AUSGABE 04/2024

VISUAL BASIC ENTWICKLER

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

IN DIESEM HEFT:

André Minhorst Verlag

Aufgabenverwaltung vollständig im Griff

In dieser Ausgabe zeigen wir Dir nicht nur, wie Du Deine Projekte und Aufgaben auf einfachste Weise mit einer kostenlosen Microsoft-Lösung verwalten kannst. Wir gehen noch einen Schritt weiter und liefern Dir gleich noch das Know-how, mit dem Du die mit dieser Lösung verwalteten Aufgaben per VBA anlegen, bearbeiten, auslesen und löschen kannst. Das ist für VBA allein etwas viel, daher holen wir uns prominente Unterstützung – nämlich von Microsofts Automatisierungtool Power Automate.

Wenn Du ein einfaches Tool zur Verwaltung von Aufgaben suchst, das sowohl auf Deinem Windows-Rechner als App zur Verfügung steht als auch für mobile Endgeräte wie Smartphones, ist vielleicht **Microsoft To Do** das Richtige für Dich. Damit kannst Du einfache Aufgabenlisten verwalten, die Du noch in Gruppen und Listen hierarchisch unterteilen kannst. Der Clou: Die Listen kannst Du auch mit anderen Menschen teilen, was sowohl für die gemeinsame Einkaufsliste mit der Familie als auch für eine Aufgabenliste für Dich und Dein Entwicklerteam interessant ist. Mehr dazu erfährst Du im Artikel **Aufgaben mit Microsoft To Do verwalten** ab Seite 4.

Außerdem streuen wir mal wieder einen Grundlagenartikel zum Thema VBA ein. Unter **VBA Basics: Mit Arrays programmieren** erfährst Du ab Seite 10 alles, was Du über die Programmierung mit Arrays unter VBA und ähnlichen Programmiersprachen wissen musst.

Wenn Du mit VBA programmierst, kommst Du aktuell um den VBA-Editor noch nicht herum. Als Start einer kleinen Artikelreihe zu diesem Thema zeigen wir einmal, wie Du das Menüsystem im VBA-Editor optimal nutzen und für Deine Zwecke einstellen kannst. Mehr dazu unter dem Titel **Menüs im VBA-Editor anpassen** ab Seite 13.

Solltest Du gelegentlich mal eigene Klassen programmieren, kennst Du dieses Problem: Zum Definieren von Eigenschaften mit Setter und Getter ist viel manuelle Tipparbeit nötig. Wie Du das viel einfacher gestalten kannst, zeigen wir Dir ab Seite 23 im Artikel **VBA-Editor: Klasseneigenschaften per Mausklick**. Damit brauchst Du nur noch die Variable selbst zu deklarieren und mit einem Aufruf unserer Hilfsfunktion wird der Rest automatisch erledigt.

Wenn Du viel mit Word arbeitest, möchtest Du Deinen Dokumenten vielleicht die eine oder andere Automatisierung hinzufügen. In unserem Artikel **Word: Dokument mit Ribbon und VBA-Funktionen** (ab Seite 29) erfährst Du, wie das mit einem in das aktuelle Dokument integrierten Ribbon und den passenden VBA-Prozeduren gelingt.

Kommen wir schließlich zum Schwerpunkt der Ausgabe: Das bereits erwähnte Microsoft To Do können wir nämlich auch per VBA steuern. Dazu brauchen wir allerdings ein wenig Unterstützung, die wir uns bei dem Automatisierungsdienst Power Automate von Microsoft holen. Das ist zwar nicht kostenlos, aber liefert spannende Features. Damit können wir zum Beispiel per VBA Aktionen anstoßen, mit denen wir Aufgaben in Microsoft To Do anlegen, bearbeiten, auslesen und löschen können. In der heutigen Zeit, wo immer mehr Informationen in Online-Diensten verwaltet werden, ist es ein unfassbarer Vorteil, wenn wir von unseren gewohnten VBA-Anwendungen auf diese Daten zugreifen können! Mehr dazu in den Artikeln **To Do-Aufgabe mit Power Automate und VBA anlegen** (ab Seite 33) und **To Do mit VBA und Power Automate steuern** (ab Seite 44).

Nun viel Spaß beim Lesen!

Dein André Minhorst

Menüs per VBA programmieren

Das Menüsystem des VBA-Editors lässt sich über die Benutzeroberfläche bereits einfach anpassen. Das haben wir im Artikel »Menüsystem im VBA-Editor anpassen« (www.vbentwickler.de/434) gezeigt. Außerdem haben wir uns im Artikel »Kontextmenüs per VBA programmieren« (www.vbentwickler.de/368) bereits angeschaut, wie wir per VBA Kontextmenüs erstellen und anpassen können. Es fehlen also noch die Informationen, wie wir die eigentlichen Menüleisten und Symbolleisten mit VBA programmieren können. Wie das gelingt, schauen wir uns im vorliegenden Artikel an. Wir zeigen, wie vorhandene Menüs angepasst und wie neue Menüs erstellt werden können.

Warum haben wir eigentlichen einen eigenen Artikel zum Thema Kontextmenüs per VBA programmieren veröffentlicht und nicht direkt etwas weiter ausgeholt und die Menü- und Symbolleisten ebenfalls beschrieben? Weil es damals darum ging, die Benutzeroberfläche der Office-Anwendungen zu erweitern.

Und da diese bekanntlich seit Office 2007 keine Menü- und Symbolleisten mehr verwenden, sondern das Ribbon, haben wir uns auf die Kontextmenüs beschränkt.

Das wirft nun die Frage auf, warum wir uns dann jetzt mit den Menü- und Symbolleisten beschäftigen wollen. Nun: Es gibt noch das gallische Dorf unter den Microsoft-Anwendungen, in denen noch Menü- und Symbolleisten verwendet werden.

Dabei handelt es sich um die Anwendung, mit der wir täglich arbeiten: den VBA-Editor. Dieser verwendet noch das gute, alte Menü- und Symbolleistensystem, das jüngere VBA-Entwickler möglicherweise von Office selbst gar nicht mehr kennen.

Und da wir beispielsweise mit twinBASIC den VBA-Editor erweitern wollen und das Menüsystem nun einmal der Hauptort zum Unterbringen von Steuerelementen zum Aufrufen der Funktionen der entsprechenden COM-Add-Ins ist, schauen wir uns nun doch noch einmal die Grundlagen zu den Menü- und Symbolleisten an.

Office-Verweis sinnvoll

Dazu ist es sinnvoll, dem VBA-Projekt erst einmal einen Verweis auf die Bibliothek **Microsoft Office 16.0 Object Library** hinzuzufügen.

Das erledigen wir über den Verweise-Dialog, den wir über den Menüeintrag **Extras|Verweise** des VBA-Editors öffnen (siehe Bild 1).

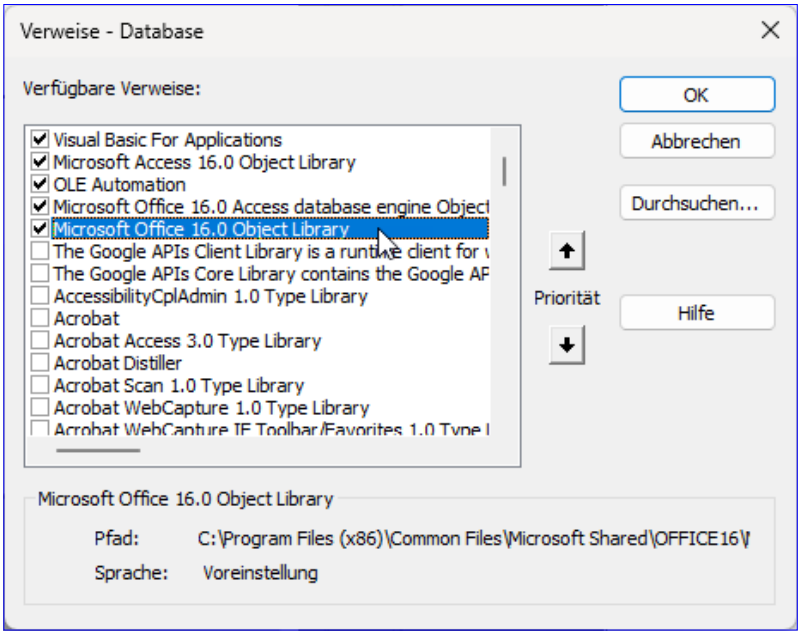

Bild 1: Hinzufügen eines Verweises auf die Office-Bibliothek

Alles ist eine Symbolleiste

Um direkt zu Beginn eine Begrifflichkeit zu klären: Im Sinne von VBA sind sowohl Menüleisten, Symbolleisten als auch Kontextmenüs alles Symbolleisten, unter VBA **CommandBar** genannt.

Menüleisten und Kontextmenüs unterscheiden sich nur durch bestimmte Eigenschaften von den anderen Elementen.

Symbolleisten durchlaufen

Mit der Office-Bibliothek holen wir uns das Objektmodell zum Verwalten von **CommandBar**-Elementen in das VBA-Projekt.

Vom VBA-Editor aus können wir gleich zwei **CommandBars**-Auflistungen durchlaufen, nämlich die des VBA-Editors als auch die der jeweiligen Host-Anwendung – also beispielsweise Access, Excel, Outlook, PowerPoint oder Word.

Die folgende Prozedur gibt zuerst alle Elemente des VBA-Editors aus (referenziert mit **VBE.CommandBars**) und dann die Elemente der aktuellen Host-Anwendung, hier Access (referenziert mit **Access.Commandbars**):

```
Public Sub AlleCommandbars()
    Dim cbr As CommandBar

    For Each cbr In VBE.CommandBars
        Debug.Print "VBA-Editor: " & cbr.Name
    Next cbr
    For Each cbr In Access.CommandBars
        Debug.Print "Access: " & cbr.Name
    Next cbr
End Sub
```

Wir wollen hier besonderes Augenmerk auf die Elemente des VBA-Editors legen. Die Menge ist überschaubar, wie Bild 2 zeigt.

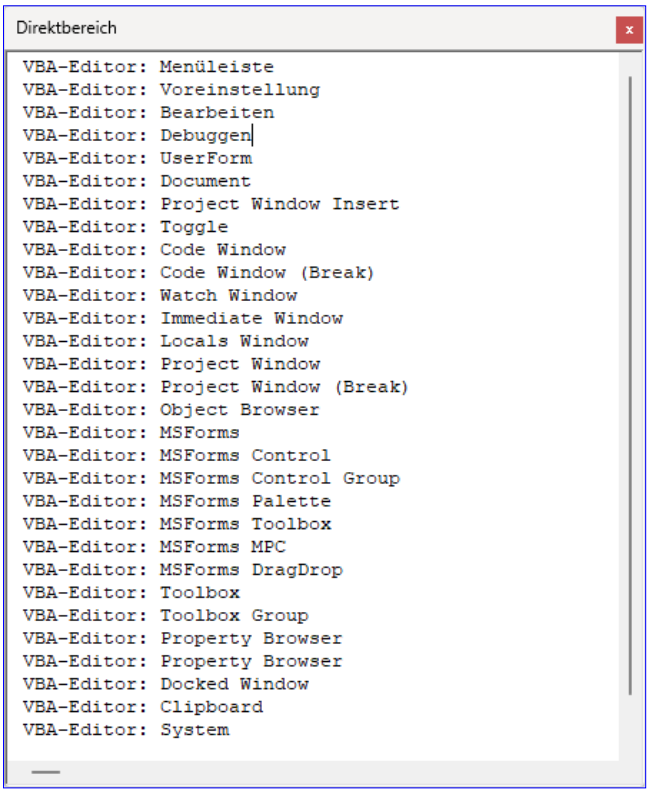

Bild 2: Alle Symbolleisten des VBA-Editors

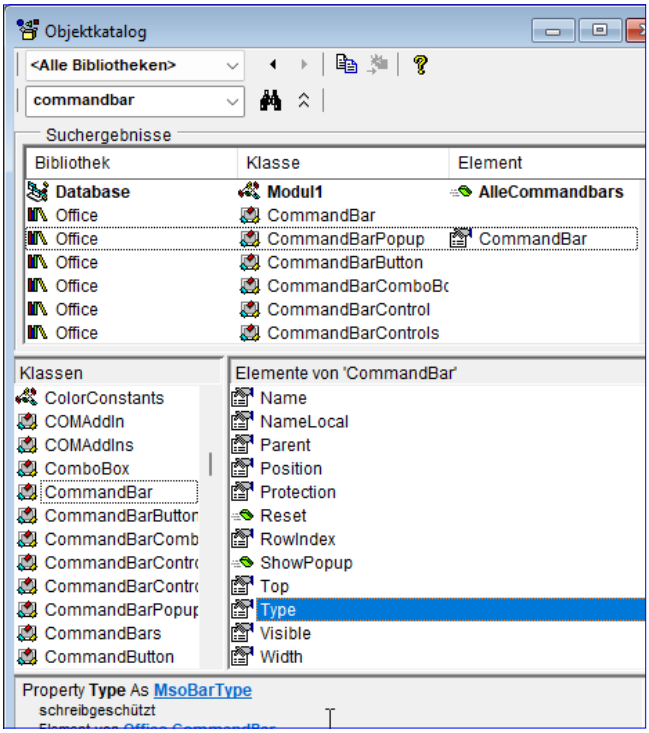

Bild 3: Ermitteln des Typs einer Eigenschaft

Unterscheidung der Symbolleisten nach Typ

Wie aber können wir nun die Menüleisten, Symbolleisten und Kontextmenüleisten voneinander unterscheiden?

Dabei hilft uns die Eigenschaft **Type**. Sie liefert uns verschiedene Werte (**0**, **1** und **2**). Wie finden wir heraus, welcher Wert welche Bedeutung hat? Da IntelliSense uns hier keine Hinweise gibt, schauen wir im Objektkatalog nach.

Hier finden wir, wenn wir unter Klassen das **CommandBar**-Element selektieren und in den Elementen den Eintrag **Type**, die Information, dass es sich hierbei um die Auflistung **msoBarType** handelt (siehe Bild 3).

Ein Klick auf diese Auflistung zeigt direkt die Elemente dieser Auflistung und somit die möglichen Werte (siehe Bild 4). Klicken wir diese nacheinander an, erhalten wir die folgenden Zahlenwerte samt zugehörigen Konstanten plus Bedeutung:

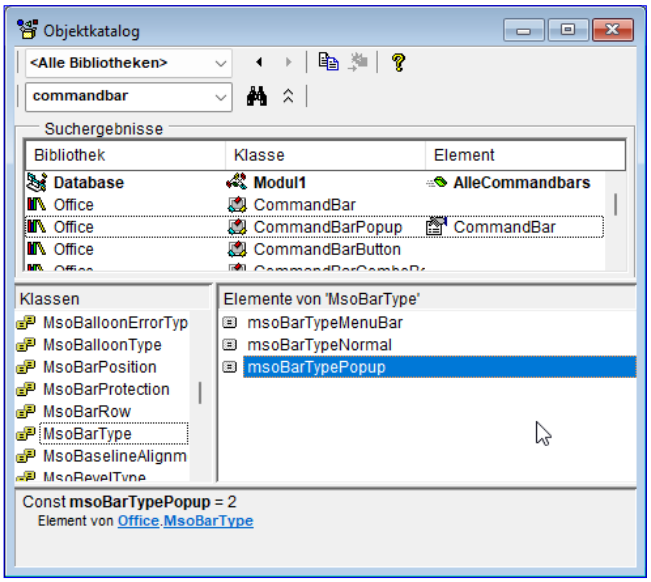

Bild 4: Ermitteln der Werte eines Eigenschaft

- **0** (**msoBarTypeNormal**): Symbolleiste
- **1** (**msoBarTypeMenuBar**): Menüleiste
- **2** (**msoBarTypePopup**): Kontextmenü

Diese Werte können wir uns gleich merken, denn wir benötigen diese später zum Anlegen neuer **CommandBar**-Elemente.

Passen wir also unsere Prozedur zur Ausgabe der VBA-Editor-Symbolleisten wie folgt an:

```
Public Sub AlleCommandbars()
    Dim cbr As CommandBar
    For Each cbr In VBE.CommandBars
        Select Case cbr.Type
            Case msoBarTypeNormal
                Debug.Print "Symbolleiste: " & cbr.Name
            Case msoBarTypeMenuBar
                Debug.Print "Menüleiste:   " & cbr.Name
```

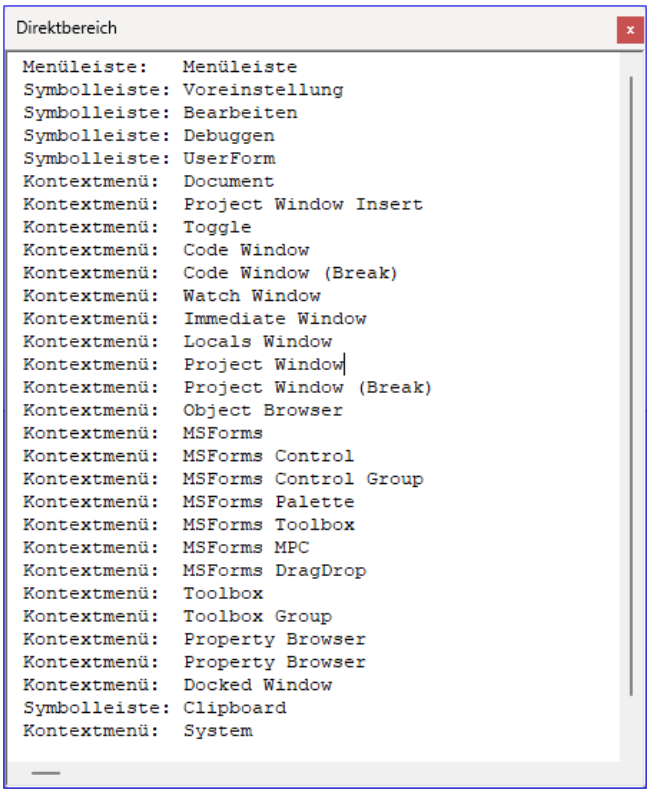

Bild 5: **CommandBar**-Typ plus Name

Menü-Steuerelemente per VBA programmieren

Im Artikel »Menüs per VBA programmieren« (www.vbentwickler.de/435) haben wir uns bereits angesehen, wie wir die Menüstruktur selbst im VBA-Editor per VBA programmieren können. Damit wissen wir nun, wie wir Hauptmenüleisten, Symbolleisten und Kontextmenüs erstellen und anzeigen können. Es fehlt allerdings noch das Salz in der Suppe, nämlich die Steuerelemente auf diesen Menüs. Welche es gibt und wie man diese hinzufügt und mit Aktionen versieht, schauen wir uns in diesem Artikel an.

Wer den VBA-Editor mit eigenen Funktionen erweitern will und das beispielsweise mit COM-Add-Ins auf Basis von twinBASIC realisiert, möchte die hinzugefügten Funktionen auch auf irgendeine Weise für den Benutzer zugänglich machen. Dazu bietet sich das Menüsystem an. Damit können wir in der Menüleiste, den Symbolleisten und in Kontextmenüs Befehle zum Aufrufen der benutzerdefinierten Funktionen hinzufügen. Allerdings brauchen wir hier nicht nur die entsprechenden Elemente, sondern wir müssen auch noch Steuerelemente hinzufügen. Dies werden meist einfache Schaltflächen sein. Wir schauen uns allerdings in diesem Artikel alle der wenigen verfügbaren Steuerelemente für die Menüs des VBA-Editors an.

Steuerelemente des CommandBar-Elements

Dem **CommandBar**-Element können wir die folgenden Steuerelemente hinzufügen:

- Schaltfläche (**msoControlButton**)
- Auswahlfeld (**msoControlComboBox**)
- Untermenü (**msoControlPopup**)

Um ein solches Steuerelement hinzuzufügen, verwenden wir die **Add**-Methode der **Controls**-Auflistung des **CommandBar**-Objekts. Das folgende, einfache Beispiel fügt einfach jeweils ein Element des jeweiligen Typs zu einer neu erstellten Symbolleiste hinzu:

```
Public Sub AddControls()
    Dim cbr As CommandBar
    Dim cbb As CommandBarButton
    Dim cbc As CommandBarComboBox
    Dim cbp As CommandBarPopup
    On Error Resume Next
    VBE.CommandBars("Symbolleiste mit Controls").Delete
    On Error GoTo 0
    Set cbr = VBE.CommandBars.Add( _
        "Symbolleiste mit Controls")
    With cbr
        Set cbb = cbr.Controls.Add(msoControlButton)
        With cbb
            .Caption = "Button"
            .Style = msoButtonCaption
        End With
        Set cbc = cbr.Controls.Add(msoControlComboBox)
        With cbc
            .Caption = "ComboBox"

        End With
        Set cbp = cbr.Controls.Add(msoControlPopup)
        With cbp
            .Caption = "Popup"
        End With
        .Visible = True
    End With
End Sub
```

Die Prozedur löscht zunächst eine gegebenenfalls bereits von einem vorherigen Test vorhandene Symbolleiste namens **Symbolleiste mit Controls** aus der

CommandBars-Auflistung. Für den Fall, dass dieses noch nicht angelegt oder zuvor auf andere Art und Weise gelöscht wurde, deaktivieren wir die eingebaute Fehlerbehandlung, damit dann kein Fehler ausgelöst wird.

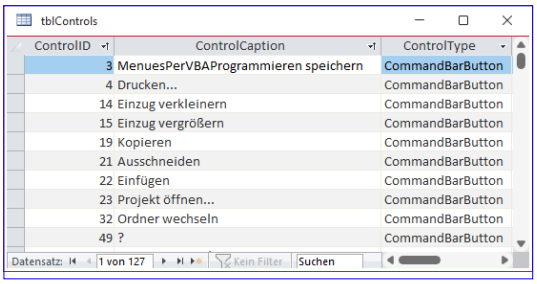

Bild 1: Beispiel für Steuerelemente im **CommandBar**

Dann legt sie ein neues **CommandBar**-Objekt mit diesem Namen an. Anschließend fügen wir mit der **Add**-Methode der **Controls**-Auflistung zunächst eine Schaltfläche mit dem Typ **msoControlButton** an und stellen dafür die Beschriftung ein. Damit diese auch erscheint, legen wir die Eigenschaft **Style** auf den Wert **msoButtonCaption** fest.

Danach legt die Prozedur ein Auswahlfeld an und verwendet dazu den Typ **msoControlComboBox** als ersten Parameter der **Add**-Methode. Diesem fügen wir auch gleich ein erstes Element hinzu.

Schließlich folgt noch ein Untermenü mit dem Typ **msoControlPopup**. Für dieses brauchen wir den Style nicht auf **msoButtonCaption** festzulegen, die Beschriftung wird automatisch angezeigt.

Das Ergebnis sehen wir in Bild 1.

Schaltflächen im CommandBar-Element

Nachdem wir eine Schaltfläche zu einem **CommandBar**-Element hinzufügt haben, ist erst einmal wichtig, dass wir dem Benutzer mitteilen, wozu er diese Schaltfläche verwenden kann. Dazu können wir primär Symbole und/oder Texte verwenden. Den Text stellen wir immer mit der Eigenschaft **Caption** ein. Damit die Beschriftung überhaupt sichtbar ist, müssen wir die **Style**-Eigenschaft auf den entsprechenden Wert einstellen. Dazu haben wir die folgenden Möglichkeiten:

- **msoButtonCaption**: Zeigt nur die Beschriftung an.
- **msoButtonIcon**: Zeigt nur ein Symbol an.
- **msoButtonIconAndCaption**: Zeigt Symbol und Text an.

Ereignisse von Schaltflächen programmieren

Wer bereits einmal Menüleisten für die verschiedenen Office-Anwendungen mit der Version 2003 und älter programmiert hat oder auch Kontextmenüs für die aktuellen Versionen, der kennt die **OnAction**-Eigenschaft. Für diese Eigenschaft eines **CommandBarButton**-Objekts hinterlegt man den Namen einer öffentlichen VBA-Prozedur, die durch das Anklicken des Menüpunktes ausgelöst werden soll.

Dies funktioniert bei der Programmierung von Menüs für den VBA-Editor so nicht mehr. Hier müssen wir ein Ereignis programmieren. Leider können wir Ereignisse nur innerhalb von Klassenmodulen implementieren und nicht in Standardmodulen. Wir benötigen also eine Klasse, die einen Verweis auf unser **CommandBarButton**-Objekt enthält und in der wir die Ereignisprozedur implementieren.

Diese Klasse heißt **clsCommandBarButton** und sieht wie in Listing 1 aus. Hier definieren wir als Erstes eine Objektvariable, mit der wir das **CommandBarButton**-Objekt referenzieren wollen. Wir fügen der Deklaration das Schlüsselwort **WithEvents** hinzu, damit wir die Ereignisse dieses Objekts innerhalb der aktuellen Klasse implementieren können.

Damit wir von außen einen Verweis auf das **CommandBarButton**-Objekt übergeben können, für das wir das Ereignis implementieren wollen, fügen wir eine **Property Set**-Methode hinzu. Diese nimmt einen Verweis auf das **CommandBarButton**-Element ent-

```
Private WithEvents m_CommandBarButton As CommandBarButton

Public Property Set CommandBarButton(cbb As CommandBarButton)
    Set m_CommandBarButton = cbb
End Property

Public Property Get CommandBarButton() As CommandBarButton
    Set CommandBarButton = m_CommandBarButton
End Property

Private Sub m_CommandBarButton_Click(ByVal Ctrl As Office.CommandBarButton, CancelDefault As Boolean)
    MsgBox "CommandBarButton '" & m_CommandBarButton.Caption & "' angeklickt."
End Sub
```

Listing 1: Klassenmodul für die Ereignisse von **CommandBarButton**-Ereignissen

gegen. Außerdem fügen wir noch eine **Property Get**-Prozedur hinzu, mit der wir den Verweis auf dieses Steuerelement abfragen können.

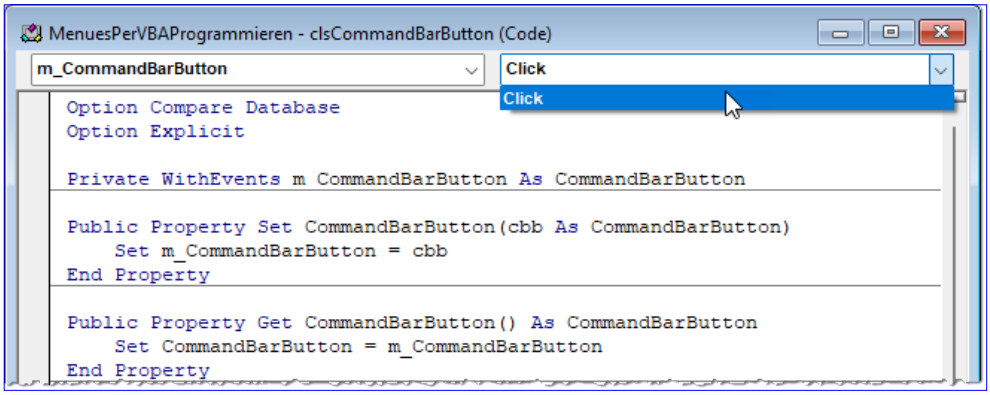

Bild 2: Anlegen der Ereignisprozedur für das **Click**-Ereignis

Um schließlich die Ereignisprozedur zu implementieren, wählen wir im linken Kombinationsfeld des Codefensters den Eintrag **m_CommandBarButton** aus und den dann im rechten Fenster erscheinenden Eintrag **Click** (siehe Bild 2). Dies legt die Ereignisprozedur **m_CommandBarButton_Click** an, der wir im Beispiel einfach eine Meldungsfenster-Funktion hinzugefügt haben.

Nun müssen wir noch das **CommandBarButton**-Objekt in einem Menü anzeigen, eine neue Instanz der Klasse **clsCommandBarButton** erzeugen und schließlich das Menü mit der Schaltfläche anzeigen.

Um zu verhindern, dass die Instanz der Klasse **clsCommandBarButton** nach dem Durchlaufen der Prozedur zum Erstellen der CommandBar mit dem CommandBarButton verloren geht, weil wir die mit einer Variablen innerhalb der Prozedur referenziert haben, verschieben wir diese Deklaration mit der folgende Anweisung nach außen (siehe Modul **mdlCommandBarButtonMitEvent**):

```
Dim objCommandBarButton As clsCommandBarButton
```

Die Prozedur selbst findest Du in Listing 2.

Diese deklariert ein **CommandBar**- und ein **CommandBarButton**-Element. Dann löscht sie eine eventuell bereits unter dem Namen **CommandBarButtonMitEvent** vorhandene Menüleiste und legt diese erneut an. Dann fügt sie mit der **Add**-Methode der **Controls**-Auflistung ein neues **CommandBarButton**-Element hinzu und stellt den Stil und die Beschriftung des Steuerelements ein.

Dann folgen die für die Ereignisprozedur wichtigen Schritte: Wir legen ein neues Objekt auf Basis der Klasse **clsCommandBarButton** an und weisen der Eigenschaft **CommandBarButton** einen Verweis auf das soeben erstellte **CommandBarButton**-Objekt zu. Schließlich blenden wir das Menü noch ein.

```
Dim objCommandBarButton As clsCommandBarButton

Public Sub CommandBarButtonMitEvent()
    Dim cbr As CommandBar
    Dim cbb As CommandBarButton
    On Error Resume Next
    VBE.CommandBars("CommandBarButtonMitEvent").Delete
    On Error GoTo 0
    Set cbr = VBE.CommandBars.Add("CommandBarButtonMitEvent")
    With cbr
        Set cbb = cbr.Controls.Add(msoControlButton)
        With cbb
            .Style = msoButtonCaption
            .Caption = "Button mit Ereignis"
        End With
        Set objCommandBarButton = New clsCommandBarButton
        Set objCommandBarButton.CommandBarButton = cbb
    End With
    cbr.Visible = True
End Sub
```

Listing 2: Anlegen eines **CommandBarButtons** mit Ereignis

Klicken wir danach auf das einzige Steuerelement, erscheint die gewünschte Meldungsbox (siehe Bild 3).

Symbole in Schaltflächen

Während wir den Text auf einfache Weise mit der Caption-Eigenschaft zuweisen können, sieht das bei Symbolen schon anders aus. Hier haben wir grundsätzlich zwei Möglichkeiten:

- Wir fügen eines der bereits vorhandenen Symbole hinzu, also eines der für die übrigen Steuerelemente verfügbaren oder
- wir fügen ein Bild im Format **StdPicture** über die Eigenschaft **Picture** hinzu.

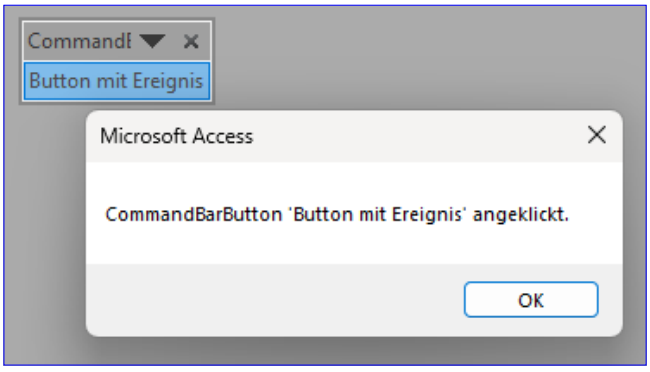

Bild 3: Menü mit **CommandBarButton** und Ereignis

Eingebaute Symbole hinzufügen

Der Einbau eingebauter Symbole ist etwas einfacher, weil wir keinen eigenen VBA-Code benötigen, um erst vorhandene Dateien in **StdPicture**-Objekte umwandeln zu müssen. Allerdings macht es dann Sinn, sich gut mit den eingebauten Symbolen auszukennen. Die eingebauten Symbole haben jeweils eine eindeutige Nummer, die wir über die Eigenschaft **FaceId** ermitteln können. Was wir also brauchen, ist eine Übersicht aller Icons mit den jeweiligen Werten für die Eigenschaft **FaceId**. Diese erhalten wir am einfachsten, indem wir eine neue Symbolleiste erstellen, der wir alle Symbole der eingebauten Schaltflächen hinzufügen.

Dies erledigen wir in der Prozedur **AlleEingebautenIcons** aus Listing 3.

Hier finden wir gleich ein spannendes Beispiel für den Fall, dass wir viele Steuerelemente in einer Menüleiste mit einem Ereignis ausstatten wollen, das beim Anklicken die gleiche Funktion ausführen soll – gegebenen-

falls noch mit kleinen Varianten durch spezielle Eigenschaften des **CommandBarButton**-Elements.

Grundlage für diese Prozedur ist, dass es **FaceId**-Werte gibt, die bis 6.000 reichen. Wir wollen all diese Werte einmal durchlaufen und in ein einziges **CommandBar**-Objekt schreiben. Allerdings liefern nicht alle Werte von **FaceId** tatsächlich ein Symbol. Deshalb wollen wir diejenigen, für die kein Symbol hinterlegt ist, auch nicht in das Menü aufnehmen. Daher haben wir – wir probieren dies hier in einer Access-Datenbank aus – eine Tabelle angelegt, in welcher wir die **FaceId**-Werte speichern, die kein Symbol enthalten (wir hätten das auch andersherum machen können, haben uns aber für diese Variante entschieden).

Um die Steuerelemente ohne Symbol für die **FaceId** in die Tabelle **tblEmptyFaceIds** zu schreiben, haben wir wiederum eine Klasse mit einer Ereignisprozedur angelegt. Diese ist grundsätzlich genauso aufgebaut wie die zuvor beschriebene Klasse **clsCommandBarButton**.

Sie heißt jedoch **clsCommandBarButtonFaceID** und enthält die folgende Ereignisprozedur:

```
Private Sub m_CommandBarButton_Click(ByVal Ctrl As _
        Office.CommandBarButton, CancelDefault As Boolean)
    CurrentDb.Execute "INSERT INTO " _
        & "tblEmptyFaceIDs(FaceID) VALUES(" _
        & m_CommandBarButton.Tag & ")", dbFailOnError
End Sub
```

```
Public Sub AlleEingebautenIconsNachID()
    Dim cbr As CommandBar
    Dim cbb As CommandBarButton
    Dim l As Long
    Dim objCommandBarButton As clsCommandBarButtonFaceID
    Set colCommandBarButtons = New Collection
    On Error Resume Next
    VBE.CommandBars("AlleIcons").Delete
    On Error GoTo 0
    Set cbr = VBE.CommandBars.Add("AlleIcons")
    With cbr
        For l = 1 To 6000
            If IsNull(DLookup("FaceID", "tblEmptyFaceIDs", "FaceID = " & l)) Then
                Set cbb = cbr.Controls.Add(msoControlButton)
                With cbb
                    .FaceId = l
                    .TooltipText = l
                    .Tag = l
                End With
                Set objCommandBarButton = New clsCommandBarButtonFaceID
                Set objCommandBarButton.CommandBarButton = cbb
                colCommandBarButtons.Add objCommandBarButton
            End If
        Next l
    End With
    cbr.Visible = True
End Sub
```

Listing 3: Funktion zum Schreiben aller Icons in eine Symbolleiste

PowerPoint: Texte automatisiert übersetzen

Neulich war es mal wieder so weit: Eine PowerPoint-Präsentation musste her. Und das auch noch auf Englisch. Okay, das Schul-Englisch ist zum Verstehen und schriftliche Kommunikation ausreichend, aber eine PowerPoint-Präsention für englischsprachiges Fachpublikum sollte schon annähernd perfekt sein. Wozu gibt es Übersetzungsdienste? Also habe ich meine Texte auf Deutsch zurechtgelegt und diese von der KI übersetzen lassen. Dann habe ich alles in die PowerPoint-Präsentation eingefügt und noch die Animationen hinzugefügt, damit beispielsweise Stichpunkte Schritt für Schritt eingeblendet werden können. All das hat so gut geklappt, dass ich die Präsentation anschließend auch noch für ein Video aufbereiten wollte – diesmal jedoch auf Deutsch. Also habe ich erstmal eine komplette Seite kopiert, übersetzen lassen und wieder zurückgeschrieben. Das habe ich für einige Folien gemacht und dann schnell festgestellt, dass so alle Animationen verloren gehen. Der nächste Ansatz dann: Absatz für Absatz in die Zwischenablage, übersetzen lassen, wieder zurückschreiben. So blieben die Animationen erhalten, aber es war zu viel Handarbeit. Wozu beherrsche ich – im Gegensatz zu Englisch – eigentlich perfekt VBA? Also habe ich mich an die Programmierung der Übersetzung der enthaltenen Texte begeben. Das Ergebnis siehst Du in diesem Artikel!

PowerPoint ist in diesem Magazin bisher noch gar nicht vorgekommen – höchstens als Randnotiz in Zusammenhang mit der Office-Programmierung. In diesem Artikel wollen wir jedoch direkt einmal eine praktische Lösung liefern.

Ausgangspunkt ist ein Dokument mit Folien wie der aus Bild 1. Wir haben hier verschiedene Absätze mit englischem Text, die wir gern übersetzen würden. Dabei gibt es verschiedene Vorgehensweisen:

- Die erste ist, einfach manuell in das Dokument zu gehen und die Texte Wort für Wort zu übersetzen. Das ist recht viel Arbeit und kostet entsprechend Zeit.

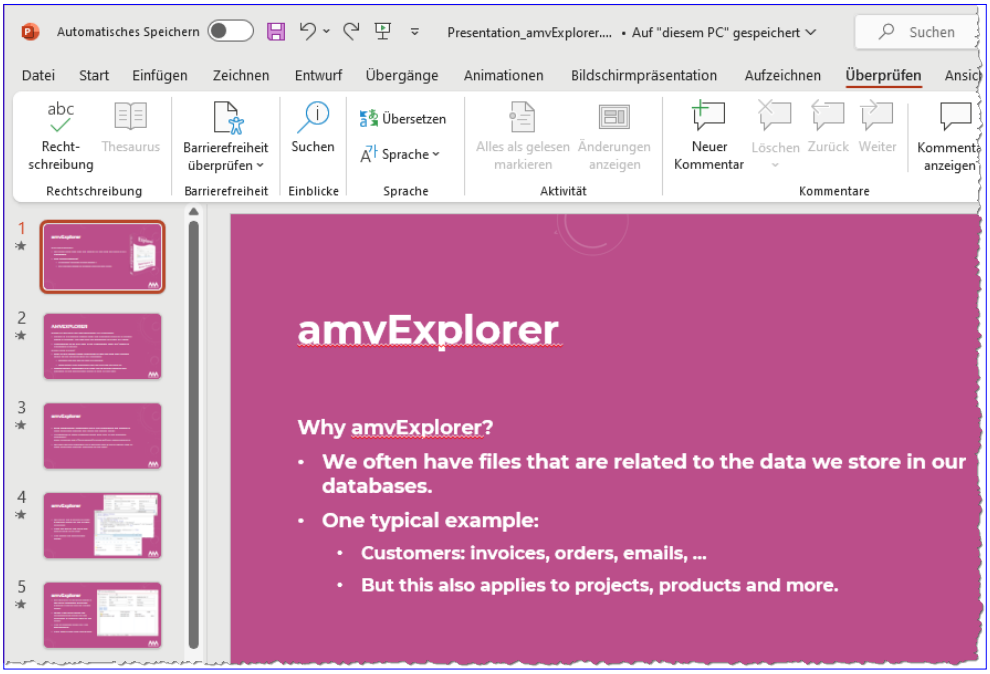

Bild 1: Ein zu übersetzendes PowerPoint-Dokument

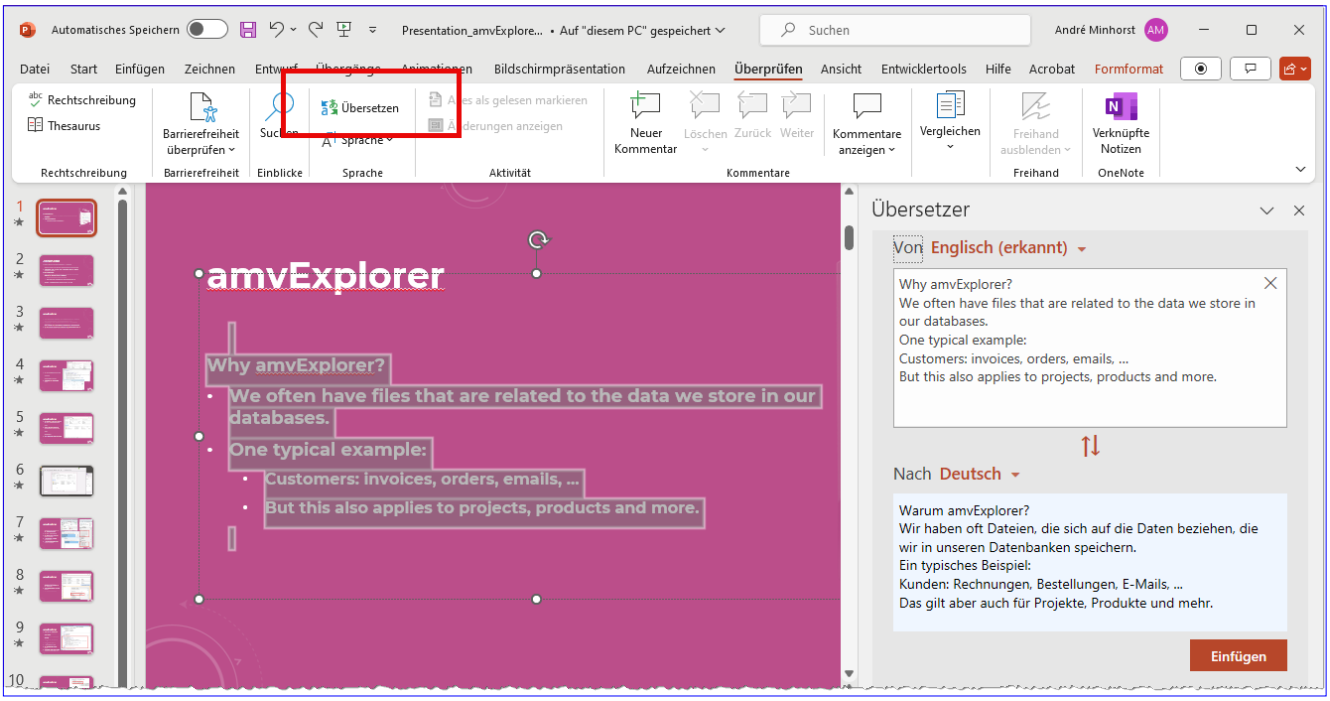

Bild 2: Der eingebaute Übersetzter

- Die zweite ist, die Texte seitenweise zu kopieren und von der KI übersetzen zu lassen. Wenn man die übersetzte Seite dann zurückkopiert, verliert man jedoch eventuell für einzelne Absätze festgelegte Animationen.

- Die dritte ist, sich einmal im Ribbon von PowerPoint umzuschauen und festzustellen, dass es dort einen Eintrag namens Übersetzen gibt. Markieren wir damit die vollständige Seite, zeigt dieser wie in Bild 2 die englische und die ins Deutsche übersetzte Version des Textes an. Wenn wir hier allerdings auf Einfügen klicken, stellen wir fest, dass auch hier einfach der ganze Text übersetzt und wieder zurückgeschrieben wird – ohne Berücksichtigung der Animationen.

- Die vierte Stufe wäre, die Texte absatzweise zu markieren und den Rest den eingebauten Übersetzer erledigen zu lassen. Auf diese Weise habe ich zuvor mit der externen KI gearbeitet, aber auch dabei sind Fehler passiert: Wenn man den Zeilenumbruch mit kopiert und die Übersetzung für den vollständigen Absatz einfügt, gehen wiederum die Animationen verloren. Man muss also genau prüfen, ob man das Zeilenumbruchzeichen nicht mit kopiert. Wie kann man das verhindern? Indem man darauf achtet, dass die Markierung sich nur genau bis zum letzten sichtbaren Zeichen des Absatzes erstreckt. Wenn der Zeilenumbruch ebenfalls markiert ist, enthält die Markierung noch einen kleinen Bereich hinter dem letzten Zeichen. Das passiert beispielsweise, wenn man den Absatz durch einen dreifachen Mausklick selektiert. Auch diese Variante kostet immer noch recht viel Zeit (siehe Bild 3).

- Die fünfte und nachfolgend vorgestellte Variante ist, alle Textabschnitte der PowerPoint-Präsentation automatisiert zu durchlaufen, die Texte zu übersetzen und diese wieder zurückzuschreiben, ohne dass Animationen verloren gehen.

Diesen Ansatz schauen wir und nun an. Dazu benötigen wir ein paar Grundlagen zur PowerPoint-Pro-

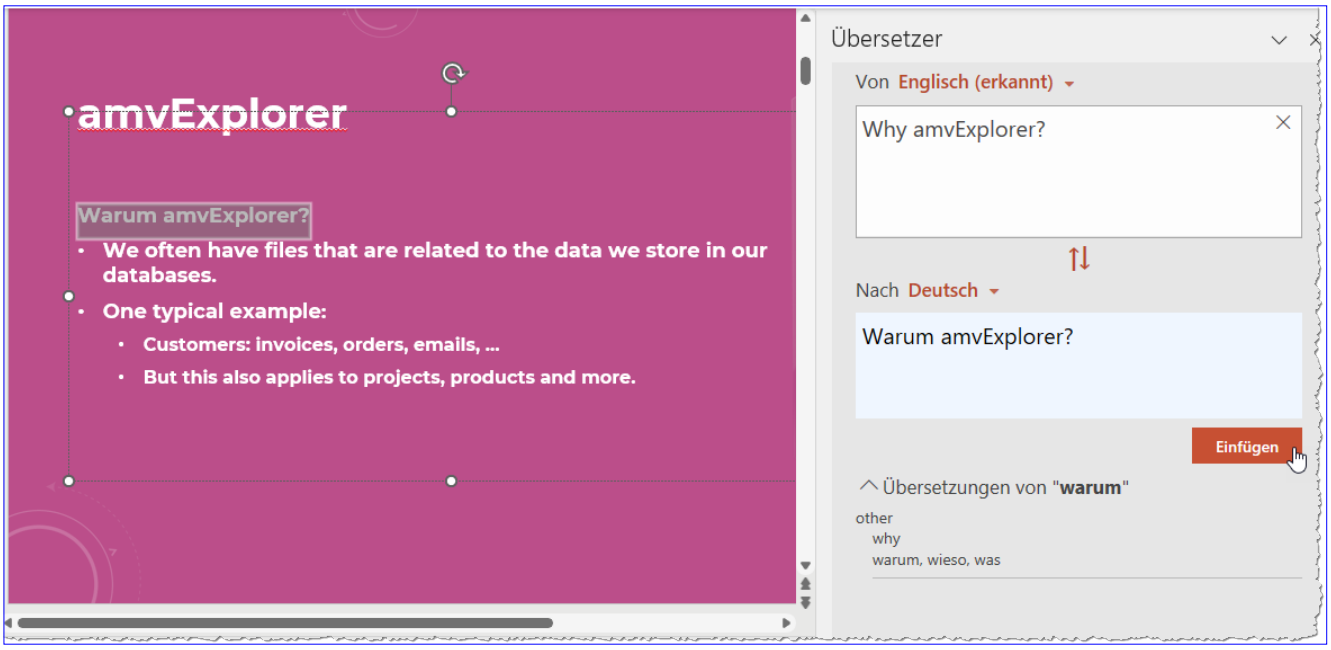

Bild 3: Texte absatzweises übersetzen

grammierung und werfen einen Blick auf das Objektmodell dieser Anwendung.

Das Objektmodell von PowerPoint

Wir werden uns nicht das vollständige Objektmodell ansehen, sondern nur die für uns relevanten Abschnitte. Bevor wir damit starten, legen wir ein neues Standardmodul im VBA-Editor von PowerPoint an, den wir mit der Tastenkombination **Alt + F11** öffnen.

Hier stellen wir fest, dass es keine Standardklassen oder -module gibt, die bereits beim Anlegen eines PowerPoint-Dokuments vorhanden sind – so, wie es beispielsweise bei Excel oder Word der Fall ist. Also können wir mit einem frischen, leeren VBA-Projekt starten.

PowerPoint referenzieren

Das **Application**-Objekt ist das oberste Element im Objektmodell von PowerPoint. Wir deklarieren es mit der folgenden Zeile:

```
Dim objPowerPoint As PowerPoint.Application
```

Je nachdem, ob wir von einem Modul innerhalb des PowerPoint-Dokuments auf das **Application**-Objekt zugreifen oder von außen, ändert sich die Referenzierung.

Für unser Beispiel reicht es erst einmal, wenn wir wie folgt darauf zugreifen:

```
Set objPowerPoint = Application
```

PowerPoint-Präsentation referenzieren

Damit haben wir bereits Zugriff auf die Anwendung. Nun wollen wir auch noch das aktuell angezeigte Dokument referenzieren. Dazu nutzen wir wie nachfolgend gezeigt eine Objektvariable des Typs **PowerPoint.Presentation**:

```
Dim objPresentation As PowerPoint.Presentation
```

Die aktuell geöffnete Präsentation erhalten wir mit der Funktion **ActivePresentation**:

```
Set objPresentation = objPowerPoint.ActivePresentation
```

Slides durchlaufen

Eine PowerPoint-Präsentation besteht zuerst einmal aus den Folien. Diese können wir mit einer Variablen des Typs **Slide** referenzieren:

```
Dim objSlide As PowerPoint.Slide
```

Auf die einzelnen Slides greifen wir zum Beispiel über den Index zu. Dieser ist praktischerweise 1-basiert, sodass wir die Slides entsprechend der Nummer der Slide-Übersicht am linken Rand des PowerPoint-Fensters ansprechen können. Mit der **Count**-Methode können wir die Anzahl der Folien ermitteln:

```
Debug.Print objPresentation.Slides.Count
```

Wir wollen diese aber ohnehin nacheinander durchlaufen, sodass uns eine **For Each**-Schleife dienen würde. Hier geben wir zuerst einmal den Wert der Eigenschaft **SlideNumber** aus:

```
For Each objSlide in objPresentation.Slides
    Debug.Print objSlide.SlideNumber
Next objSlide
```

Shapes durchlaufen

Unseren Texten noch ein wenig näher kommen wir mit der **Shapes**-Auflistung. Shapes sind alle Elemente, die wir über die Benutzeroberfläche direkt anklicken können.

Diese werden auch im Taskpane names **Auswahl** angezeigt (siehe Bild 4). Diesen blendest Du am einfachsten ein, wenn Du oben in der Suche die Zeichenfolge Pane eingibst und dann auf Auswahlbereich anzeigen klickst.

Um mit den Shapes zu arbeiten, deklarieren wir eine Variable des Typs **Shape**:

```
Dim objShape As PowerPoint.Shape
```

Diese Elemente können wir für jedes **Slide**-Objekt ebenfalls in einer **For Each**-Schleife durchlaufen. In diesem Fall wollen wir einfach nur den Namen eines jeden **Shape**-Elements ausgeben:

```
For Each objShape In objSlide.Shapes
    Debug.Print objShape.Name
Next objShape
```

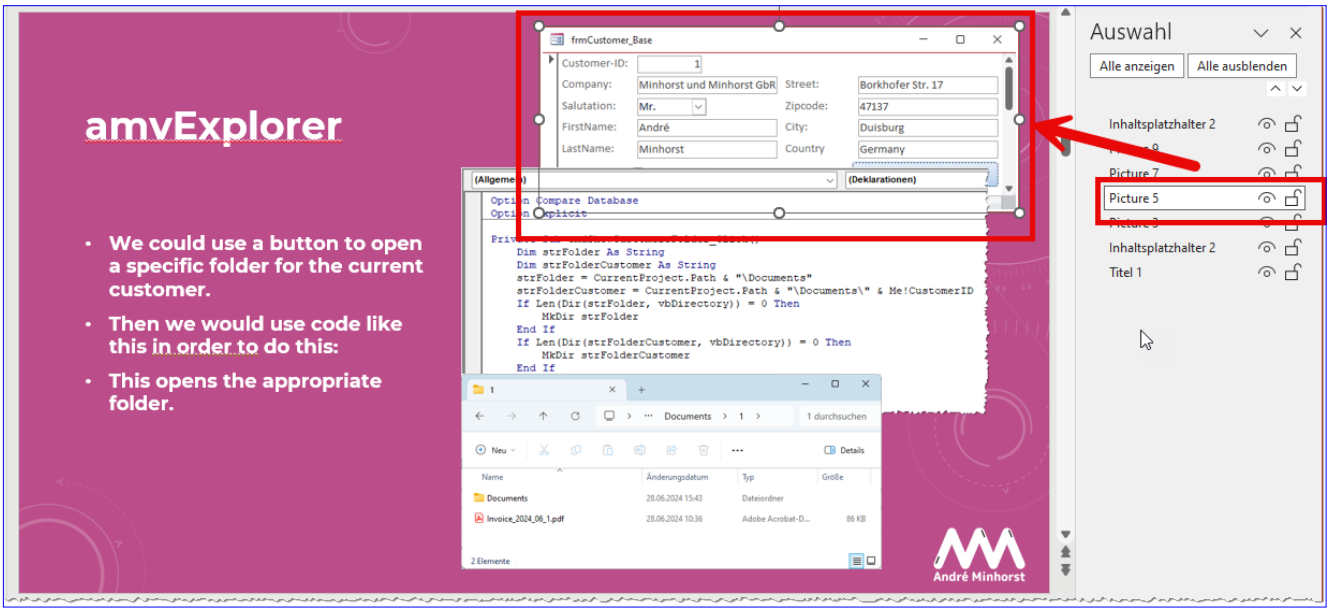

Bild 4: Auflistung der **Shape**-Elemente in der Benutzeroberfläche

PowerPoint-Übersetzung per COM-Add-In

Im Artikel »PowerPoint: Texte automatisiert übersetzen« (www.vbentwickler.de/437) haben wir VBA-Code produziert, mit dem wir alle Absätze aller Folien in einem PowerPoint-Dokument automatisch übersetzen können. Dabei nutzen wir den Dienst DeepL. Leider müssen wir, um diesen Code in einem PowerPoint-Dokument verwenden zu können, das Modul erst in das jeweilige Dokument integrieren. Wenn man oft PowerPoint-Folien übersetzen muss, ist das recht aufwändig. Da nehmen wir lieber den Aufwand in Kauf, einmal ein COM-Add-In für diesen Zweck zu programmieren, dass wir dann auch noch an Dich weitergeben können, damit Du es für Dich und Deine Mitarbeiter und/oder Kunden einsetzen kannst.

Das Werkzeug für COM-Add-Ins: twinBASIC

Wie üblich nutzen wir das in der 32-Bit-Version sogar kostenlose Produkt twinBASIC für die Erstellung des COM-Add-Ins. Im Download findest Du, auch wenn Du Dir noch nicht die kostenpflichtige Version von twinBASIC gekauft hast, aber auch eine 64-Bit-Version der DLL, die das COM-Add-In enthält. Da wir keine DLL-Funktionen nutzen, ist der Code für beide Varianten allerdings identisch.

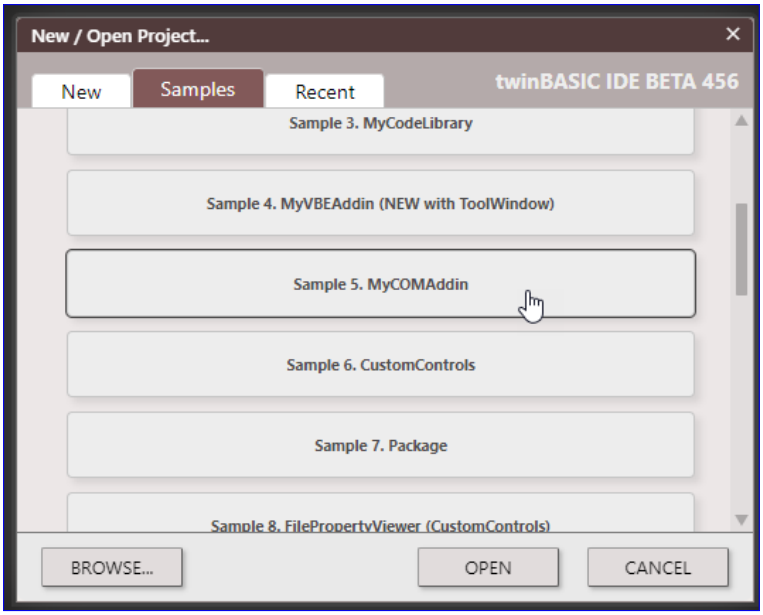

Bild 1: Auswahl des Projekttyps

twinBASIC findest Du beispielsweise unter dem folgenden Link:

```
https://github.com/twinbasic/twinbasic/releases/tag/beta-x-0585
```

Nach dem Start (es ist keine Installation nötig) erscheint der Dialog aus Bild 1. Hier wählen wir unter Samples den Eintrag **Sample 5. MyCOMAddin** aus.

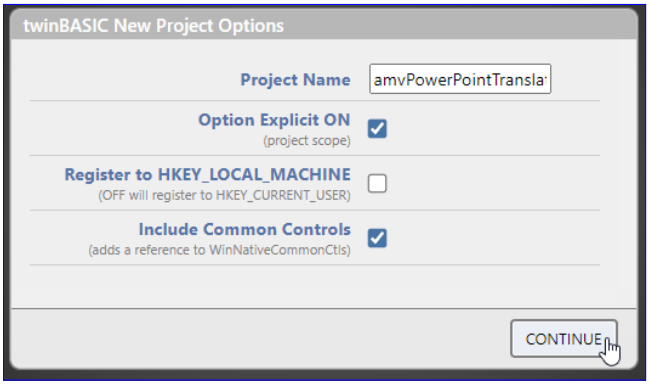

Bild 2: Projektname für das neue Projekt

Anschließend erscheint der Dialog aus Bild 2, wo wir den Namen des Projekts festlegen – hier auf **amvPowerPointTranslator**. Danach finden wir bereits die Entwicklungsumgebung vor, in der wir die ersten grundlegenden Änderungen durchführen.

Änderungen am Beispielprojekt für das COM-Add-In

Als Erstes stellen wir den Klassennamen sowohl im Code der Klasse als auch für die Klassendatei auf **amvPowerPointTranslator** ein (siehe Bild 3). Nach dem Ändern im Modul speichern wir das Projekt zunächst, damit wir im nächsten Schritt das Modul schließen können, um den Namen der **.twin**-Datei anzupassen.

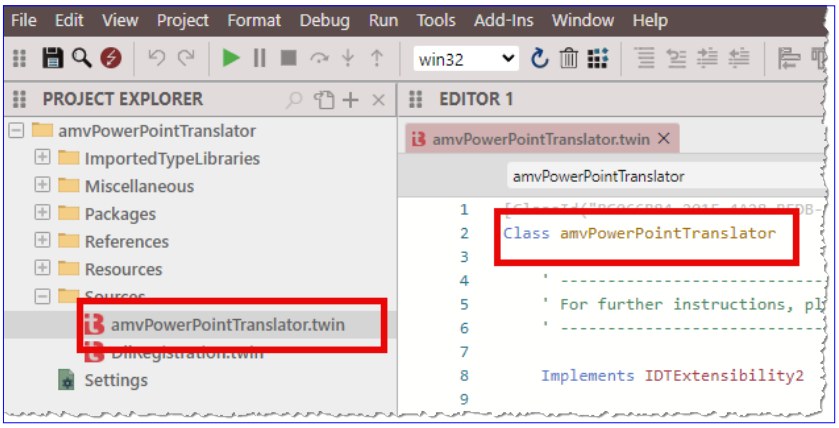

Bild 3: Eintragen des neuen Klassennamens

Verweis auf PowerPoint hinzufügen

Damit wir die PowerPoint-Elemente unter Verwendung von IntelliSense nutzen können, fügen wir dem Projekt unter **Project Settings** im Bereich **Library References** einen Verweis auf die Bibliothek **Microsoft PowerPoint 16.0 Object Library** hinzu (siehe Bild 4).

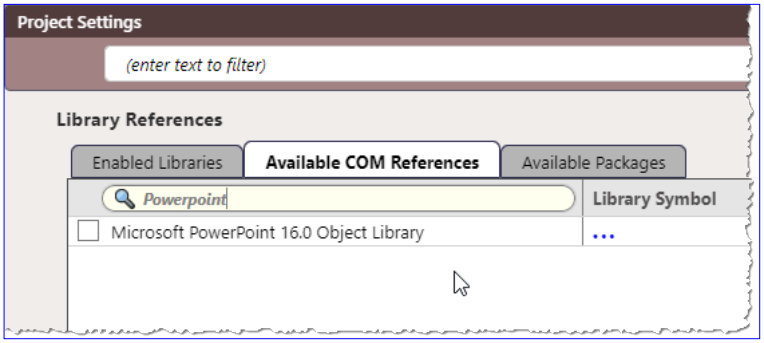

Bild 4: Auswahl der **PowerPoint**-Bibliothek

Funktionen für die Registrierung des COM-Add-In anpassen

Die Beispielfunktionen zum Durchführen der Registrierung in der Datei **DllRegistration.twin** sind für den Einsatz in Access- und Excel-Anwendungen ausgelegt. Dies passen wir noch an. Dabei stellen wir vor allem die Konstante **AddinClassName** ebenfalls auf **amvPowerPointTranslator** ein.

Außerdem ändern wir die Konstante **RootRegistryFolder_Excel** auf den Namen **RootRegistryFolder_PowerPoint** und fügen auch in dem Wert den Eintrag **PowerPoint** statt **Excel** ein. Die Konstante **RootRegistryFolder_Access** löschen wir.

In den beiden Funktionen **DllRegisterServer** und **DllUnregisterServer** ersetzen wir **RootRegistryFolder_Excel** durch **RootRegistryFolder_PowerPoint** und löschen alle Einträge für **RootRegistryFolder_Access**. Das Ergebnis sieht in gekürzter Form wie in Listing 1 aus.

Klasse amvPowerPointTranslator anpassen

Auch den Inhalt der Datei **amvPowerPointTranslator.twin** passen wir an. Hier löschen wir die Variable **applicationObject** durch **objPowerPoint** und legen ihren Datentyp auf **PowerPoint.Application** fest. Diese weisen wir in der Ereignisprozedur **OnConnection** wie folgt zu:

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

```
Module DllRegistration
    Const AddinProjectName As String = VBA.Compilation.CurrentProjectName
    Const AddinClassName As String = "amvPowerPointTranslator"
    Const AddinQualifiedClassName As String = AddinProjectName & "." & AddinClassName
    Const RootRegistryFolder_PowerPoint As String = "HKCU\SOFTWARE\Microsoft\Office\PowerPoint\Addins\" _
        & AddinQualifiedClassName & "\"

    Public Function DllRegisterServer() As Boolean
        On Error GoTo RegError
        Dim wscript As Object = CreateObject("wscript.shell")
        wscript.RegWrite RootRegistryFolder_PowerPoint & "FriendlyName", AddinProjectName, "REG_SZ"
        wscript.RegWrite RootRegistryFolder_PowerPoint & "Description", AddinProjectName, "REG_SZ"
        wscript.RegWrite RootRegistryFolder_PowerPoint & "LoadBehavior", 3, "REG_DWORD"
        ...
    End Function

    Public Function DllUnregisterServer() As Boolean
        On Error GoTo RegError
        Dim wscript As Object = CreateObject("wscript.shell")
        wscript.RegDelete RootRegistryFolder_PowerPoint & "FriendlyName"
        wscript.RegDelete RootRegistryFolder_PowerPoint & "Description"
        wscript.RegDelete RootRegistryFolder_PowerPoint & "LoadBehavior"
        wscript.RegDelete RootRegistryFolder_PowerPoint
        ...
    End Function
End Module
```

Listing 1: Funktionen zum Registrieren und Deregistrieren

In der Prozedur **OnHelloWorldClicked** finden wir auch noch ein Vorkommen der Variablen **applicationObject**, das wir durch **objPowerPoint** ersetzen. Den Rest belassen wir erst einmal so, wie er ist, und kompilieren das Projekt erstmalig mit dem Befehl **File|Build**.

Der Kompiliervorgang sollte genau wie die Registrierung nun erfolgreich verlaufen sein. Sollte das nicht der Fall, nutze entweder das Beispielprojekt aus dem Download oder gehe die bisherige Anleitung nochmals durch.

Normalerweise sollte aber nun beim Öffnen von PowerPoint ein Tab namens **twinBASIC Test** erscheinen und ein Klick auf die enthaltene Schaltfläche sollte die Meldung aus Bild 5 anzeigen.

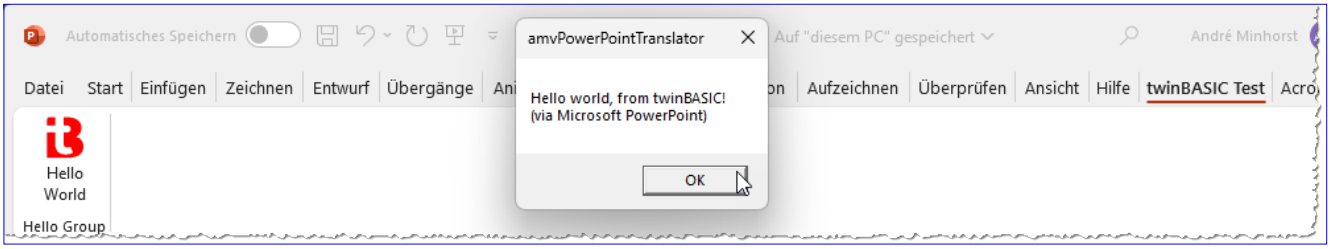

Bild 5: Das COM-Add-In ist bereits startbereit.

Funktion implementieren

Nun brauchen wir eigentlich erst einmal nur die Funktionen aus der Lösung aus dem Artikel **PowerPoint: Texte automatisiert übersetzen** (**www.vbentwickler.de/437**) in ein neues Modul im twinBASIC-Projekt einzufügen.

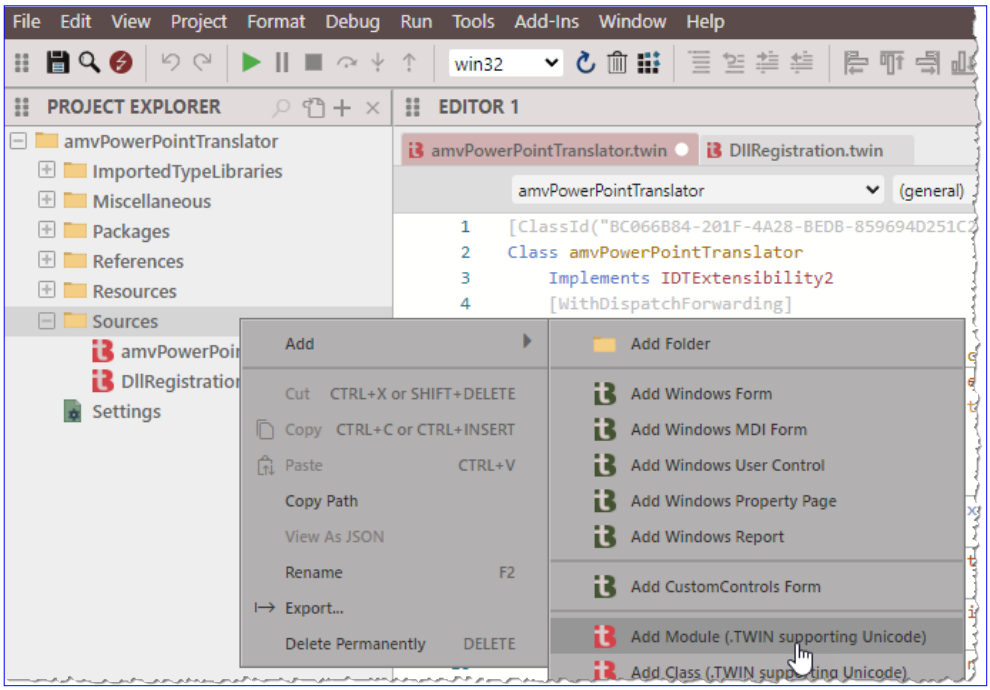

Bild 6: Neues Modul hinzufügen

Dafür benötigen wir alle vier Module:

- **mdlDeepL**
- **mdlJSON**
- **mdlJSONDOM**
- **mdlUebersetzen**

Diese können wir leider nicht per Drag and Drop einfügen, daher müssen wir jeweils ein neues Modul erstellen und den Code des jeweiligen Moduls dort hineinkopieren.

Neue Module erstellen wir wie in Bild 6 und füllen dann die Inhalte der Module des VBA-Projekts der **PowerPoint**-Datei zwischen die Anweisungen **Module ...** und **End Module** ein.

Außerdem benötigen wir noch zwei weitere Verweise auf die Bibliotheken **Microsoft Scripting Runtime** und **Microsoft XML, v6.0**, die wir wie oben gezeigt hinzufügen.

Die Variable **objPowerPoint** nehmen wir nun aus der Klasse **amvPowerPointTranslator.twin** heraus, da wir dort nicht von anderen Modulen aus darauf zugreifen können, und fügen sie dem Modul **mdlUebersetzen** hinzu:

```
Private objPowerPoint As PowerPoint.Application
```

Aufruf der Prozedur Translate

Die gleich angepasste Prozedur **Translate** aus dem oben genannten PowerPoint-Artikel rufen wir auf, wenn der Benutzer auf den einzigen bisherigen Button des twinBASIC-Projekts klickt:

```
Public Sub OnHelloWorldClicked(Control As IRibbonControl)
    MsgBox "Hello world, from twinBASIC!" & vbCrLf & _
           "(via " & objPowerPoint.Name & ")"
    Call Translate()
End Sub
```

Code der Lösung anpassen

Danach passen wir den Code unserer Lösung an. Wir können vorab prüfen, ob wir schon Stellen finden, die wir für den Betrieb in einem COM-Add-In anpassen

Klassenprogrammierung mit COM-Add-In vereinfachen

Im Artikel »VBA-Editor: Klasseneigenschaften per Mausklick« (www.vbentwickler.de/422) haben wir eine Prozedur vorgestellt, mit denen man aus einer einfachen Variablendeklaration innerhalb eines Klassenmoduls eine private Membervariable und jeweils eine Property Get-Methode und eine Property Set/Let-Methode erzeugen kann. Der einzige Haken bei dieser Lösung ist, dass man diese bisher noch über den Direktbereich aufrufen musste. Da das nicht sonderlich komfortabel ist, stellen wir in diesem Artikel eine Lösung vor, bei der wir mit twinBASIC ein COM-Add-In für den VBA-Editor erstellen, mit dem wir die Funktion aus dem oben genannten Artikel über die Menüleiste, die Symbolleiste oder auch über das Kontextmenü des VBA-Editors aufrufen kann.

Wir bauen hier auf den Vorbereitungen auf, die wir im Artikel **twinBASIC: COM-Add-In für den VBA-Editor** (**www.vbentwickler.de/421**) vorgestellt haben. Hier haben wir ein Basis-COM-Add-In für den VBA-Editor vorgestellt, das einfach nur ein paar Dummy-Einträge im Menü und im Kontextmenü anzeigt, die Meldungsfenster liefern. Den ersten sehen wir in Bild 1.

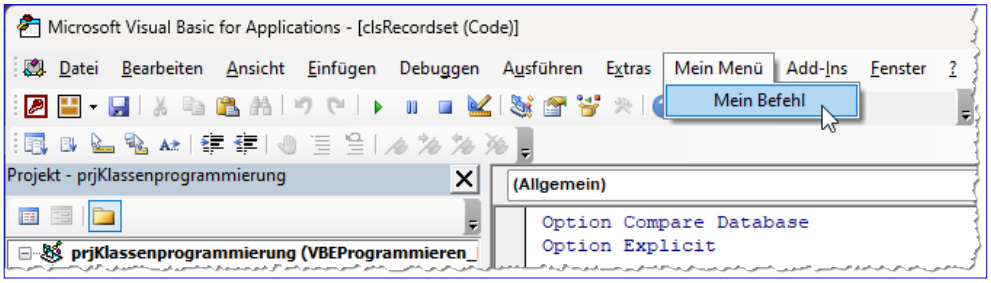

Bild 1: Menüeintrag unseres COM-Add-Ins

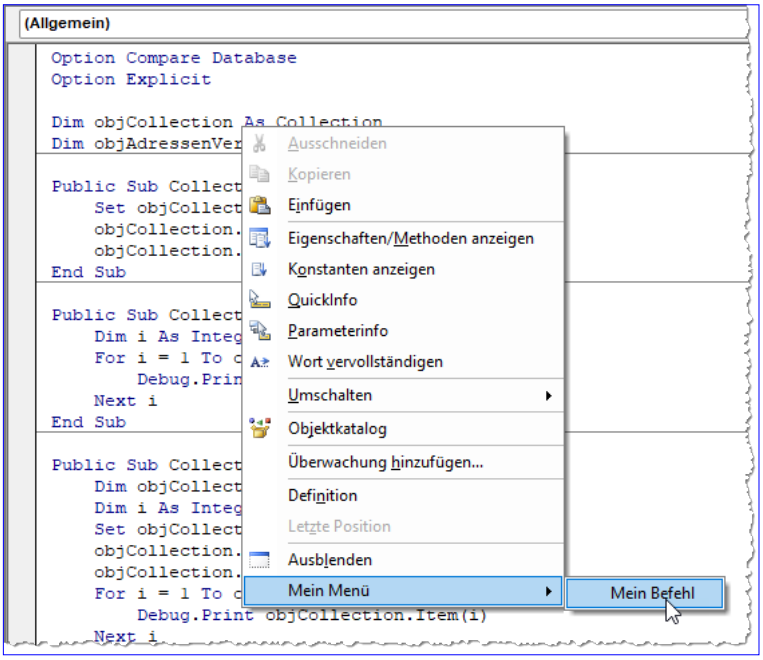

Bild 2: Menüeintrag im Kontextmenü

Den zweiten Eintrag, den wir im Kontextmenü untergebracht haben, finden wir im Kontextmenü in Bild 2 ganz unten.

In der Symbolleiste haben wir noch keinen Eintrag untergebracht. Auch das wollen wir gegebenenfalls noch erledigen. Erst einmal schauen wir uns jedoch die zu programmierende Funktion an – wo im Menü wir diese optimal unterbringen.

Umwandeln von Variablen in Property Get/Let/Set

Die Aufgabe lautet, die Prozedur, die wir im oben genannten Artikel entwickelt haben, über ein COM-Add-In für den VBA-Editor

verfügbar zu machen. Diese Aufgabe ist einfach – wir brauchen eigentlich nur unsere Menüpunkte umzubenennen und diesen den Aufruf der gewünschten Prozedur hinzuzufügen.

Allerdings wollen wir das mit Bedacht tun, damit wir den bestmöglichen Ort für den Menüeintrag finden. Es geht darum, eine oder mehrere Variablen im Code eines Klassenmoduls zu markieren und dann die Prozedur **VariableToProperty** aufzurufen. Diese holt sich die aktuelle Markierung und passt den gefundenen Code entsprechend an.

Um diese Funktion nach dem Selektieren der zu bearbeitenden Code-Passagen aufzurufen: Was wäre der geeignete Ort? Wir wollen mit möglichst wenig Klicks und möglichst wenig Mausmetern zum Ziel kommen. Wenn wir also einen Eintrag im Menü verwenden, sollte dieser direkt erreichbar sein. Dazu bietet sicher eher die Symbolleiste an, in der wir den Befehl über ein geeignetes Icon anbieten können. Allerdings muss man für eine solche erklärungsbedürftige Aufgabe erst einmal ein passendes Icon finden.

Die zweite Möglichkeit ist, diesen Befehl in der Menüleiste selbst unterzubringen. Hier sollten wir allerdings keine Befehle in der Hauptmenüleiste unterbringen, sondern vielleicht einen Hauptmenüpunkt unterbringen, der nicht nur diesen, sondern gegebenenfalls noch andere Einträge mit Aufrufen unserer selbst definierten Funktionen enthält. Die Struktur mit **Mein Menü|Mein Befehl** ist daher schon passend. Damit wären wir nach dem Markieren der zu bearbeitenden Codezeilen mit zwei Mausklicks am Ziel.

Die nächste Möglichkeit ist das Kontextmenü. Hier benötigen wir grundsätzlich einen Klick mehr, weil wir diese erst einmal mit einem Rechtsklick anzeigen müssen. Wenn wir den Befehl dann direkt im Kontextmenü anzeigen, kommen wir schnell zum Ziel. Wenn das Kontextmenü noch keine Einträge weiterer COM-Add-Ins enthält, kann man das so machen. Wenn wir jedoch planen, noch weitere benutzerdefinierte Funktionen zu diesem oder anderen COM-Add-Ins hinzuzufügen, können wir auch einen Unterpunkt hinzufügen.

Das wäre dann zwar ein Mausklick mehr, aber letztlich sparen wir durch die Funktion an sich schon so viel Zeit, dass sich das nicht besonders auswirken sollte.

Also starten wir wie folgt:

- Wir fügen der Menüleiste einen Menüpunkt beispielsweise namens **AMV-Tools** hinzu. Unter diesem legen wir den Aufruf zum Umwandeln der Variablen in **Property Get/Let/Set**-Prozeduren an.

- Wir fügen auch dem Kontextmenü einen Menüpunkt namens **AMV-Tools** hinzu. Dieser enthält wiederum eine Schaltfläche zum Aufrufen der gewünschten Funktion.

- Schließlich wollen wir den Befehl auch noch in der Symbolleiste unterbringen. Hier können wir den Befehl direkt unterbringen, sollten uns jedoch auf ein Icon beschränken.

- Außerdem fügen wir die gewünschte Funktion zum Umwandeln von Variablen in Propertys hinzu und testen diese ausführlich.

Passende Symbolleiste finden

Der VBA-Editor enthält einige Symbolleisten. Welche verwenden wir optimalerweise für unsere Funktion? Schauen wir uns erst einmal an, welche eingebauten Symbolleisten der VBA-Editor alle anbietet. Dazu lassen wir in einer VBA-Anwendung die folgende Prozedur laufen:

```
Public Sub ListMenus()
    Dim cbr As Office.CommandBar
```

```
    For Each cbr In VBE.CommandBars
        If cbr.Type = msoBarTypeNormal Then
            Debug.Print cbr.Name
        End If
    Next cbr
End Sub
```

Dies liefert das folgende Ergebnis:

```
Voreinstellung
Bearbeiten
Debuggen
UserForm
Clipboard
```

Wenn wir uns diese Symbolleisten einmal anschauen, können wir besser entscheiden, welche der richtige Ort ist. Dazu blenden wir einmal alle Symbolleisten ein. Das erledigen wir über den Menüpunkt **Ansicht|Symbolleisten**. Hier finden wir alle verfügbaren Symbolleisten vor – wobei aus der per VBA ausgegebenen Liste der Eintrag **Clipboard** fehlt. Dafür finden wir einen zusätzlichen Eintrag namens **Anpassen...** (siehe Bild 3).

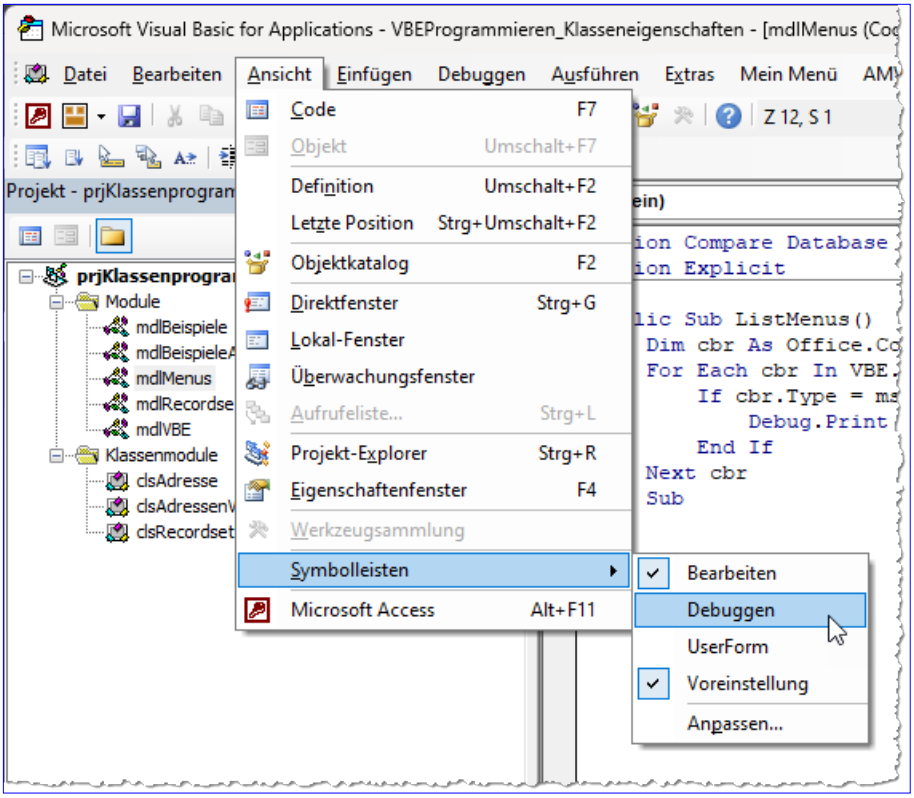

Bild 3: Einblenden aller Symbolleisten

Die eingeblendeten Symbolleisten erscheinen zunächst freischwebend, was normalerweise nicht erwünscht ist – außer, man benötigt die entsprechende Funktion direkt in der Nähe der damit zu bearbeitenden Elemente. Dann stellt sich jedoch die Frage, ob der Einsatz eines Kontextmenüs nicht ohnehin die bessere Lösung wäre.

Für unseren Zweck scheint uns jedenfalls die Symbolleiste **Bearbeiten** am geeignetsten zu sein. Also fügen die unseren Befehl dort hinzu.

Einfügen eines Befehls in der Bearbeiten-Symbolleiste und mehr

Nun schauen wir uns an, wie wir die im Artikel **twinBASIC: COM-Add-In für den VBA-Editor** (**www.vbentwickler.de/421**) vorgestellte Vorlage genau anpassen, um unsere Funktionen im VBA-Editor bereitzustellen. Wir gehen an dieser Stelle nicht mehr auf alle dort beschriebenen Details ein.

Um die Menüs mit unseren Schaltflächen zu füllen, benötigen wir zuerst einmal die Prozedur, die beim Verbinden des VBA-Editors mit dem COM-Add-In ausgelöst wird.

Diese weist der Objektvariablen **objVBE**, die wir öffentlich in einem Modul namens **mdlGlobal** definieren, den mit dem Para-

```
Sub OnConnection(ByVal Application As Object, ByVal ConnectMode As ext_ConnectMode, ByVal AddInInst As Object, _
        ByRef custom As Variant()) Implements IDTExtensibility2.OnConnection
    Set objVBE = Application
    Set objAddIn = AddInInst
    isConnected = True
    CreateToolBar()
End Sub
```

Listing 1: Diese Prozedur wird beim Verbinden des COM-Add-Ins ausgelöst.

meter **Application** gelieferten Verweis auf die aufrufende Anwendung zu:

```
Public objVBE As VBIDE.VBE
```

Außerdem rufen wir in dieser Prozedur, die in Listing 1 abgebildet ist, die Prozedur **CreateToolbar** auf.

Diese referenziert zuerst die Menüleiste mit der Variablen **cbr** und fügt dieser ein Untermenü hinzu (siehe Listing 2). Dieses platziert sie vor dem achten Untermenü, stellt die Beschriftung auf den Wert aus der Konstanten **cStrMenu** ein (**AMV-Tools**) und fügt diesem ein mit der Variablen **cbb** referenziertes **CommandBarButton**-Element hinzu.

Für dieses legt es die Beschriftung aus der Konstanten **cStrCommandButton** fest (**Getter und Setter für Variable**) und stellt das Icon mit der **Picture**-Eigenschaft auf das Ergebnis der Funktion **GetImage** mit dem Parameter **GetSet.png** ein. Schließlich weisen wir der Objektvariablen **cbbMenuEvent** die Ereignisse des **CommandBarButton**-Objekts zu.

Die Objektvariablen für die Menüs deklarieren wir im allgemeinen Teil des Menüs wie folgt:

```
Private cbbToolbar As CommandBarButton
Private cbbMenu As CommandBarButton
Private cbbContext As CommandBarButton
```

Direkt danach folgen die Deklarationen der Event-Klassen für die drei anzulegenden Schaltflächen:

```
Private WithEvents cbbToolbarEvent As _
    VBIDE.CommandBarEvents
Private WithEvents cbbContextEvent As _
    VBIDE.CommandBarEvents
Private WithEvents cbbMenuEvent As VBIDE.commandbarEvents
```

Damit legen wir nun den Eintrag in der Symbolleiste an. Dazu referenzieren wir die Symbolleiste **Bearbeiten** mit der Variablen **cbr**. Dieser fügen wir ein neues Element hinzu und referenzieren es mit **cbbToolbar**. Dieses erhält keine Beschriftung, aber ebenfalls das Icon. Außerdem referenzieren wir seine Ereignisse mit der Variablen **cbbToolbarEvent**.

Schließlich folgt noch der Eintrag im Kontextmenü. Dazu referenzieren wir das Kontextmenü mit dem Namen **Code Window** und weisen diesem ein neues Untermenü hinzu, das wir mit der Variablen **cbpContext** referenzieren und das wie das Untermenü in der Menüleiste die Beschriftung aus **cStrMenu** erhält.

Diesem Untermenü fügen wir eine Schaltfläche hinzu, die genauso aufgebaut ist wie die im Hauptmenü. Wir referenzieren sie aber mit der Variablen **cbbContext** und seine Events mit der Variablen **cbbContextEvent**.

Damit haben wir alle Menüeintrage hinzugefügt. Nun müssen wir die Ereignisprozeduren definieren.

Ereignisse für die drei Menübefehle programmieren

Dazu wählen wir im Codefenster von twinBASIC jeweils oben im mittleren Auswahlfeld den Namen des

DHL-Versandetiketten erstellen per VBA

Wer Kunden und Bestellungen mit einer Access-Datenbank verwaltet oder gegebenenfalls auch mit einer Excel-Tabelle, möchte vielleicht Zeit sparen und die Etiketten für den Versand von Lieferungen an seine Kunden automatisieren. Das gelingt mit den verschiedenen Webservices von DHL. Wir haben bereits einmal eine solche Lösung vorgestellt, aber DHL hat seine Schnittstellen für die Erstellung von Versandetiketten aktualisiert. In diesem Artikel schauen wir uns an, wie die neuen Schnittstellen funktionieren: Welche Daten benötige ich? Welche URL muss für den Zugriff verwendet werden? In welcher Form übergebe ich beispielsweise die Adressdaten an den Webservice?

Voraussetzung für die Verwendung der Lösung

Wenn wir die Webservices von DHL für die Erstellung von Versandetiketten nutzen wollen, sind einige Voraussetzungen zu erfüllen. Als Erstes benötigen wir ein Konto als Geschäftskunde bei DHL. Wie Du dieses anlegst, wollen wir hier nicht im Detail beschreiben. Wir wollen Dir aber zumindest den Link zum Geschäftskundenportal mit auf den Weg geben:

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

Nach der Anmeldung benötigen wir später Deine Zugangsdaten für Dein Kundenkonto, also Benutzername und Kennwort.

Als Zweites benötigst Du ein Entwicklerkonto bei DHL. Dazu besuchst Du die folgende Adresse und erstellst einen Account:

```
https://developer.dhl.com/
```

Damit können wir bereits starten.

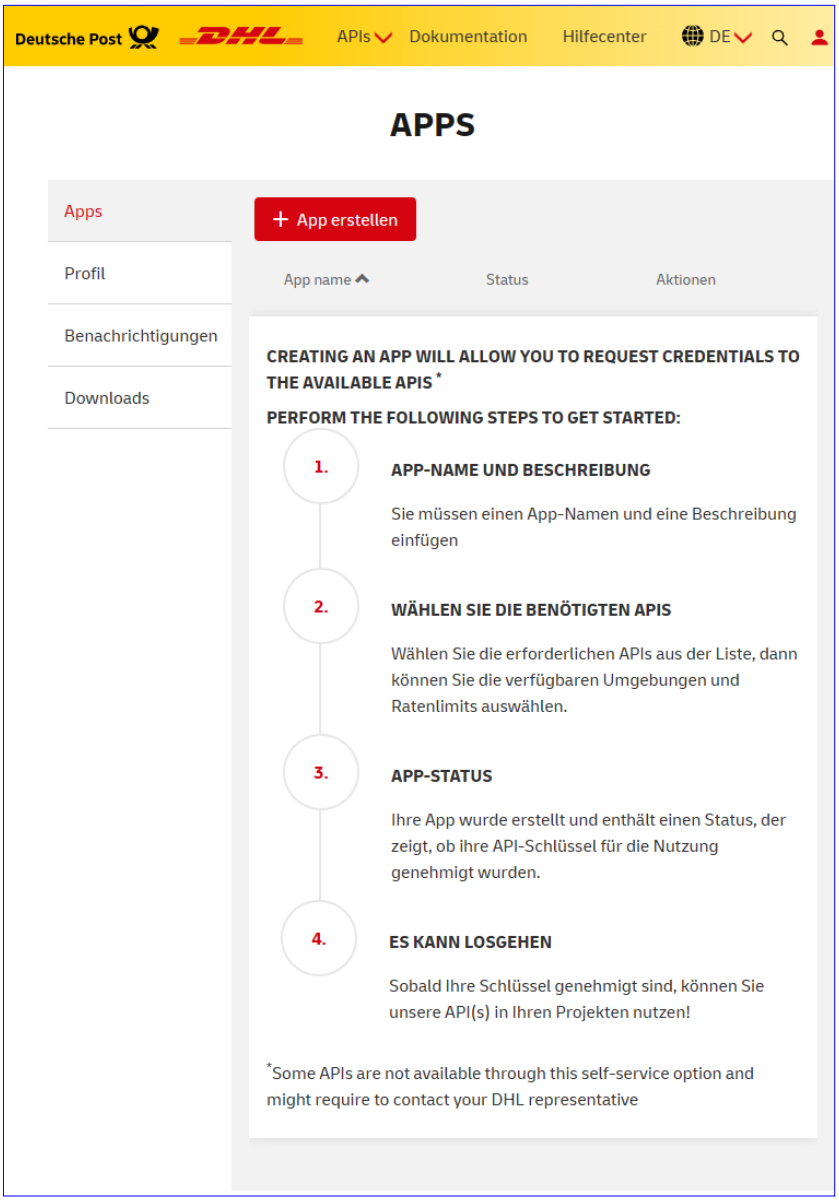

Bild 1: App anlegen für weitere Daten

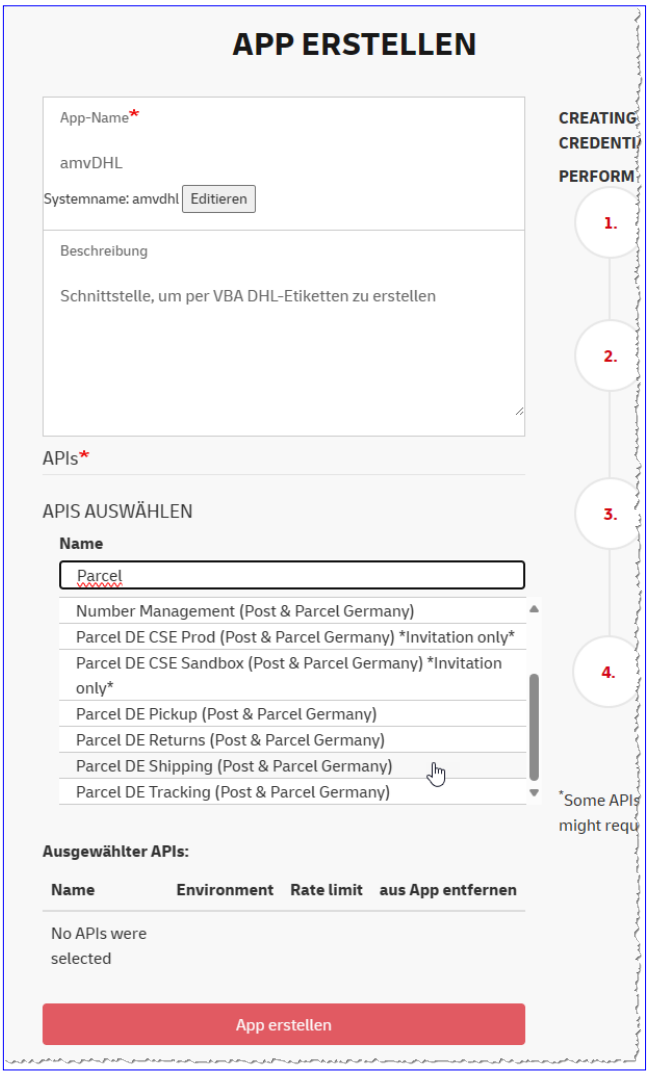

Bild 2: Angeben von App-Name und Beschreibung sowie Auswahl der APIs

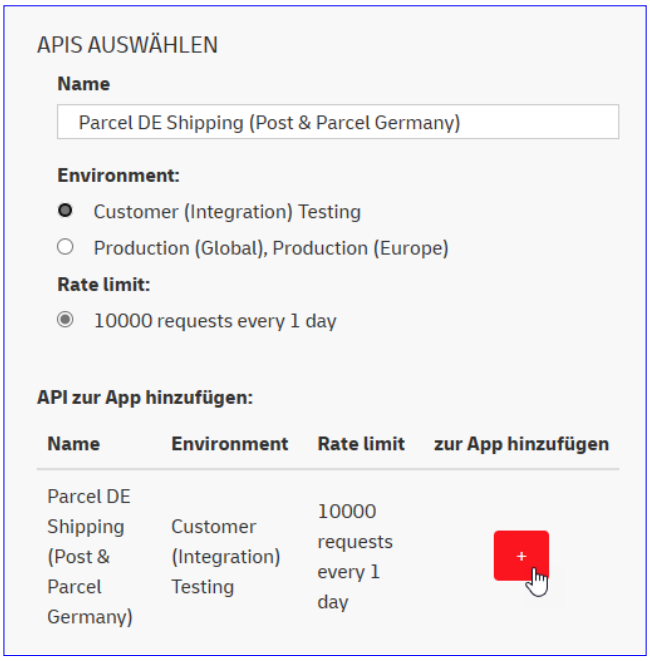

Bild 3: Angabe, ob die App zum Testen oder produktiv genutzt werden soll

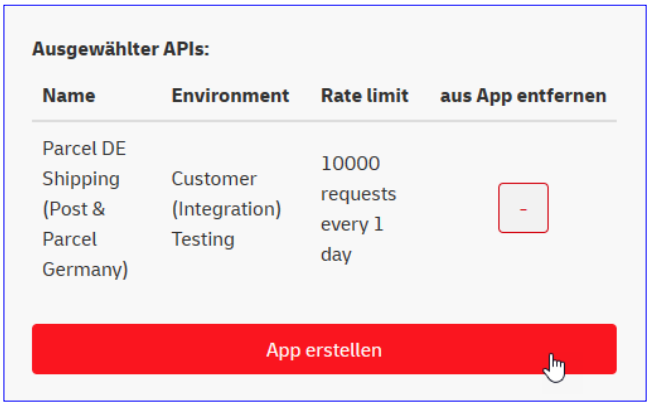

Bild 4: Erstellen der App

App erstellen

Nachdem wir einen Entwickler-Account erstellt haben, benötigen wir eine App. Wozu brauchen wir diese? Weil eine App uns einen API Key und ein API Secret liefert, mit dem wir auf die API-Schnittstelle zugreifen können. Also klicken wir, nachdem wir den Developer Account erstellt haben, auf die Schaltfläche **App erstellen** (siehe Bild 1).

Dies führt uns zu der Seite aus Bild 2, wo wir den Namen der App und eine Beschreibung eingeben. Außerdem wählen wir hier bereits die API aus, die wir in dieser App verwenden wollen – in diesem Fall **Parcel DE Shipping (Post & Parcel Germany)**.

Im nächsten Schritt wählen wir weiter unten aus, ob wir die API zunächst testen oder direkt produktiv einsetzen wollen. Wir entscheiden uns zunächst für den Testbetrieb (siehe Bild 3).

Nach diesem Schritt erstellen wir schließlich mit einem Klick auf **App erstellen** die App (siehe Bild 4).

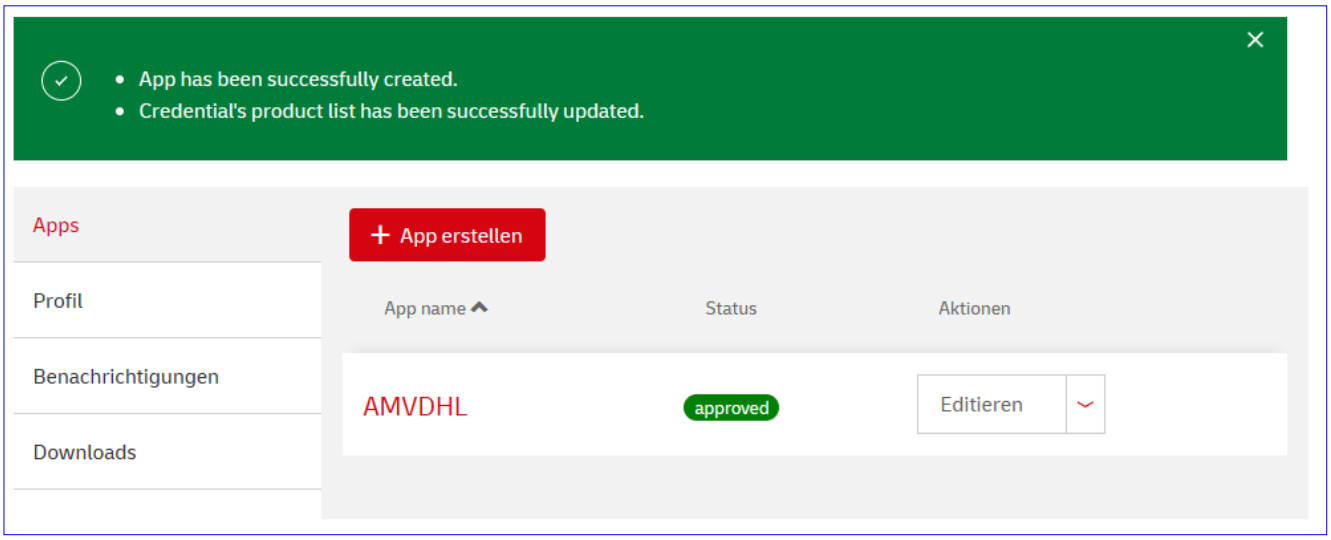

Bild 5: Ein Klick auf den App-Namen zeigt die Details an.

Die App wird nun in der Übersicht angezeigt, wo wir mit einem Klick auf den Namen der App die Detailseite öffnen können (siehe Bild 5). Außerdem können wir die App hier löschen oder Analysedaten anzeigen.

Diese liefert uns unter **Credentials** die gesuchten Informationen – den API Key und das API Secret (siehe Bild 6). Diese können wir mit einem Klick auf den Link **Show key** jeweils einblenden.

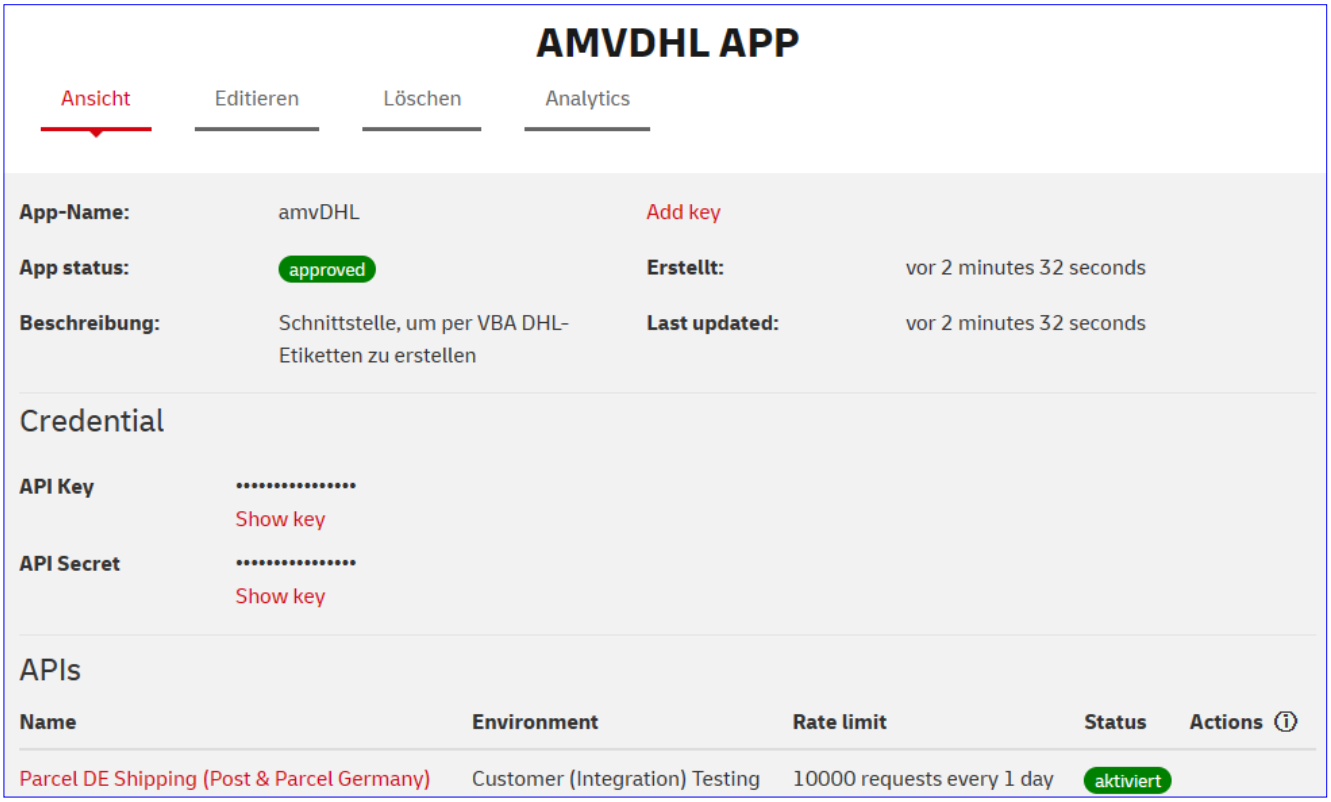

Bild 6: Anzeige von API Key und API Secret

Bevor es weitergeht: Produktiv-Version

Wenn wir schon auf dieser Seite sind, können wir auch gleich die Produktiv-Version der API hinzufügen. Wenn wir die **API Parcel DE Shipping** erneut auswählen, wird unter **Environment** direkt **Production** angeboten. Dieses behalten wir bei und klicken auf die Schaltfläche zum Hinzufügen (siehe Bild 7).

Dies wird jedoch nicht direkt freigegeben, sondern wir erhalten die Meldung, dass sich die API im Status **Pending** befindet.

Als wir die produktive Version hinzugefügt haben, dauerte die Freigabe allerdings nur wenige Minuten. Es empfiehlt sich jedoch, die ersten Tests zunächst einmal mit der Sandbox-Version durchzuführen.

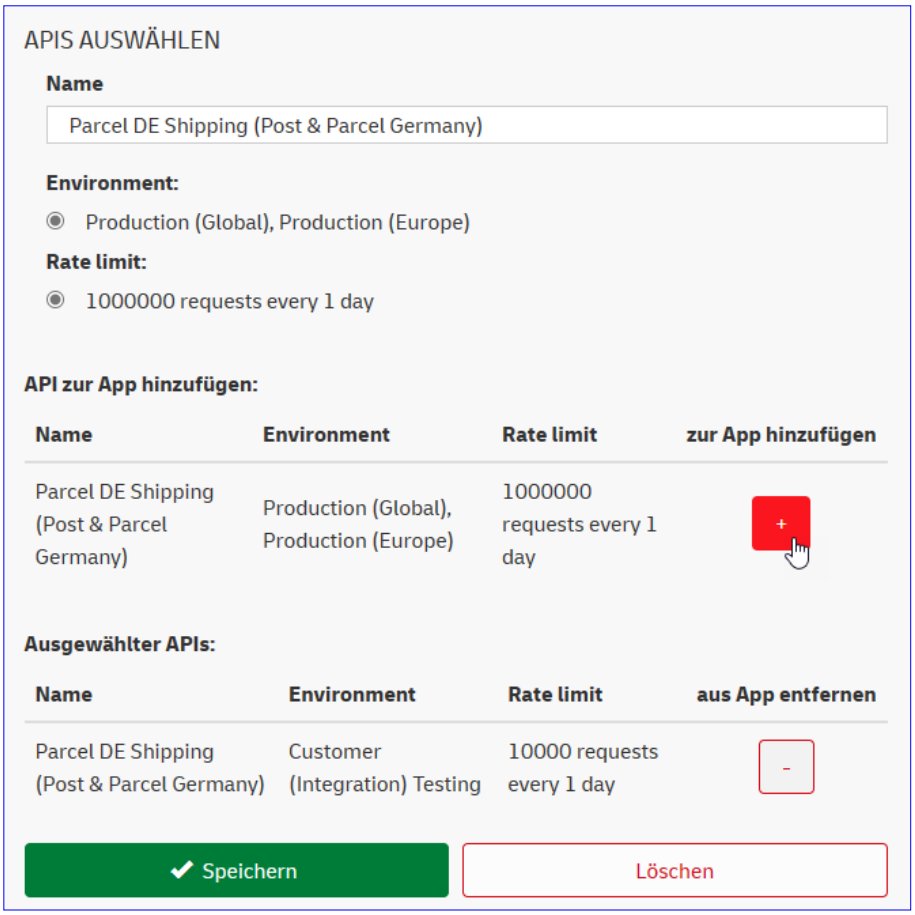

Bild 7: API produktiv machen

Informationen über die API suchen

Bevor wir den Zugriff auf eine REST Api programmieren können, benötigen wir einige Informationen – zum Beispiel die URL, an die wir die Anfrage schicken, die Art der Authentifizierung und die dafür notwendigen Zugangsdaten, die verschiedenen API-Funktionen und deren Syntax.

Dazu wechseln wir von der Seite **https://developer.dhl.com** per Klick auf den Link **Browse Shipping APIs** zu den gesuchten APIs.

Hier finden wir eine Liste aller API, die wir durch Auswahl verschiedener Kriterien auf die Einträge aus Bild 8 begrenzen.

All diese APIs sind entweder mit der deutschen Bezeichnung **Post & Paket Deutschland** oder der englischen Übersetzung **Post & Parcel Germany** markiert, also gehen wir davon aus, dass wir diese über die von uns gewählte, gleichnamige API nutzen können (siehe Bild 8).

Mit dieser API können wir beispielsweise die folgenden Aufgaben erledigen:

- Erstellung von Paketmarken für Privatkunden für den Inlands- und internationalen Versand (Absenderadresse in Deutschland)
- Herunterladen erstellter Versandetiketten

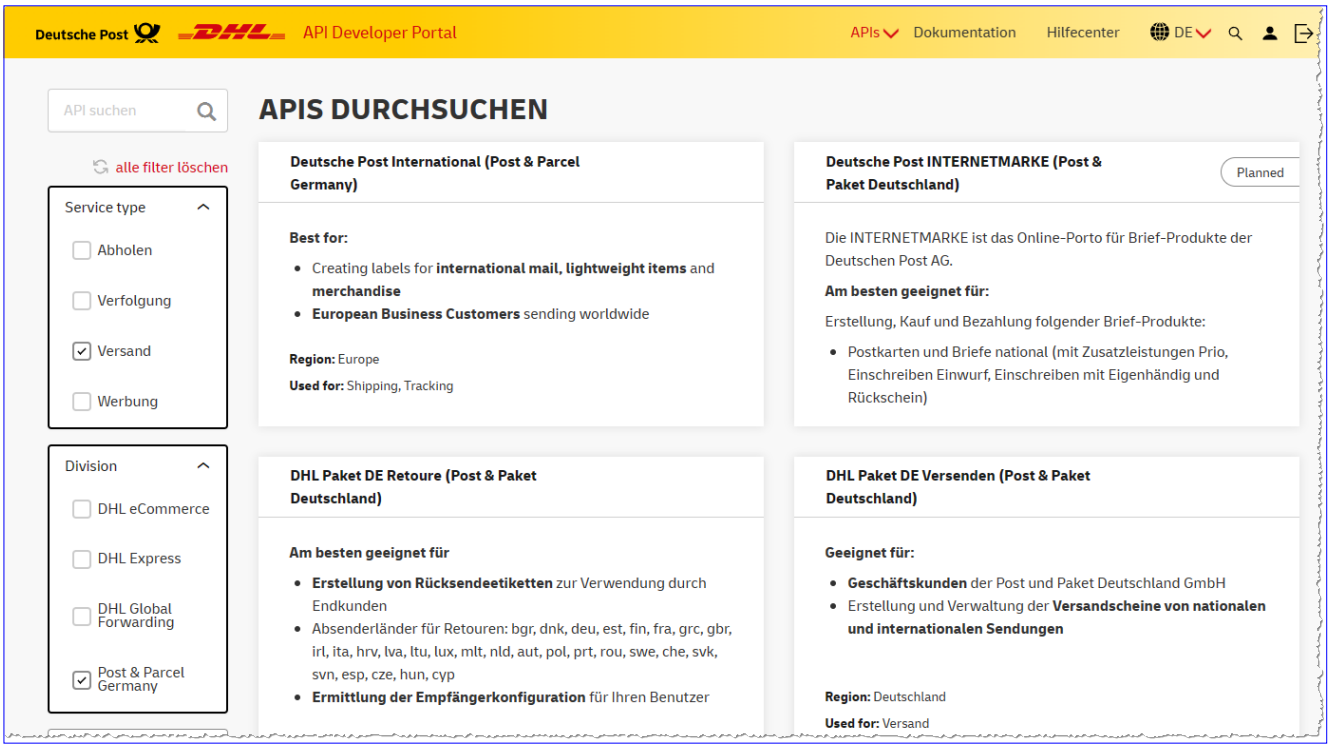

Bild 8: APIs für den Versand

- Herunterladen von mobilen Paketmarken entweder als QR-Code (PNG) oder Wallet-Datei
- Die Zahlungsfunktion wird sicher über die Web-Schnittstelle von DHL während des Checkouts bereitgestellt
- Erstellung von Zoll- und Exportdokumenten

Programmierung der API

Wir starten mit der Programmierung der API mit einem einfachen Beispiel. Dazu wollen wir zunächst die Sandbox nutzen, also einen Testserver.

Einfachste Abfrage: Version ermitteln

Es ist immer schön, eine kleine Test-API zu haben, mit der man erstmal den grundlegenden Zugriff testen kann – bevor es dann ans Eingemachte geht und wir kompliziertere und umfangreichere Daten hin- und herschicken.

Also freuen wir uns, dass es eine API gibt, die einfach Versionsinformationen zurückliefert.

Diese rufen wir mit der folgenden einfachen Funktion auf:

```
Public Function APIVersionAbfragen() As String
    Dim strMethod As String
    Dim strResponse As String
    Dim lngStatus As Long
    strMethod = "GET"
    lngStatus = HTTPRequest(GetURL(False), _
        strMethod, , , False, strResponse)
    Select Case lngStatus
        Case 200
            Debug.Print GetJSONDOM(strResponse, True)
        Case Else
            Debug.Print GetJSONDOM(strResponse, True)
    End Select
End Function
```

Natürlich ist das nicht alles, es hängt noch eine weitere Funktion namens **HTTPRequest** dahinter, die den eigentlichen API-Zugriff durchführt.

Diese sehen wir in Listing 1. Die Funktion erwartet die folgenden Parameter:

- **strURL**: Nimmt die Basis-URL der Rest-API entgegen.
- **strMethod**: Nimmt die Methode entgegen, zum Beispiel **POST**, **GET** oder **PUT**.
- **strContentType**: Nimmt die Art des Contents entgegen, hier **application/json**.
- **strData**: Nimmt eine zu übermittelnde JSON-Datei entgegen.
- **strResponse**: Liefert die Antwort der Rest-API.
- **bolProductive**: Erwartet die Angabe, ob die Sandbox- oder die Produktions-API aufgerufen werden soll.

Um nun die Version der Rest-API zu ermitteln, starten wir mit dem Aufruf der Prozedur **APIVersionAbfragen**. Diese legt die Methode mit **GET** fest und übermittelt die mit **GetURL(False)** geholte URL. Die Funktion **GetURL** erwartet den Wert **True** oder **False**, je nachdem, ob die Produktiv- oder die Sandboxumgebung genutzt werden soll:

```
Public Function GetURL(Optional bolProductive As Boolean)
    If bolProductive = True Then
        GetURL = "https://api-eu.dhl.com/" _
            & "parcel/de/shipping/v2/"
    Else
        GetURL = "https://api-sandbox.dhl.com/" _
            & "parcel/de/shipping/v2/"
    End If
End Function
```

```
Public Function HTTPRequest(strURL As String, Optional strMethod As String = "POST", _
        Optional strContentType As String = "application/json", Optional strData As String, _
        Optional strResponse As String, Optional bolProductive As Boolean) As Integer
    Dim objHTTP As ServerXMLHTTP60
    Set objHTTP = New MSXML2.ServerXMLHTTP60
    With objHTTP
        .Open strMethod, strURL, False
        .setRequestHeader "Accept", "application/json"
        .setRequestHeader "Content-Type", "application/json; charset=UTF-8"
        .setRequestHeader "CharSet", "utf-8"
        .setRequestHeader "dhl-api-key", GetAPIKey
        .setRequestHeader "Authorization", "Basic " & Base64Encode(GetUsername(bolProductive) & ":" _
            & GetPassword(bolProductive))
        If Not Len(strData) = 0 Then
            .setRequestHeader "Body", strData
        End If
        .send strData
        strResponse = .responseText
        HTTPRequest = .status
    End With
End Function
```

Listing 1: Die Funktion **HTTPRequest** führt den eigentlichen API-Aufruf durch.

Außerdem übergeben wir noch den Parameter **strResponse**, mit dem wir die Antwort der Rest-API entgegennehmen. Die Antwort geben wir schließlich sowohl für den Erfolgsfall als auch für den Fall eines Fehlers im Direktbereich des VBA-Editors aus – genau genommen das mit der Funktion **GetJSONDOM** bearbeitete Ergebnis. Wie diese funktioniert, haben wir bereits in den Artikeln **Mit JSON arbeiten** (**www.vbentwickler.de/361**) und **JSON-Dokumente per Objektmodell zusammenstellen** (**www.vbentwickler.de/412**) beschrieben.

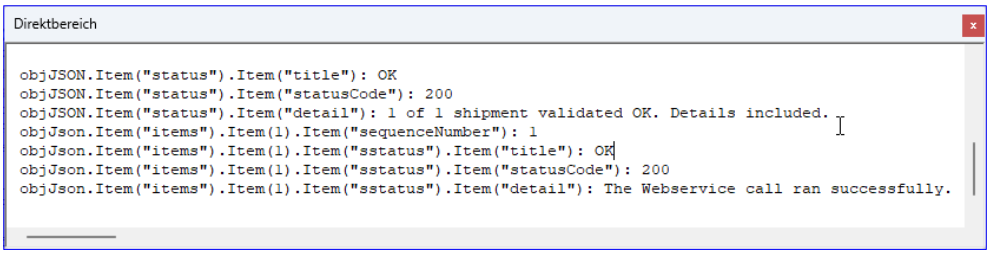

Bild 9: Ergebnis der Versionsabfrage

Das Ergebnis sieht schließlich wie in Bild 9 aus. Hier sehen wir die Ausdrücke, mit denen wir direkt auf die Werte des mit der Funktion **ParseJSON** ermittelten Objekts zugreifen können.

Erstellung von Versandmarken testen

Nachdem der Zugriff auf die Version funktioniert hat, gehen wir einen Schritt weiter und schauen uns direkt die Hauptfunktion an. Diese wiederum testen wir mit der folgenden Prozedur, die wie folgt aussieht und gar nicht so kompliziert aussieht:

```
Public Sub Test_APIOrder()
    Dim strResponse As String
    Dim lngStatus As Long
    Dim strData As String
    Dim strPathPDF As String
    strPathPDF = CurrentProject.Path & "\test.pdf"
    strData = JSONBeispielsendung
    If APIOrder(strData, strPathPDF, strResponse, _
            lngStatus, True, False) = True Then
        MsgBox "Erfolgreich"
        OpenDocument strPathPDF
    Else
        MsgBox "Nicht erfolgreich" & vbCrLf & vbCrLf _
            & strResponse
    End If
End Sub
```

Das liegt daran, dass wir darin eine weitere Funktion namens **JSONBeispielsendung** aufrufen, die uns das komplette JSON-Dokument mit den Daten für die Sendung zusammenstellt – inklusive Absender- und Empfängeradresse und den Details der Sendung wie die Abmessungen und das Gewicht (siehe Listing 2).

Danach ruft die die Funktion **APIOrder** auf, der sie alle notwendigen Variablen übergibt – diese beschreiben wir im Anschluss mit der Funktion **APIOrder**.

Funktion zum Ausführen der API-Funktion Order

Die Funktion **APIOrder** erwartet die folgenden Parameter (siehe Listing 3):

- **strData**: JSON-Dokument mit den Daten für die DHL-Versandmarke
- **strPathPDF**: Pfad, unter dem das erzeugte PDF-Dokument gespeichert werden soll
- **strResponse**: Antwort des API-Aufrufs
- **lngStatus**: Status des Aufrufs
- **bolValidate**: Gibt an, ob der Aufruf nur der Validierung dienen soll. Der Wert **False** sorgt dafür, dass der Aufruf nur getestet wird.

```
Public Function JSONBeispielsendung() As String
    Dim strJSON As String

    strJSON = strJSON & "{" & vbCrLf
    strJSON = strJSON & "  ""shipments"": [" & vbCrLf
    strJSON = strJSON & "    {" & vbCrLf
    strJSON = strJSON & "      ""product"": ""V01PAK""," & vbCrLf
    strJSON = strJSON & "      ""billingNumber"": ""33333333330102""," & vbCrLf
    strJSON = strJSON & "      ""refNo"": ""Order No. 1234""," & vbCrLf
    strJSON = strJSON & "      ""shipper"": {" & vbCrLf
    strJSON = strJSON & "        ""name1"": ""Minhorst und Minhorst GmbH""," & vbCrLf
    strJSON = strJSON & "        ""addressStreet"": ""Borkhofer Str. 17""," & vbCrLf
    strJSON = strJSON & "        ""postalCode"": ""47137""," & vbCrLf
    strJSON = strJSON & "        ""city"": ""Duisburg""," & vbCrLf
    strJSON = strJSON & "        ""country"": ""DEU""," & vbCrLf
    strJSON = strJSON & "        ""email"": ""andre@minhorst.com""," & vbCrLf
    strJSON = strJSON & "        ""phone"": ""+49 123456789""" & vbCrLf
    strJSON = strJSON & "      }," & vbCrLf
    strJSON = strJSON & "      ""consignee"": {" & vbCrLf
    strJSON = strJSON & "        ""name1"": ""Maria Musterfrau""," & vbCrLf
    strJSON = strJSON & "        ""addressStreet"": ""Kurt-Schumacher-Str. 20""," & vbCrLf
    strJSON = strJSON & "        ""additionalAddressInformation1"": ""Apartment 107""," & vbCrLf
    strJSON = strJSON & "        ""postalCode"": ""53113""," & vbCrLf
    strJSON = strJSON & "        ""city"": ""Bonn""," & vbCrLf
    strJSON = strJSON & "        ""country"": ""DEU""," & vbCrLf
    strJSON = strJSON & "        ""email"": ""maria@musterfrau.de""," & vbCrLf
    strJSON = strJSON & "        ""phone"": ""+49 987654321""" & vbCrLf
    strJSON = strJSON & "      }," & vbCrLf
    strJSON = strJSON & "      ""details"": {" & vbCrLf
    strJSON = strJSON & "        ""dim"": {" & vbCrLf
    strJSON = strJSON & "          ""uom"": ""mm""," & vbCrLf
    strJSON = strJSON & "          ""height"": 100," & vbCrLf
    strJSON = strJSON & "          ""length"": 200," & vbCrLf
    strJSON = strJSON & "          ""width"": 150" & vbCrLf
    strJSON = strJSON & "        }," & vbCrLf
    strJSON = strJSON & "        ""weight"": {" & vbCrLf
    strJSON = strJSON & "          ""uom"": ""g""," & vbCrLf
    strJSON = strJSON & "          ""value"": 500" & vbCrLf
    strJSON = strJSON & "        }" & vbCrLf
    strJSON = strJSON & "      }" & vbCrLf
    strJSON = strJSON & "    }" & vbCrLf
    strJSON = strJSON & "  ]" & vbCrLf
    strJSON = strJSON & "}" & vbCrLf

    JSONBeispielsendung = strJSON
End Function
```

Listing 2: Diese Funktion stellt das JSON-Dokument mit den Paketdaten zusammen.