON-Dokuments.
- **401 – Unauthorized**: Die Authentifizierung ist fehlgeschlagen. Token oder API-Key fehlt oder ist abgelaufen.
- **403 – Forbidden**: Der Zugriff ist verweigert. Das Token ist gültig, aber die nötigen Rechte fehlen.
- **404 – Not Found**: Die angeforderte Ressource existiert unter dieser URL nicht.
- **422 – Unprocessable Entity**: Die Anfrage ist syntaktisch korrekt, scheitert aber an der inhaltlichen Validierung (zum Beispiel Pflichtfeld fehlt).
- **429 – Too Many Requests**: Das Anfrage-Limit (Rate Limit) des Dienstes wurde überschritten.
- **500 – Internal Server Error**: Auf dem Server ist ein unerwarteter Fehler aufgetreten.
- **503 – Service Unavailable**: Der Dienst ist vorübergehend nicht erreichbar, zum Beispiel wegen Wartungsarbeiten.

Die Codes lassen sich grob in Gruppen einteilen: 2xx steht generell für Erfolg, **4xx** für einen Fehler auf der Client-Seite (zum Beispiel falscher API-Key oder ungültige URL) und **5xx** für einen Fehler auf der Server-Seite. Der Code **401** bedeutet »nicht autorisiert« und weist meist auf ein fehlendes oder abgelaufenes Token hin. **404** kennst Du vielleicht schon vom Surfen im Web und bedeutet hier, dass die angeforderte Ressource unter dieser URL nicht existiert.

## Das XMLHTTP-Objekt in VBA

VBA bringt von Haus aus kein eigenes HTTP-Objekt mit. Wir greifen stattdessen auf die Bibliothek **Microsoft XML, v6.0** zurück, die Windows standard-

mäßig mitliefert. Diese enthält das Objekt **MSXML2.XMLHTTP60**, das uns alle nötigen Methoden und Eigenschaften für HTTP-Aufrufe bereitstellt.

Alternativ steht **MSXML2.ServerXMLHTTP60** zur Verfügung. Der Unterschied: **XMLHTTP60** nutzt die Proxy-Einstellungen des angemeldeten Windows-Benutzers, während **ServerXMLHTTP60** eine eigene Verbindung ohne Benutzerkontext aufbaut und sich deshalb besser für serverseitige Szenarien oder Situationen eignet, in denen kein Benutzerprofil vorliegt.

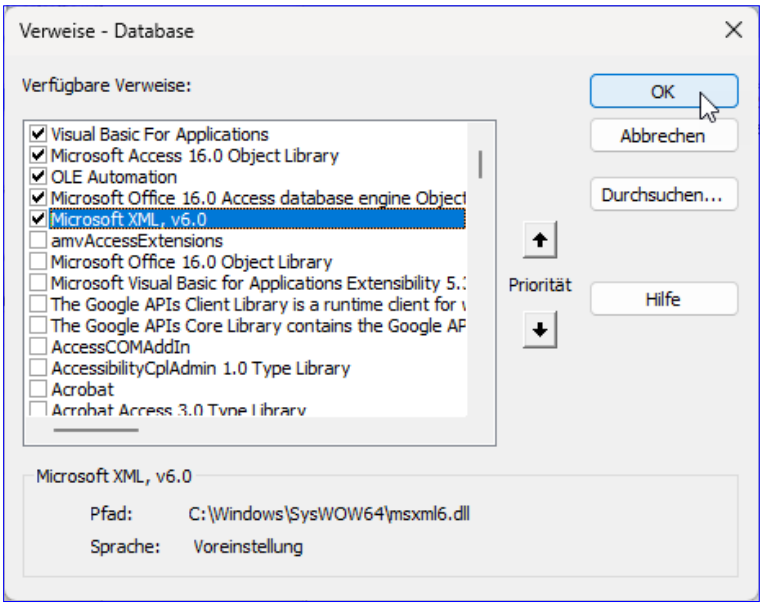


**Bild 1:** Den Verweis auf **Microsoft XML, v6.0** im **Verweise**-Dialog aktivieren

Für typische Aufgaben am Entwickler-PC sind beide gleichwertig. In diesem Artikel verwenden wir **XMLHTTP60**.

## Verweis hinzufügen

Damit wir das Objekt mit früher Bindung, also mit voller IntelliSense-Unterstützung, nutzen können, müssen wir dem VBA-Projekt einen Verweis auf die XML-Bibliothek hinzufügen.

Dazu öffnen wir im VBA-Editor den Dialog über den Befehl **Extras|Verweise** und aktivieren den Eintrag **Microsoft XML, v6.0** (siehe Bild 1).

```
Public Function HTTPRequest(strURL As String, strMethod As String, strResponse As String, Optional strData As String, _
        Optional strAuthorization As String, Optional strContentType As String = "application/json") As Integer
    Dim objHTTP As MSXML2.XMLHTTP60

    Set objHTTP = New MSXML2.XMLHTTP60
    With objHTTP
        .Open strMethod, strURL, False
        .setRequestHeader "Accept", strContentType
        .setRequestHeader "Content-Type", strContentType
        If Not Len(strAuthorization) = 0 Then
            .setRequestHeader "Authorization", strAuthorization
        End If
        .send strData
        strResponse = .responseText
        HTTPRequest = .status
    End With
End Function
```

**Listing 1:** Die Funktion HTTPRequest kapselt alle HTTP-Aufrufe an Rest-APIs.

Ab sofort können wir Variablen als **MSXML2.XMLHTTP60** deklarieren und profitieren von der Typprüfung beim Kompilieren.

## Die Kernmethoden von XMLHTTP60

Das Objekt arbeitet nach einem festen Ablauf, den wir uns merken sollten:

- **Open**: Öffnet eine Verbindung und legt Methode, URL und Synchronität fest.
- **setRequestHeader**: Setzt einen einzelnen HTTP-Header.
- **send**: Versendet die Anfrage – optional mit einem Body, zum Beispiel einem JSON-Dokument.
- **status**: Eigenschaft, die nach dem Aufruf den HTTP-Statuscode enthält.
- **responseText**: Eigenschaft, die die Antwort des Servers als Text enthält.

Der dritte Parameter von **Open** legt fest, ob der Aufruf asynchron erfolgen soll.

Wir übergeben hier immer **False**, damit der Code solange wartet, bis die Antwort eingetroffen ist.

## Eine wiederverwendbare HTTPRequest-Funktion

Anstatt in jedem Modul das **XMLHTTP60**-Objekt neu aufzubauen, lohnt es sich, eine zentrale Funktion anzulegen, die alle API-Aufrufe kapselt.

Listing 1 zeigt diese Funktion namens **HTTPRequest**.

Die Funktion nimmt folgende Parameter entgegen:

- **strURL**: Der vollständige Endpunkt des API-Aufrufs.
- **strMethod**: Die HTTP-Methode, zum Beispiel **GET**, **POST**, **PUT** oder **DELETE**.
- **strResponse**: Eine leere String-Variable, in die die Antwort des Servers geschrieben wird.
- **strData**: Optionaler Body der Anfrage, zum Beispiel ein JSON-Dokument beim Anlegen einer neuen Ressource.
- **strAuthorization**: Optionaler Autorisierungsheader, zum Beispiel **Bearer <Token>** oder **ApiKey <Key>**.
- **strContentType**: Optionaler Content-Type, Standardwert ist **application/json**.

Als Rückgabewert liefert die Funktion den HTTP-Statuscode als **Integer**. Die aufrufende Prozedur kann diesen Wert auswerten und entsprechend reagieren.

## Statuscodes auswerten

Eine typische Auswertung des Statuscodes sieht wie folgt aus. Der **Select Case**-Block behandelt die häufigsten Fälle.

```
Dim intStatus As Integer
Dim strResponse As String

intStatus = HTTPRequest( _
    "https://api.beispiel.de/v1/daten", _
    "GET", strResponse)

Select Case intStatus
    Case 200, 201
        'Erfolg: Antwort verarbeiten
        Debug.Print strResponse
    Case 401
        MsgBox "Authentifizierung fehlgeschlagen." _
            & " Token prüfen."
    Case 404
```

# PLZ-Lookup per Rest-API

**Im Grundlagenartikel zu Rest-APIs haben wir gezeigt, wie das Konzept funktioniert und wie eine wiederverwendbare HTTPRequest-Funktion aussieht. Jetzt setzen wir das Gelernte das erste Mal gegen eine echte API ein: Wir fragen zu einer deutschen Postleitzahl den zugehörigen Ort und die Koordinaten ab – ohne Anmeldung, ohne API-Key und vollkommen kostenlos. Nebenbei lernen wir, wie wir die JSON-Antwort mit wenigen Handgriffen auslesen.**

## Vorbereitungen

Um die nachfolgenden Beispiele auszuprobieren, benötigst Du die beiden in der Beispieldatatenbank enthaltenen Module **mdlJSON** und **mdlJSONDOM**.

Außerdem müssen wir zwei Verweise zu Bibliotheken hinzufügen, was wir über das Verweise-Fenster des VBA-Editors erledigen.

Hier fügen wir die beiden Verweise auf die Bibliotheken **Microsoft Scripting Runtime** und **Microsoft XML, v6.0** hinzu (siehe Bild 1).

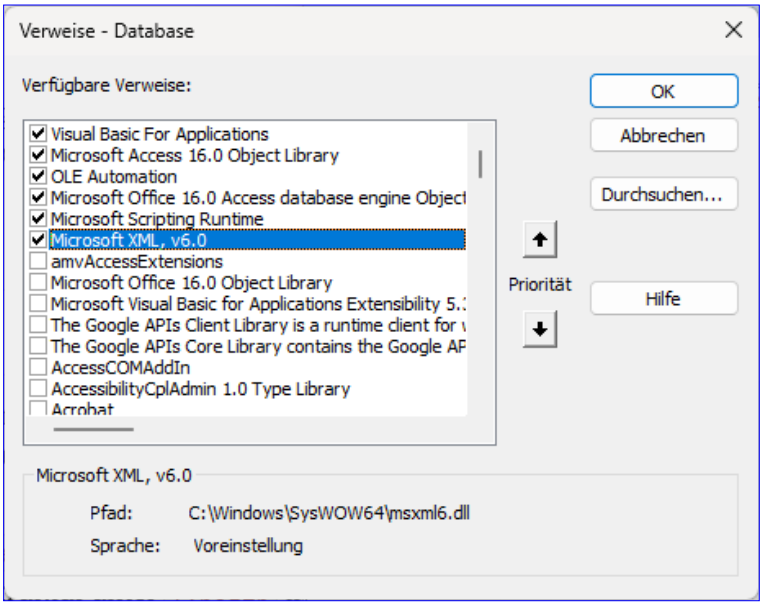


**Bild 1:** Hinzufügen der benötigten Verweise

## Die API: Zippopotam.us

Der Dienst **Zippopotam.us** ist eine schlanke, öffentlich zugängliche REST-API für Postleitzahlen aus über 60 Ländern. Es gibt kein Entwicklerkonto, keinen API-Key und keine Ratenbegrenzung, die für unsere Zwecke relevant wäre.

Ein einziger GET-Aufruf genügt, um zu einer Postleitzahl den Ort und die geografischen Koordinaten zu erhalten.

Die URL hat immer denselben Aufbau:

```
https://api.zippopotam.us/<Länderkürzel>/<PLZ>
```

Für die Postleitzahl 44787 in Deutschland sieht der Aufruf also so aus:

```
https://api.zippopotam.us/de/44787
```

Das Länderkürzel entspricht dem zweistelligen ISO-3166-Code in Kleinbuchstaben, also zum Beispiel **de** für Deutschland, **at** für Österreich oder **ch** für die Schweiz.

## Die JSON-Antwort verstehen

Wenn wir die URL im Browser aufrufen, erhalten wir eine JSON-Antwort, die in etwa so aussieht:

```
{
  "post code": "44787",
  "country": "Germany",
  "country abbreviation": "DE",
  "places": [
    {
```

```
      "place name": "Bochum",
      "longitude": "7.2167",
      "state": "North Rhine-Westphalia",
      "state abbreviation": "NW",
      "latitude": "51.4833"
    }
  ]
}
```

Die Struktur ist flach: Außen liegen einfache Felder wie **post code** und **country**. Das Feld **places** ist ein Array – erkennbar an den eckigen Klammern –, das ein oder mehrere Objekte enthalten kann.

In Deutschland ist einer PLZ nicht zwingend genau ein Ort zugeordnet – das Array kann also mehrere Elemente enthalten. Wie wir damit umgehen, zeigt die Funktion **PLZLookup** weiter unten.

## JSON-Antworten auslesen mit ParseJson

VBA kennt JSON nicht von Natur aus. Um auf die einzelnen Felder zuzugreifen, verwenden wir die Funktion **ParseJson** aus dem Modul **mdlJSON**. Diese wandelt die JSON-Zeichenkette in ein Objektmodell aus **Dictionary**- und **Collection**-Elementen um.

Auf die einzelnen Werte greifen wir dann über verkettete **Item**-Aufrufe zu. Die vollständige Beschreibung dieser Module findest Du in den Artikeln **Mit JSON arbeiten** (**www.vbentwickler.de/361**) und **JSON-Dokumente per Objektmodell zusammenstellen** (**www.vbentwickler.de/412**).

Damit wir aber gar nicht erst rätselmüßig blättern müssen, gibt es den praktischen Helfer **GetJSONDOM** aus dem Modul **mdlJSONDOM**. Dieser nimmt die JSON-Zeichenkette entgegen und gibt für jedes enthaltene Feld genau den VBA-Ausdruck aus, mit dem wir auf den Wert zugreifen.

Diese Prozedur können wir ganz einfach wie folgt gestalten:

```
Public Sub JSONDOMAnzeigen()
    Dim strURL As String
    Dim strResponse As String
    Dim intStatus As Integer

    strURL = "https://api.zippopotam.us/de/44787"
    intStatus = HTTPRequest(strURL, "GET", strResponse)

    If intStatus = 200 Then
        Debug.Print GetJSONDOM(strResponse, True)
    End If
End Sub
```

Die Prozedur führt den Aufruf der Rest-API aus und bekommt das Ergebnis im Rückgabeparameter **strResponse**. Für unsere PLZ-Antwort liefert **GetJSONDOM** im Direktbereich folgendes Ergebnis:

```
objJSON.Item("post code"): 44787
objJSON.Item("country"): Germany
objJSON.Item("country abbreviation"): DE
objJSON.Item("places").Item(1)
    .Item("place name"): Bochum
```

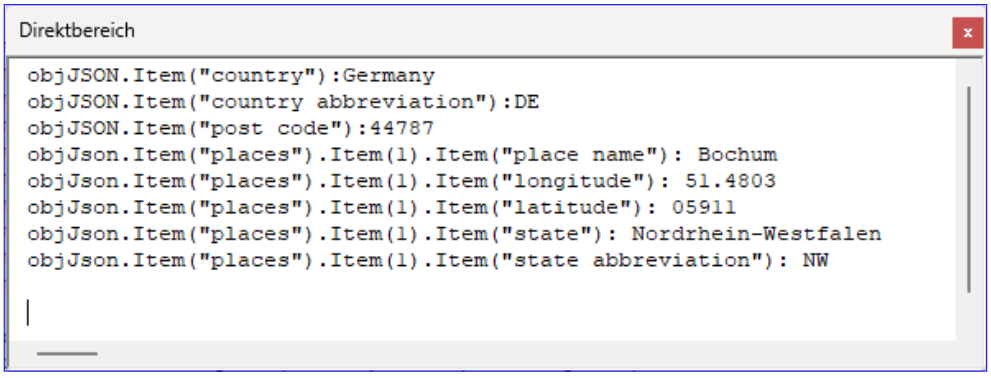


**Bild 2:** Ausgabe von **GetJSONDOM** im Direktbereich des VBA-Editors

```
objJSON.Item("places").Item(1)
    .Item("longitude"): 7.2167
objJSON.Item("places").Item(1)
    .Item("state"): North Rhine-Westphalia
objJSON.Item("places").Item(1)
    .Item("latitude"): 51.4833
```

Wir sehen sofort: Die einfachen Felder sprechen wir direkt mit **objJSON.Item(»Feldname«)** an. Das verschachtelte Feld **places** erfordert einen zusätzlichen **Item(1)**-Aufruf für das erste Element des Arrays – und dahinter nochmals **Item(»Feldname«)** für das gewünschte Unterfeld (siehe Bild 2).

Wenn wir beispielsweise mehrere Elemente für places zurückbekommen würden, könnten wir diese in einer **For...Next**-Schleife durchlaufen.

Damit brauchen wir für keine JSON-Antwort mehr zu rätseln. Wir rufen die API einmal testweise auf, geben das Ergebnis durch **GetJSONDOM** und haben sofort alle benötigten Zugriffspfade schwarz auf weiß.

## Beispieldatenbank

Als Basis dient eine einfache Access-Datenbank mit einer Tabelle namens **tblAdressen**. Die Tabelle enthält die Felder **ID** (Autowert), **Name**, **Strasse**, **PLZ**, **Ort**, **Bundesland**, **Breitengrad** und **Laengengrad**.

Die letzten vier Felder sollen per API befüllt werden, sobald eine neue PLZ eingetragen wird.

Die Funktion **HTTPRequest** aus dem Grundlagenartikel legen wir in einem eigenen Modul namens **mdlHTTP** ab.

## PLZ-Lookup: Die Kernfunktion

Die Funktion **PLZLookup** nimmt eine Postleitzahl und ein optionales Länderkürzel entgegen und gibt ein **Dictionary**-Objekt mit den Ergebnisfeldern zurück. Enthält die Antwort mehrere Orte für dieselbe PLZ, zeigt die Funktion eine **InputBox** an, in der der Benutzer den gewünschten Ort auswählen kann. Liefert die API keinen Treffer, gibt die Funktion **Nothing** zurück. Listing 1 zeigt die vollständige Implementierung.

Die Funktion baut zunächst die URL zusammen und übergibt sie an **HTTPRequest**. Im Erfolgsfall parst sie die Antwort mit **ParseJson** und legt das **places**-Array in der Variable **objPlaces** ab.

Enthält dieses mehr als ein Element, stellt die Funktion eine nummerierte Liste aller Ortsnamen zusammen und zeigt diese in einer **InputBox** an. Der Benutzer gibt die gewünschte Nummer ein – bei ungültiger Eingabe wird automatisch der erste Eintrag verwendet, bei Abbruch gibt die Funktion **Nothing** zurück.

Enthält das Array nur ein Element, wird **intIndex** direkt auf 1 gesetzt. Anschließend füllt die Funktion das **Dictionary** mit den vier Feldern des gewählten Eintrags.

Das Dictionary enthält dann die Elemente **Ort**, **Bundesland**, **Breitengrad** und **Längengrad**.

Ein Beispielaufruf sieht wie folgt aus. Hier holen wir das **Dictionary**-Objekt mit dem Ergebnis und geben den Wert des Eintrags **Ort** aus:

```
Public Sub Test_PLZLookup()
    Dim dic As Scripting.Dictionary
    Set dic = PLZLookup("47137")
    Debug.Print dic("Ort")
End Sub
```

## Einzelnen Datensatz aktualisieren

Mit **PLZLookup** im Gepäck ist die Prozedur zum Aktualisieren eines Datensatzes überschaubar. Die Prozedur **AdresseAktualisieren** in Listing 2 nimmt eine PLZ entgegen, ruft die Funktion auf und schreibt die Ergebnisse per DAO in die Tabelle.

# SQL Server-Verbindungen per Backstage verwalten

**Wer in Access mit verknüpften SQL Server-Tabellen arbeitet, kennt das Problem: Die Verbindungszeichenfolge muss korrekt zusammengesetzt sein, Treiber und Authentifizierungsart müssen stimmen – und bei jeder neuen Datenbank fängt man von vorn an. Dieser Artikel zeigt, wie Du einen eigenen Tab im Backstage-Bereich von Access einrichtest, der Dir das Zusammenbauen der Verbindungszeichenfolge abnimmt.**

## Beispieldatenbank

Die Beispieldatenbank zum Artikel enthält die Tabelle **USysRibbons** mit dem XML für den Backstage-Tab sowie das Modul **mdlBackstage** mit allen Callback-Prozeduren. Du kannst die Datenbank direkt öffnen und den Backstage-Tab sofort verwenden – oder den Code als Vorlage für Deine eigene Datenbank nutzen.

## Was der Backstage-Tab leistet

Der Tab **SQL Server-Verbindung** erscheint im Backstage-Bereich von Access – also in dem Bereich, der sich öffnet, wenn Du auf **Datei** klickst (siehe Bild 1). Er enthält auf der linken Seite alles, was zum manuellen Aufbau einer Verbindungszeichenfolge nötig ist:

- **Treiber-Auswahl:** Ein Dropdown mit den drei gängigsten ODBC-Treibern für SQL Server – **ODBC Driver 17 for SQL Server**, **ODBC Driver 18 for SQL Server** und **SQL Server Native Client 11.0**.

- **Servername mit History:** Eine ComboBox, die sowohl freie Eingabe als auch die Auswahl aus zuvor verwendeten Servernamen erlaubt. Neue Servernamen werden automatisch gespeichert. Ein **Löschen**-Button entfernt den aktuell eingetragenen Namen aus der History.

- **Datenbanken laden:** Ein Button verbindet sich mit dem eingetragenen Server und liest alle verfügbaren Datenbanken aus. Das Ergebnis erscheint als Auswahlliste in der Datenbankname-ComboBox.

- **Authentifizierung:** Eine **RadioGroup** schaltet zwischen Windows-Authentifizierung und SQL Server-

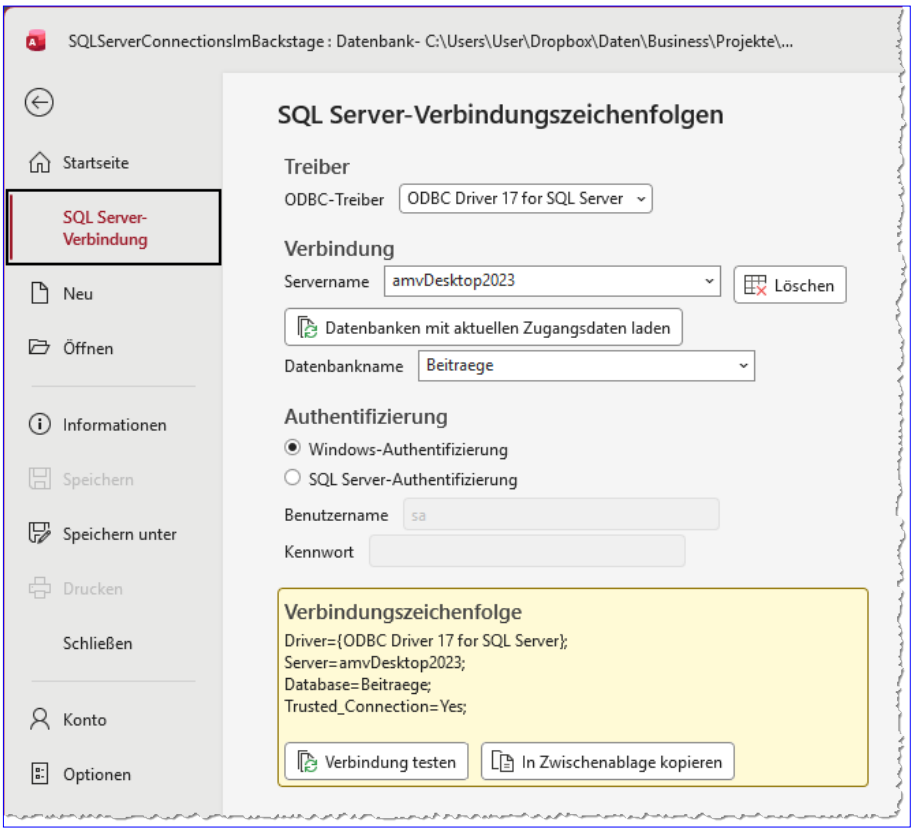


**Bild 1:** Der Backstage-Tab **SQL Server-Verbindung** in Access

Authentifizierung um. Bei SQL Server-Authentifizierung werden die Felder für Benutzername und Kennwort automatisch aktiviert.

- **Verbindungszeichenfolge:** Das Ergebnis aller Eingaben wird in Echtzeit als fertige Verbindungszeichenfolge angezeigt. Zwei Buttons ermöglichen das Testen der Verbindung und das Kopieren in die Zwischenablage.

## So verwendest Du den Backstage-Tab

Nach dem Einrichten des Tabs – wie das geht, beschreibt der nächste Abschnitt – öffnest Du ihn über **Datei|SQL Server-Verbindung**. Beim ersten Aufruf sind alle Felder leer.

- **Schritt 1 – Treiber wählen:** Wähle im Dropdown **ODBC-Treiber** den auf Deinem System installierten Treiber aus. Meist ist das **ODBC Driver 17 for SQL Server**. Der **ODBC Driver 18** erfordert zusätzliche Parameter für die verschlüsselte Verbindung, die automatisch ergänzt werden. Falls Du noch den älteren **Native Client** im Einsatz hast, steht auch **SQL Server Native Client 11.0** zur Verfügung (siehe Bild 2).

- **Schritt 2 – Servername eingeben:** Trage den Servernamen in die ComboBox **Servername** ein. Das kann ein Rechnername, eine IP-Adresse oder ein benannter Instanzname im Format **Rechnername\Instanzname** sein. Sobald Du den Namen bestätigst, wird er automatisch in der History gespeichert und beim nächsten Mal in der Auswahlliste angeboten (siehe Bild 3). Nicht mehr benötigte Einträge entfernst Du über den **Löschen**-Button rechts neben der ComboBox – nach einer Rückfrage wird der aktuell angezeigte Name aus der Liste entfernt.

- **Schritt 3 – Authentifizierung festlegen:** Wähle zwischen **Windows-Authentifizierung** und **SQL Server-Authentifizierung**. Bei Windows-Authentifizierung werden die Felder für Benutzername und Kennwort deaktiviert – die Anmeldedaten des aktuellen Windows-Benutzers werden automatisch verwendet. Bei SQL Server-Authentifizierung werden beide Felder aktiv und müssen ausgefüllt werden (siehe Bild 4).

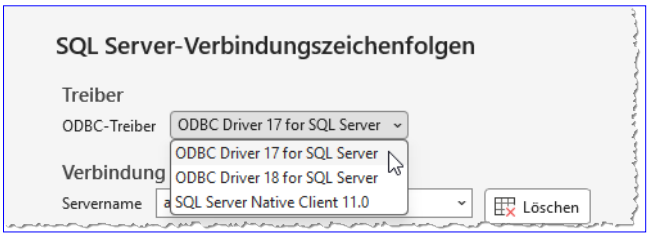


**Bild 2:** Treiber-Auswahl im Dropdown

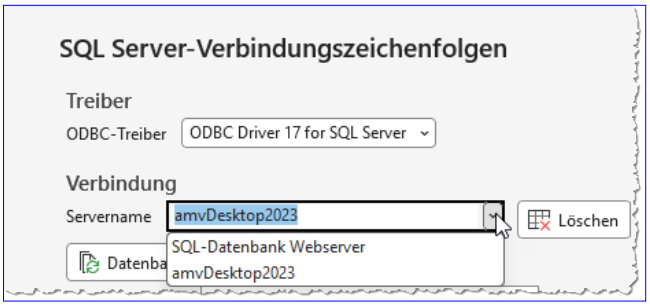


**Bild 3:** Servername-ComboBox mit History-Einträgen

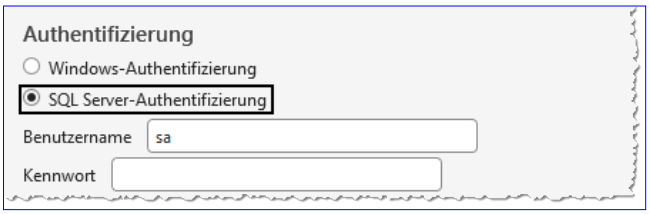


**Bild 4:** Authentifizierungsoptionen

- **Schritt 4 – Datenbanken laden:** Klicke auf **Datenbanken mit aktuellen Zugangsdaten laden**. Das Add-In baut eine Verbindung zum Server auf – ohne Datenbankangabe – und liest alle verfügbaren Datenbanken aus. Das kann je nach Netzwerk und Server einige Sekunden dauern. Die gefundenen Datenbanken erscheinen anschließend in der ComboBox **Datenbankname** (siehe Bild 5). Du kannst den Datenbanknamen alternativ auch direkt eingeben, falls Du ihn kennst.

- **Schritt 5 – Verbindungszeichenfolge prüfen und testen:** Die fertig zusammengesetzte Verbindungszeichenfolge erscheint im unteren Bereich der linken Spalte. Sie wird nach jeder Änderung automatisch aktualisiert und zeilenweise dargestellt, damit sie gut lesbar ist (siehe Bild 6). Über **Verbindung testen** prüfst Du, ob die Verbindung tatsächlich aufgebaut werden kann. Bei Erfolg erscheint eine kurze Bestätigungsmeldung, bei einem Fehler die Fehlerbeschreibung von ADODB – das hilft beim schnellen Eingrenzen von Problemen wie falschem Kennwort, nicht erreichbarem Server oder fehlendem Treiber.

Mit **In Zwischenablage kopieren** übernimmst Du die Zeichenfolge in einem Schritt – ohne die Darstellung im Label manuell markieren zu müssen. Von dort aus kannst Du sie zum Beispiel direkt in den VBA-Editor einfügen oder in eine andere Anwendung übertragen.

## Einrichtung in der Datenbank

Der Backstage-Tab wird über die Systemtabelle **USys-Ribbons** in der jeweiligen Access-Datenbank eingerichtet. Die Tabelle muss einmalig angelegt werden – die Beispieldatenbank zum Artikel enthält sie bereits. Sie besteht aus drei Feldern: **ID** (Primärschlüsselfeld mit Autowert), Ribbon **RibbonName** (Text) und **RibbonXml** (Memo). In **RibbonXml** steht das vollständige XML, das den Tab beschreibt. In den Access-Optionen unter **Aktuelle Datenbank** trägst Du im Feld **Name des Menübands** den Wert aus **RibbonName** ein. Nach dem nächsten Öffnen der Datenbank ist der Tab aktiv.

Die Callback-Prozeduren befinden sich im Modul **mdlBackstage**. Die Verbindungseinstellungen werden in der Windows-Registry unter dem Schlüssel **HKCU\Software\VB and VBA Program Settings\amvSQLServerConnection\SQLServer** gespeichert. Das hat den Vorteil, dass die Einstellungen datenbankübergreifend gelten – Du musst Servernamen und Zugangsdaten nicht in jeder Datenbank neu eingeben.

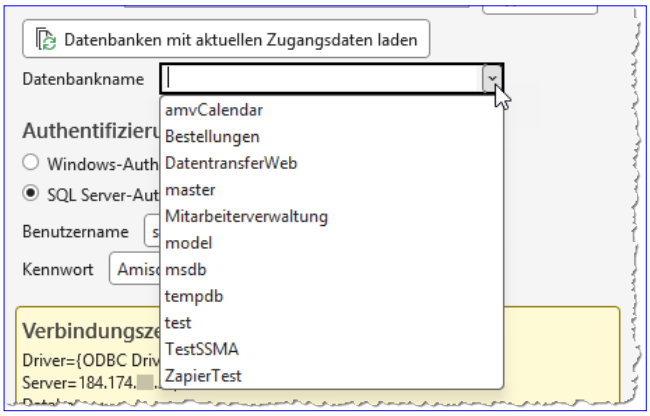


**Bild 5:** Datenbankname-ComboBox nach dem Laden der verfügbaren Datenbanken

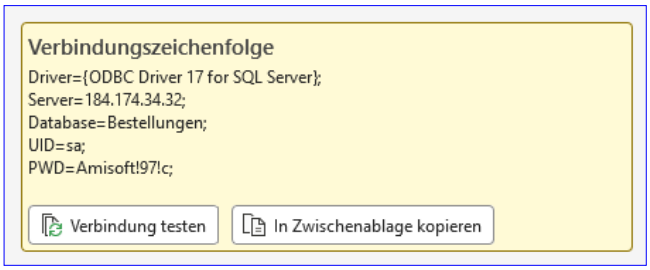


**Bild 6:** Fertige Verbindungszeichenfolge mit zeilenweiser Darstellung

Die Servernamen-History wird unter dem Registry-Wert **Servernamen** als pipe-separierte Liste gespeichert, zum Beispiel:

```
194.163.171.28|SQLSERVER01|LAPTOP\SQLEXPRESS
```

## XML-Definition für den Backstage-Bereich

Das XML beginnt mit dem üblichen **customUI**-Wurzelelement, gefolgt von **<backstage>** und einem **<tab>**-Element mit einer eigenen ID. Das Attribut **insertAfterMso="TabInfo"** sorgt dafür, dass der Tab hinter dem **Informationen**-Tab erscheint. Das Attribut **title** legt die Überschrift fest, die im Backstage über dem Inhalt angezeigt wird:

```
<customUI xmlns="http://schemas.microsoft.com/of-
fice/2009/07/customui"
```

```
    onLoad="customUI_OnLoad">
  <backstage>
    <tab id="tabSQLServer"
         label="SQL Server-Verbindung"
         insertAfterMso="TabInfo"
         title="SQL Server-Verbindungszeichenfolgen">
      <firstColumn>
        ...
      </firstColumn>
    </tab>
  </backstage>
</customUI>
```

Das Attribut **onLoad** verweist auf eine Callback-Funktion, die beim Laden des Ribbons aufgerufen wird und eine Referenz auf das **IRibbonUI**-Objekt liefert. Diese Referenz wird in einer Modulvariablen gespeichert und später benötigt, um das Backstage nach Änderungen neu zu zeichnen:

```
Private m_ribbon As IRibbonUI

Public Function customUI_OnLoad(ribbon As IRibbonUI) _
        As IRibbonUI
    Set m_ribbon = ribbon
    Call ServernamenLaden
    Call DatenbanknamenLeeren
End Function
```

Alle Callback-Prozeduren kommen in ein Standardmodul namens **mdlBackstage**. Für die Registry-Speicherung verwendet die Lösung zwei Konstanten:

```
Public Const cStrAppName As String = _
    "amvSQLServerConnection"
Public Const cStrSection As String = _
    "SQLServer"
```

Die Registry eignet sich hier gut als Speicherort, weil die Verbindungsdaten datenbankübergreifend verfügbar sein sollen – einmal konfiguriert, stehen sie in jeder Datenbank zur Verfügung, die diesen Backstage-Tab verwendet.

Klassen wie **IRibbonUI** sind in der Bibliothek **Microsoft Office 16.0 Object Library** definiert, daher müssen wir noch einen Verweis auf diese Bibliothek zu den Verweisen des VBA-Projekts hinzufügen.

## Steuerelemente im Backstage-Tab

Der Tab verwendet eine einzelne Spalte (**firstColumn**), die in mehrere Gruppen unterteilt ist. Jede Gruppe enthält ein **topItems**-Element, das die eigentlichen Steuerelemente aufnimmt. Im Backstage stehen nicht alle Steuerelemente des Ribbons zur Verfügung – die wichtigsten für diese Lösung sind **dropDown**, **comboBox**, **editBox**, **radioGroup**, **labelControl**, **button** und **layoutContainer**.

Ein wesentlicher Unterschied zum Ribbon: Backstage-Callbacks, die einen Wert zurückliefern, verwenden in VBA **ByRef**-Parameter statt Funktionsrückgabewerten. Die Signatur für einen Text-Callback sieht zum Beispiel so aus:

```
Public Sub ebServername_GetText(control As IRibbonControl, _
        ByRef text)
    text = GetSetting(cStrAppName, _
        cStrSection, "Servername", "")
End Sub
```

Falsche Signaturen führen zu stillen Fehlern – das Steuerelement zeigt einfach nichts an, ohne eine Fehlermeldung zu erzeugen.

## Treiber-Auswahl

Die erste Gruppe enthält ein **dropDown**-Element für die Auswahl des ODBC-Treibers. Die drei verfügbaren Treiber sind im Code fest hinterlegt: **ODBC Driver 17 for SQL Server**, **ODBC Driver 18 for SQL Server** und **SQL Server Native Client 11.0**. Das XML für das Dropdown sieht so aus:

```
<group id="grpTreiber" label="Treiber">
  <topItems>
    <dropDown id="ddTreiber" label="ODBC-Treiber"
              getItemCount="ddTreiber_GetItemCount"
              getItemLabel="ddTreiber_GetItemLabel"
              getItemID="ddTreiber_GetItemID"
              getSelectedItemIndex=_
                  "ddTreiber_GetSelectedItemIndex"
              onAction="ddTreiber_OnAction"/>
  </topItems>
</group>
```

Die Callbacks für ein **dropDown** im Backstage folgen einem einheitlichen Muster. **GetItemCount** liefert die Anzahl der Einträge, **GetItemLabel** den anzuzeigenden Text für jeden Index, **GetItemID** eine eindeutige Zeichenkette je Eintrag und **GetSelectedItemIndex** den aktuell gewählten Index. **OnAction** wird aufgerufen, wenn der Benutzer einen Eintrag auswählt (siehe Listing 1).

Die Hilfsfunktion **GetTreiberName** liefert den Treibernamen für einen gegebenen Index und wird später auch beim Zusammenbauen der Verbindungszeichenfolge verwendet:

```
Private Function GetTreiberName(index As Integer) As String
    Select Case index
        Case 0
            GetTreiberName = "ODBC Driver 17 for SQL Server"
        Case 1
            GetTreiberName = "ODBC Driver 18 for SQL Server"
        Case 2
            GetTreiberName = "SQL Server Native Client 11.0"
        Case Else
            GetTreiberName = "ODBC Driver 17 for SQL Server"
    End Select
End Function
```

Um die Funktion **VerbindungszeichenfolgeAktualisieren** kümmern wir uns weiter unten.

```
Public Sub ddTreiber_GetItemCount(control As IRibbonControl, ByRef count)
    count = 3
End Sub

Public Sub ddTreiber_GetItemLabel(control As IRibbonControl, index As Integer, ByRef label)
    label = GetTreiberName(index)
End Sub

Public Sub ddTreiber_GetItemID(control As IRibbonControl, index As Integer, ByRef id)
    id = "treiber" & index
End Sub

Public Sub ddTreiber_GetSelectedItemIndex(control As IRibbonControl, ByRef selectedIndex)
    selectedIndex = Val(GetSetting(cStrAppName, cStrSection, "TreiberIndex", "0"))
End Sub

Public Sub ddTreiber_OnAction(control As IRibbonControl, selectedID As String, selectedIndex As Integer)
    SaveSetting cStrAppName, cStrSection, "TreiberIndex", selectedIndex
    Call VerbindungszeichenfolgeAktualisieren
    m_ribbon.Invalidate
End Sub
```

**Listing 1:** Callbacks für das Treiber-Dropdown

## Servername mit History

Für den Servernamen kommt eine **comboBox** zum Einsatz – sie erlaubt sowohl freie Eingabe als auch die Auswahl aus einer Liste zuvor verwendeter Servernamen. Die verwendeten Namen werden pipe-separiert in der Registry gespeichert. Daneben gibt es einen **Löschen**-Button, mit dem ein Servername aus der History entfernt werden kann. Beide Steuerelemente werden in einem **layoutContainer** nebeneinander angeordnet:

```
<group id="grpServer" label="Verbindung">
  <topItems>
    <layoutContainer id="lcServername"
                     layoutChildren="horizontal">
      <comboBox id="cbServername"
                label="Servername"
                sizeString="WWWWWWWWWWWWWWWWWW"
                getText="cbServername_GetText"
                onChange="cbServername_OnChange"
                getItemCount="cbServername_GetItemCount"
                getItemLabel="cbServername_GetItemLabel"
                getItemID="cbServername_GetItemID"/>
      <button id="btnServernameLoeschen"
              label="Löschen"
              imageMso="DeleteTable"
              onAction=_
                 "btnServernameLoeschen_OnAction"/>
    </layoutContainer>
    ...
  </topItems>
</group>
```

Die **comboBox** kennt dieselben Item-Callbacks wie das **dropDown**, zusätzlich aber **getText** und **onChange** für den frei eingebbaren Text. **onChange** wird sowohl bei freier Eingabe als auch bei Auswahl aus der Liste aufgerufen. Der neue Servername wird dabei automatisch in der History gespeichert, sofern er noch nicht vorhanden ist.

Bevor wir uns die Callbacks anschauen, werfen wir einen Blick auf die Prozeduren, mit denen wir die Servernamen laden und einen neu eingegebenen Servernamen speichern. Diese finden wir in Listing 2.

Hier verwenden wir das Array **m_astrServernamen** zum Speichern der Servernamen und die Variable **m_intServernamenAnzahl** zum Speichern der Anzahl der enthaltenen Einträge.

Die Prozedur **ServernamenLaden** holt mit **GetSetting** die aktuelle Liste der Servernamen aus der Registry und prüft, ob diese leer ist. In diesem Fall werden die Variablen auf die Anzahl **0** eingestellt beziehungsweise das Array geleert. Anderenfalls lesen wir die Liste mit der **Split**-Funktion als Array in **m_astrServernamen** ein und ermitteln die Anzahl mit der **UBound**-Funktion.

Die Prozedur **ServernameHinzufuegen** hängt den als Parameter angegebenen Servernamen an die Liste der Servernamen an. Dazu lädt sie die Liste und durchläuft alle Einträge. Ist der Eintrag bereits vorhanden, wird er nicht erneut angelegt. Anderenfalls wird er hinten an die Liste angehängt. Danach werden die beiden Variablen **m_astrServernamen** und **m_intServernamenAnzahl** aktualisiert und die Liste der Servernamen wieder in der Registry gespeichert.

## Callback-Funktionen für die Server-ComboBox

Nun folgen die **ComboBox**-Callbacks, die auf das Array zurückgreifen (siehe Listing 3):

- **cbServername_GetText** liest den Text für den aktuellen Eintrag direkt aus der Registry-Einstellung Servername ein.
- **cbServername_OnChange** wird ausgelöst, wenn der Benutzer einen anderen Eintrag auswählt. Dies speichert den gewählten Eintrag für die Einstellung

```
Private m_astrServernamen() As String
Private m_intServernamenAnzahl As Integer

Private Sub ServernamenLaden()
    Dim strListe As String

    strListe = GetSetting(cStrAppName, cStrSection, "Servernamen", "")
    If Len(strListe) = 0 Then
        m_intServernamenAnzahl = 0
        ReDim m_astrServernamen(0)
        Exit Sub
    End If
    m_astrServernamen = Split(strListe, "|")
    m_intServernamenAnzahl = UBound(m_astrServernamen) + 1
End Sub

Private Sub ServernameHinzufuegen(strName As String)
    Dim i As Integer

    If Len(strName) = 0 Then Exit Sub
    Call ServernamenLaden
    For i = 0 To m_intServernamenAnzahl - 1
        If LCase(m_astrServernamen(i)) = LCase(strName) Then
            Exit Sub
        End If
    Next i
    If m_intServernamenAnzahl = 0 Then
        ReDim m_astrServernamen(0)
    Else
        ReDim Preserve m_astrServernamen(m_intServernamenAnzahl)
    End If
    m_astrServernamen(m_intServernamenAnzahl) = strName
    m_intServernamenAnzahl = m_intServernamenAnzahl + 1
    SaveSetting cStrAppName, cStrSection, "Servernamen", Join(m_astrServernamen, "|")
End Sub
```

**Listing 2:** History der Servernamen laden und ergänzen

Servername in der Registry. Außerdem werden die drei Prozeduren **ServernameHinzufuegen**, **DatenbanknamenLeeren** und **Verbindungszeichenfolge-Aktualisieren** aufgerufen, damit die übrigen Elemente aktualisiert werden – gefolgt vom Aufruf der **Invalidate**-Methode, um die Aktualisierungen im Backstage-Bereich sichtbar zu machen.

- **cbServername_GetItemCount** ruft die Prozedur **ServernamenLaden** auf, was die aktuellen Servernamen aus der Registry einliest und die Anzahl der Einträge für das **ComboBox**-Element zurückgibt.

- **cbServername_GetItemLabel** wird für jeden Eintrag einmal aufgerufen und trägt den Text für den jeweiligen Server ein.