AUSGABE 06/2022

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

Paketversand einfach gemacht

Wer regelmäßig Versandetiketten vorbereitet, kennt die langweilige Arbeit, Adressen aus der Kundenverwaltung in das Onlineformular eines der Paketversanddienstleister einzutippen. Wer die Daten aus einer eigenen Anwendung entnimmt, beispielsweise einer Access-Datenbank oder aus einer Excel-Tabelle, kann diesen Vorgang automatisieren: Mit der Lösung aus diesem Heft lernst Du, wie Du als Geschäftskunde von DHL die für den Versand nötigen Daten ganz einfach über die Webservice-API schickst und ein druckfertiges Label im PDF-Format geliefert bekommst.

Mir war diese Arbeit immer ein Graus: Datenbank öffnen, den Kundendatentsatz mit der Bestellung aufrufen, das Onlineportal meines präferierten Dienstleisters öffnen und dann die Daten eintippen, den Bezahlvorgang durchführen, das Versandetikett herunterladen und dieses ausdrucken.

Diese Schritte können wir drastisch vereinfachen! Wenn Du eine Excel-Tabelle oder eine Access-Datenbank mit den Empfängerdaten hast, brauchst Du nur noch ein Geschäftskundenkonto bei DHL und unseren Artikel **DHL-Paketlabel per VBA erstellen** (ab Seite 52). Hier zeigen wir Dir, wie Du per VBA den Webservice von DHL aufrufst, die Versanddaten übermittelst und ein fertiges PDF mit dem Versandetikett herunterlädst – alles per Mausklick.

Falls Du die Grundlagen der VBA-Programmierung noch nicht komplett verinnerlicht hast oder wenn Du hin und wieder nachschlagen musst, wie die eine oder Schleife aufgebaut ist, findest Du im Artikel **VBA Basics: Schleifen** ab Seite 4 alles, was Du brauchst. Hier beschreiben wir die verschiedenen Schleifentypen unter VBA wie **For... Next**, **For Each**, **Do While** oder **Do Until**.

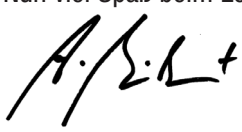
Viele Anwendungen benötigen irgendwann einen Dateidialog – also einen Dialog, mit dem Du eine zu öffnende Datei auswählst, den Namen einer zu speichernden Datei festlegst oder ein Verzeichnis selektierst. All dies kannst Du mit der **FileDialog**-Klasse erledigen, die Teil der Office-Bibliothek ist. Im Artikel **Dateien und Ordner auswählen per FileDialog** lernst Du ab Seite 11 alle Feinheiten kennen, die für den professionellen Einsatz dieser Dialoge nötig ist.

Viele Excel-Workbooks kommen nicht mit einem einzigen Arbeitsblatt aus – die Daten verteilen sich schnell auf verschiedene Arbeitsblätter, die dann die Registerreiter am unteren Rand des Excel-Fensters füllen. Manchmal hast Du sogar mehrere Arbeitsblätter mit ähnlichen Inhalten beziehungsweise Du möchtest mehrere Arbeitsblätter mit den gleichen Daten füllen. Dann ist es hilfreich zu wissen, wie man mehrere Arbeitsblätter gleichzeitig markiert – und das nicht nur manuell, sondern auch per VBA. Im Artikel **Excel: Mit markierten Arbeitsblättern arbeiten** zeigen wir Dir ab Seite 21, wie das gelingt!

In der Outlook-Abteilung findest Du in dieser Ausgabe gleich zwei Artikel: Unter **Outlook: Ordner per VBA im Griff** lernst Du ab Seite 23 alles, was Du für das Anlegen, Löschen, Selektieren oder Bearbeiten von Ordnern in Outlook benötigst. Und unter **Outlook: E-Mail-Anlagen verarbeiten** erfährst Du ab Seite 34, wie Du die Anlagen von E-Mails per VBA im Dateisystem speichern kannst.

Und schließlich liefern wir noch ein schickes COM-Add-In, mit dem Du eine normale Excel-Datei per Ribbonbefehl schnell in eine Excel-Datei mit Makros (.xlsm) umwandeln kannst – das alles ab Seite 42 unter dem Titel **Excel-Datei per COM-Add-In als .xlsm speichern**.

Nun viel Spaß beim Lesen!



Dein André Minhorst

VBA Basics: Schleifen

Wenn wir in VBA einen Vorgang mehr als einmal durchführen wollen, verwenden wir dazu eine sogenannte Schleife. Davon gibt es verschiedene Arten: Einige, wie die »For...Next«-Schleife und die »For Each«-Schleife, werden entsprechend einer vorgegebenen Anzahl durchlaufen, andere solange, wie eine bestimmte Bedingung erfüllt ist – so zum Beispiel die »Do While«-Schleife. Dieser Artikel stellt die verschiedenen Schleifenarten vor und zeigt, welche sich für welchen Einsatzzweck eignet.

Schleifen

Grundsätzlich sind Schleifen Code-Konstrukte und ähneln vom Aufbau her den VBA-Routinen. Sie besitzen einen Schleifenkopf und eine abschließende Anweisung, die allerdings in diesem Fall nur das Ende eines Durchlaufs der Anweisungen innerhalb der Schleife bedeutet. Innerhalb der Schleife werden Anweisungen ausgeführt, solange die Bedingungen für das Fortsetzen der Schleife erfüllt sind. Diese Bedingungen können auf verschiedene Arten aufgebaut sein.

Unter VBA kennen wir die folgenden Schleifentypen:

- **For...Next**-Schleife: Diese Schleife hat eine Laufvariable mit einem Zahlendatentyp sowie einen Start- und einen Endwert. Mit einer zusätzlichen Option können wir festlegen, dass nicht jeder, sondern nur jeder x-te Wert berücksichtigt wird oder auch dass die Werte in umgekehrter Reihenfolge durchlaufen werden.

- **For Each**-Schleife: Die **For Each**-Schleife durchläuft alle Elemente einer Auflistung. Dabei wird der Laufvariablen dieses Schleifentyps jeweils das aktuelle Element der Auflistung zugewiesen.

- **Do While**-Schleife: Diese Schleife wird solange durchlaufen, wie die angegebene Bedingung erfüllt ist. Dementsprechend haben die in der Schleife enthaltenen Anweisungen in der Regel Einfluss auf den als Bedingung angegebenen Ausdruck. Diese Schleife gibt es in zwei Ausführungen – mit der Abbruchbedingung in der ersten und in der letzten Zeile.

- **Do Until**-Schleife: Im Gegensatz zur **Do While**-Schleife, die solange durchlaufen wird, wie die Bedingung erfüllt ist, läuft die **Do Until**-Schleife solange, bis die Bedingung erfüllt ist. Auch die **Do Until**-Schleife gibt es in zwei Ausführungen, von denen die eine die Bedingung in der ersten Zeile prüft und die andere in der letzten Zeile.

Beispiele

Die ersten Beispiele zu diesem Artikel findest Du in in der Excel-Datei **VBABasics_Schleifen.xlsm** oder in der Access-Datenbank **VBABasics_Schleifen.accdb**.

For...Next-Schleife

Die **For...Next**-Schleife zeichnet sich dadurch aus, dass zu Beginn festgelegt wird, wie oft die enthaltenen Anweisungen durchlaufen werden.

Um nachzuhalten, wie oft die Anweisungen innerhalb der Schleife bereits durchlaufen wurden, wird eine sogenannte Laufvariable verwendet, die meistens **i** genannt wird. Für diese Variable legt man einen Zahlendatentyp wie **Integer** oder **Long** fest. Wenn Du den Typ **Integer** wählst, musst Du den recht kleinen Wertebereich von **-32.768** bis **32.767** beachten.

Die erste Zeile der **For...Next**-Schleife enthält außerdem noch den ersten und den letzten Wert sowie ge-

gegebenfalls eine Schrittweite. Die Standardschrittweite lautet **1**. Man gibt nur in Ausnahmefällen eine alternative Schrittweite an – dazu später mehr.

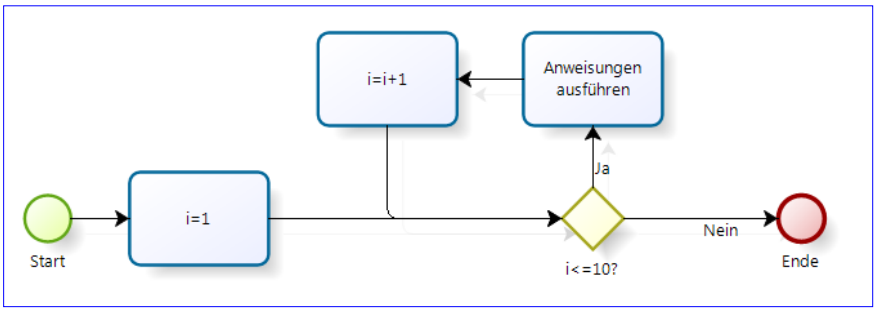


Bild 1: Diagramm für den Ablauf einer einfachen **For...Next**-Schleife

Die letzte Zeile einer **For...Next**-Schleife enthält lediglich das **Next**-Schlüsselwort und optional den Namen der Laufvariablen. Insgesamt sieht eine **For...Next**-Schleife, die beispielsweise mit den Werten von **1** bis **10** für die Variable **i** durchlaufen werden soll, wie folgt aus:

```
Dim i As Integer
For i = 1 To 10
    Debug.Print i
Next i
```

Innerhalb der **For...Next**-Schleife wird der Wert von **i** im Direktfenster ausgeben. Dort erscheinen beim Aufrufen der Prozedur **ForNext_Einfach** des Moduls **mdlBeispieleSchleifen** der Beispieldatenbank also die Zahlen von **1** bis **10**. Der Ablauf der **For...Next**-Schleife wird auch vom Flussdiagramm in Bild 1 skizziert. Die Schleife startet mit dem Wert **1** für die Variable **i**.

Enthält **i** einen Wert kleiner oder gleich dem Endwert der Schleife, werden die Anweisungen des Schleifenkörpers ausgeführt und **i** um eins erhöht.

Monate ausgeben

Die Laufvariable können wir gleich sinnvoll nutzen. Das folgende Beispiel durchläuft etwa die Zahlen von **1** bis **12** und gibt die entsprechenden Monatsnamen aus:

```
Dim i As Integer
For i = 1 To 12
    Debug.Print Format("1." & i & "." & Year(Date), _
       "mmmm")
Next i
```

Schrittweite mit Step einstellen

Der **For...**-Zeile können wir noch das Schlüsselwort **Step** mit der gewünschten Schrittweite hinzufügen.

Wenn wir die Zahlen von **10** bis **1** rückwärts durchlaufen möchten, geben wir als Startwert **10**, als Endwert **1** und für **Step** den Wert **-1** an:

```
Dim i As Integer
For i = 10 To 1 Step -1
    Debug.Print i
Next i
```

Exit For

Gelegentlich werden wir eine **For...Next**-Schleife vorzeitig verlassen wollen. In diesem Fall können wir die **Exit For**-Anweisung verwenden:

```
Dim i As Integer
For i = 1 To 10
    Debug.Print i
    If i = 5 Then
        Exit For
    End If
Next i
```

Verschachtelte Schleifen

Möglicherweise möchten wir einmal zwei oder mehr verschachtelte Schleifen verwenden. Dazu benötigen wir entsprechend viele Laufvariablen, im folgenden Beispiel **i** und **j**:

```
Dim i As Integer
Dim j As Integer
For i = 1 To 5
    For j = 1 To 5
        Debug.Print "(" & j & "," & i & ")",
    Next j
    Debug.Print
Next i
```

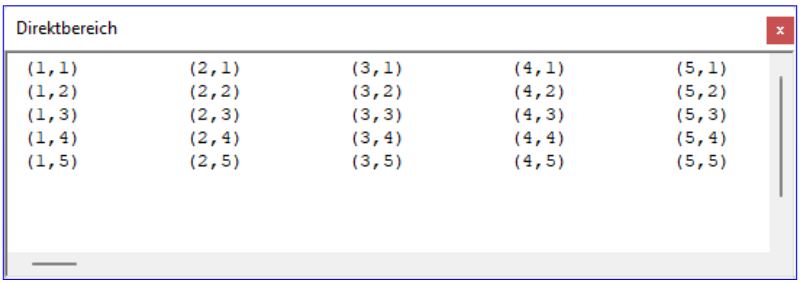


Bild 2: Ausgabe einer Zahlenmatrix mit zwei verschachtelten Schleifen

Die obigen Schleifen geben eine Zahlenmatrix wie in Bild 2 aus. Damit jeweils fünf Einträge nebeneinander ausgegeben werden, beenden wir die **Debug.Print**-Anweisung der inneren Schleife mit dem Komma-Zeichen.

Dies bedeutet, dass die folgende **Debug.Print**-Anweisung in die gleiche Zeile drucken soll – und zwar mit einem Tabulator-Schritt als Abstand.

Wir können auch das Semikolon als letztes Zeichen angeben. Die Ausgabe wird dann unmittelbar hinter dem letzten Zeichen fortgesetzt.

Damit die Ausgabe nach fünf Elementen in der folgenden Zeile fortgesetzt wird, ruft die Prozedur nach der Abarbeitung der inneren Schleife eine **Debug.Print**-Anweisung ohne auszugebenden Text auf.

Dies bewirkt lediglich den Sprung in die nächste Zeile.

Alternative Laufvariablen

Eine Laufvariable muss nicht zwingend eine Ganzzahl des Typs **Integer** oder **Long** sein. Wir können auch mit Dezimalzahlen arbeiten und mit dem **Step**-Schlüsselwort beispielsweise Schritte von **0,1** verarbeiten:

```
Dim s As Single
For s = 0 To 1 Step 0.1
    Debug.Print s
Next s
```

Das Beispiel hat allerdings noch einen kleinen Schönheitsfehler, denn die Ausgabe im Direktfenster sieht so aus:

```
0
0,1
0,2
0,3
0,4
0,5
0,6
0,7
0,8000001
0,9000001
```

Hier fällt erstens auf, dass nach **0,7** ein Rundungsfehler auftritt. Außerdem fehlt der eigentlich erwartete Wert **1**.

Dieses Problem ist typisch für Gleitkommazahlen. Im vorliegenden Fall sollten wir also besser einen Festkomma-Datentyp wie **Currency** verwenden:

```
Dim c As Currency
For c = 0 To 1 Step 0.1
    Debug.Print c
Next c
```

Datum durchlaufen

Wir können auch beispielsweise ein Datum als Laufvariable angeben und alle Tage zwischen zwei angegebenen Datumsangaben durchlaufen.

Dateien und Ordner auswählen per FileDialog

Wenn Du mit VBA programmierst, wirst Du immer wieder mit Dateien arbeiten. Eine der Hauptaufgaben dabei ist, zu öffnende Dateien auszuwählen, Verzeichnisse zu selektieren oder einen Namen für eine zu speichernde Datei festzulegen. Alles drei lässt sich mit verschiedenen Methoden erledigen, aber es gibt eine Klasse, die alles gleichzeitig anbietet – und zwar die FileDialog-Klasse der Office-Bibliothek. In diesem Artikel schauen wir uns an, wie Du Dateidialoge für die verschiedenen Anwendungszwecke öffnen und auswerten kannst.

Beispielumgebung

Wie immer benötigen wir eine Office-Anwendung oder Entwicklungsumgebung, um mit den Objekten, Eigenschaften und Methoden der Klasse **FileDialog** zu experimentieren. In diesem Fall wollen wir einmal ein Excel-Workbook nutzen, das wir logischerweise als **.xlsm**-Datei speichern. Das Worksheet dieser Exceldatei nutzen wir zur Ausgabe der mit der **FileDialog**-Klasse ermittelten Dateien und dort bringen wir auch die zum Aufruf notwendigen Schaltflächen unter.

Die FileDialog-Klasse verfügbar machen

Um auf die **FileDialog**-Klasse und ihre Methoden und Eigenschaften zugreifen zu können, benötigen wir entweder einen Verweis auf die Bibliothek **Microsoft Office x.0 Object Library** oder wir referenzieren diese per Late Binding. Da wir die Vorzüge von IntelliSense zu schätzen wissen, nutzen wir hier die Bibliothek, die wir im VBA-Editor einbinden können.

Von der Excel-Benutzeroberfläche wechseln wir dazu mit der Tastenkombination **Alt + F11** zum VBA-Editor. Hier wählen wir den Menüeintrag **Extras|Verweise** aus. Im nun erscheinenden **Verweise**-Dialog finden wir gegebenenfalls bereits den markierten Eintrag aus Bild 1 vor. Anderenfalls suchen Sie diesen in der alphabetisch sortieren Liste und fügen ihn durch Setzen eines Hakens in das entsprechende Kästchen hinzu.

Einen Dateidialog anzeigen

Grundsätzlich zeigt man einen **FileDialog** wie folgt an:

```
Dim objFiledialog As Office.FileDialog
Set objFiledialog = _
    Application.FileDialog(msoFileDialogFilePicker)
objFiledialog.Show
```

Wir benötigen also eine Objektvariable, die den Typ **Office.FileDialog** aufweist und füllen diese mit einem Verweis auf die **FileDialog**-Klasse unter Angabe des Typs, den wir benötigen – in diesem Fall **msoFileDialogPicker**.

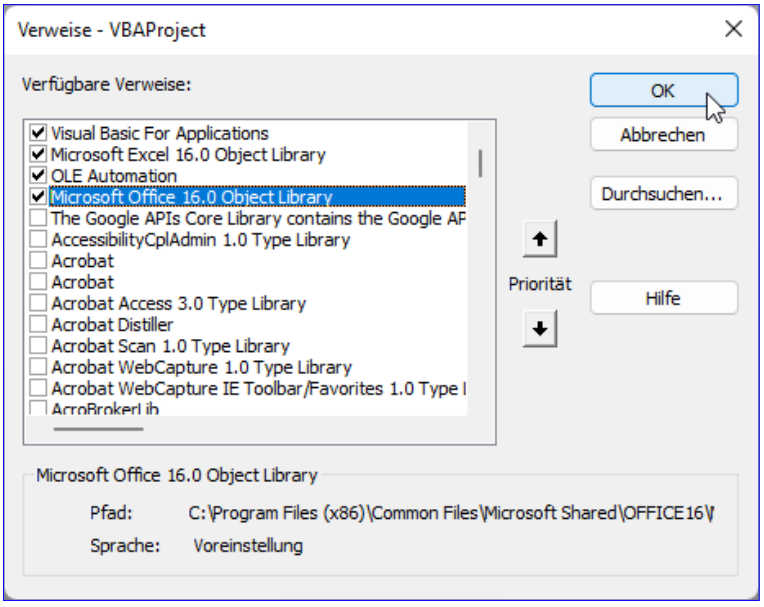


Bild 1: Verweis auf die Objektbibliothek **Microsoft Office 16.0 Object Library**

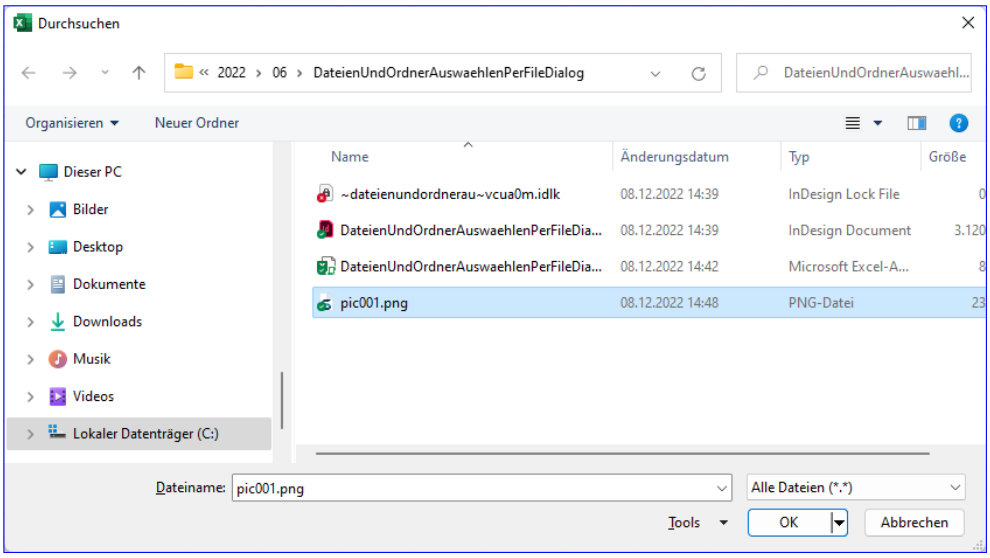


Bild 2: Ein erster Dateidialog zur Auswahl einer Datei

Dann zeigen wir den Dialog mit der Methode **Show** an. Das Ergebnis dieses einfachen Aufrufs finden wir in Bild 2.

Auswahl im Dateidialog auswerten

Nun können wir nach der Auswahl wie im oben angegebenen Beispiel mit dem Ergebnis noch nichts anfangen. Das erledigen wir, indem wir das Ergebnis des Aufrufs prüfen und dann die selektierten Daten auswerten. Die ersten beiden Zeilen bleiben gleich:

```
Dim objFiledialog As Office.FileDialog
Set objFiledialog = _
    Application.FileDialog(msoFileDialogFilePicker)
```

Danach rufen wir die **Show**-Methode jedoch als Teil einer **If...Then**-Bedingung auf und prüfen in dieser das Ergebnis.

Dieses kann **True** oder **False** lauten. Es lautet **True**, wenn der Benutzer die Eingabe mit der **OK**-Schaltfläche abgeschlossen hat, und **False**, wenn er die **Abbrechen**-Schaltfläche wählt. Die **OK**-Schaltfläche kann der Benutzer übrigens erst betätigen, wenn er eine Datei ausgewählt hat.

Im ersten Fall, also wenn der Benutzer eine Datei ausgewählt und auf die **OK**-Schaltfläche gedrückt hat, greifen wir mit **objFileDialog.SelectedItems(1)** auf den ersten ausgewählten Eintrag zu und geben diesen im Direktbereich des VBA-Editors aus.

Im zweiten Fall, also wenn der Benutzer die Abbrechen-Schaltfläche betätigt hat, zeigen wir den Text **Keine Datei ausgewählt** an:

```
If objFiledialog.Show = True Then
    Debug.Print objFiledialog.SelectedItems(1)
Else
    Debug.Print "Keine Datei ausgewählt"
End If
```

Verschiedene Dialogtypen

Wir können verschiedene Dialogtypen nutzen. Dazu gibst Du unterschiedliche Werte beim Zuweisen des **FileDialog**-Objekts an. Diese lauten:

- **msoFileDialogFilePicker**: Zeigt einen Dialog zum Auswählen einer oder mehrerer Dateien an.
- **msoFileDialogFolderPicker**: Zeigt einen Dialog zum Auswählen eines Verzeichnisses an.
- **msoFileDialogOpen**: Zeigt den gleichen Dialog an wie **msoFileDialogFilePicker**, jedoch mit dem Titel **Öffnen**. Ist für Office-Anwendungen wie Word oder Excel geeignet, wo Dokumente so direkt geöffnet werden können.

- **msoFileDialogSaveAs**: Zeigt einen Dialog zum Angeben einer zu speichernden Datei an. Es kann eine vorhandene Datei ausgewählt werden oder auch ein neuer Dateiname angegeben werden. Die Schaltfläche zum Bestätigen der Eingabe ist hier mit **Speichern** beschriftet.

Mehrfachauswahl erlauben

Mit der Eigenschaft **AllowMultiSelect** können Sie festlegen, ob der Benutzer nur einen Eintrag (**False**) oder mehrere Einträge auswählen kann (**True**). Diese Eigenschaft stellen wir nach dem Zuweisen der **FileDialog**-Klasse an die Variable **objFileDialog** ein:

```
objFiledialog.AllowMultiSelect = True
```

Wenn Du so einen **msoFileDialogFilePicker**-Dialog öffnest, kannst Du beispielsweise bei gedrückter **Strg**-Taste mehrere Dateien wie in Bild 3 auswählen.

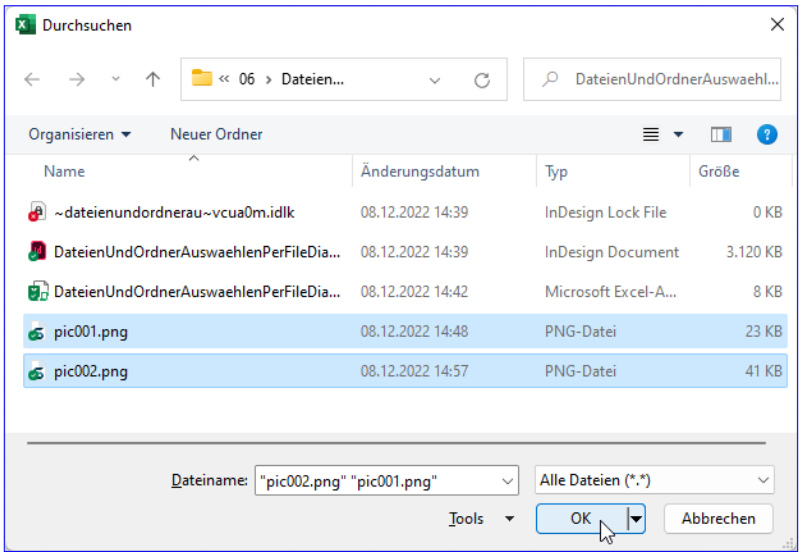


Bild 3: Mehrfachauswahl per **FileDialog**-Objekt

Danach ist allerdings eine andere Auswertung erforderlich als bei der einfachen Auswahl, denn wir erhalten ja nicht nur einen, sondern gegebenenfalls auch mehrere Einträge zurück. Im Beispiel aus Listing 1 durchlaufen wir in einer Schleife alle Einträge der Auflistung **SelectedItems** und geben diese im Direktbereich aus. Dabei muss die als Laufvariable verwendete Variable den Typ **Variant** aufweisen, auch wenn diese lediglich **String**-Werte zugewiesen bekommt.

Beschriftung der Schaltfläche zum Auswählen ändern

Mit der Eigenschaft **ButtonName** können wir die sichtbare Bezeichnung der Schaltfläche zum Auswäh-

```
Public Sub Mehrfachauswahl()
    Dim objFiledialog As Office.FileDialog
    Dim varFilename As Variant
    Set objFiledialog = Application.FileDialog(msoFileDialogFilePicker)
    objFiledialog.AllowMultiSelect = True
    If objFiledialog.Show = True Then
        For Each varFilename In objFiledialog.SelectedItems
            Debug.Print varFilename
        Next varFilename
    Else
        Debug.Print "Keine Datei ausgewählt"
    End If
End Sub
```

Listing 1: Anzeigen und Auswerten einer Mehrfachauswahl

len der gewählten Dateien oder Verzeichnisse ändern. Das gelingt allerdings nicht bei allen **FileDialog**-Typen auf Anhieb, sondern bei manchen erst nach der Auswahl eines Eintrags. Bei den Dialogen der Typen **msoFileDialogFolderPicker** und **msoFileDialogSaveAs** erscheint der gewünschte Text sofort. Das sieht beispielsweise wie folgt aus:

```
Public Sub Schaltflaechenbeschriftung()
    Dim objFiledialog As Office.FileDialog
    Set objFiledialog = _
        Application.FileDialog(msoFileDialog-
FolderPicker)
    objFiledialog.ButtonName = "Ordner aus-
wählen"
    If objFiledialog.Show = True Then
        Debug.Print objFiledialog.SelectedItems(1)
    Else
        Debug.Print "Keine Datei ausgewählt"
    End If
End Sub
```

Das Ergebnis sieht wie in Bild 4 aus. Statt der Beschriftung **Öffnen** erscheint hier also nun der Text **Ordner auswählen**.

Bei den anderen Einstellungen wird der gewünschte Text erst für den Standardtext angezeigt, wenn der Benutzer mindestens eine Datei ausgewählt hat.

Eigenschaften werden beibehalten

Wenn wir diese und die nachfolgend beschriebenen Eigenschaften nutzen, werden die Einstellungen beim erneuten Öffnen eines **FileDialog**-Fensters aus der gleichen Anwendung heraus beibehalten, wenn wir nicht explizit neue Werte für die Eigenschaften angeben. Um eine Eigenschaft wie beispielsweise die Schaltflächenbeschriftung zurückzusetzen, stellen wir diese einmalig auf eine leere Zeichenkette ein:

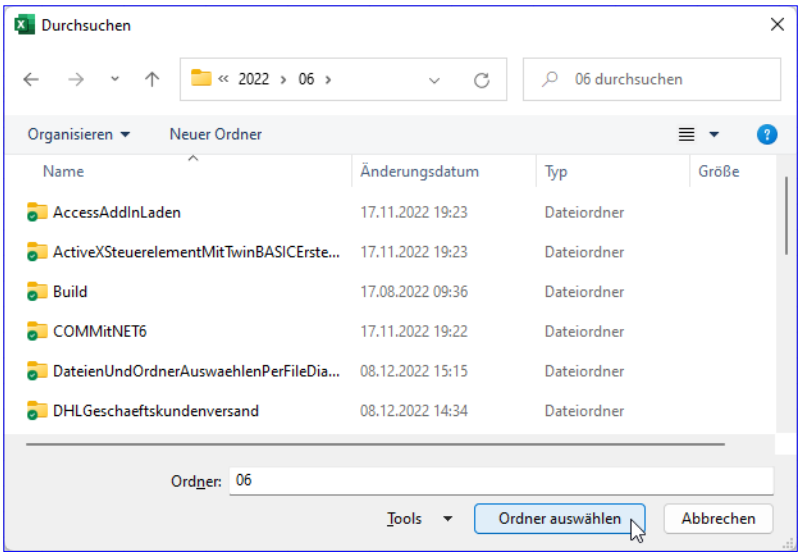


Bild 4: Angepasste Schaltflächenbeschriftung beim Auswählen eines Ordners

```
...
objFiledialog.ButtonName = ""
...
```

Grundsätzlich empfiehlt es sich aber, immer alle Eigenschaften auf die gewünschten Werte einzustellen, um eventuelle Einstellungen, die in der gleichen Sitzung vorgenommen wurden und die eventuell nicht mehr erwünscht sind, zu verwerfen beziehungsweise zu ersetzen.

Überschrift des Dialogs einstellen

Mit der Eigenschaft **Title** können wir den Text in der Titelzeile festlegen:

```
objFiledialog.Title = "Wähle ein oder mehrere Dateien aus."
```

Der Titel erscheint dann wie in Bild 5.

Beim Öffnen anzuzeigenden Ordner einstellen

Wenn wir direkt beim Öffnen des Dialogs einen bestimmten Ordner anzeigen wollen, übergeben wir diesen für die Eigenschaft **InitialFileName**. Den Ordnernamen müssen wir immer mit einem Backslash (\)

Excel: Mit markierten Arbeitsblättern arbeiten

Unter Excel zeigt man in der Regel nur ein einziges Arbeitsblatt an. Allerdings lassen sich, und das wissen nur wenige Benutzer, auch mehrere Arbeitsblätter gleichzeitig markieren. Der Clou: Aktionen, die dann im aktuell angezeigten Arbeitsblatt durchgeführt werden, wirken sich auch auf alle anderen markierten Arbeitsblätter aus. Dies ist Grund genug, dass wir uns ansehen, wie wir die aktuell markierten Arbeitsblätter auslesen können oder wie wir sogar per VBA einige oder alle Arbeitsblätter markieren können.

Eigentlich arbeitet man immer nur in einem einzigen Arbeitsblatt. Manchmal möchte man aber vielleicht Änderungen vornehmen oder ein Arbeitsblatt vorbereiten, von dem man weiß, dass man die Änderungen anschließend auch noch in weitere Arbeitsblätter übertragen muss.

Das Selektieren mehrerer Arbeitsblätter gelingt, wenn Du bei gedrückter **Strg**-Taste die gewünschten Arbeitsblätter markierst. Wenn Du mehrere Arbeitsblätter mit nebeneinander liegenden Registerreitern markieren willst, markierst Du zuerst den ersten Eintrag und dann bei gedrückter **Umschalt**-Taste den letzten.

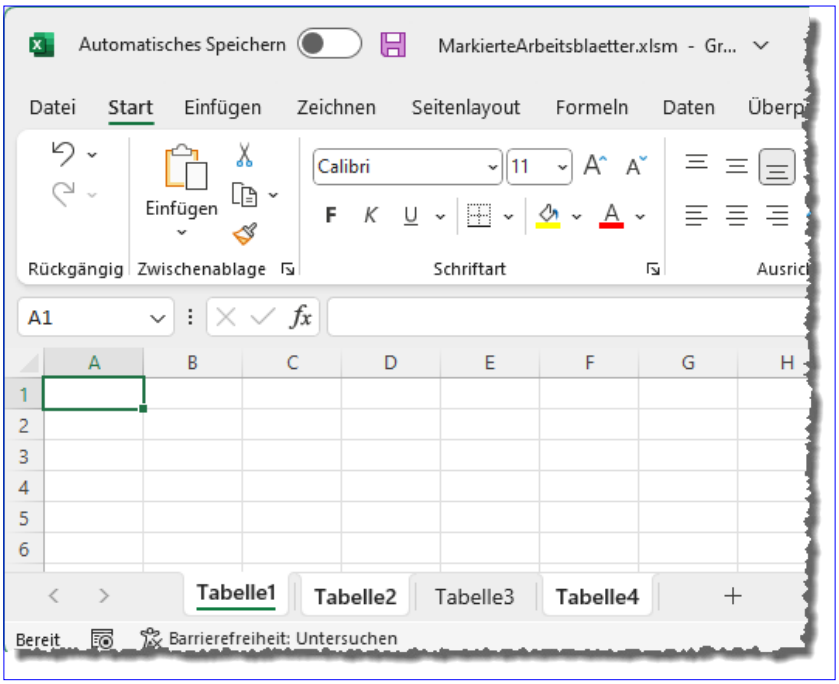


Bild 1: Mehrere markierte Arbeitsblätter

Wenn Du beispielsweise **Tabelle1**, **Tabelle2** und **Tabelle4** wie in Bild 1 markiert hast, und einen Wert in die Zelle **A1** von **Tabelle1** einträgst, dann wird dieser Wert automatisch auch in diese Zelle der anderen markierten Arbeitsblätter eingetragen. Zum Auflösen der Markierung klickst Du beispielsweise einfach ein nicht markiertes Arbeitsblatt an.

Aktuell markierte Arbeitsblätter per VBA auslesen

Um alle aktuell markierten Arbeitsblätter zu ermitteln, nutzen wir die **SelectedSheets**-Auflistung. Diese ist eine Auflistung der **Windows**-Klasse, wobei wir das aktive Fenster mit **ActiveWindow** ermitteln und dann die Elemente der **SelectedSheets**-Auflistung durchlaufen:

```
Public Sub AlleMarkiertenAuslesen()
    Dim wks As Worksheet
    For Each wks In ActiveWindow.SelectedSheets
        Debug.Print wks.Name
    Next wks
End Sub
```

Outlook: Ordner per VBA im Griff

Bei einer frischen Outlook-Installation ist die Ordner-Hierarchie recht übersichtlich. Je mehr E-Mails man mit Outlook erhält oder je mehr man mit den übrigen Objekten arbeitet, umso mehr Ordner legt man in den bereits vorhandenen Ordnern als Unterordner an. Für uns ist natürlich vor allem interessant, wie wir per VBA auf die einzelnen Ordner zugreifen, Ordner ermitteln, anlegen, bearbeiten oder auch löschen. Interessant ist auch, gezielt nach einem Ordner zu suchen und diesen zu referenzieren. Gegebenenfalls möchten wir auch einmal alle vorhandenen Ordner durchlaufen, um die enthaltenen Elemente zu verarbeiten. Wie all dies funktioniert, zeigen wir im vorliegenden Artikel!

Wer eine neue Outlook-Installation öffnet und ein erstes E-Mail-Konto angelegt hat, findet erst einmal nur die Standardordner im linken Bereich des Outlook-Fensters vor. Dazu gehören der Posteingang, der Postausgang, Ordner für gelöschte Elemente, Entwürfe, gesendete Elemente und für Junk-E-Mail (siehe Bild 1).

Wechseln wir über die Schaltflächen unten links zu einem der anderen Bereiche wie beispielsweise dem Kalender, den Kontakten oder den Aufgaben, finden wir im linken Bereich wiederum verschiedene Ordner für die jeweilige Kategorie vor. Wer immer alle Ordner aus den verschiedenen Bereichen im Überblick sehen

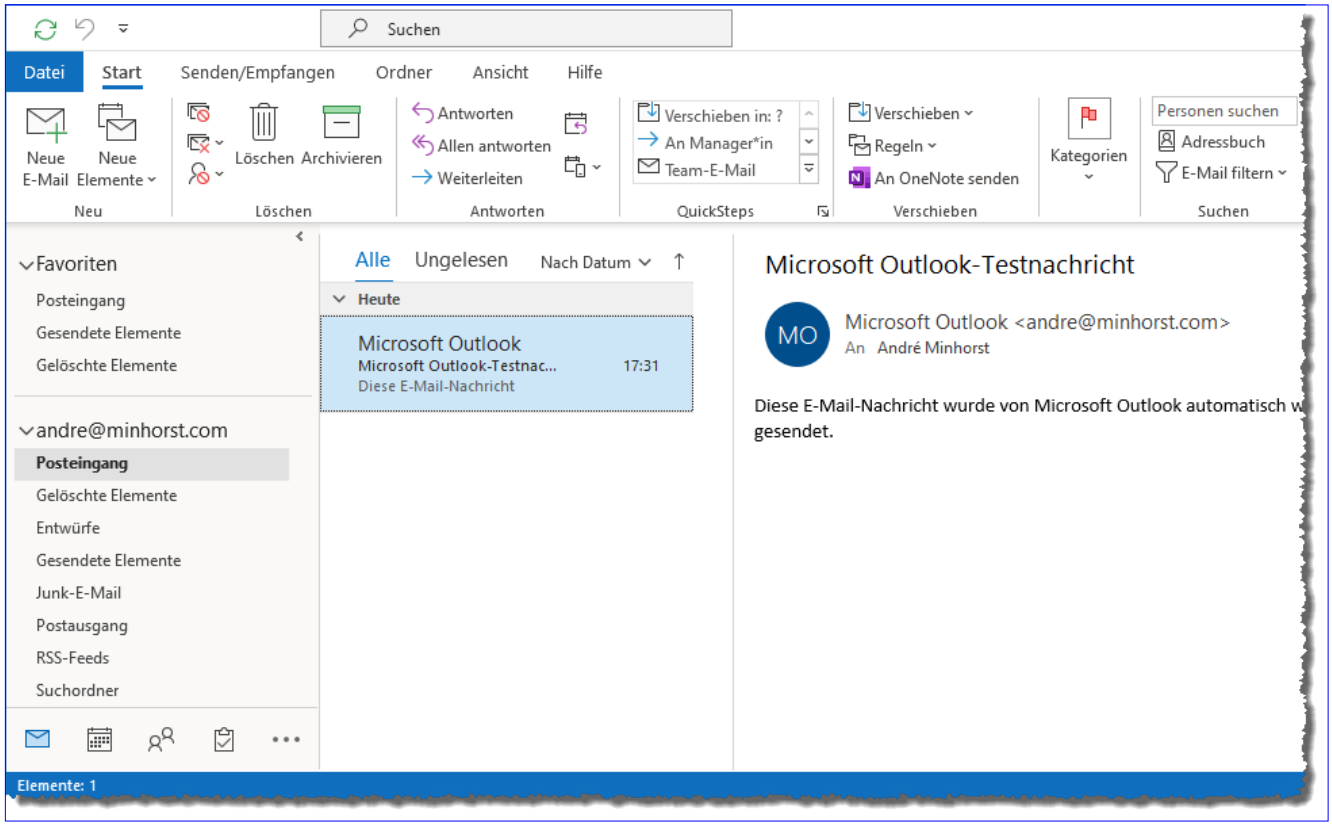


Bild 1: Outlook-Ordner für E-Mails

möchte, kann über die Schaltfläche mit den drei Punkten weitere Befehle anzeigen (sofern diese noch nicht sichtbar sind) und über den Befehl **Ordner** eine Übersicht aller Ordner einblenden. Diese liefert schließlich alle Ordner der verschiedenen Kategorien in einer Liste (siehe Bild 2).

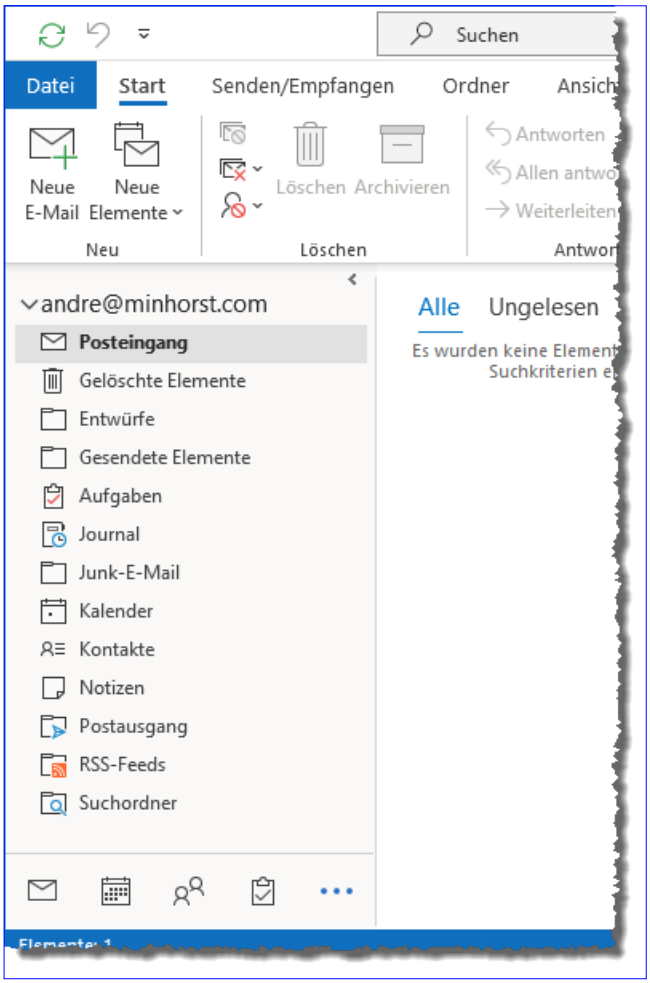


Bild 2: Alle Outlook-Ordner im Überblick

Zugriff von innen und außen

Die Beispiele zu diesem Artikel haben wir so programmiert, dass Du sie sowohl direkt in den Modulen des VBA-Projekts von Outlook verwenden kannst, aber auch in anderen Anwendungen wie Access, Excel oder Word.

Wir greifen also nicht einfach über die Klasse Application auf die davon bereitgestellten Objekte, Eigenschaften, Methoden und Ereignisse zu, sondern über eine Objektvariable namens **objOutlook**, die wir zuvor deklarieren und füllen.

Außerdem wollen wir per Early Binding auf die Klassen zugreifen, damit wir IntelliSense beim Programmieren nutzen können. Wenn Du nicht vom VBA-Projekt von Outlook aus auf die Outlook-Objekte zugreifen möchtest, musst Du dazu noch einen Verweis auf die Objektbibliothek von Outlook hinzufügen.

Dazu öffnest Du den VBA-Editor von der jeweiligen Anwendung aus und wählst dort den Menübefehl **Extras|Verweise** aus.

Es erscheint der **Verweise**-Dialog, wo Du in der Liste den Eintrag **Microsoft Outlook 16.0 Object Library** auswählst (siehe Bild 3). Danach schließt Du den Dialog wieder.

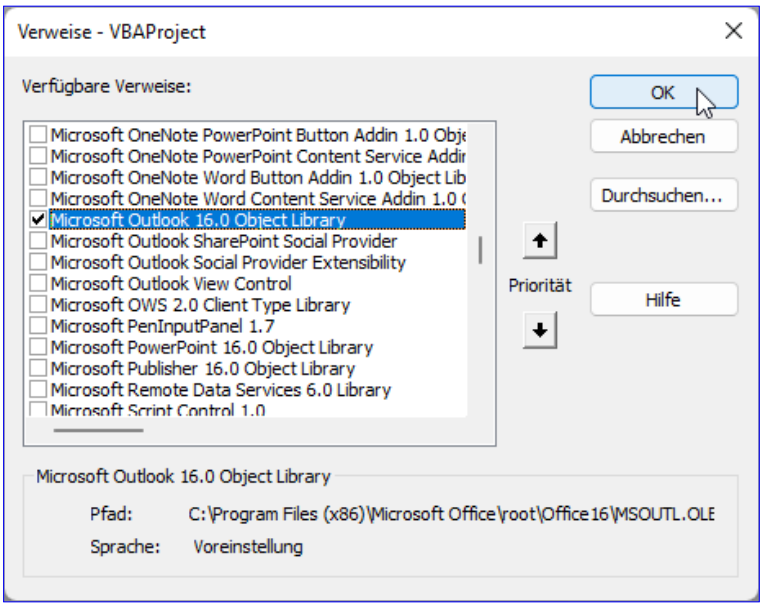


Bild 3: Hinzufügen eines Verweises auf die Outlook-Objektbibliothek

Auf die Outlook-Anwendung greifen wir dann mit den folgenden beiden Befehlen zu:

```
Dim objOutlook As Outlook.Application
Set objOutlook = New Outlook.Application
```

Auf diese Weise können wir auch beispielsweise vom VBA-Projekt einer Excel-Datei oder von anderen Anwendungen darauf zugreifen.

Mutter aller Ordner: Der MAPI-Namespace

Wenn wir direkt auf die Ordner von Outlook zugreifen wollen, können wir nicht direkt irgendeine **Folders**-Auflistung des **Application**-Objekts von Outlook nutzen, sondern es gibt noch eine Zwischenschicht. Diese ist das **Namespace**-Objekt, das wir allerdings nicht mit dem **New**-Schlüsselwort erzeugen, sondern mit der Methode **GetNamespace** holen.

Für diese übergeben wir den Wert **MAPI** als Parameter. Das Namespace-Objekt deklarieren wir beispielsweise als **objMAPI** oder **objNamespace** und weisen es dann wie folgt zu:

```
Dim objMAPI As Outlook.NameSpace
Set objMAPI = objOutlook.GetNamespace("MAPI")
```

Zugriff über die Folders-Auflistung

Die einfachste Methode, auf ein **Folder**-Element zuzugreifen ist die **Folders**-Auflistung des **NameSpace**-Objekts. Mit seiner **Count**-Eigenschaft können wir die Anzahl der **Folder**-Objekte ermitteln, allerdings nur für die der ersten Ebene.

Was im Neuzustand einer Outlook-Datei noch nicht zu erkennen ist: Wir können den vorhandenen Ordnern durchaus Unterordner hinzufügen. Genaugenommen ist das auch schon der Fall, wie der erste Aufruf der **Count**-Eigenschaft der **Folders**-Auflistung zeigt:

```
Debug.Print objMAPI.Folders.Count
1
```

Folder-Element referenzieren

Wir erhalten nämlich genau ein **Folder**-Element. Auf dieses wollen wir nun zugreifen und seinen Namen ermitteln. Dazu referenzieren wir es, was auf verschiedene Arten gelingt. Die erste ist die über den Index. Damit greifen wir wie folgt auf das Element zu:

```
Debug.Print objMAPI.Folders(1).Name
andre@minhorst.com
```

Als Ergebnis erhalten wir den Namen des Hauptordners, in diesem Fall nach der E-Mail-Adresse benannt, für die wir Outlook eingerichtet haben. Je nach Einrichtung kann dieser Ordner auch **Outlook** heißen. Eine weitere wichtige Information hieraus ist: Der Index der **Folders**-Auflistung ist **1**-basiert.

Die zweite Möglichkeit, ein **Folder**-Element über die **Folders**-Auflistung zu referenzieren, ist die Angabe des Namens, sofern dieser bekannt ist.

```
Debug.Print objMAPI.Folders("andre@minhorst.com").Name
```

Was die Eigenschaften und Methoden der **Folder**-Klasse angeht, tappen wir hier etwas im Dunkeln, denn wenn wir ein Element über die Auflistung ansprechen, werden diese nicht per IntelliSense bereitgestellt. Dazu benötigen wir eine entsprechende Objektvariable.

Variable für Folder-Element

Deshalb referenzieren wir als Nächstes ein **Folder**-Element mit einer Variablen. Diese deklarieren wir beispielsweise wie folgt:

```
Dim objFolder As Outlook.Folder
```

Dieser Variablen weisen wir nun das **Folder**-Element hinzu:

```
Set objFolder = objMAPI.Folders(1)
```

Danach können wir leicht per IntelliSense auf die einzelnen Eigenschaften des **Folder**-Elements zugreifen. Wie das gelingt, sehen wir in Bild 4.

Folders-Auflistung durchlaufen

Wenn wir für das Root-Folder-Element die Anzahl der untergeordneten **Folder**-Elemente ermitteln, erhalten wir für unsere frisch installierte Outlook-Anwendung bereits einige Unterordner:

```
Debug.Print objFolder.Folders.Count
14
```

Damit können wir schon eher etwas anfangen und wollen diese nun per Schleife durchlaufen.

Das gelingt sowohl in einer **For...Next**-Schleife als auch in einer **For Each**-Schleife.

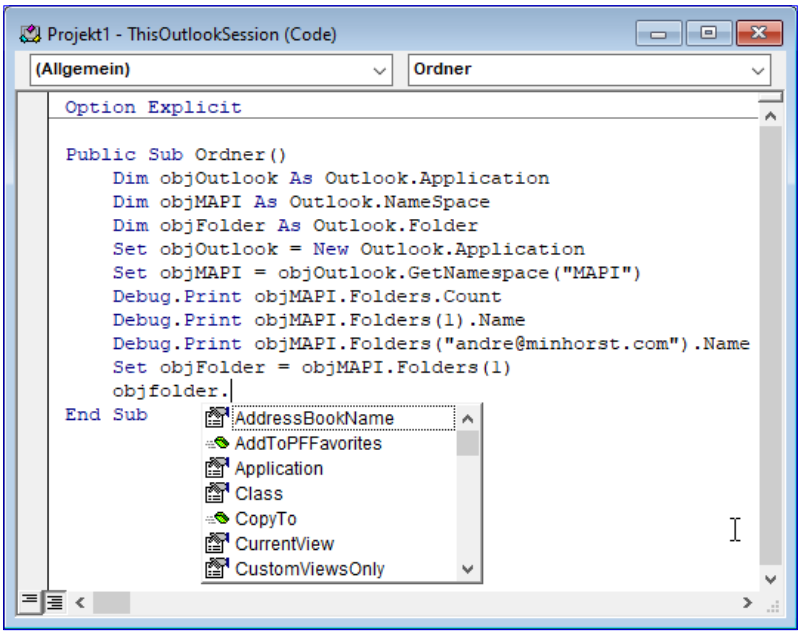


Bild 4: Eigenschaften des **Folder**-Elements per IntelliSense nutzen

Mit der **For...Next**-Schleife durchlaufen wir die Elemente von **1** bis zur Anzahl der Elemente:

```
Dim i As Long
For i = 1 To objFolder.Folders.Count
    Debug.Print i, objFolder.Folders(i).Name
Next i
```

Das Ergebnis sieht wie folgt aus:

```
1           Gelöschte Elemente
2           Posteingang
3           Postausgang
4           Gesendete Elemente
5           Kalender
6           Kontakte
7           Journal
8           Notizen
9           Aufgaben
10          Entwürfe
11          RSS-Feeds
12          Einstellungen für Unterhaltungsaktionen
13          Einstellungen für QuickSteps
14          Junk-E-Mail
```

Mit einer **For Each**-Schleife durchlaufen wir die **Folder**-Elemente wie folgt:

```
Dim objSubfolder As Folder
For Each objSubfolder In objFolder.Folders
    Debug.Print objSubfolder.Name
Next objSubfolder
```

Das Ergebnis ist identisch – mit Ausnahme der fehlenden Werte der Laufvariablen.

Standardordner mit GetDefaultFolder referenzieren

Bevor wir uns die Eigenschaften der **Folder**-Klasse ansehen, wollen wir noch weitere Möglichkeiten zum Referenzieren von **Folder**-Elementen betrachten. Wenn wir die Eigenschaften und Methoden des MAPI-Namespaces per IntelliSense anzeigen, finden wir hier beispielsweise noch die Methode **GetDefaultFolder** (siehe Bild 5).

Wählen wir diese aus und tippen noch eine öffnende Klammer ein, erhalten wir eine Auflistung der möglichen Ordner. Diese lauten:

- **olFolderCalendar**: Ordner für Termine
- **olFolderContacts**: Ordner für Kontakte
- **olFolderDeletedItems**: Ordner für gelöschte Elemente
- **olFolderDrafts**: Ordner für Entwürfe
- **olFolderInbox**: Ordner für den Posteingang
- **olFolderJournal**: Ordner für Journaleinträge
- **olFolder Junk**: Ordner für Junk-E-Mail
- **olFolderNotes**: Ordner für Notizen
- **olFolderOutbox**: Ordner für den Postausgang
- **olFolderRssFeeds**: Ordner für RSS
- **olFolderSentMails**: Ordner für gesendete Objekte
- **olFolderTasks**: Ordner für Aufgaben

Wenn wir also einen dieser Ordner referenzieren wollen, können wir das direkt über die Methode **GetDefaultFolder** erledigen.

Um beispielsweise den Ordner mit dem Parameter **olFolderInbox** zu referenzieren, verwenden wir die folgenden Anweisungen. Die letzte gibt den Anzeigenamen dieses Ordners im Direktbereich des VBA-Editors aus:

```
Dim objInbox As Outlook.Folder
Set objInbox = objMAPI.GetDefaultFolder(olFolderInbox)
Debug.Print objInbox.Name
```

Auf die gleiche Art lassen sich auch die übrigen Ordner referenzieren.

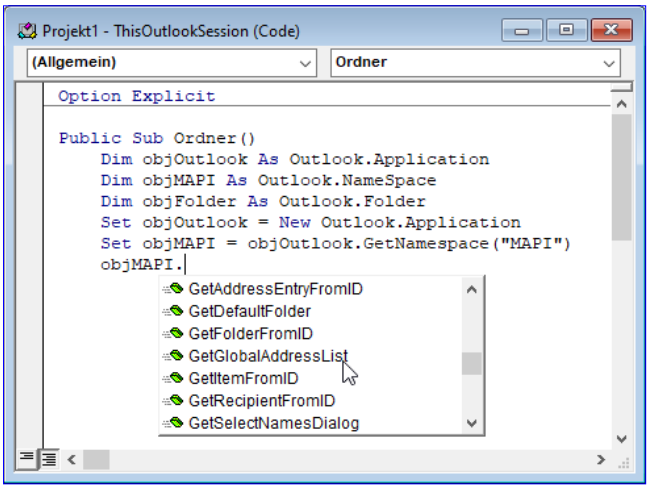


Bild 5: Weitere Methoden zum Referenzieren von **Folder**-Elementen

Ordner per ID referenzieren

Eine weitere Möglichkeit zum Referenzieren eines Ordners ist die Verwendung der Methode **GetFolderFromID**, dem man die eindeutige ID des Ordners übergeben muss. Diese kennen wir üblicherweise nicht, sodass wir diese zuerst ermitteln müssten.

Das erledigen wir bei dem soeben referenzierten Ordner Posteingang wie folgt:

```
Debug.Print objInbox.EntryID
00000000B683105189780 34DA785666C7768FEE282800000
```

Das Ergebnis ist eine 48-stellige Zahl, die sogenannte **EntryID**. Diese können wir der Methode **GetFolderFromID** als Parameter übergeben:

```
Set objInbox = objMAPI.GetFolderFromID("00000000B683105189
78034DA785666C7768FEE282800000")
Debug.Print objInbox.Name
```

Ordner per Dialog auswählen

Eine weitere interessante Möglichkeit, die wir nutzen können, wenn wir den Benutzer einen Ordner auswählen lassen wollen, um diesen später weiterzuverarbeiten, ist die Methode **PickFolder** der **NameSpace**-

Outlook: E-Mail-Anlagen verarbeiten

E-Mails können unterschiedlichste Dateien als Anlagen enthalten. Von der Rechnung über Beispieldatenbanken, Word-Dokumente, PDFs oder Excel-Datenbanken. In manchen Fällen möchte man diese automatisiert weiterverarbeiten – beispielsweise, um Rechnungsdokumente einzulesen oder auch um die Dateien einfach nur an einer Stelle im Dateisystem zu speichern, die gesichert wird. In diesem Artikel zeigen wir, wie wir auf die Anlagen von Outlook-E-Mails zugreifen können. Dazu nutzen wir die Outlook-Bibliothek und greifen per VBA auf die Attachments-Auflistung des MailItem-Objekts zu.

E-Mails und Anlagen

Wie wir E-Mails erstellen und ihnen Anlagen hinzufügen, haben wir bereits in einem früheren Artikel beschrieben. Dieser heißt **E-Mails per VBA erstellen mit CreateItem** (**www.vbentwickler.de/309**). Im vorliegenden Artikel schauen wir uns nun an, wie wir den umgekehrten Weg gehen – also Anlagen aus E-Mails auslesen und weiterverarbeiten.

Um welche E-Mails geht es?

Die erste Frage, die man sich stellen muss, ist die der betroffenen E-Mails. Es gibt verschiedene Möglichkeiten, zum Beispiel:

- nur eine bestimmte E-Mail
- alle E-Mails eines Ordners
- alle E-Mails eines bestimmten Empfängers
- alle E-Mails aus allen Ordnern

Was soll mit den Anlagen geschehen?

Wenn wir Anlagen von E-Mails weiterverarbeiten wollen, geschieht das meist im Kontext einer bestimmten Anwendung. Also beispielsweise in einer Access-Anwendung wie etwa einer Kundenverwaltung.

In dieser will man vielleicht alle E-Mails, die in Zusammenhang mit der E-Mail-Adresse des Kunden stehen, in einer Tabelle der Datenbank speichern und im Detailformular zur Ansicht des Kunden anzeigen. Ein schönes Feature wäre, wenn man per Doppelklick auf eine solche E-Mail direkt die entsprechende E-Mail im Mailfenster öffnen könnte.

Vielleicht hat man auch einen Ordner, in den man alle E-Mails verschiebt, die eine Rechnung enthalten, die man für die Buchhaltung benötigt. Man könnte dann alle E-Mails dieses Ordners durchlaufen und die Rechnungen in Ordner verschieben, die nach dem Empfangsdatum der E-Mail benannt sind – zum Beispiel **2022/12** oder **2023/01**.

Man könnte die Anlagen auch in Ordnern speichern, die als Name die E-Mail-Adresse des Absenders enthalten. So kann man schnell die Anlagen durchsuchen, die von einem bestimmten Absender geschickt wurden.

Wann sollen die Anlagen verarbeitet werden?

Auch hier gibt es verschiedene Möglichkeiten:

- Manueller Abruf. Hier würde man eine Prozedur aufrufen, welche die gewünschten E-Mails oder auch vollständige Ordner durchläuft und alle enthaltenen Anlagen auf die gewünschte Art speichert. Dieser Aufruf würde vermutlich im VBA-Projekt der Anwendung untergebracht werden, mit der die Anlagen weiterverarbeitet werden.

- Automatischer Abruf: Man könnte auch Ereignisprozeduren im Outlook-VBA-Projekt definieren, mit denen man auf das Eintreffen neuer E-Mails reagiert – oder auch auf das Verschieben von E-Mails in einen bestimmten, für das Verarbeiten von Anlagen vorgesehenen Ordner.

In diesem Artikel wollen wir jedoch erst einmal die grundlegenden Techniken für das Speichern von Anlagen betrachten. Dazu müssen wir wissen, wie wir die entsprechenden E-Mails referenzieren und dann die passenden Methoden der **Attachments**-Auflistung nutzen.

Alle Anlagen aus dem Posteingang speichern

Der Einfachheit halber beginnen wir mit den E-Mails, die sich aktuell im Posteingang befinden. Den Ordner für den Posteingang können wir, wie wir auch im Artikel **Outlook: Ordner per VBA im Griff** (**www.vbentwickler.de/340**) beschrieben haben, relativ einfach referenzieren. Wir verwenden nachfolgend die folgende Funktion, um den Ordner **Posteingang** zu referenzieren. Da diese das **Application**-Objekt von Outlook explizit deklariert und referenziert, können wir diesen Code auch außerhalb von Outlook nutzen – also beispielsweise in Access, Excel oder Word:

```
Public Function GetInbox() As Outlook.Folder
    Dim objOutlook As Outlook.Application
    Dim objMAPI As Outlook.NameSpace
    Set objOutlook = New Outlook.Application
    Set objMAPI = objOutlook.GetNamespace("MAPI")
    Set GetInbox = objMAPI.GetDefaultFolder(olFolderInbox)
End Function
```

Damit können wir leicht auf den Posteingangsordner zugreifen und beispielsweise seinen Namen ausgeben:

```
Debug.Print GetInbox.Name
Posteingang
```

E-Mails mit Anlagen im Posteingang durchlaufen

Die nächste Prozedur erlaubt uns, alle E-Mails im Posteingang zu durchlaufen und nur die Betreffzeile derjenigen E-Mails auszugeben, die mindestens eine Anlage enthalten. Dazu durchlaufen wir erst einmal alle Elemente der **Items**-Auflistung in einer **For Each**-Schleife und weisen das aktuelle Objekt der Variablen **objItem** zu. Diese hat den Datentyp **Object**. Warum das und nicht direkt MailItem? Wir wollen doch E-Mails untersuchen? Der Grund ist, dass der Posteingang durchaus auch einmal Termine oder andere Objekte enthalten kann. Wenn wir diese dann mit einer **MailItem**-Objektvariablen referenzieren, lösen wir einen Fehler aus. Also prüfen wir erst einmal anhand der **Class**-Eigenschaft von **objItem**, ob es sich um ein **MailItem**-Objekt handelt. Diese Eigenschaft liefert dann den Wert **olMail**. Ist das der Fall, weisen wir das Objekt aus **objItem** der Objektvariablen **objMailItem** zu. Warum überhaupt wechseln? Weil wir mit einem **MailItem**-Objekt besser IntelliSense nutzen können.

Schließlich prüft die Prozedur mit der **Count**-Eigenschaft der **Attachments**-Auflistung, ob die E-Mail mindestens eine Anlage enthält und gibt für diese E-Mails den Betreff im Direktbereich des VBA-Editors aus:

```
Public Sub AlleMailsImPosteingang()
    Dim objItem As Object
    Dim objMailItem As MailItem
    For Each objItem In GetInbox.Items
        Select Case objItem.Class
            Case olMail
                Set objMailItem = objItem
                If Not objMailItem.Attachments.Count = 0 Then
                    Debug.Print objMailItem.Subject
                End If
        End Select
    Next objItem
End Sub
```

E-Mails mit Anlagen per Restrict ermitteln

Bevor wir uns an die Verarbeitung der Anlagen begeben, schauen wir uns noch eine weitere Möglichkeit an, die E-Mails mit Anlagen aus einem Ordner zu ermitteln.

Im folgenden Beispiel nutzen wir dazu die **Restrict**-Methode der **Items**-Auflistung.

Als Erstes referenzieren wir mit der Variablen **objItems** mit dem Typ **Items** alle Elemente des Posteingangs und geben ihre Anzahl aus:

```
Public Sub MailsMitAttachmentFiltern()
    Dim objItems As Items
    Dim objItemsWithAttachment As Items
    Dim strSQL As String
    Set objItems = GetInbox.Items
    Debug.Print "Alle Mails: " & objItems.count
```

Dann stellen wir in **strSQL** einen Abfrageausdruck zusammen, er alle E-Mails mit Anlage ermittelt:

```
    strSQL = "@SQL=""urn:schemas:httpmail:hasattachment""=1"
```

Diesen wenden wir mit der **Restrict**-Methode auf die Liste **objItems** an, weisen das Ergebnis der Variablen **objItemsWithAttachments** zu und geben die mit Count ermittelte Anzahl der Elemente im Direktbereich des VBA-Editors aus:

```
    Set objItemsWithAttachment = _
        objItems.Restrict(strSQL)
    Debug.Print "Alle Mails mit Anlage: " _
        & objItemsWithAttachment.count
End Sub
```

An dieser Stelle reicht die Information, wie der **Restrict**-Ausdruck aussieht:

```
@SQL="urn:schemas:httpmail:hasattachment"=1
```

Diesen Ausdrücken werden wir einen eigenen Artikel in einer späteren Ausgabe widmen.

Funktion zum Lesen der E-Mails mit Anlage aus einem Ordner

Aus diesen Anweisungen bauen wir uns eine Funktion, der wir als Parameter einen Ordner übergeben und die als Ergebnis eine Liste der Elemente mit Anlagen liefert:

```
Public Function GetMailsWithAttachment( _
        objFolder As Outlook.Folder) As Items
    Dim objItems As Items
    Dim objItemsWithAttachment As Items
    Dim strSQL As String
    Set objItems = objFolder.Items
    strSQL = "@SQL=""urn:schemas:httpmail:hasattachment""=1"
    Set objItemsWithAttachment = objItems.Restrict(strSQL)
    Set GetMailsWithAttachment = objItemsWithAttachment
End Function
```

Diese Funktion können wir dann gemeinsam mit **GetInbox** wie folgt nutzen:

```
Public Sub AlleMailsMitAnlageImPosteingang()
    Dim objItemsWithAttachment As Items
    Dim objMailItem As Outlook.MailItem
    Set objItemsWithAttachment = _
        GetMailsWithAttachment(GetInbox)
    For Each objMailItem In objItemsWithAttachment
        Debug.Print objMailItem.Subject
    Next objMailItem
End Sub
```

Alle Anlagen im Posteingang im Dateisystem speichern

Nun wollen wir die in den E-Mails enthaltenen Anlagen in einem Verzeichnis im Dateisystem des aktuellen Rechners speichern. Dazu nutzen wir eine Prozedur namens **SaveAttachments**, die zwei Parameter erwartet:

Excel-Datei per COM-Add-In als .xlsm speichern

Wer viel mit Excel-Worksheets arbeitet und diesen regelmäßig VBA-Code hinzufügt, muss diese als .xlsm-Datei speichern, damit die Änderungen am VBA-Projekt beim Schließen nicht verlorengehen. Dazu muss man immer den Backstage-Bereich von Excel öffnen und einige Mausklicks durchführen. Wie wäre es, wenn man diese Aktion direkt im Backstage-Bereich finden würde – und nur noch einen Mausklick tätigen müsste, damit das aktuelle Excel-Worksheet nicht nur unter dem gleichen Namen und der Dateiendung .xlsm gespeichert wird, sondern auch noch die ursprüngliche .xlsx-Datei gelöscht wird? Wie das geht, zeigen wir in diesem Artikel.

Vorbereitung: twinBASIC

Wie Du twinBASIC herunterlädst und installierst, haben wir im Artikel **twinBASIC: Visual Basic für die Zukunft** (**www.vbentwickler.de/310**) beschrieben.

COM-Add-In erstellen

Um das COM-Add-In zu erstellen, startest Du **twinBASIC** und wählst im Startbildschirm unter **Samples** den Eintrag **Sample 5. MyCOMAddin** aus (siehe Bild 1). Danach speichern wir das neu angelegte Projekt erst einmal unter dem gewünschten Namen, in diesem Fall **MakeXLSM**.

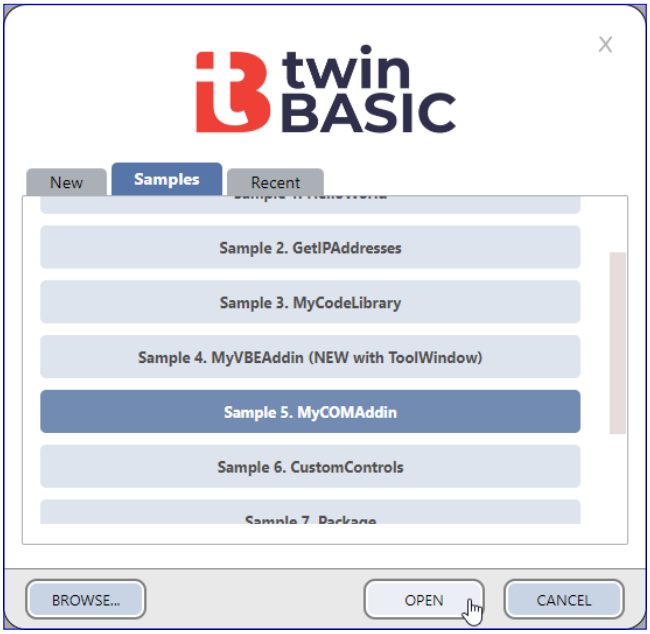


Bild 1: Projekt auf Basis der Vorlage **MyCOMAddin** erstellen

Verweis auf die Excel-Bibliothek hinzufügen

Damit wir direkt per IntelliSense auf die Elemente der Excel-Bibliothek zugreifen können, fügen wir dem Projekt einen entsprechenden Verweis hinzu. Dazu wechseln wir in twinBASIC zum Bereich Settings. Hier scrollen wir nach unten zum Bereich **COM Type Library / ActiveX References**.

Unter **All Available COM References** aktivieren wir mit einem Klick auf das Lupe-Symbol ein Suchfeld und geben **Excel** ein.

Der Eintrag **Microsoft Excel 16.0 Object Library** erscheint und wir fügen einen Verweis auf diese Bibliothek hinzu, indem wir das Kästchen vor diesem Eintrag anhaken (siehe Bild 2).

Name und Beschreibung des Projekts anpassen

Wenn wir schon die Einstellungen anpassen, können wir auch gleich den Namen und die Beschreibung des Projekts ändern. Dazu passen wir die beiden Eigenschaften **Project: Name** und **Project: Description** im oberen Bereich der Einstellungen an (siehe Bild 3).

Einstellungen speichern

Diese Änderungen speichern wir mit der Tastenkombination **Strg + S**, was in der Regel einen Neustart von twinBASIC auslöst.

Projektdateien anpassen

Schließlich ändern wir auch noch den Dateinamen von der Vorgabe **COMAddIn.twin** in **MakeXLSM.twin** und den Klassennamen von **COMAddIn** in **MakeXLSM** (siehe Bild 4).

Die Klasse MakeXLSM anpassen

In der Klasse **MakeXLSM**, die beispielsweise die Implementierung der Schnittstellen **IDTExtensibility2** und **IRibbonExtensibility** übernimmt, nehmen wir einige Änderungen vor. Die wichtigsten fassen wir nachfolgend zusammen.

Wir deklarieren eine Variable namens **objExcel** mit dem Datentyp **Excel.Application**, der wir in der Ereignismethode **OnConnection** den Verweis auf die Excel-Instanz zuweisen, durch die das COM-Add-In aufgerufen wird:

```
Private objExcel As Excel.Application
```

Außerdem deklarieren wir eine Variable, mit der wir später die Ribbondefinition referenzieren wollen. Dies hat den Nutzen, dass wir dann die Methode **Invalidate** aufrufen können, um unsere Ribbon-Steuerelemente per Code zu aktualisieren. Die Deklaration sieht wie folgt aus:

```
Public objRibbon As IRibbonUI
```

Referenz zur Excel-Anwendung herstellen

Die Variable **objExcel** füllen wir wie folgt in der Prozedur **OnConnection**:

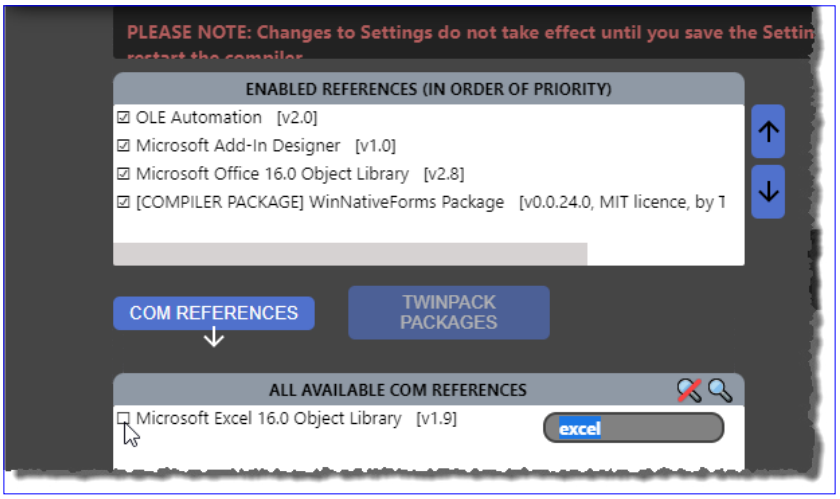


Bild 2: Hinzufügen eines Verweises auf die Excel-Bibliothek

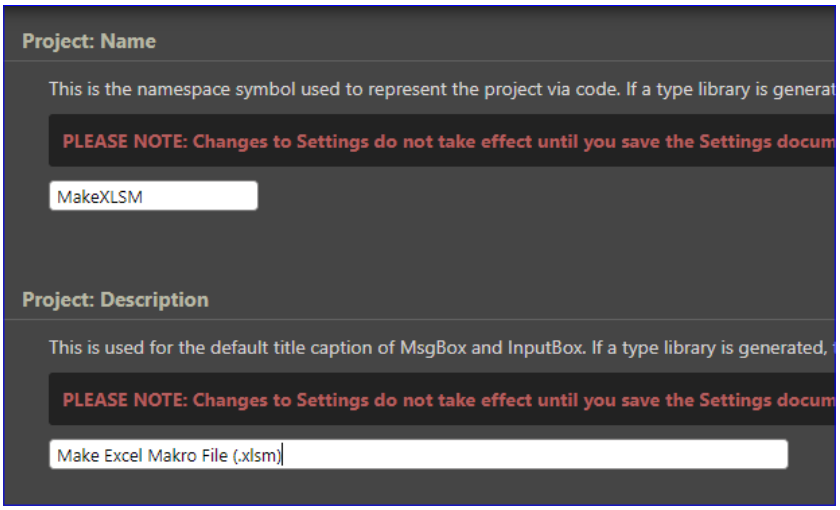


Bild 3: Einstellen von Projektname und -beschreibung

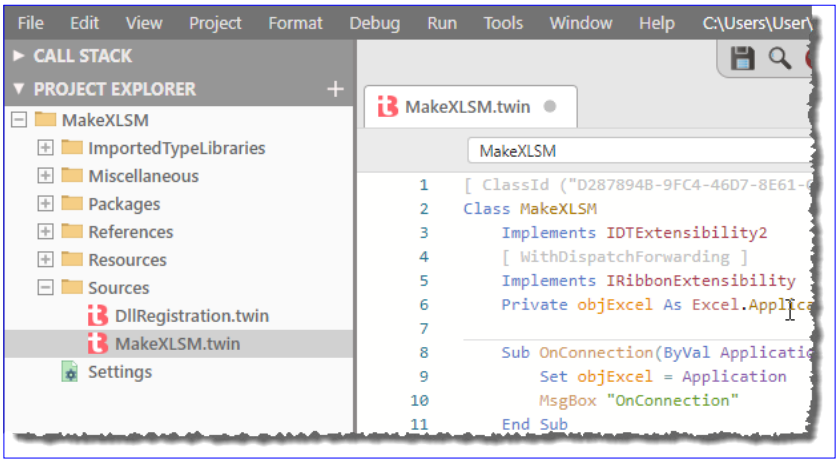


Bild 4: Einstellen des Klassennamens und des Dateinamens

```
Private Function GetCustomUI(ByVal RibbonID As String) As String Implements IRibbonExtensibility.GetCustomUI
    Dim strXML As String
    strXML &= "<?xml version=""1.0""?>" & vbCrLf
    strXML &= "<customUI xmlns=""http://schemas.microsoft.com/office/2009/07/customui"" loadImage=""loadImage"" " _
        & "onLoad=""onLoad"">" & vbCrLf
    strXML &= "  <backstage>" & vbCrLf
    strXML &= "    <button id=""btnMakeXLSM"" onAction=""onAction"" insertAfterMso=""TabInfo"" " _
        & "getEnabled=""getEnabled"" label=""In .xlsm umwandeln"" image=""console.ico""/>" & vbCrLf
    strXML &= "  </backstage>" & vbCrLf
    strXML &= "</customUI>" & vbCrLf
    Return strXML
End Function
```

Listing 1: Die Funktion **LoadCustomUI** lädt die in **strXML** zusammengestellte Ribbondefinition.

```
Sub OnConnection(ByVal Application As Object, _
        ByVal ConnectMode As ext_ConnectMode, _
        ByVal AddInInst As Object, _
        ByRef custom As Variant()) _
        Implements IDTExtensibility2.OnConnection
    Set objExcel = Application
End Sub
```

Befehl zum Backstagebereich hinzufügen

Bevor wir uns um die eigentliche Funktion kümmern, wollen wir uns erst einmal in die Benutzeroberfläche von Excel einnisten, und zwar in den Backstage-Bereich. Hier wollen wir den Befehl **In .xlsm umwandeln** direkt links in der Liste der übrigen Befehle zum Öffnen der verschiedenen Bereiche oder zum Aufrufen von Funktionen integrieren.

Dazu definieren wir in der Funktion **GetCustomUI** die benötigte Ribbon-Anpassung. Die Funktion sieht anschließend wie in Listing 1 aus.

Die Anpassung fügt im Wesentlichen einen Eintrag zum **Backstage**-Bereich von Excel hinzu. Dieser erhält die Beschriftung **In .xlsm umwandeln** und soll das Icon **console.ico** anzeigen, welches wir gleich zum Projekt hinzufügen. Außerdem soll das neue Backstage-Element bei mir relativ weit oben angezeigt werden, weil ich es oft benötige. Wenn Du es lieber an einer anderen Position ablegen möchtest, kannst Du den Wert des Attributs **insertAfterMso** nutzen und die **idMso** des jeweiligen Elements dort eintragen. Wie findest Du die **idMso**? Indem Du in den Excel-Optionen zum Bereich **Menüband anpassen** wechselst, unter **Befehle auswählen** den Eintrag **Registerkarte "Datei"** selektierst und mit der Maus über den Befehl fährst, vor oder hinter dem Du das neue Steuerelement einfügen möchtest (siehe Bild

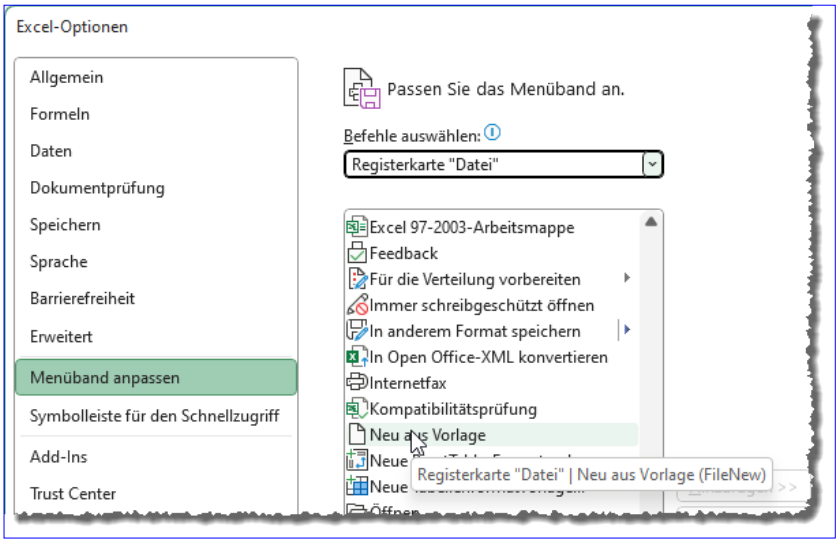


Bild 5: Ermitteln der idMso für ein Backstage-Element

5). Aber Achtung: Es funktioniert leider nicht für alle Elemente. Zumindest **FileSave** haben wir aber noch erfolgreich ausprobiert.

Ein weiteres wichtiges Element in der Ribbondefinition des **button**-Elements ist das Attribut **onAction**, für das wir den Namen der VBA-Prozedur angeben, die dadurch aufgerufen werden soll und die wir weiter unten beschreiben.

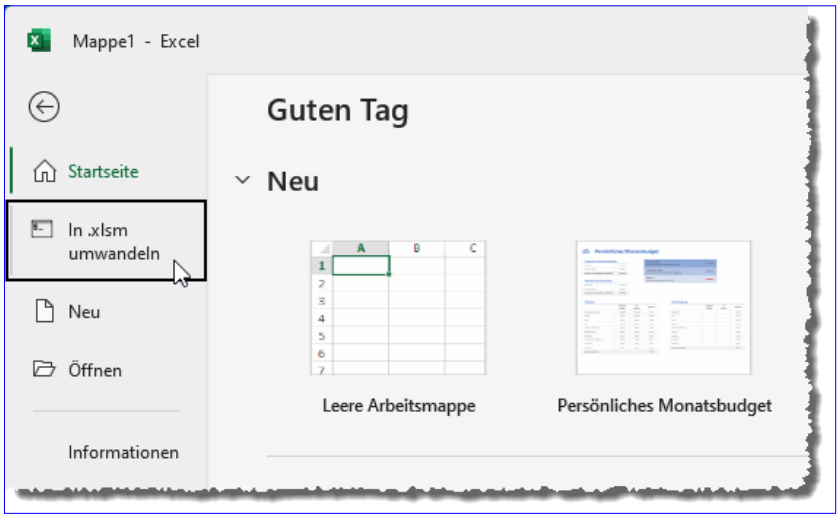


Bild 6: Unser Befehl im Backstage-Bereich

Das Attribut **getEnabled** benötigen wir, weil wir das **button**-Element dynamisch aktivieren oder deaktivieren wollen. Die für dieses Attribut hinterlegte Funktion liefert den Wert **True** oder **False** und aktiviert oder deaktiviert so die Schaltfläche. Warum sollten wir den Button deaktivieren? Weil es passieren kann, dass das geladene Excel-Workbook bereits das Format **.xlsm** aufweist. Dann benötigen wir den Befehl natürlich nicht mehr.

Auch im Element **customUI** haben wir noch zwei Attribute definiert. Für das Attribut **loadImages** hinterlegen wir den Namen der Funktion, die für jedes Element, welches das Attribut **image** aufweist, das jeweilige Icon zurückliefert. Und für **onLoad** legen wir die die Funktion fest, die beim Laden der Ribbondefinition ausgelöst wird und mit der wir eine Objektvariable mit einem Verweis auf die Ribbondefinition füllen können. Die Variable namens **objRibbon** haben wir weiter oben bereits deklariert. Nun fehlt noch die Definition der Funktion **onLoad**, die wie folgt aussieht:

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

Icon hinzufügen

Damit wir mit unserem Befehl ein Icon anzeigen können (siehe Bild 6), fügen wir dieses zuerst zum Ordner **Resources|ICON** hinzu.

Dazu wählen wir per Rechtsklick den Kontextmenübefehl **Add|Import File...** dieses Ordners aus und selektieren dann die gewünschte **.ico**-Datei.

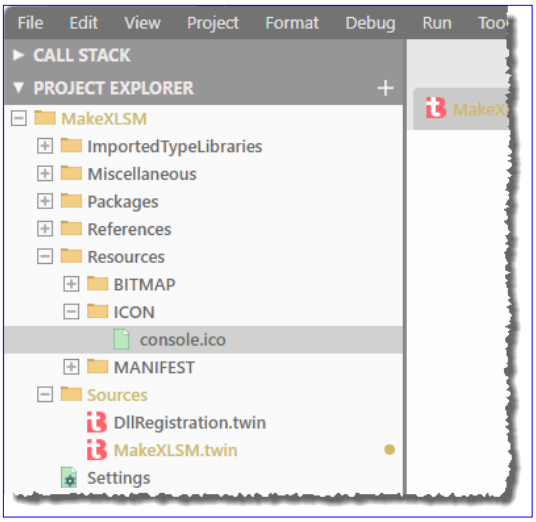


Bild 7: Hinzufügen eines Icons

In unserem Fall heißt die Datei **console.ico** (siehe Bild 7).

Zum Einlesen der Bilddateien in das Ribbon fügen wir die Funktion **LoadImage** aus Listing 2 zum Modul **MakeXLSM.twin** hinzu. Diese nimmt den Namen der zu ladenden Bilddatei entgegen und prüft, ob es sich dabei um eine **.bmp**- oder **.ico**-Datei handelt.

DHL-Paketlabel per VBA erstellen

Beim Versand von Paketen kommt recht oft DHL zum Einsatz. Die einfachste Form der computergestützten Erstellung von Paketlabeln ist dabei das Anmelden bei DHL und das Erstellen eines Labels durch Eingabe der Adressdaten und Bezahlung auf der Webseite. Etwas mehr Möglichkeiten bietet ein Geschäftskundenkonto beim Anbieter DHL. Hier können wir einen Schritt weitergehen und die Adressdaten beispielsweise per CSV übermitteln. Noch schöner wäre es, wenn wir von der jeweiligen Anwendung aus – ob es sich nun um eine Access-Datenbank, eine Excel-Tabelle oder sogar Outlook handelt – direkt per Mausklick ein Paketlabel zur Sendung eines Pakets an den jeweiligen Kontakt erstellen könnten, dass dann beispielsweise als PDF auf unserem Rechner landet. Wie das gelingt, zeigen wir im vorliegenden Artikel.

Voraussetzung: Geschäftskundenkonto bei DHL

Bevor Du weiter in diesen Artikel einsteigst, hier ein wichtiger Hinweis: Um die Beispiele in der Praxis einzusetzen, benötigst Du ein Geschäftskundenkonto bei DHL – beziehungsweise der Kunde, für den Du diese Anwendung erstellst. Zum Ausprobieren der Beispiele reicht ein einfaches Entwicklerkonto bei DHL aus. Wie Du ein Geschäftskundenkonto erstellst, wollen wir hier nicht demonstrieren – aber zumindest den Weg dorthin. Dazu folgst Du diesem Link und klickst dann auf Geschäftskunden (siehe Bild 1):

```
https://www.dhl.de
```

Entwickler-Konto bei DHL erstellen

Wenn Du das für die Umsetzung benötigte Entwicklerkonto bei DHl erstellen möchtest, folgst Du diesem Link:

```
https://entwickler.dhl.de/
```

Hier findest Du im unteren Bereich einen Link mit dem Text **Registrieren Sie sich jetzt**. Dieser führt bereits zur Registrierungsseite, wo zunächst nur eine Entwickler-ID, die E-Mail sowie ein Kennwort einzugeben sind – nebst Bestätigung von AGB und Datenschutzerklärung.

Übersicht der Dienste

Nach der Registrierung des Entwicklerkontos und der ersten Anmeldung sehen wir die zur Verfügung stehenden Dienste (siehe Bild 2). Uns interessiert als Erstes der **Geschäftskundenversand**. Nach einem Klick auf den entsprechenden Link landen wir direkt auf der Seite **Geschäftskundenversand API**, die erste Informationen für uns bereithält. Hier finden wir zum Zeitpunkt der Erstellung des Artikels die Information, dass es eine neue Version mit der Nummer 3.3 mit neuer

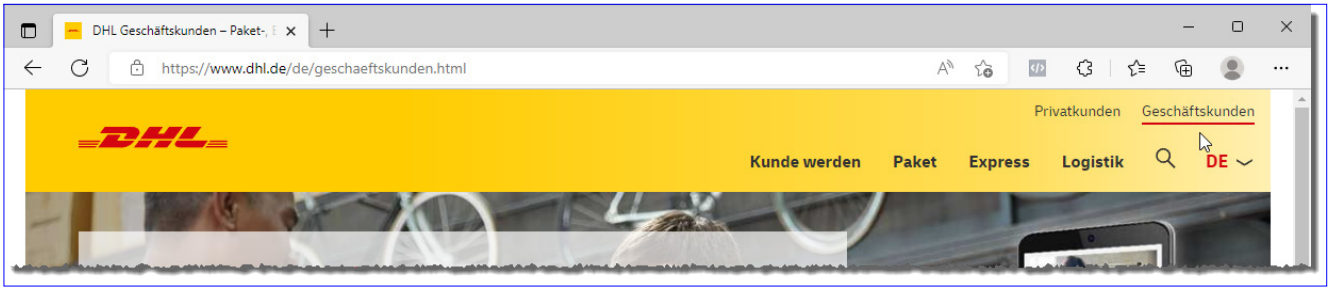


Bild 1: Hier gelangst Du zum Geschäftskundenbereich von DHL.

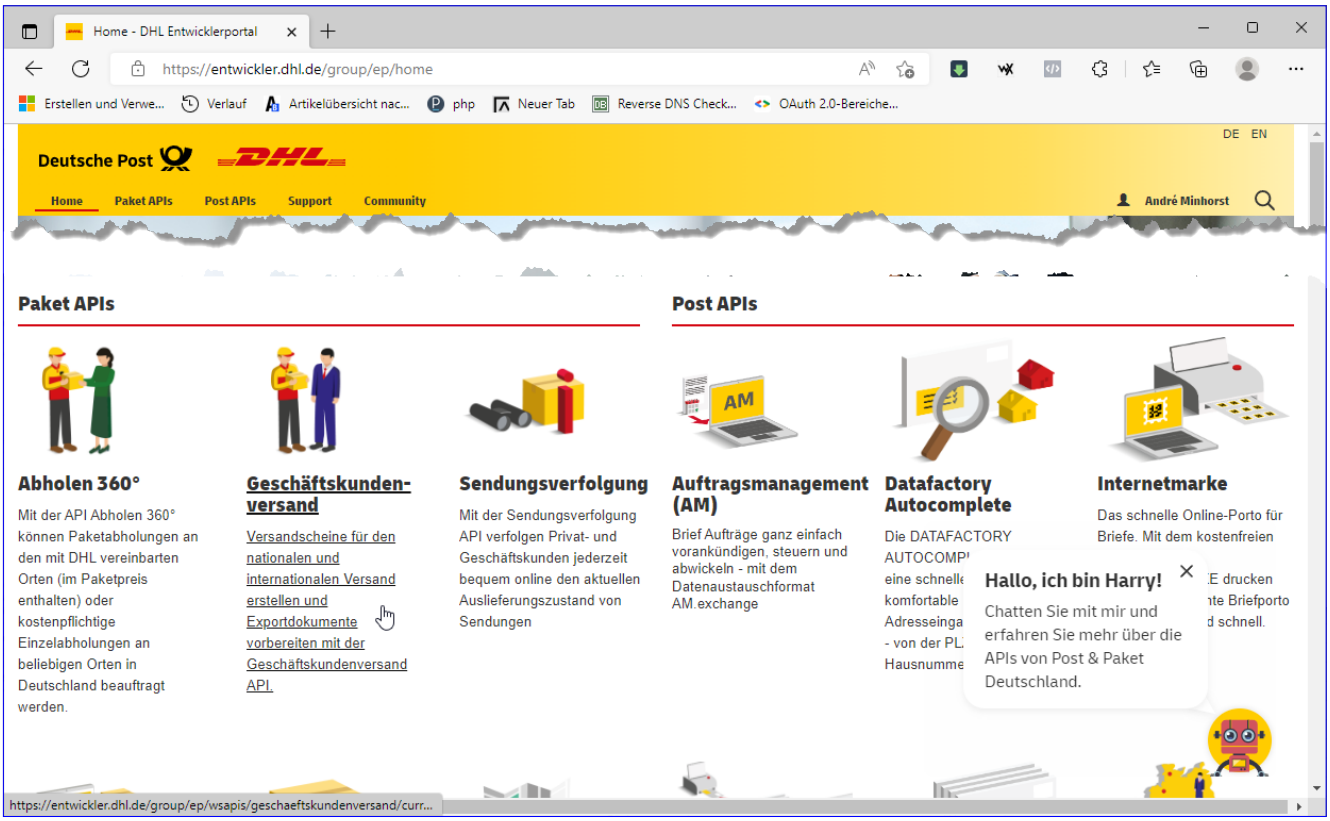


Bild 2: Bereiche im Entwicklerportal

Fehlerbehandlung gibt. Und da dies beim Zugriff auf APIs über das Internet immer besonders wichtig ist, wollen wir direkt mit dieser Version einsteigen. Dazu erhalten wir den wichtigen Tipp, die gewünschte Version im XML-Block namens **Version** zu übergeben.

Anlegen einer Applikation

Bevor wir richtig einsteigen können, legen wir eine Applikation an. Das erledigen wir im Bereich **Freigabe & Betrieb** der Webseite (siehe Bild 3). Hier tragen wir den technischen Namen, den Namen und eine Beschreibung ein.

Außerdem fügen wir alle Operationen hinzu, die wir verwenden wollen. In diesem Fall klappen wir den Bereich **Geschäftskundenversand/Business customer shipment** auf und selektieren dort alle Einträge. Anschließend schließen wir das Anlegen der Applikation mit einem Klick auf **Applikation speichern** ab.

Authentifizierung

Nach dem Anlegen der Applikation können wir zum nächsten Menüpunkt springen und uns die Informationen rund um die Authentifizierung ansehen. Wichtig ist hier für uns die Kenntnis der sogenannten Endpunkte für Test und Entwicklung sowie später für die produktive Nutzung. Vor dieser steht allerdings noch die Freigabe. Wir wollen den Zugriff per SOAP nutzen, also können wir uns den folgenden Endpunkt für die Tests mit der Sandbox merken:

```
https://cig.dhl.de/services/sandbox/soap
```

Außerdem erhalten wir hier noch wichtige Informationen über die zu verwendenden Zugangsdaten. Für den Testzugriff auf die Sandbox verwenden wir dabei als Benutzername die zuvor festgelegte Entwickler-ID und als Kennwort das für das Entwicklerkonto festgelegte Kennwort.

Später, wenn wir die Freigabe der Anwendung erhalten haben, können wir als Benutzernamen den Namen der Anwendung angeben und ein dafür festgelegtes Token. Beides erhalten wir im Bereich **Freigabe & Betrieb** der Webseite, also an der gleichen Stelle, an der wir auch die Applikation erstellt haben.

Neben den Informationen über die Authentifizierung erhalten wir auch noch den wichtigen Hinweis, dass wir für den produktiven Betrieb im Header noch die jeweilige Aktion angeben müssen – dazu kommen wir später noch.

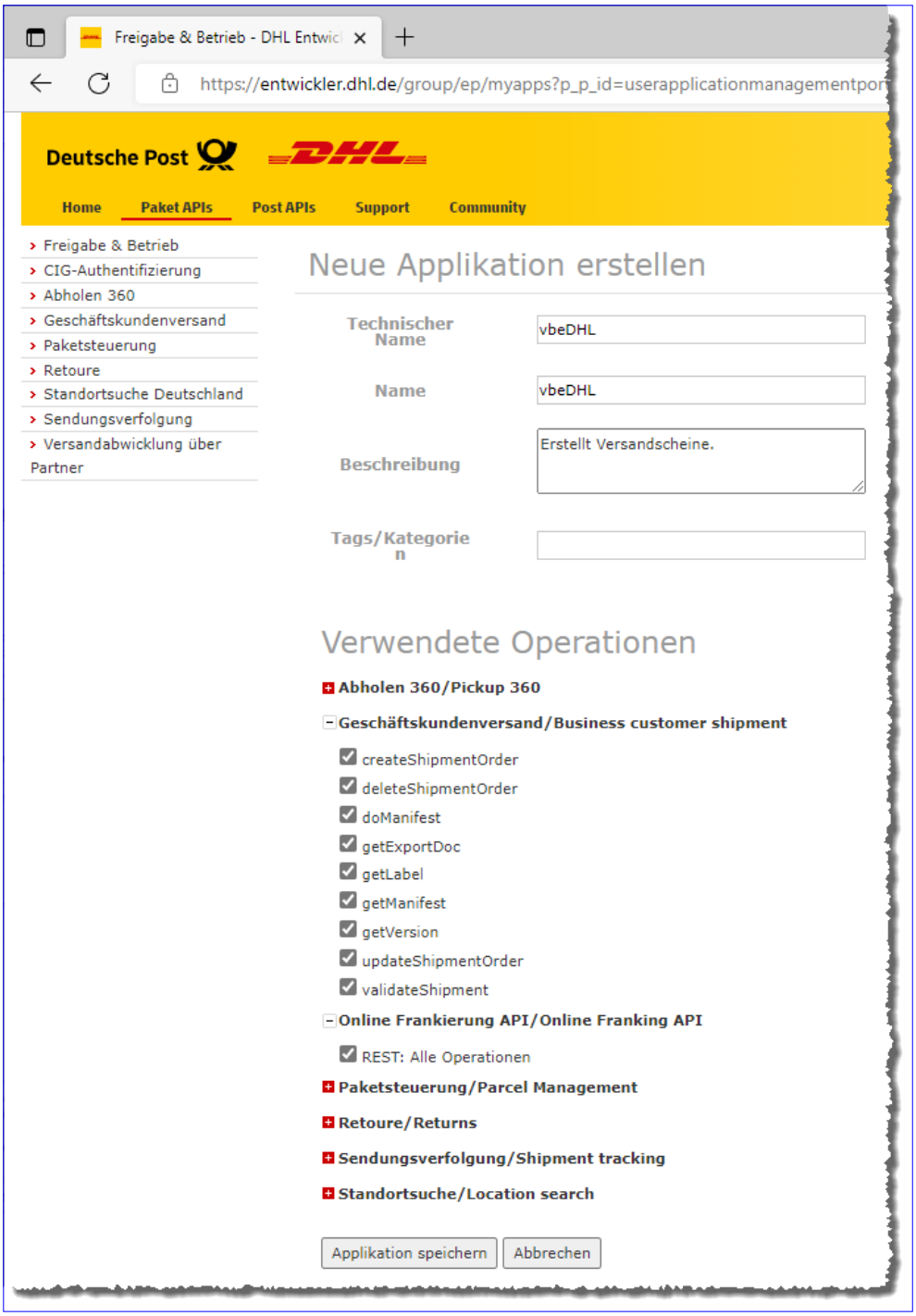


Bild 3: Anlegen einer neuen Anwendung

Die Geschäftskundenversand-API

Danach folgen im Menü die einzelnen Bereiche, die wir programmieren können. Wie weiter oben festgelegt, wollen wir den Geschäftskundenversand automatisieren und wechseln daher zum Menü **Geschäftskundenversand**.

Fingerübung: Version abfragen

Bevor wir uns an die Programmierung der Erstellung von Versandlabeln begeben, wollen wir zum Warmmachen die aktuell verfügbar maximale Version abfragen. Dazu erstellen wir als Erstes eine Funktion, die ein entsprechendes SOAP-Dokument zusammenstellt und dieses dann an eine Funktion zum Senden der Anfrage an die API schickt.

Die Funktion heißt **VersionErmitteln** und erwartet die beiden Parameter **intMajor** und **intMinor**, welche beides Rückgabeparameter sind, also die als leere Variablen übergeben und von der Funktion gefüllt werden (siehe Listing 1).

Sie deklariert zwei Variablen des Typs **MSXML2.DOMDocument60**, womit wir gleich bei einer Voraussetzung landen – dem Hinzufügen eines Verweises auf die Bibliothek **Microsoft XML, v6.0**. Dies erledigen wir im Dialog **Verweise**, den wir vom VBA-Editor der verwendeten Anwendung aus mit dem Menübefehl **Extras|Verweise** öffnen. Die benötigte Bibliothek fügen wir wie in Bild 4 hinzu. Anschließend können wir die enthaltenen Klassen, Methoden und Eigenschaften im VBA-Editor unter Verwendung von IntelliSense eingeben.

Neben diesen beiden Objektvariablen nutzen wir zwei weitere namens **objMinor** und **objMajor**, jeweils mit

```
Public Function VersionErmitteln(intMajor As Integer, intMinor As Integer)
    Dim objXMLRequest As MSXML2.DOMDocument60
    Dim objXMLResponse As MSXML2.DOMDocument60
    Dim objMinor As MSXML2.IXMLDOMNode
    Dim objMajor As MSXML2.IXMLDOMNode
    Dim strRequest As String
    strRequest = "<soapenv:Envelope xmlns:soapenv=""http://schemas.xmlsoap.org/soap/envelope/"" " _
        & "xmlns:ns=""http://dhl.de/webservices/businesscustomershipping/3.0"">" & vbCrLf
    strRequest = strRequest & "   <soapenv:Body>" & vbCrLf
    strRequest = strRequest & "      <ns:Version>" & vbCrLf
    strRequest = strRequest & "         <majorRelease>?</majorRelease>" & vbCrLf
    strRequest = strRequest & "         <minorRelease>?</minorRelease>" & vbCrLf
    strRequest = strRequest & "         <build>?</build>" & vbCrLf
    strRequest = strRequest & "      </ns:Version>" & vbCrLf
    strRequest = strRequest & "   </soapenv:Body>" & vbCrLf
    strRequest = strRequest & "</soapenv:Envelope>" & vbCrLf
    Set objXMLRequest = New MSXML2.DOMDocument60
    If Not objXMLRequest.loadXML(strRequest) Then
        Debug.Print objXMLRequest.parseError.Line, objXMLRequest.parseError.linepos, objXMLRequest.parseError.reason, _
            objXMLRequest.parseError.errorCode
    Else
        If RequestSandbox(strRequest, objXMLResponse) Then
            Set objMinor = objXMLResponse.selectSingleNode("//minorRelease")
            If Not objMinor Is Nothing Then
                intMinor = objMinor.nodeTypedValue
            End If
            Set objMajor = objXMLResponse.selectSingleNode("//majorRelease")
            If Not objMajor Is Nothing Then
                intMajor = objMajor.nodeTypedValue
            End If
        End If
    End If
End Function
```

Listing 1: Prozedur zum Zusammenstellen einer Abfrage der aktuellen Version

dem Typ **MSXML2.IXMLDOMNode**, um Elemente mit Versionsangabe zu referenzieren. Die Funktion stellt nun den Code der Anfrage zusammen, die wir an die API von DHL schicken wollen. Zusammengefasst sieht diese in der Variablen **strRequest** erfasste Zeichenkette wie folgt aus:

```
<soapenv:Envelope
        xmlns:soapenv="http://schemas.xmlsoap.org/soap/
        envelope/" xmlns:ns="http://dhl.de/webservices/
        businesscustomershipping/3.0">
    <soapenv:Body>
        <ns:Version>
            <majorRelease>?</majorRelease>
            <minorRelease>?</minorRelease>
            <build>?</build>
        </ns:Version>
    </soapenv:Body>
</soapenv:Envelope>
```

Nach dem Zusammenstellen dieser Zeichenkette erstellt die Funktion ein neues Objekt des Typs **DOMDocument60** und ruft dessen Funktion **loadXML** auf. Dieser übergibt sie den Inhalt von **strRequest** und füllt das Objekt aus **objXMLRequest** damit mit der XML-Anfrage. Die **loadXML**-Funktion prüft gleichzeitig die Gültigkeit des übergebenen XML-Dokuments und gibt eine Meldung im Direktbereich aus, sollte dieses nicht gültig sein.

Anderenfalls ruft sie die Funktion **Request** auf und übergibt dieser das gefüllte Objekt **objXMLRequest** sowie das noch leere Objekt **objXMLResponse**. Diese Funktion beschreiben wir gleich im Anschluss. Wenn die Funktion den Wert **True** zurückgibt, die Abfrage also erfolgreich war, dann ist **objXMLResponse** anschließend mit einem XML-Dokument wie dem folgenden gefüllt:

```
<soapenv:Envelope
        xmlns:bus="http://dhl.de/webservices/
        businesscustomershipping/3.0"
        xmlns:soapenv="http://schemas.xmlsoap.org/soap/
        envelope/">
    <soapenv:Header/>
    <soapenv:Body>
        <bus:GetVersionResponse>
            <bus:Version>
                <majorRelease>3</majorRelease>
                <minorRelease>0</minorRelease>
                <build>0</build>
            </bus:Version>
        </bus:GetVersionResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

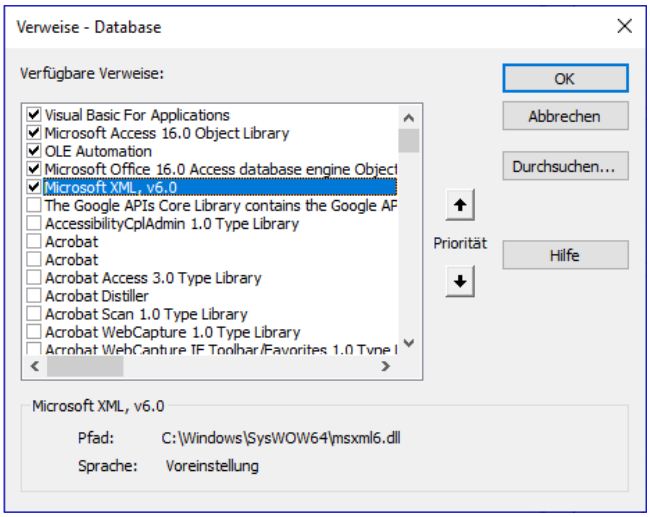


Bild 4: Hinzufügen eines Verweises auf die XML-Bibliothek

Die Aufgabe lautet nun noch, den Inhalt der beiden Elemente **majorRelease** und **minorRelease** auszulesen und in die Rückgabeparameter zu schreiben. Das erledigen wir, indem wir mit der Funktion **selectSingleNode** nach den entsprechenden Elementen suchen. Finden wir diese, lesen wir daraus über die Eigenschaft **nodeTypedValue** die enthaltenen Werte aus. Diese füllen wir in die Rückgabeparameter **intMinor** und **intMajor**.

Diese Funktion rufen wir beispielsweise mit der folgenden Prozedur auf:

```
Public Sub Test_VersionErmitteln()
    Dim intMajor As Integer
    Dim intMinor As Integer
    VersionErmitteln intMajor, intMinor
    Debug.Print "Version " & intMajor & "." & intMinor
End Sub
```

Die Request-Funktion

Die eigentliche Arbeit beim Aufruf der DHL-API übernimmt die Funktion **RequestSandbox** (siehe Listing 2). Diese nimmt über den Parameter **strRequest** den Aufruf entgegen und soll den Rückgabeparameter **objXMLResponse** mit dem **DOMDocument60**-Objekt mit der Antwort füllen. Dieses erstellt die Funktion mit dem **New**-Schlüsselwort zunächst neu. Außerdem erstellt sie ein Objekt des Typs **XMLHTTP60** und referenziert es mit der Objektvariablen **objXMLHTTP**. Hier kommt nun eine Konstante ins Spiel, die wir im Kopf des Moduls deklarieren und mit der weiter oben angegebenen URL für den Zugriff auf die API in der Sandbox-Version füllen – also mit der URL für unsere Tests der API:

```
Public Const cStrURLBase As String = _
    "https://cig.dhl.de/services/sandbox/soap"
```

Diese übergeben wir als zweiten Parameter der **Open**-Methode von **objXMLHTTP**. Der erste Parameter lautet **post** und bedeutet, dass wir später mit der **send**-Methode noch Informationen senden statt diese an die URL anzuhängen.

Danach stellen wir drei Header ein, zunächst **Content-Type** und **Content-Length**. Der erste gibt an, dass wir mit Daten im Format **text/xml** mit dem Zeichensatz **utf-8** arbeiten. Der zweite legt die Länge des übermittelten XML-Dokuments fest. Der dritte Header nimmt die Zugangsdaten auf. Diese speichern wir in Form entsprechender Konstanten wieder im Kopf des Moduls **mdlDHL**. Die Konstanten sehen wie folgt aus:

```
Public Function RequestSandbox(strRequest As String, objXMLResponse As MSXML2.DOMDocument60) As Boolean
    Dim strURL As String
    Dim objXMLHTTP As MSXML2.XMLHTTP60
    Set objXMLHTTP = New MSXML2.XMLHTTP60
    Set objXMLResponse = New MSXML2.DOMDocument60
    With objXMLHTTP
        .Open "post", cStrURLBase, False
        .setRequestHeader "Content-Type", "text/xml; charset=utf-8"
        .setRequestHeader "Content-Length", Len(strRequest)
        .setRequestHeader "Authorization", "Basic " + Base64Encode(strUserSandbox + ":" + strPasswordSandbox)
        .send strRequest
        Select Case .status
            Case 200
                Request = True
                objXMLResponse.loadXML .responseText
            Case Else
                MsgBox "Fehler beim Request:" & vbCrLf & .statusText
        End Select
    End With
End Function
```

Listing 2: Die **Request**-Funktion zum Durchführen eines API-Aufrufs

```
Public Const strUserSandbox As String = "[EntwicklerID]"
Public Const strPasswordSandbox As String = "[Kennwort]"
```

Für **strUserSandbox** geben wir die Entwickler-ID aus dem Entwicklerportal an, die wir unter folgendem Link finden:

```
https://entwickler.dhl.de/group/ep/mein-konto
```

Für **strPasswordSandbox** hinterlegen wir das Kennwort für die Anmeldung am Entwicklerportal. Die Anmeldedaten werden in einer durch einen Doppelpunkt getrennten Zeichenkette zusammengefasst und Base64-kodiert.

Dann stellen wir diesem Ausdruck den Text **Basic** und ein Leerzeichen voran und übergeben dies für den Header **Authorization**. Schließlich ruft die Funktion die **send**-Methode auf und übergibt dieser als Parameter die in **strRequest** gespeicherte XML-Anfrage.

Nachdem diese ausgeführt wurde, können wir das Ergebnis in Form der Eigenschaft **status** auswerten. Der Wert **200** entspricht dem Ergebnis **OK**. In diesem Fall stellen wir das Ergebnis der Funktion auf **True** ein und lesen den Inhalt der Eigenschaft **responseText**, die von der API gefüllt wurde, mit der **loadXML**-Methode in das **DOMDocument60**-Objekt **objXMLResponse** ein. Anderenfalls gibt die Funktion den Statustext mit der Fehlermeldung in einem Meldungsfenster aus.

Benutzerdaten manuell eingeben

Wenn Du die Benutzerdaten nicht im Code speichern möchtest, kannst Du die Übergabe der dritten Header-Zeile auch weglassen. Dann erscheint eine Meldung wie in Bild 5, um die Zugangsdaten abzufragen.

Diese muss allerdings nicht für jeden Aufruf eingegeben werden – wir konnten nach einmaliger Eingabe innerhalb der gleichen Access-Session weitere Aufrufe absetzen, ohne uns erneut anmelden zu müssen.

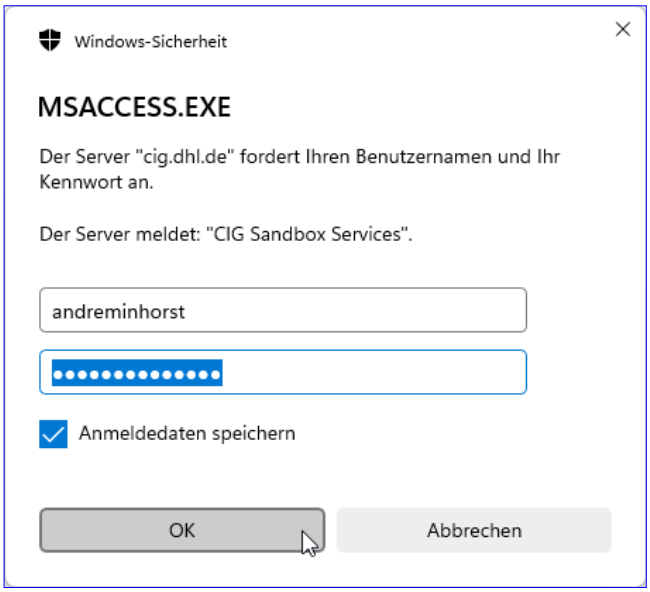


Bild 5: Manuelle Eingabe der Zugangsdaten

Weitere Funktionen der API für den Geschäftskundenversand

Im Bereich **Geschäftskundenversand|Operationen** des Entwicklerportals finden wir neben **getVersion** noch weitere API-Funktionen:

- **validateShipment**: Ermöglicht die Validierung einer Sendung, bevor diese mit Versandschein und Rechnungsnummer angelegt wird.
- **createShipmentOrder**: Erzeugt eine Sendung mit Versandscheinen. Diese können über die mit der Antwort übergebenen URL heruntergeladen werden.
- **updateShipmentOrder**: Legt eine neue Sendung an und löscht die zu aktualisierende Sendung.
- **deleteShipmentOrder**: Storniert eine Sendung, was nur vor Tagesabschluss möglich ist.
- **getLabel**: Ermöglicht das Abrufen von Versandscheinen für eine bestimmte zuvor angelegte Sendungsnummer.