AUSGABE 03/2025

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

Microsoft 365- E-Mails per Access-Formular senden

Im Artikel »Microsoft 365 E-Mails mit Make per Klasse senden« (www.vbentwickler.de/465) und den dort referenzierten Artikeln haben wir am Beispiel von Microsoft 365 und Make.com gezeigt, wie wir per VBA E-Mails versenden können. Dazu haben wir eine Klasse programmiert, mit der wir leicht die notwendigen Daten für die E-Mail übergeben können. Im vorliegenden Artikel stellen wir nun ein Formular vor, mit dem wir leicht die Daten für eine solche E-Mail eingeben und diese schließlich absenden können. Die dazu eingegebenen Daten speichern wir in entsprechenden Tabellen, sodass wir diese später einsehen können.

Wenn man eine kleine Access-Anwendung zum Versenden von E-Mails programmieren möchte, denkt man zunächst, man würde mit einigen wenigen Tabellen auskommen. Eine zum Speichern von E-Mails und vielleicht noch eine für die Kontakte. Tatsächlich haben wir auch so angefangen, allerdings wuchs das Datenmodell dann schnell an.

Datenmodell der Beispiellösung

Wir schauen uns das Datenmodell zunächst in der Übersicht an und gehen später ins Detail (siehe Bild 1). Dass wir eine Haupttabelle zum Verwalten der einzelnen Informationen der E-Mail benötigen, ist logisch (**tblEMails**). Hinzu kommen aber erst einmal zwei Tabellen mit den Werten für den Content-Type (**tblContentTypes**) und für die Priorität (**tblImportanceLevels**). Eine weitere Tabelle benötigen wir für die Adressen, die wir als ReplyTo-Adressen angeben (**tblReplyToContacts**).

Außerdem kann jede E-Mail einen oder mehrere To-Adressaten aufweisen und keinen, einen oder mehrere CC- oder BCC-Adressaten. Diese wollen wir komfortabel auswählen können. Also schreiben wir diese in die Tabelle **tblEMailContacts**.

Um diese den verschiedenen Empfängertypen der E-Mail zuordnen zu können, verwenden wir die Tabelle

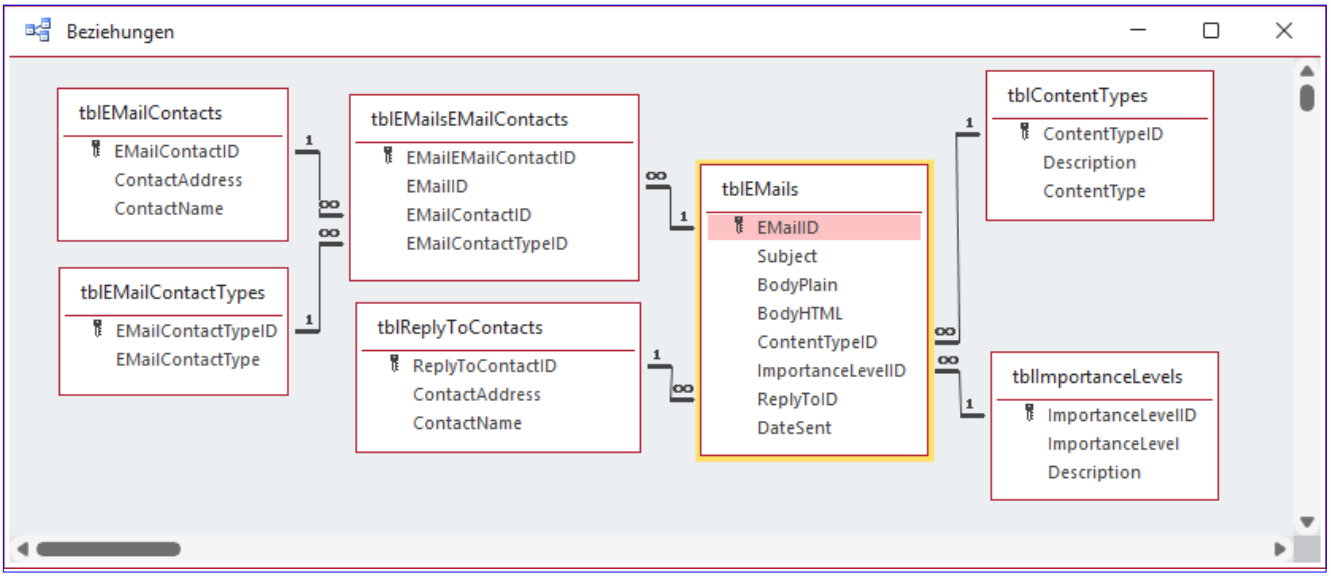

Bild 1: Datenmodell der Beispiellösung

tblEMailsEMailContacts. Diese dient als m:n-Verknüpfungstabelle zwischen den Tabellen **tblEMailContacts** und **tblEMails**.

Um festzulegen, ob es sich um einen To-, CC- oder BCC-Empfänger handelt, enthält diese Tabelle außerdem noch ein Auswahlfeld, dessen Werte aus der Tabelle **tblEMailContactTypes** stammen.

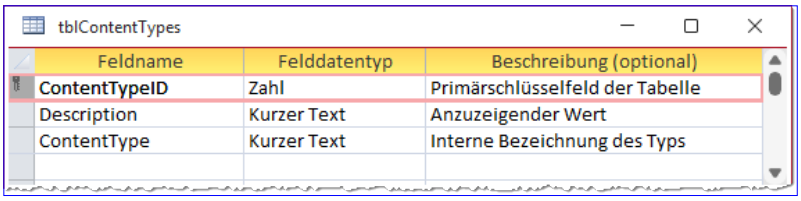

Bild 2: Entwurf der Tabelle **tblContentTypes**

Die Tabelle tblContentTypes

Wir starten mit den Lookuptabellen für die Tabelle **tblEMails**. Die erste ist die Tabelle **tblContentTypes**. Sie enthält neben dem Primärschlüsselfeld noch ein Textfeld für die anzuzeigende Bezeichnung und eines für interne Zwecke (siehe Bild 2).

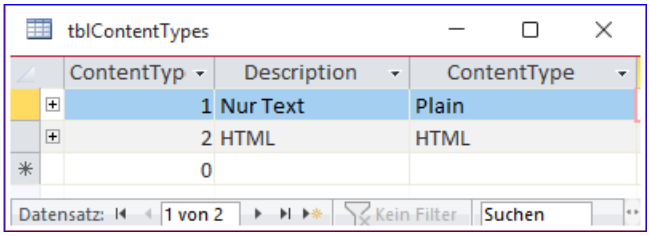

Bild 3: Werte der Tabelle **tblContentTypes**

Dieser Tabelle fügen wir die Werte aus Bild 3 hinzu. Mehr benötigen wir hier nicht.

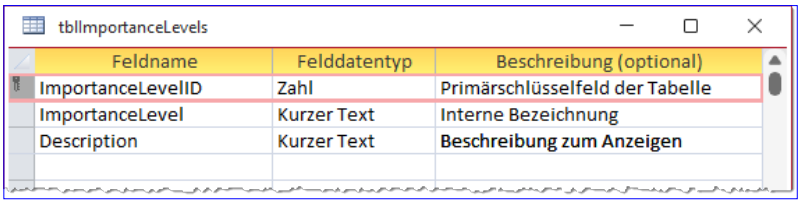

Bild 4: Entwurf der Tabelle **tblImportanceLevels**

Die Tabelle tblImportanceLevels

Die Tabelle **tblImportanceLevels** ist genauso aufgebaut wie die Tabelle **tblContentTypes**. Sie enthält ebenfalls ein Feld für interne Zwecke (**ImportanceLevel**) und eins für die Anzeige (**Description**) – siehe Bild 4.

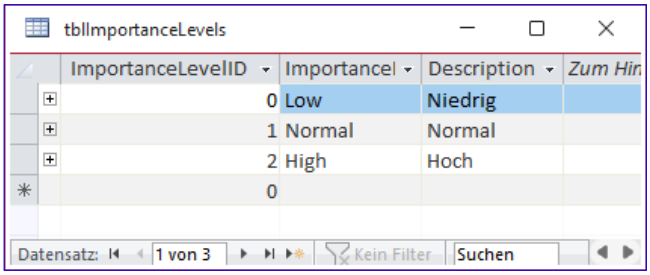

Bild 5: Werte der Tabelle **tblImportanceLevels**

Hier finden wir mit **Low**, **Normal** und **High** beziehungsweise **Niedrig**, **Normal** und **Hoch** drei verschiedene Werte vor (siehe Bild 5).

Die Tabelle tblReplyToContacts

In der Tabelle **tblReplyToContacts** speichern wir die Kontakte, die wir für das Feld **ReplyTo** verwenden wollen, also als alternative Antwortadresse zu der eigentlichen Absenderadresse. Antwort der Empfänger auf eine E-Mail mit einer Antwortadresse, wird diese entsprechend adressiert.

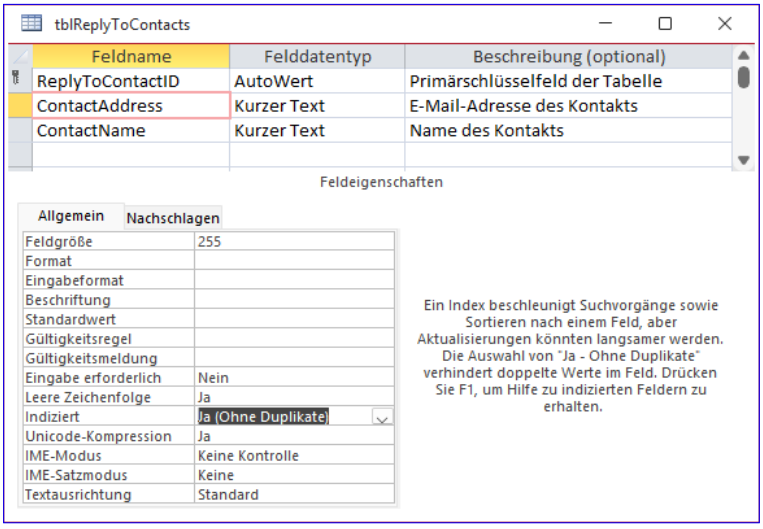

Bild 6: Entwurf der Tabelle **tblReplyContacts**

Die Tabelle aus speichert die E-Mailadresse und den Namen des Kontaktes in zwei getrennten Feldern (siehe Bild 6).

Für das Feld **ContactAddress** haben wir außerdem einen eindeutigen Index definiert. Auf diese Weise kann ein Kontakt nicht doppelt angelegt werden, was Redundanzen vermeidet.

Die Haupttabelle tblEMails

Damit haben wir den Grundstein für die Beschreibung der Tabelle **tblEMails** gelegt (siehe Bild 7).

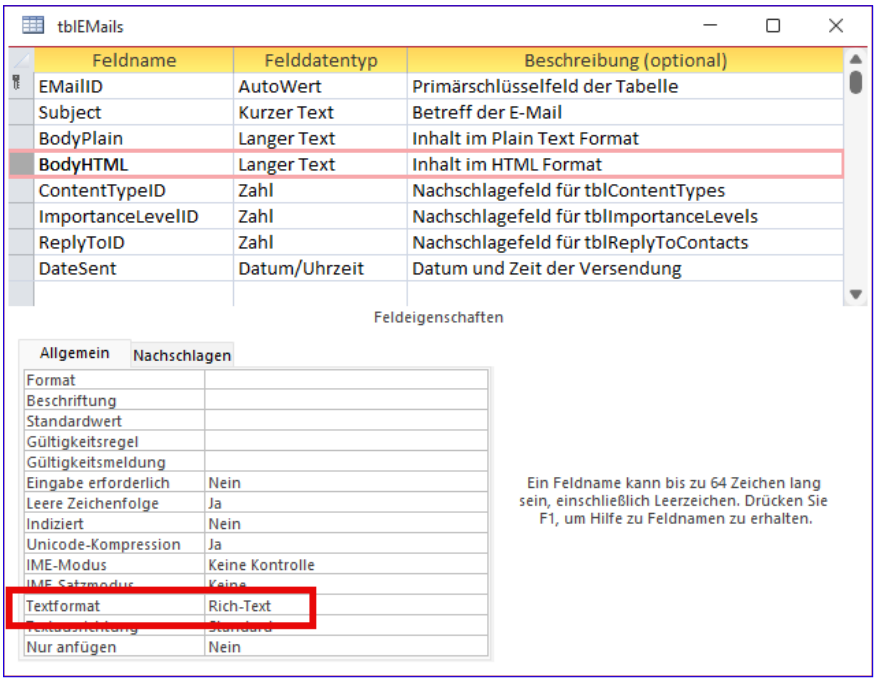

Bild 7: Entwurf der Tabelle **tblEMails**

Sie enthält die üblichen Daten für eine E-Mail, also zunächst einmal den Betreff (**Subject**).

Für den Inhalt gibt es zwei Felder, die wir je nach der Einstellung für das Feld **ContentTypeID** beim Versenden heranziehen.

Das Feld **BodyPlain** nimmt den Inhalt für reine Text-E-Mails auf, das Feld **BodyHTML** den für E-Mails mit Markierungen. Damit wir diese Markierungen später im Formular hinzufügen können, haben wir das Feld mit dem Wert **Rich-Text** für die Eigenschaft **Textformat** ausgestattet.

Die übrigen Felder sind Nachschlagefelder für die Daten aus den Tabellen **tblContentTypes**, **tblImportanceLevels** und **tblReplyToContacts**. Diese haben wir jeweils mit den Nachschlage-Assistenten eingerichtet und Beziehungen mit referenzieller Integrität definiert.

Nach dem Anlegen eines Datensatzes sieht die Tabelle wie in Bild 8 aus.

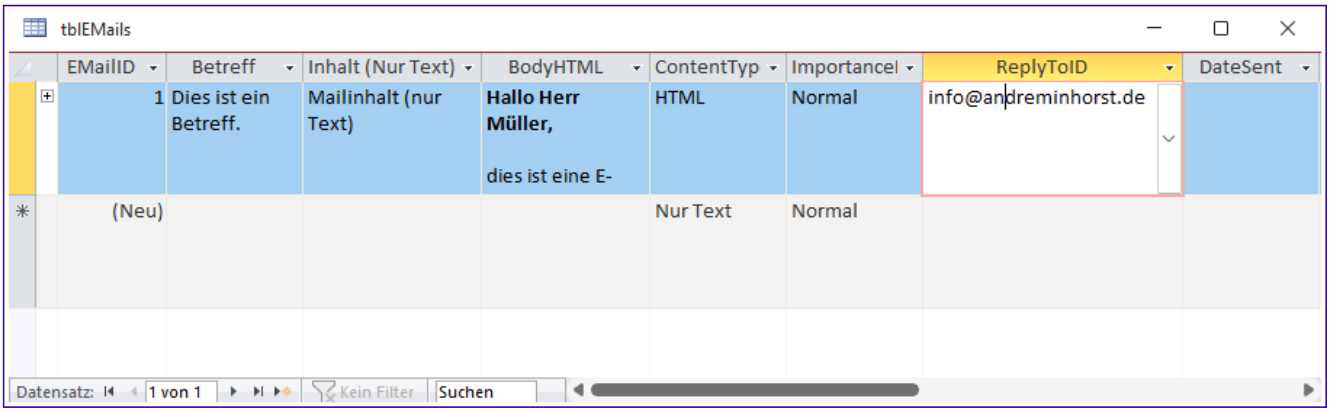

Bild 8: Beispieldatensatz in der Tabelle **tblEMails**

Die Tabelle tblEMailContacts

Die Tabelle **tblEMailContacts** ist identisch aufgebaut wie **tblReplyToContacts**. Auch für diese Tabelle haben wir für das Feld **ContactAddress** einen eindeutigen Index definiert (siehe Bild 9).

Somit können wir jede E-Mail-Adresse nur einmal in dieser Tabelle speichern. Diese sieht mit einigen Daten wie in Bild 10 aus.

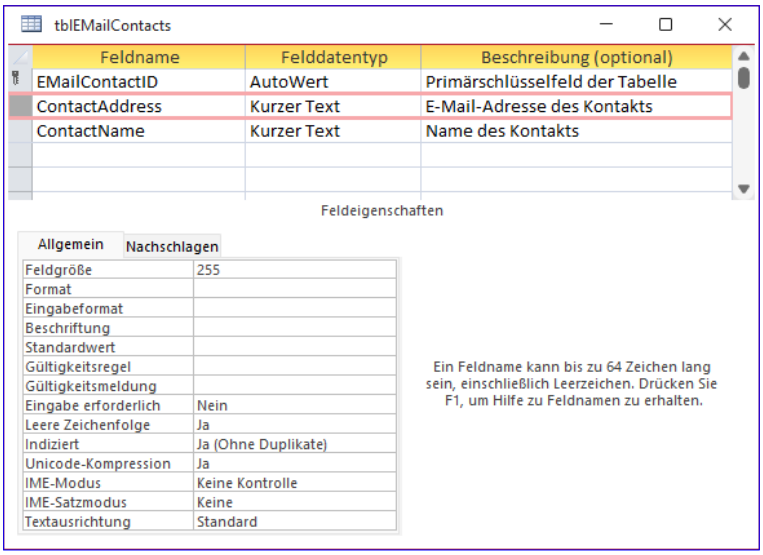

Bild 9: Entwurf der Tabelle **tblEMailContacts**

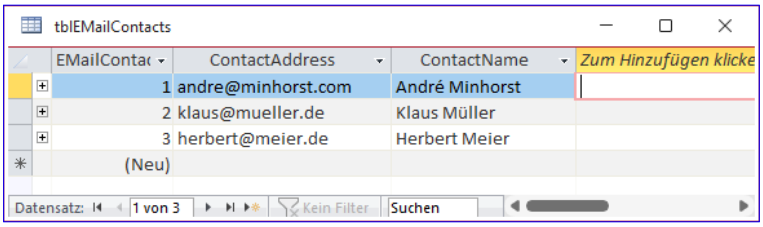

Bild 10: Werte der Tabelle **tblEMailContacts**

Die Tabelle tblEMailContactTypes

Bevor wir uns der Verknüpfungstabelle zur Zuordnung von Kontakten zu E-Mails zuwenden, legen wir noch die Tabelle **tblEMailContactTypes** an. Wir wollen später für jede E-Mail nach den unterschiedlichen Empfängertypen filtern können, also nach **To**, **CC** und **BCC**.

Diese Typen holen wir wieder aus einer Lookuptabelle, die wir wie in Bild 11 entwerfen.

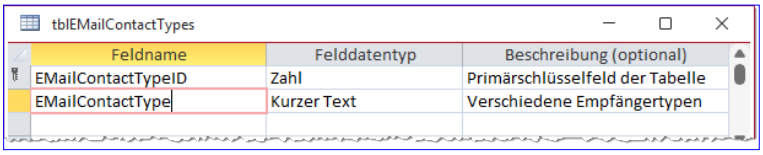

Bild 11: Entwurf der Tabelle **tblEMailContactTypes**

Diese enthält die drei Werte **To-Empfänger**, **CC-Empfänger** und **BCC-Empfänger** (siehe Bild 12).

Die Tabelle tblEMailsEMailContacts

Damit kommen wir zur letzten Tabelle, welche die Kontakte zu den E-Mails zuordnet und gleichzeitig festlegen, ob es sich

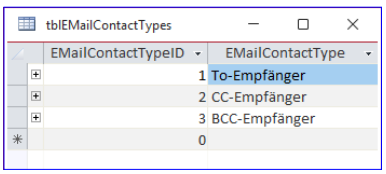

Bild 12: Werte der Tabelle **tblEMailContactTypes**

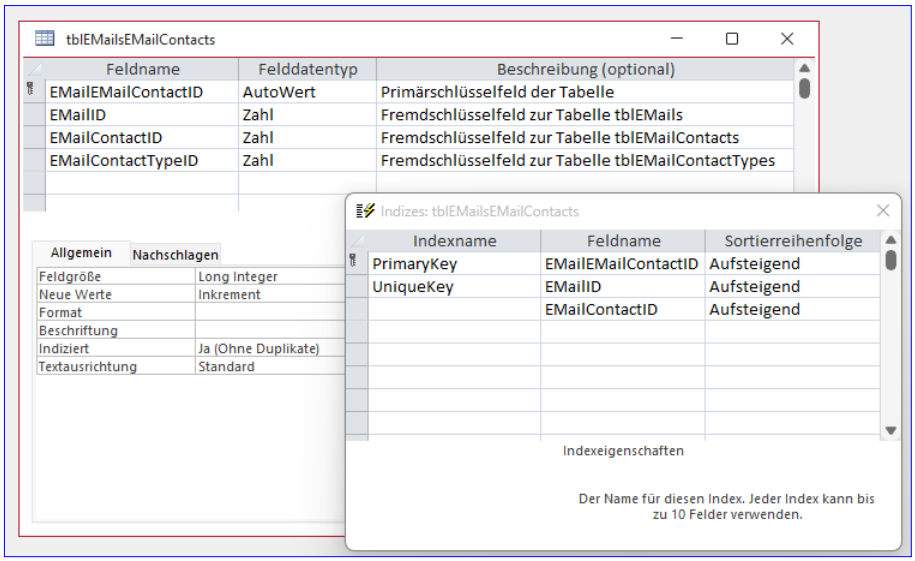

Bild 13: Entwurf der Tabelle **tblEMailsEMailContacts**

um einen **To**-, **CC**- oder **BCC**-Empfänger handelt.

Diese Tabelle enthält neben dem Primärschlüsselfeld jeweils ein Fremdschlüsselfeld mit Beziehung zu den Tabellen **tblEMails**, **tblEMailContacts** und **tblEMailContactTypes** (siehe Bild 13).

Für diese Tabelle haben wir außerdem einen zusammengesetzten, eindeutigen Index hinzugefügt, der sicherstellt, dass jeder Kontakt jeder E-Mail nur einmal hinzugefügt werden kann. Dieser umfasst daher die Felder **EMailID** und **EMailContactID**.

Wenn wir eine E-Mail angelegt und dieser Kontakte für die drei Kategorien von Empfängern hinzugefügt haben, liefert die Datenblattansicht der Tabelle die Werte aus Bild 14.

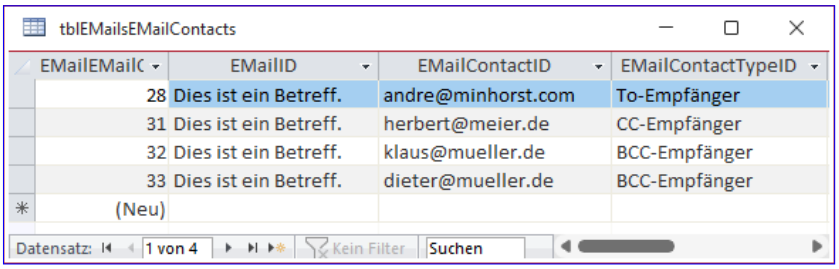

Bild 14: Werte der Tabelle **tblEMailsEMailContacts**

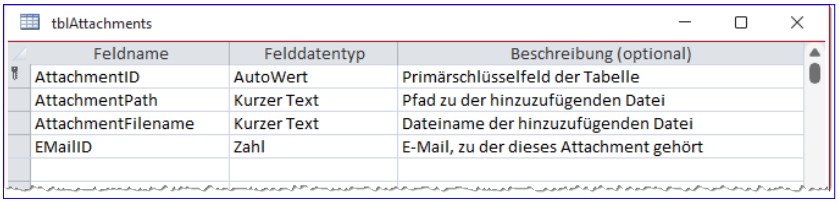

Bild 15: Entwurf der Tabelle **tblAttachments**

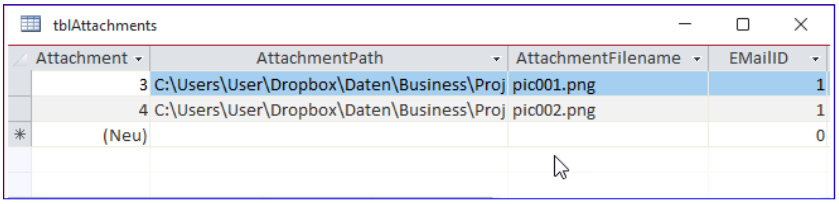

Bild 16: Werte der Tabelle **tblAttachments**

Die Tabelle tblAttachments

Diese Tabelle speichert die Informationen zu den Dateien, die als Anlage zu einer E-Mail hinzugefügt werden sollen (siehe Bild 15). Sie enthält ein Feld für den vollständigen Pfad sowie eines für den Dateinamen. Dieses Feld dient allein zur Anzeige im Listenfeld der Lösung. Den Pfad verwenden wir schließlich zum Hinzufügen der Datei zur E-Mail.

In Bild 16 sehen wir auch noch einige Beispielwerte für diese Tabelle in der Datenblattansicht.

Benutzeroberfläche zum Eingeben der E-Mail-Daten

Nun programmieren wir die Formulare, die wir zum Eingeben der Daten einer E-Mail und zum Absenden benötigen. Dazu nutzen wir zwei Formulare.

Wir schauen uns zuerst die Funktionen an, anschließend zeigen wir, wie diese programmiert werden.

Das Hauptformular heißt **frmEMails** und ist in Bild 17 zu sehen. Im obersten Textfeld können wir den Betreff eingeben. Darunter finden wir die Auswahl, ob wir eine reine Text-E-Mail schreiben wollen oder eine HTML-E-Mail. Je nachdem, welcher Eintrag gewählt ist, wird entweder das Feld **Inhalt (Nur Text)** oder **Inhalt (HTML)** aktiviert und das jeweils andere deaktiviert.

Im Feld **Inhalt (HTML)** können wir den aktuell markierten Text mit individuellen Formatierungen versehen.

Darunter stellen wir die Wichtigkeit dieser E-Mail ein und wählen dazu einen der Werte **Niedrig**, **Normal** oder **Hoch** aus.

Anschließend folgt der Kontakt, den wir für die Eigenschaft **ReplyTo** hinterlegen wollen. Diesen können wir entweder aus den bestehenden Kontakten der Tabelle **tblReplyToContacts** auswählen oder wir klicken auf die **Bearbeiten**-Schaltfläche, um das Formular **frmEditEMailContacts** zu öffnen.

Dieses sieht wie in Bild 18 aus und zeigt alle E-Mails aus der Tabelle **tblReplyToContacts** an. Wir können hier eine Adresse auswählen und auf die Schaltfläche rechts neben der Liste klicken, um diese hinzuzufügen.

Oder wir geben unten den Namen und die E-Mail-Adresse eines Kontakts ein, den wir der Liste hinzufügen wollen.

Beim Doppelklicken auf einen der Einträge oder beim Betätigen der Schaltfläche rechts neben der Liste wird der aktuelle Eintrag in das Kombinationsfeld übernommen und das Formular geschlossen.

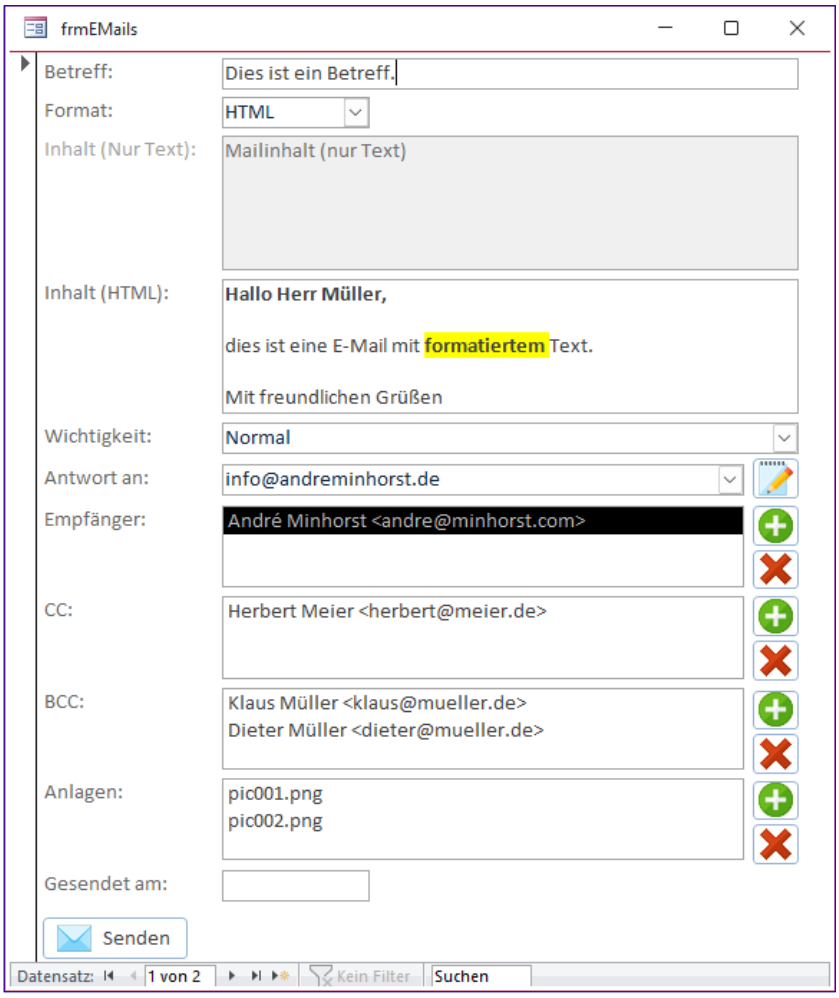

Bild 17: Formular zur Eingabe der E-Mail-Daten

Empfänger auswählen

Darunter befinden sich drei Listenfelder, mit denen wir die Empfänger der drei Kategorien To, CC und BCC verwalten können.

Ein Klick auf die Plus-Schaltfläche öffnet den gleichen Dialog wie bei der **ReplyTo**-Adresse, allerdings werden hier nun die Einträge der Tabelle **tblEMailContacts** angezeigt. Hier können wir nun einen oder mehrere Einträge auswählen und mit einem Klick auf die Schaltfläche rechts zum Listenfeld hinzufügen.

Wenn mindestens ein Empfänger vorhanden ist und wir diesen entfernen wollen, können wir diesen markieren und mit einem Klick auf die entsprechende Löschen-Schaltfläche löschen.

Das Gleiche gilt für die beiden weiteren Listenfelder mit den CC- und den BCC-Empfängern.

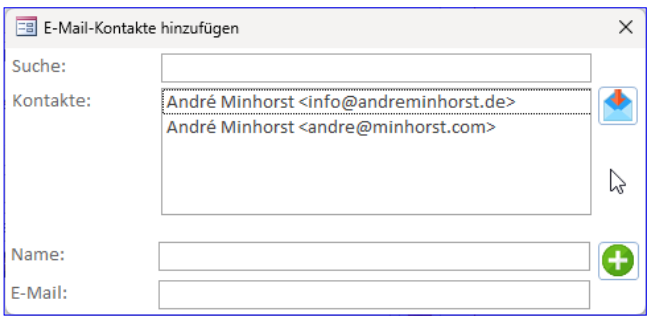

Bild 18: Das Formular **frmEditEMailContacts**

Anlagen hinzufügen

Auch für die Anlagen einer E-Mail gibt es ein Listenfeld. Durch Anklicken der Plus-Schaltfläche öffnen wir einen Dateiauswahl-Dialog, mit dem wir ein oder mehrere Dateien auswählen können.

Diese Dateien werden anschließend im Listenfeld angezeigt.

Um eine Anlage zu entfernen, markieren wir die gewünschte Anlage und betätigen die Löschen-Schaltfläche.

Schließlich lösen wir mit der **Senden**-Schaltfläche die bereits im Artikel **Microsoft 365 E-Mails mit Make per Klasse senden** (**www.vbentwickler.de/465**) vorgestellte Prozedur zum Versenden einer E-Mail aus.

Programmieren des Formulars frmEMails

Das Formular **frmEMails** legen wir in der Entwurfsansicht wie in Bild 19 an.

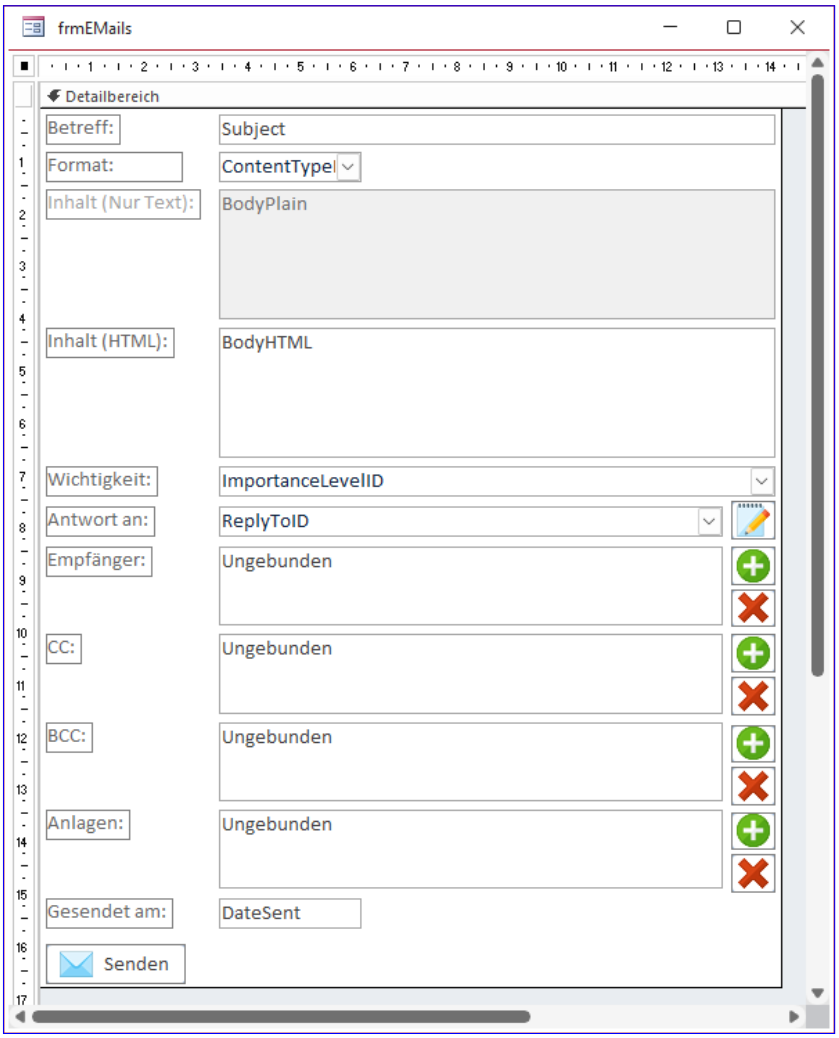

Bild 19: Das Formular **frmEMails** in der Entwurfsansicht

Wir gehen nun Schritt für Schritt die für den Einsatz notwendigen Prozeduren durch.

Laden des Formulars und Füllen der Listenfelder

Den Start macht die Ereignisprozedur **Beim Anzeigen**. Sie löst diese Prozedur aus:

```
Private Sub Form_Current()
    Call RequeryBodyfields
    Call UpdateListboxes
    Me.Subject.SetFocus
End Sub
```

Die erste aufgerufene Prozedur **RequeryBodyFields** aus Listing 2 prüft den Wert des Feldes **ContentType**.

Hat dieser den Wert **1**, wird das Textfeld **BodyPlain** für reine Text-E-Mails aktiviert und das Feld **BodyHTML** deaktiviert.

Außerdem wird **VerticalAnchor** für **BodyPlain** auf **acVerticalAnchorBoth** eingestellt, was bedeutet, dass dieses Feld nun beim Vergrößern des Formulars in der Höhe ebenfalls vergrößert wird. Damit dies beim Feld **BodyPlain** nicht ebenfalls geschieht, stellen wir hier **VerticalAnchor** auf **acVerticalAnchorBottom** ein.

```
Private Sub RequeryBodyfields()
    If Me.ContentTypeID = 1 Then
        Me.BodyPlain.Enabled = True
        Me.BodyHTML.Enabled = False
        Me.BodyPlain.VerticalAnchor = acVerticalAnchorBoth
        Me.BodyHTML.VerticalAnchor = acVerticalAnchorBottom
    Else
        Me.BodyPlain.Enabled = False
        Me.BodyHTML.Enabled = True
        Me.BodyPlain.VerticalAnchor = acVerticalAnchorTop
        Me.BodyHTML.VerticalAnchor = acVerticalAnchorBoth
    End If
End Sub
```

Listing 2: Diese Prozedur aktualisiert die Steuerelemente zum Eingeben des Inhalts der E-Mail.

Ist **ContentType** hingegen auf **2** eingestellt, wird **BodyPlain** deaktiviert und **BodyHTML** aktiviert und **BodyHTML** passt sich nun in der Höhe an das Formular an.

Listenfelder aktualisieren

Die zweite in **Form_Current** aufgerufene Funktion **UpdateListboxes** sehen wir in Listing 1.

Sie stellt in **strSQL** eine SQL-Anweisung zusammen, die alle E-Mail-Kontakte liefert, die zu der aktuell angezeigten E-Mail gehören und hängt hinten noch ein unvollständiges Kriterium für **EMailContactTypeID** an.

Diesem fehlt noch der Vergleichswert, den wir beim Zuweisen von **strSQL** zur jeweiligen Datensatzherkunft der Listenfelder **lstTo**, **lstCC** und **lstBCC** noch hinzufügen.

Außerdem weisen wir hier die Datensatzherkunft für das Listenfeld **lstAttachments** zu, das alle Anlagen aus der Tabelle **tblAttachments** liefert, die zur aktuellen E-Mail gehören.

Fokus auf das Betreff-Feld setzen

Die dritte Anweisung in **Form_Current** stellt den Fokus auf das Feld **Betreff** ein, damit der Benutzer hier als Erstes einen Text eintragen kann.

```
Private Sub UpdateListboxes()
    Dim strSQL As String
    strSQL = "SELECT tblEMailsEMailContacts.EMailEMailContactID, [ContactName] & ' <' & [ContactAddress] _
        & '>' AS Contact FROM tblEMailContacts INNER JOIN (tblEMails INNER JOIN tblEMailsEMailContacts " _
        &  "ON tblEMails.EMailID = tblEMailsEMailContacts.EMailID) ON tblEMailContacts.EMailContactID = " _
        & "tblEMailsEMailContacts.EMailContactID WHERE tblEMailsEMailContacts.EMailID = " & Me.EMailID _
        & " AND tblEMailsEMailContacts.EMailContactTypeID = "
    Me.lstTo.RowSource = strSQL & " 1"
    Me.lstCC.RowSource = strSQL & " 2"
    Me.lstBCC.RowSource = strSQL & " 3"
    Me.lstAttachments.RowSource = "SELECT AttachmentID, AttachmentFilename FROM tblAttachments WHERE EMailID = " _
        & Me.EMailID
End Sub
```

Listing 1: Diese Prozedur aktualisiert die Listenfelder.

ContentType aktualisieren

Wenn der Benutzer einen anderen Eintrag im Feld **ContentType** auswählt, sollen die beiden Felder **BodyPlain** und **BodyHTML** angepasst werden.

Deshalb rufen wir in der Prozedur, die durch das Ereignis **Nach Aktualisierung** aufgerufen wird, erneut die Routine **RequeryBodyFields** auf:

```
Private Sub ContentTypeID_AfterUpdate()
    Call RequeryBodyfields
End Sub
```

Es kann sein, dass der Benutzer das Kombinationsfeld leert. Dies prüft die Ereignisprozedur **Vor Aktualisierung** wie folgt und gibt gegebenenfalls eine Meldung aus, dass ein Contenttyp eingestellt werden muss:

```
Private Sub ContentTypeID_BeforeUpdate(Cancel As Integer)
    If IsNull(Me.ContentTypeID) Then
        MsgBox "Bitte einen Contenttype einstellen.", _
            vbOKOnly + vbExclamation, "Contenttype fehlt."
        Me.ContentTypeID.Undo
        Cancel = True
    End If
End Sub
```

Verhindern, dass keine Wichtigkeit angegeben wird

Auf ähnliche Weise sorgt die folgende Prozedur, die durch das Ereignis **Vor Aktualisierung** des Feldes **ImportanceLevelID** ausgelöst wird, dafür, dass dieses Feld nicht geleert wird:

```
Private Sub ImportanceLevelID_BeforeUpdate(Cancel _
        As Integer)
    If IsNull(Me.ImportanceLevelID) Then
        MsgBox "Bitte eine Wichtigkeit einstellen.", _
            vbOKOnly + vbExclamation, "Wichtigkeit fehlt."
        Me.ImportanceLevelID.Undo
        Cancel = True
    End If
End Sub
```

ReplyTo-Kontakt hinzufügen

Wenn der Benutzer die Schaltfläche zum Bearbeiten der ReplyTo-Kontakte **cmdEditReplyTo** anklickt, löst er die folgende Prozedur aus:

```
Private Sub cmdEditReplyTo_Click()
    If CheckNewRecord = False Then Exit Sub
    DoCmd.OpenForm "frmEditEMailContacts", _
        WindowMode:=acDialog, OpenArgs:="4|" & Me.EMailID
    Me.ReplyToID.Requery
End Sub
```

Diese ruft zuerst die Funktion **CheckNewRecord** auf. War der Aufruf erfolgreich öffnet sie das Formular **frmEditMailContacts** und übergibt als Öffnungsargument eine Zeichenkette bestehend aus der Zahl **4**, einem Pipe-Zeichen (|) und der aktuellen **EMailID**.

Dort werden die Kontakte bearbeitet und nach dem Schließen des Formulars wird das Kombinationsfeld **ReplyToID** aktualisiert. Wie das im Detail aussieht, sehen wir weiter unten bei der Beschreibung des Formulars **frmEditEMailContacts**.

Prüfen, ob ein neuer Datensatz vorliegt

Wenn wir einen neuen Kontakt zu der aktuellen E-Mail hinzufügen wollen, öffnen wir dazu das Formular **frmEditEMailContacts**. Hier würden wir im Normalfall mindestens einen Kontakt auswählen, der dann über einen neuen Datensatz in der Tabelle **tblEMails-EMailContacts** der aktuellen E-Mail zugewiesen wird. Das ist jedoch nur möglich, wenn im Hauptformular bereits ein gespeicherter Datensatz mit einer E-Mail vorhanden ist.

Deshalb rufen wir gleich zu Beginn die Funktion **CheckNewRecord** auf (siehe Listing 3). Diese prüft als Erstes, ob es sich bei dem aktuellen Datensatz im For-

```
Private Function CheckNewRecord() As Boolean
    If Me.NewRecord = True Then
        If Me.Dirty = True Then
            Me.Dirty = False
            CheckNewRecord = True
        Else
            MsgBox "Bitte erst einen Betreff angeben.", vbOKOnly + vbExclamation, "Betreff fehlt"
            Me.Subject.SetFocus
            CheckNewRecord = False
        End If
    Else
        Me.Dirty = False
        CheckNewRecord = True
    End If
End Function
```

Listing 3: Prüfen, ob ein speicherbarer Datensatz vorliegt

mular **frmEMails** um einen neuen, leeren Datensatz handelt.

Falls ja, folgt eine zweite Prüfung, ob dieser bereits in Bearbeitung ist (**Me.Dirty = True**). Ist auch das der Fall, wurde der Datensatz also bereits bearbeitet und wir können diesen speichern (mit **Me.Dirty = False**).

In diesem Fall soll **CheckNewRecord** den Wert **True** zurückliefern.

Ist der Datensatz noch nicht bearbeitet, erscheint die Meldung, dass zunächst ein Betreff angegeben werden muss. Außerdem wird der Fokus auf das Textfeld **Subject** gesetzt und **CheckNewRecord** liefert **False** zurück.

Es kann auch sein, dass wir es mit einem bereits gespeicherten Datensatz zu tun haben – dann speichern wir den Datensatz mit **Me.Dirty = False** und geben **CheckNewRecord = True** zurück.

To-Empfänger hinzufügen

Die **Plus**-Schaltfläche neben dem Listenfeld **lstToRecipients** ruft zunächst ebenfalls die **CheckNewRecord** auf.

Liefert diese den Wert **True** zurück, öffnet die Prozedur das Formular **frmEditEMailContacts** und übergibt diesmal ein Öffnungsargument, das als ersten Teil den Wert **1** und als zweiten die aktuelle **EMailID** enthält. Was in diesem Formular geschieht, beschreiben wir ebenfalls weiter unten. Nach dem Hinzufügen der gewünschten Kontakte über dieses Formular läuft die aufrufende Prozedur weiter und aktualisiert die Anzeige im Listenfeld.

```
Private Sub cmdAddTo_Click()
    If CheckNewRecord = False Then Exit Sub
    DoCmd.OpenForm "frmEditEMailContacts", WindowMode:=acDialog, OpenArgs:="1|" & Me.EMailID
    Me.lstTo.Requery
End Sub
```

CC-Empfänger hinzufügen

Das Hinzufügen von CC-Empfängern funktioniert auf die gleiche Weise. Diesmal übergeben wir beim Aufrufen des Formulars **frmEditEMailContacts** allerdings als ersten Wert des Öffnungsarguments den Wert **2** (für **CC**).

```
Private Sub cmdAddCC_Click()
    If CheckNewRecord = False Then Exit Sub
```

```
    DoCmd.OpenForm "frmEditEMailContacts", _
        WindowMode:=acDialog, OpenArgs:="2|" & Me.EMailID
    Me.lstCC.Requery
End Sub
```

BCC-Empfänger hinzufügen

Schließlich fehlt noch das Hinzufügen von BCC-Empfängern, für die wir das Formular mit der folgenden Prozedur öffnen:

```
Private Sub cmdAddBCC_Click()
    If CheckNewRecord = False Then Exit Sub
    DoCmd.OpenForm "frmEditEMailContacts", _
        WindowMode:=acDialog, OpenArgs:="3|" & Me.EMailID
    Me.lstBCC.Requery
End Sub
```

Anlagen hinzufügen

Etwas anders sieht die Prozedur aus, mit der wir Anlagen zur Tabelle **tblAttachments** hinzufügen und damit zum Listenfeld **lstAttachments**.

Diese wird durch die Schaltfläche **cmdAddAttachment** ausgelöst und ist in Listing 4 abgebildet.

In dieser Prozedur nutzen wir die Dateidialoge von Office. Dazu müssen wir dem VBA-Projekt noch einen Verweis auf die Bibliothek **Microsoft Office 16.0 Object Library** hinzufügen.

Damit der Benutzer die hinzuzufügende Datei auswählen kann, deklarieren wir eine Variable des Typs **Office.FileDialog**.

```
Private Sub cmdAddAttachment_Click()
    Dim db As DAO.Database
    Dim objFileDialog As Office.FileDialog
    Dim strTemp As String
    Dim strFilename As String
    Dim varPath As Variant
    If CheckNewRecord = False Then Exit Sub
    Set objFileDialog = Application.FileDialog(msoFileDialogFilePicker)
    Set db = CurrentDb
    With objFileDialog
        .Title = "Anlagen auswählen"
        .ButtonName = "Auswählen"
        .AllowMultiSelect = True
        .InitialFileName = CurrentProject.Path & "\"
        .Filters.Clear
        .Filters.Add "Alle Dateitypen", "*.*"
        If .Show = True Then
            For Each varPath In .SelectedItems
                strFilename = Mid(varPath, InStrRev(varPath, "\") + 1)
                db.Execute "INSERT INTO tblAttachments(AttachmentPath, AttachmentFilename, EMailID) VALUES('" _
                    & varPath & "', '" & strFilename & "', " & Me.EMailID & ")", dbOpenDynaset
            Next varPath
        End If
    End With
    Me.lstAttachments.Requery
End Sub
```

Listing 4: Hinzufügen eines Attachments

Wie auch bei den übrigen Schaltflächen, mit denen wir Datensätze zu Tabellen hinzufügen, die wir mit dem aktuellen Datensatz der Tabelle **tblEMails** verknüpfen wollen, rufen wir auch hier die Funktion **CheckNewRecord** auf, um zu prüfen, ob bereits ein Datensatz im Formular **frmEMails** vorliegt.

Dann erstellt die Prozedur ein neues **FileDialog**-Objekt mit dem Typ **msoFileDialogPicker** und weist es **objFileDialog** zu.

Für dieses Objekt legen wir den Titel, die Buttonbeschriftung, den initialen Pfad sowie als Filter ***.*** fest. Außerdem stellen wir die Eigenschaft **AllowMultiSelect** auf **True** ein, damit der Benutzer mehrere Dateien gleichzeitig auswählen kann.

Mit der **Show**-Methode rufen wir den Dialog auf und verarbeiten nach dem Schließen seine Auflistung **SelectedItems**. Diese durchlaufen wir in einer **For Each**-Schleife und speichern den aktuellen Pfad jeweils in der Variablen **varPath**.

Aus dem Pfad extrahieren wir den Dateinamen und schreiben anschließend den Pfad und den Dateinamen in die Tabelle **tblAttachments**. Außerdem tragen wir dort noch die ID des aktuellen E-Mail-Datensatzes ein.

Schließlich aktualisieren wir noch das Listenfeld **lstAttachments**, damit neu hinzugefügte Anlagen direkt angezeigt werden.

Anlagen entfernen

Zum Entfernen von Anlagen programmieren die Schaltfläche **cmdRemoveAttachment**. Diese löst die Prozedur aus Listing 5 aus.

Hier ermitteln wir zunächst den aktuell im Anlagefeld markierten Datensatz und entfernen diesen mit einer **DELETE**-Anweisung.

Auch hier folgt anschließend die Aktualisierung des Listenfeldes, damit die entfernten Anlagen auch hier nicht mehr angezeigt werden.

Entfernen von Empfängern

Für die drei Listenfelder **lstTo**, **lstCC** und **lstBCC** haben wir ebenfalls jeweils eine Schaltfläche zum Entfernen des jeweils markierten Eintrags vorgesehen.

Diese rufen jeweils die Prozedur **RemoveRecipients** auf:

```
Private Sub cmdRemoveTo_Click()
    RemoveRecipient Me.lstTo
End Sub

Private Sub cmdRemoveCC_Click()
    RemoveRecipient Me.lstCC
End Sub

Private Sub cmdRemoveBCC_Click()
    RemoveRecipient Me.lstBCC
End Sub
```

```
Private Sub cmdRemoveAttachment_Click()
    Dim lngID As Long
    Dim db As DAO.Database
    Set db = CurrentDb
    lngID = Nz(Me.lstAttachments, 0)
    db.Execute "DELETE FROM tblAttachments WHERE AttachmentID = " & lngID, dbFailOnError
    Me.lstAttachments.Requery
End Sub
```

Listing 5: Entfernen eines Attachments

ADODB: Datenzugriff mit der Command-Klasse

In einem anderen Artikel namens »ADODB: SQL-Befehle schnell ausführen mit Execute« (www.vbentwickler.de/447) haben wir bereits gezeigt, wie wir unter ADO mit der Execute-Methode der Connection-Klasse schnell SQL-Anweisungen zum Manipulieren von Daten ausführen oder Daten abrufen und mit einem Recordset durchlaufen können. Allerdings gibt es gerade für den Austausch von Daten mit dem SQL Server noch einige weitere Möglichkeiten. Dazu benötigen wir allerdings die Command-Klasse. Sie bietet primär auch eine Execute-Methode, mit der wir die gleichen Dinge erledigen können, wie mit der gleichnamigen Methode der Connection-Klasse. Sie bietet allerdings viele weitere Optionen, mit denen wir zum Beispiel Parameter an eine gespeicherte Prozedur übergeben können.

Warum die Command-Klasse?

Die **Command**-Klasse ermöglicht es Dir, Abfragen und gespeicherte Prozeduren sehr flexibel und sicher auszuführen.

Während einfache SQL-Strings in der **Connection.Execute**-Methode direkt übergeben werden, kannst Du mit Command auch Parameter definieren. Das ist gerade bei dynamischen Abfragen oder bei wiederverwendbaren SQL-Prozeduren entscheidend.

Wichtige Eigenschaften und Methoden der Command-Klasse

Hier findest Du die wichtigsten Eigenschaften und Methoden der **Command**-Klasse:

- **ActiveConnection:** Gibt an, mit welcher Connection das Command-Objekt verbunden ist.
- **Cancel:** Bricht eine laufende Ausführung eines Command-Objekts ab.
- **CommandStream:** Ermöglicht es, einen Stream (zum Beispiel XML) anstelle von SQL-Text zu verwenden.
- **CommandText:** Enthält den SQL-Befehl oder den Namen einer gespeicherten Prozedur.
- **CommandTimeout:** Gibt an, wie viele Sekunden gewartet wird, bevor ein Command abgebrochen wird.
- **CommandType:** Bestimmt, wie der Inhalt von **CommandText** interpretiert wird (zum Beispiel Text, Tabelle, Prozedur).
- **CreateParameter:** Erstellt einen neuen Parameter für das **Command**-Objekt.
- **Dialect:** Gibt die Syntax an, die für **CommandText** verwendet wird (zum Beispiel SQL-Dialect).
- **Execute:** Führt den Befehl aus und liefert optional ein Recordset zurück.
- **Name:** Name des **Command**-Objekts, kann für spätere Referenzen verwendet werden.
- **NamedParameters:** Gibt an, ob Parameter per Namen statt nach Reihenfolge übergeben werden.
- **Parameters:** Sammlung aller Parameter, die dem **Command**-Objekt zugeordnet sind.
- **Prepared:** Gibt an, ob das **Command**-Objekt vorbereitet/kompiliert ist, um schneller ausgeführt zu werden.

- **Properties:** Sammlung zusätzlicher Eigenschaften, die für das **Command**-Objekt gelten.

- **State:** Zeigt den aktuellen Status des **Command**-Objekts an (zum Beispiel geöffnet oder geschlossen).

Grundaufbau eines Command-Objekts

Ein **Command**-Objekt besteht grundsätzlich aus der Zuweisung der Connection, der Angabe des SQL-Befehls (oder Prozedurnamens) und der Einstellung des **CommandType**.

Danach kannst Du über die **Parameters**-Auflistung **Parameter** hinzufügen und konfigurieren. Schließlich rufst Du die **Execute**-Methode auf, um den Befehl auszuführen oder ein Recordset zu erhalten.

Im folgenden Beispiel öffnen wir ein Recordset mit Hilfe eines **Command**-Objekts und geben die Inhalte direkt im Direktbereich aus:

```
Public Sub KundenPerCommand()
    Dim cnn As ADODB.Connection
    Dim cmd As ADODB.Command
    Dim rst As ADODB.Recordset
    Set cnn = CurrentProject.Connection
    Set cmd = New ADODB.Command
    cmd.ActiveConnection = cnn
    cmd.CommandText = "SELECT * FROM tblKunden"
    cmd.CommandType = adCmdText
    Set rst = cmd.Execute
    Do Until rst.EOF
        Debug.Print rst!Vorname & " " & rst!Nachname
        rst.MoveNext
    Loop
    rst.Close
    Set rst = Nothing
    Set cmd = Nothing
    Set cnn = Nothing
End Sub
```

ActiveConnection

Die Eigenschaft **ActiveConnection** legt fest, welche Verbindung das **Command**-Objekt verwendet. Meist weist Du hier eine bestehende Connection zu, damit das **Command**-Objekt genau weiß, gegen welche Datenbank es ausgeführt werden soll. Beispiel:

```
Set cmd.ActiveConnection = cnn
cmd.CommandText = "SELECT * FROM tblKunden"
```

Cancel

Mit der Methode **Cancel** kannst Du einen laufenden **Command**-Vorgang abbrechen. Dies ist hilfreich, wenn ein Befehl sehr lange dauert oder der Benutzer den Vorgang abbricht. Beispiel:

```
cmd.Execute
cmd.Cancel
```

CommandStream

Die Eigenschaft **CommandStream** erlaubt es, anstelle von SQL-Text einen Datenstrom (Stream) an ein **Command**-Objekt zu übergeben. Meist handelt es sich dabei um einen XML-Stream oder einen anderen binären Datenstrom, der als Eingabe für einen Befehl dient.

Ein typisches Szenario ist der Import großer XML-Datenmengen in eine Datenbank oder das Ausführen von gespeicherten XML-Abfragen, die nicht als reiner Text übergeben werden sollen. In solchen Fällen wird ein **Stream**-Objekt erstellt (zum Beispiel mit der **ADODB.Stream**-Klasse), befüllt und dann der Eigenschaft **CommandStream** zugewiesen.

Beispiel (verkürzt dargestellt):

```
Dim stm As ADODB.Stream
Set stm = New ADODB.Stream
stm.Open
stm.WriteText "<root><datensatz>...</datensatz></root>"
Set cmd.CommandStream = stm
```

```
cmd.CommandType = adCmdText
cmd.Execute
```

Die Verwendung von **CommandStream** ist eher selten und wird vor allem in komplexeren Szenarien mit XML-Daten, BLOBs (Binary Large Objects) oder bei bestimmten Webservice-Integrationen eingesetzt. In klassischen Access- und SQL Server-Szenarien spielt diese Eigenschaft nur eine untergeordnete Rolle.

CommandText

Hier gibst Du den eigentlichen SQL-Befehl oder den Namen einer gespeicherten Prozedur an. Beispiel:

```
cmd.CommandText = "SELECT * FROM tblKunden"
cmd.CommandType = adCmdText
```

CommandTimeout

Mit **CommandTimeout** legst Du die maximale Zeit in Sekunden fest, die auf die Ausführung des Befehls gewartet wird. Beispiel:

```
cmd.CommandTimeout = 30
```

CommandType

Mit **CommandType** steuerst Du, wie Access den Inhalt von **CommandText** interpretiert (Text, Tabelle, Prozedur et cetera). Beispiel:

```
cmd.CommandType = adCmdText
```

Die folgenden Werte können wir für die Eigenschaft **CommandType** verwenden:

- **adCmdUnknown (8):** Standardwert, wenn der Typ des Befehls nicht angegeben ist. ADO versucht selbst zu erkennen, ob es sich um einen SQL-Text, eine Tabelle oder eine Prozedur handelt. Sollte möglichst vermieden werden, da es zu unerwartetem Verhalten führen kann.
- **adCmdText (1):** Gibt an, dass der Befehl ein SQL-Textbefehl ist (zum Beispiel **SELECT**-, **UPDATE**- oder **DELETE**-Befehl). Dies ist der am häufigsten verwendete Typ für direkte SQL-Strings.
- **adCmdTable (2):** Verweist auf eine ganze Tabelle. ADO erzeugt intern ein **SELECT * FROM [Tabelle]**. Ideal, wenn alle Datensätze einer Tabelle ohne Filter geladen werden sollen.
- **adCmdStoredProc (4):** Kennzeichnet eine gespeicherte Prozedur. Wird verwendet, um auf SQL Server oder anderen Datenbanken gespeicherte Prozeduren auszuführen.
- **adCmdFile (256):** Verweist auf eine Persisted Recordset-Datei, also eine gespeicherte Daten-Datei (zum Beispiel **.adtg**). Wird selten genutzt, wenn Daten aus einer Datei gelesen oder in eine Datei geschrieben werden.
- **adCmdTableDirect (512):** Öffnet die Tabelle direkt, ohne SQL-Parsing oder weitere Verarbeitung. Sehr schnelle Methode, aber ohne Filter oder Sortierungen. Nur für einfache Lesezugriffe geeignet.

CreateParameter

Mit **CreateParameter** erstellst Du einen neuen Parameter für Dein **Command**-Objekt. Dies ist vor allem nützlich bei gespeicherten Prozeduren oder dynamischen SQL-Abfragen. Der folgende Parameter übergibt den Wert **5** als Parameter für das Feld **KundenID**:

```
Set prm = cmd.CreateParameter("KundenID", adInteger, adPa-
ramInput, , 5)
cmd.Parameters.Append prm
```

Dialect

Die Eigenschaft **Dialect** gibt an, welche SQL-Syntax oder welcher Dialekt für **CommandText** verwendet

Backstage-Bereich per COM-Add-In anpassen

Wenn man eine der Office-Anwendungen um ein selbst programmiertes COM-Add-In erweitern möchte, kann man den Backstage-Bereich nutzen, um eventuell notwendige Optionen für dieses COM-Add-In dort abzubilden. Wir würden dann einen eigenen Reiter auf der linken Seite des Backstage-Bereichs platzieren, über den wir einen eigenen Bereich anzeigen können. Diesem wiederum können wir verschiedene Steuerelemente für die Anzeige und Eingabe von Optionen hinzufügen. Dieser Artikel zeigt, wie wir ein einfaches COM-Add-In mit twinBASIC erstellen und dieses mit der Definition einer Backstage-Erweiterung ausstatten und wie diese Erweiterung beim Starten der Anwendung angezeigt wird.

Mit **twinBASIC**, der Entwicklungsumgebung von Wayne Philips, lassen sich einfach COM-Add-Ins erstellen, mit denen wir auch den Backstage-Bereich anpassen können.

twinBASIC in der jeweils aktuellen Version kannst Du hier herunterladen:

```
https://github.com/twinbasic/twinbasic/releases
```

Nach dem Kopieren der enthaltenen Dateien in einen beliebigen Ordner können wir twinBASIC mit einem Doppelklick auf die **.exe**-Datei starten. Wir landen dann im Startbildschirm, wo wir auf der Seite Samples den Eintrag **Sample 5. MyCOMAddin** auswählen (siehe Bild 1).

Im nächsten Dialog können wir bereits den Projektnamen festlegen (siehe Bild 2). Die übrigen Optionen behalten wir bei.

Damit haben wir bereits ein COM-Add-In-Projekt angelegt, das wir nun direkt testen können. Dazu schließen wir alle eventuell geöffneten Access- und Excel-Anwendungen, denn für dieses ist die Demo ausgelegt.

Danach betätigen wir den Menübefehl **File|Build** (siehe Bild 3). Dies erstellt, nachdem wir den Speicherort

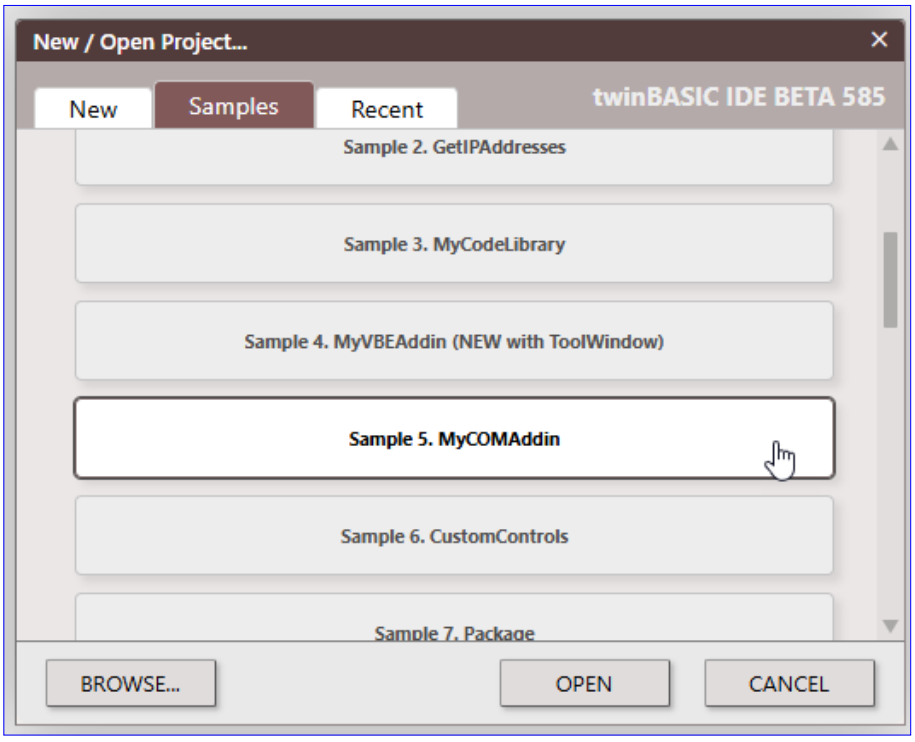

Bild 1: Erstellen eines COM-Add-Ins auf Basis des Beispiels

für die Projektdatei angegeben haben, das COM-Add-In.

Das bedeutet, dass die DLL für das COM-Add-In erstellt und in die Registry eingetragen wird. Diese Registry-Einträge werden beim Start der Office-Anwendungen ausgelesen und sorgen dafür, dass die im COM-Add-In angegebenen Ereignisprozeduren ausgeführt werden, sodass das COM-Add-In bereits beim nächsten Start ausführbereit ist.

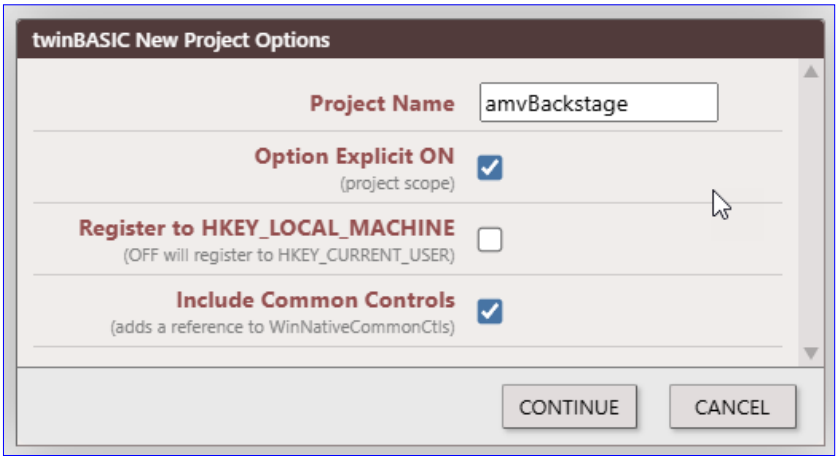

Bild 2: Festlegen des Projektnamens

Erster Test in Excel

Starten wir nun Excel und öffnen ein Workbook, sehen wir im Ribbon einen neuen Registerreiter namens **twinBASIC Test**, der eine Schaltfläche namens **Hello World** enthält. Klicken wir diese an, wird die provisorische Methode zum Anzeigen einer Meldung ausgeführt (siehe Bild 4).

Zielanwendung definieren

In twinBASIC finden wir im **PROJECT EXPLORER** unter

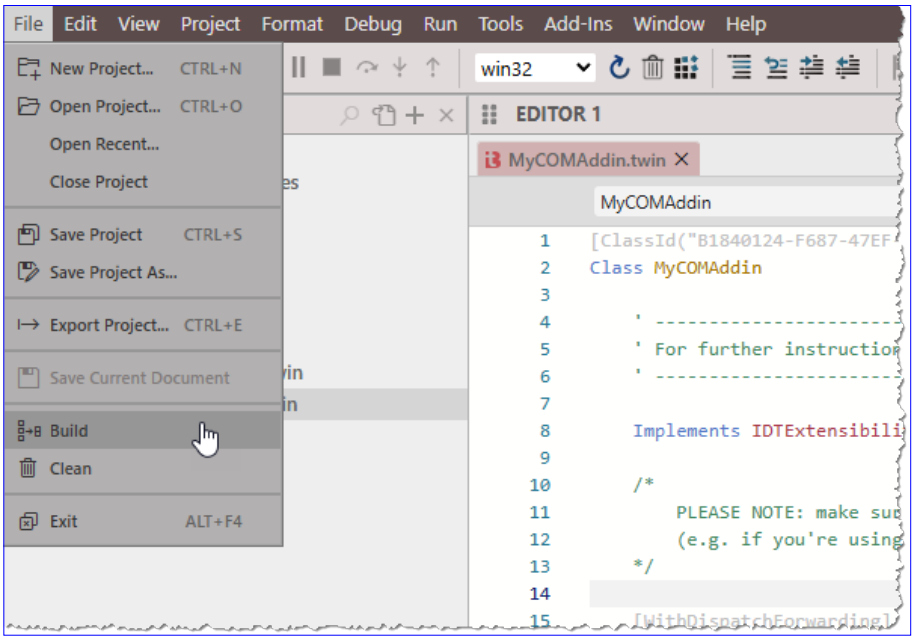

Bild 3: Projekt erstellen

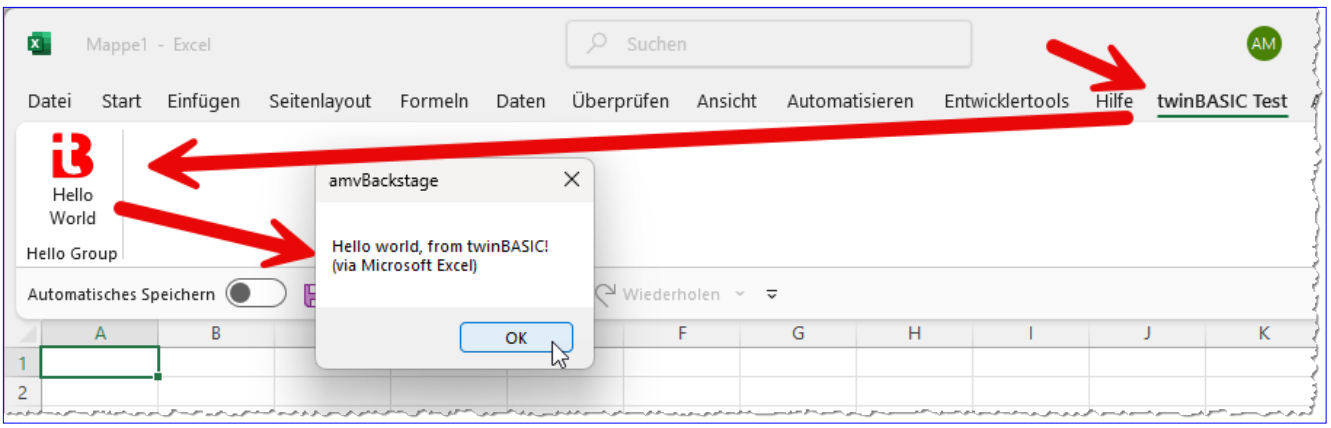

Bild 4: Die Funktion des neuen COM-Add-Ins ist nun direkt in der jeweiligen Office-Anwendung verfügbar.

```
Const AddinProjectName As String = VBA.Compilation.CurrentProjectName
Const AddinClassName As String = "Backstage"
Const AddinQualifiedClassName As String = AddinProjectName & "." & AddinClassName
Const RootRegistryFolder_ACCESS As String = "HKCU\SOFTWARE\Microsoft\Office\Access\Addins\" & AddinQualifiedClassName & "\"
Const RootRegistryFolder_EXCEL As String = "HKCU\SOFTWARE\Microsoft\Office\Excel\Addins\" & AddinQualifiedClassName & "\"
Const RootRegistryFolder_WORD As String = "HKCU\SOFTWARE\Microsoft\Office\Word\Addins\" & AddinQualifiedClassName & "\"
Const RootRegistryFolder_OUTLOOK As String = "HKCU\SOFTWARE\Microsoft\Office\Outlook\Addins\" & AddinQualifiedClassName & "\"
Const RootRegistryFolder_POWERPOINT As String = "HKCU\SOFTWARE\Microsoft\Office\Powerpoint\Addins\" _
        & AddinQualifiedClassName & "\"
```

Listing 1: Konstanten für das Registrieren des COM-Add-Ins für die jeweilige Office-Anwendung

Sources zwei Module. Das Modul **DllRegistration** enthält den Code, der das COM-Add-In für die gewünschten Office-Anwendungen installiert. Im oberen Bereich werden verschiedene Konstanten definieren mit den Pfaden der Registry, in denen die Registry-Informationen beim Registrieren eingetragen werden sollen (siehe Listing 1). Wichtig: Hier muss für **AddinClassName** der Name der soeben umbenannten Klas-

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

    wscript.RegWrite RootRegistryFolder_EXCEL & "FriendlyName", AddinProjectName, "REG_SZ"
    wscript.RegWrite RootRegistryFolder_EXCEL & "Description", AddinProjectName, "REG_SZ"
    wscript.RegWrite RootRegistryFolder_EXCEL & "LoadBehavior", 3, "REG_DWORD"

    wscript.RegWrite RootRegistryFolder_WORD & "FriendlyName", AddinProjectName, "REG_SZ"
    wscript.RegWrite RootRegistryFolder_WORD & "Description", AddinProjectName, "REG_SZ"
    wscript.RegWrite RootRegistryFolder_WORD & "LoadBehavior", 3, "REG_DWORD"

    wscript.RegWrite RootRegistryFolder_OUTLOOK & "FriendlyName", AddinProjectName, "REG_SZ"
    wscript.RegWrite RootRegistryFolder_OUTLOOK & "Description", AddinProjectName, "REG_SZ"
    wscript.RegWrite RootRegistryFolder_OUTLOOK & "LoadBehavior", 3, "REG_DWORD"

    wscript.RegWrite RootRegistryFolder_POWERPOINT & "FriendlyName", AddinProjectName, "REG_SZ"
    wscript.RegWrite RootRegistryFolder_POWERPOINT & "Description", AddinProjectName, "REG_SZ"
    wscript.RegWrite RootRegistryFolder_POWERPOINT & "LoadBehavior", 3, "REG_DWORD"
    Return True
RegError:
    MsgBox "DllRegisterServer -- An error occured trying to write to the system registry:" & vbCrLf & _
            Err.Description & " (" & Hex(Err.Number) & ")"
    Return False
End Function
```

Listing 2: Funktion zum Eintragen der COM-Add-In-Informationen in die Registry

Ereignisse aus VB.NET-COM-DLLs implementieren

Viele Szenarien, in denen man eine COM-DLL benötigt, kann man mit einer VB6-COM-DLL auf Basis von twinBASIC abbilden. Manchmal möchte man aber auf Funktionen zugreifen, die einfacher mit einem VB.NET-Projekt zu erledigen sind. In diesem Artikel schauen wir uns nicht nur an, wie wir einen COM-DLL mit VB.NET erstellen, sondern legen den Fokus auf das Bereitstellen von Ereignissen und wie wir diese in dem VBA-Projekt, welches die COM-DLL nutzt, implementieren können.

Beispielprojekt anlegen

Um das Beispielprojekt zu erstellen, benötigen wir Visual Studio, zum Beispiel in der Version 2022. Hier legen wir ein neues Projekt an und wählen dazu den Typ **Klassenbibliothek (.NET-Framework)** aus (siehe Bild 1).

Name festlegen

Anschließend erscheint der Dialog, mit dem wir den Namen des Projekts festlegen sowie das Verzeichnis, in dem es angelegt werden soll. Hier stellen wir für die Eigenschaft **Name** den Wert **amvEvents** ein (siehe Bild 2).

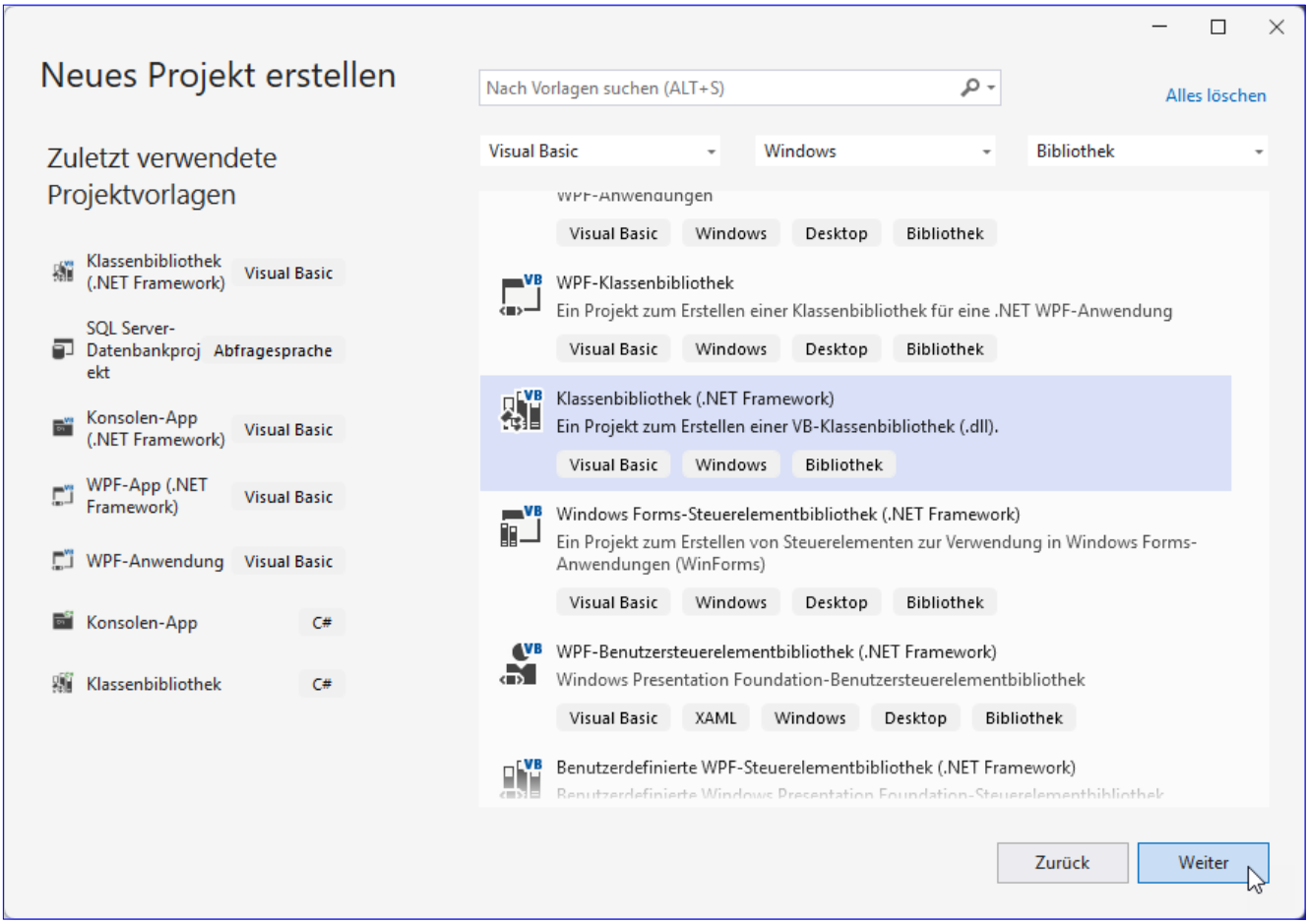

Bild 1: Projekttyp für das neue Projekt auswählen

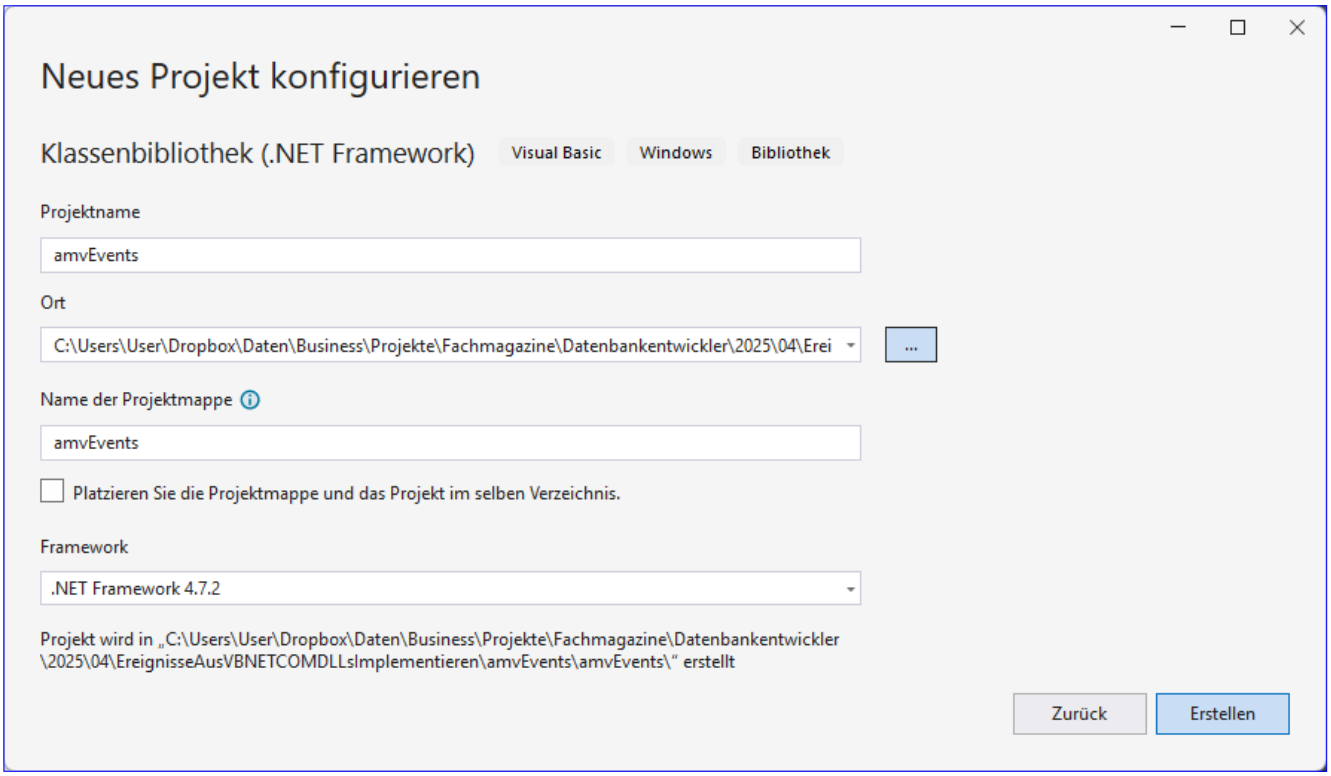

Bild 2: Projekteigenschaften vor dem Speichern festlegen

Klasse umbenennen

Im nun erscheinenden Projekt-Explorer benennen wir als Erstes die Klasse **Class1.vb** in **Ereignis.vb** um. Visual Studio fragt uns nun, ob wir auch die Verweise auf das entsprechende Code-Element anpassen wollen, was wir bestätigen (siehe Bild 3).

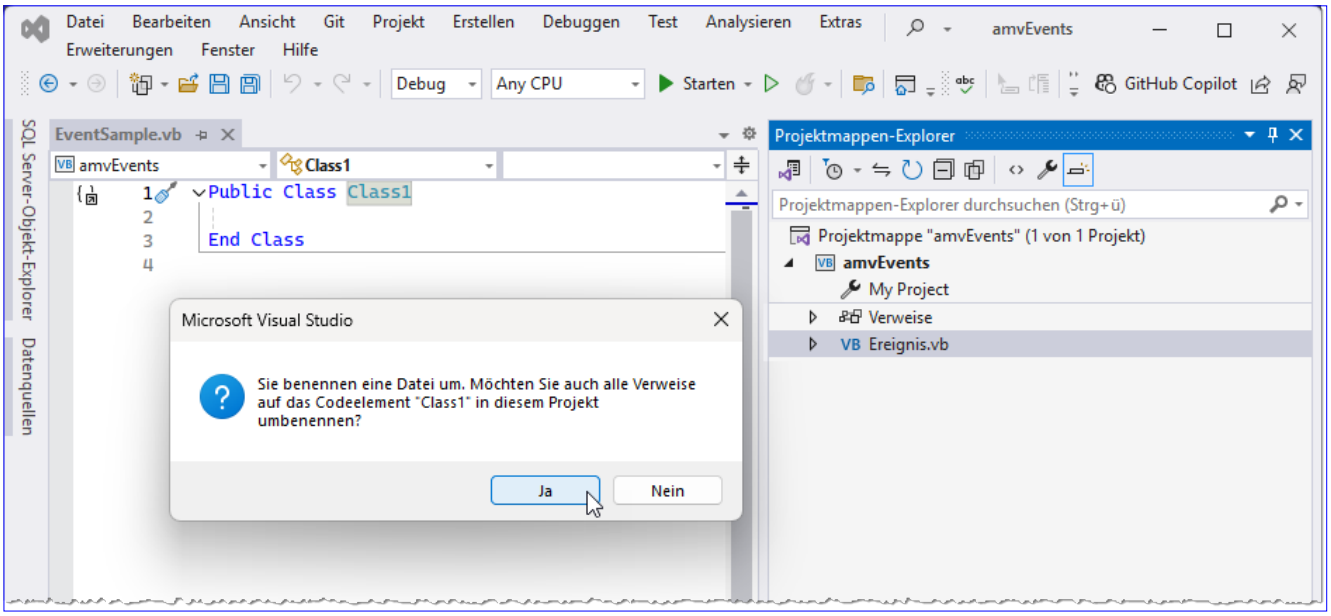

Bild 3: Umbenennen der Klasse des Projekts

Assemblyinformationen anpassen

Damit kommen wir zu einigen Einstellungen, die für die Verwendung des Projekts als COM-DLL wichtig sind.

Als Erstes klicken wir im Projektmappen-Explorer doppelt auf den Eintrag **My Project**.

Dies öffnet den Bereich mit den Eigenschaften des Projekts. Hier sehen wir unter **Anwendung** die Schaltfläche **Assemblyinformationen...** (siehe Bild 4).

Diese klicken wir an und öffnen so den Dialog **Assemblyinformationen**. Hier aktivieren wir im unteren Bereich die Option **Assembly COM-sichtbar machen** (siehe Bild 5).

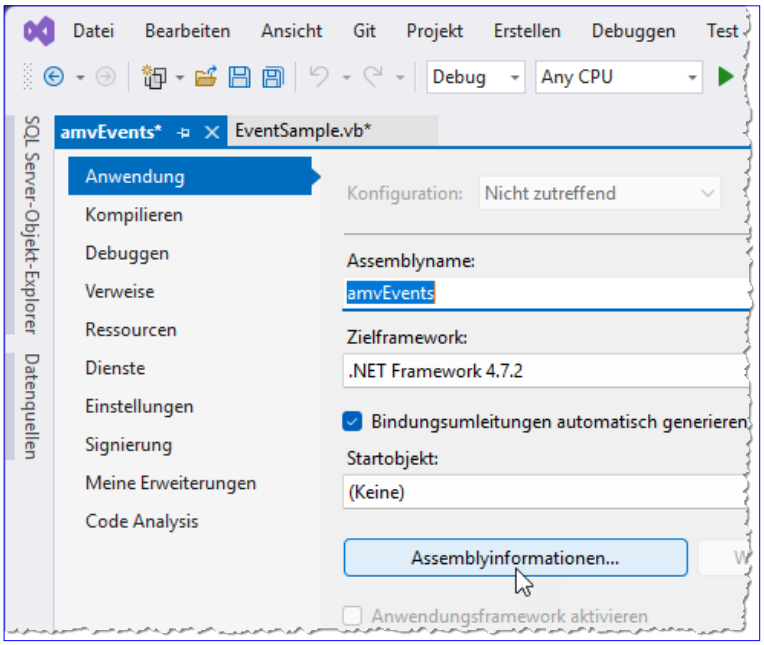

Bild 4: Bearbeiten der Projekteigenschaften

Nachdem wir den Dialog wieder geschlossen haben, wechseln wir in den Projekt-Optionen zum Bereich **Kompilieren**. Hier finden wir ganz unten die Option **Für COM-Interop registrieren**. Indem wir diese aktivieren, sorgen wir dafür, dass die COM-DLL beim Erstellen in die Registry eingetragen wird, sodass sie für den Zugriff von VBA aus verfügbar ist (siehe Bild 6).

Damit haben wir die Vorbereitungen abgeschlossen und können uns dem eigentlichen Programmieren zuwenden.

Beispiel für ein COM-DLL-Ereignis programmieren

Das Ziel der folgenden Schritte ist die Programmierung des einfachsten denkbaren Beispiels für das Bereitstellen eines Ereignisses durch die COM-DLL. Dazu wollen wir die folgende Struktur realisieren:

- Programmierung eines Ereignisses mit dem Schlüsselwort **Event**, so wie es auch in VBA realisierbar ist

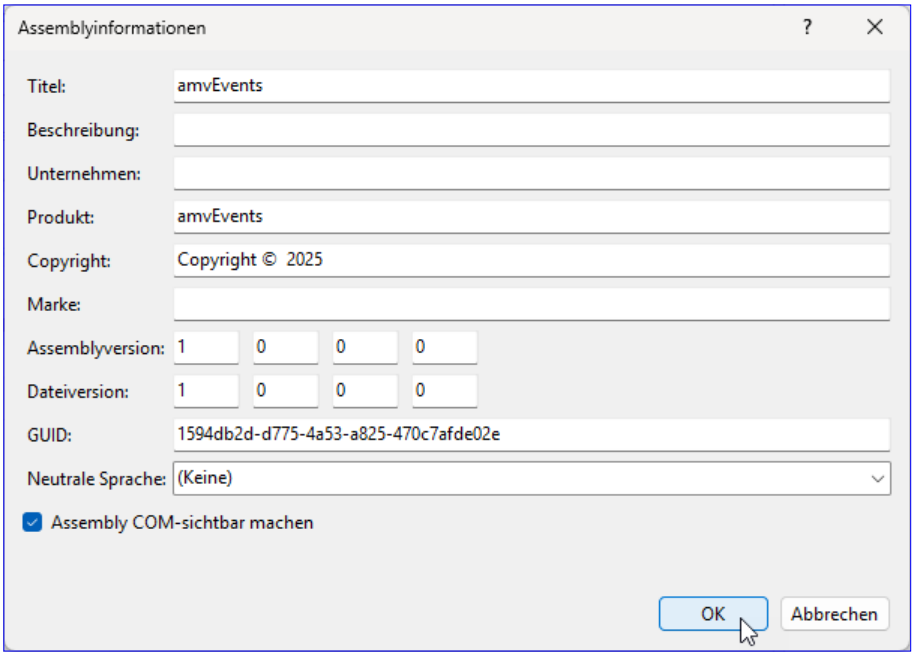

Bild 5: Aktivieren der COM-Sichtbarkeit

- Hinzufügen einer öffentlichen Methode, mit der wir das Ereignis auslösen können.

Damit die Elemente der COM-DLL optimal im VBA-Editor genutzt werden können, können wir nicht einfach eine Klasse mit der Definition des Ereignisses und der auslösenden Methode programmieren.

Unter VBA könnten wir das etwa wie folgt in einem Klassenmodul programmieren:

```
Public Event Ereignis()
Public Sub EreignisAusloesen()
    RaiseEvent Ereignis()
End Sub
```

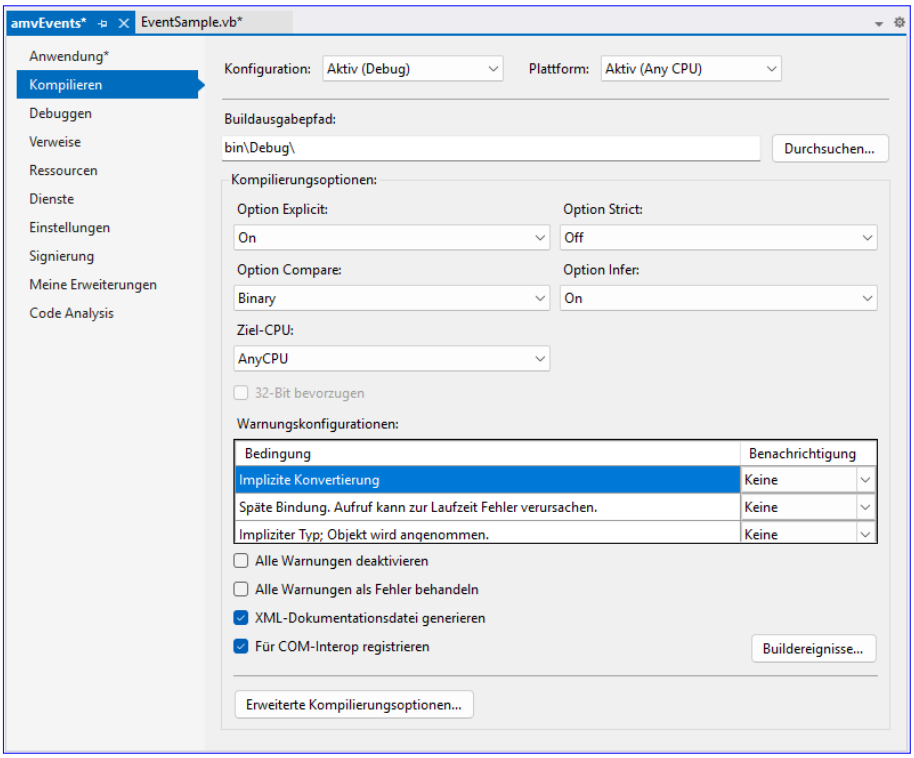

Bild 6: Aktivieren der Registrierung für COM-Interop

In VB.NET reicht das aber nicht aus. Hier benötigen wir einige weitere Elemente, damit das Ereignis problemlos funktioniert und nicht mehr Elemente als nötig unter VBA angezeigt werden. Die Klasse wird in Listing 1 im Überblick dargestellt, in den folgenden Abschnitten erläutern wir die einzelnen Elemente.

Notwendiger Namespace: InteropServices

Damit wir die gleich im Detail beschriebenen speziellen COM-Interfaces verwenden können, fügen wir dem Modul als erste Zeile den Import des Namensraums **System.Runtime.InteropServices** hinzu:

```
Imports System.Runtime.InteropServices
```

Trennung von COM-Methoden und COM-Ereignissen

Wenn wir in anderen Artikeln einmal COM-DLLs auf Basis von VB.NET vorgestellt haben, wurde dort auch schon immer ein Interface verwendet, welches die Klasse dann implementiert hat. Das hat den Sinn, dass wirklich nur die in dem Interface definierten Elemente wie Eigenschaften oder Methoden nach außen sichtbar werden – zum Beispiel unter VBA.

Wenn wir kein Interface verwenden würden, sehen wir unter VBA unnötige Elemente wie **Equals**, **GetHashCode** oder **GetType**. Diese benötigen wir aber nicht und sie machen die Auswahl der Elemente per IntelliSense auch noch unübersichtlicher. Das ist also der Grund, warum wir ein Interface wie das folgende definieren:

```
<InterfaceType(ComInterfaceType.InterfaceIsDual)>
Public Interface IEreignis
    Sub EreignisAusloesen()
End Interface
```

Hier legen wir zunächst den Typ des Interfaces fest, und zwar **ComInterfaceType.InterfaceIsDual**. damit stellen wir sicher, dass alle in diesem Interface dekla-

```
Imports System.Runtime.InteropServices

<InterfaceType(ComInterfaceType.InterfaceIsDual)>
Public Interface IEreignis
    Sub EreignisAusloesen()
End Interface

<InterfaceType(ComInterfaceType.InterfaceIsIDispatch)>
Public Interface IEreignisEvents
    Sub Ereignis()
End Interface

<ClassInterface(ClassInterfaceType.None)>
<ComSourceInterfaces(GetType(IEreignisEvents))>
Public Class Ereignis
    Implements IEreignis

    Public Event Ereignis()
    Public Sub EreignisAusloesen() Implements IEreignis.EreignisAusloesen
        RaiseEvent Ereignis()
    End Sub
End Class
```

Listing 1: Die Klasse **Ereignis** im Überblick

rierten Methoden, Eigenschaften und so weiter auch unter VBA angezeigt und verwendet werden können – sowohl per IntelliSense als auch im Objektkatalog.

Hier sehen wir allerdings nur die Methode **EreignisAusloesen** und nicht die Definition des Ereignisses selbst. Dieses müssen wir in einem eigenen Interface definieren.

Für dieses verwenden wir wiederum einen anderen Interface-Typ, nämlich **ComInterfaceType.InterfaceIsDispatch**. Nur so können wir Ereignisse empfangen. Das Ereignis definieren wir durch eine einfache **Sub**-Definition wie folgt:

```
<InterfaceType(ComInterfaceType.InterfaceIsIDispatch)>
Public Interface IEreignisEvents
    Sub Ereignis()
End Interface
```

Logik der Klasse und Implementierung der Elemente

Schließlich folgt die eigentliche Logik beziehungsweise die Implementierung der beiden Elemente, also der Methode und des Ereignisses.

Dies geschieht in der eigentlichen Klasse **Ereignis**, die das Interface **IEreignis** mit der Methode **EreignisAusloesen** implementiert. Diese Klasse deklarieren wir als Klassen-Interface mit dem Typ **ClassInterfaceType.None**. Die Elemente dieser Klasse werden dadurch nicht automatisch veröffentlicht und erscheinen somit nicht im VBA-Projekt, wenn wir diesem einen Verweis auf die COM-DLL hinzufügen. Das ist auch nicht nötig, denn alle Elemente haben wir bereits über die zuvor beschriebenen Elemente mit den Interface-Typen **InterfaceIsDual** (für die Eigenschaften und Methoden) sowie **InterfaceIsDispatch** (für die Ereignisse) veröffentlicht.

Wie aber machen wir in dieser Klasse kenntlich, durch welche öffentlichen Elemente die Methode und das Ereignis nach außen verfügbar gemacht werden?

Für das Interface **IEreignis** gelingt dies durch zwei Maßnahmen:

- Wir fügen der Klasse **Ereignis** über das Schlüsselwort **Implements** das Interface **IEreignis** hinzu.

eBay per VBA steuern: Angebote suchen

Im ersten Teil der Artikelreihe zum Thema Steuerung von eBay mit VBA haben wir gezeigt, wie man einen Entwickler-Account anlegt, eine neue Anwendung bei eBay erstellt, grundlegende Zugriffsdaten holt und die für die Authentifizierung im Kontext eines bestimmten Benutzerkontos notwendigen Informationen holt – hier speziell das Authentifizierungstoken. Damit haben wir die Basis geschaffen, um per VBA auf die Rest API von eBay zuzugreifen. Damit fahren wir in diesem Teil fort: Wir wollen den grundlegenden Zugriff auf die Rest-API von eBay herstellen und verwenden dazu einfache Beispiele. Wie sich zeigt, gibt es zwei verschiedene Kontexte, in denen wir hier arbeiten, den Anwendungs- und den Benutzerkontext. Beide beschreiben wir in diesem Artikel.

Anwendungs- und Benutzerkontext

Wir im ersten Teil der Artikelreihe namens **eBay per VBA steuern: Zugangsdaten holen** (**www.vbentwickler.de/460**) beschrieben, gibt es zwei verschiedene Kontexte, in denen wir auf die Rest-API von eBay zugreifen können:

- **Anwendungskontext**: Dies erlaubt uns die Nutzung allgemein zugänglicher Informationen – also beispielsweise solche, die man auch auf der Webseite **ebay.com** als nicht angemeldeter Benutzer einsehen kann. Hier für reicht das im oben genannten Artikel ermittelte Anwendungstoken zur Authentifizierung aus.

- **Benutzerkontext**: Hiermit können wir solche Aktionen durchführen, die man auch als angemeldeter Benutzer erledigen kann. Dazu benötigen wir das Benutzertoken aus dem vorherigen Artikel. Damit wir die Benutzerfunktionen nutzen können, muss das Benutzertoken auch im Kontext der Anmeldung des entsprechenden Benutzers geholt werden.

Zu beachten ist, dass man je nach der verwendeten API und Funktion keine Authentifizierung, die Anwendungsauthentifizierung oder die Benutzerauthentifizierung benötigt. Wann welche Authentifizierung benötigt wird, erfährt man am einfachsten in der API-Dokumentation beziehungsweise im API-Explorer.

Zurechtfinden im API-Dschungel

eBay hat einen umfangreichen Bestand an API-Funktionen. Einen ersten Überblick findet man auf der folgenden Webseite:

```
https://developer.ebay.com/develop
```

Von hier aus geht es weiter zu den absoluten Grundlagen, zum Entwickeln von Anwendungen zum Verkaufen oder Kaufen oder zu weiteren Themen.

Im Bereich **Kaufen** interessiert uns beispielsweise, wie wir eine Suche nach Artikeln per VBA realisieren können. Im Bereich **Verkaufen** wäre es spannend, eigene Artikel voll automatisiert per VBA bei eBay einstellen zu können. Oder auch herauszufinden, welche Artikel wir gerade bei eBay anbieten und in welcher Anzahl, um hier gegebenenfalls aufzustocken. Beide Bereiche bieten zahlreiche Funktionen bereit, wir wollen uns jedoch auf den Bereich des Verkaufens konzentrieren.

Über diesen können wir uns in den weiterführenden Artikeln des oben genannten Links einen ersten Überblick verschaffen. Diesen finden wir zum Zeitpunkt der Erstellung dieses Artikels unter dem folgenden Link:

`https://developer.ebay.com/api-docs/sell/static/selling-ig-landing.html`

Hier finden wir einen Überblick über die Unterbereiche zum Thema Verkaufen auf eBay (siehe Bild 1).

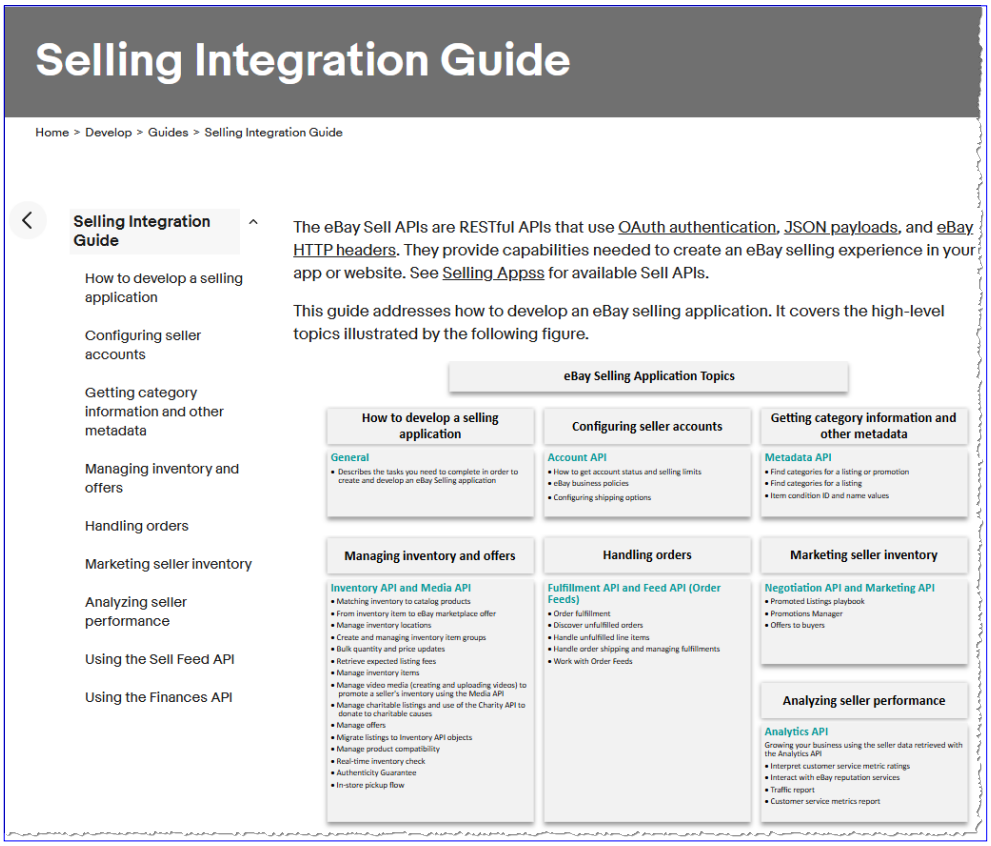

Bild 1: Überblick über die eBay Sell API

Einstellen von Artikeln

Zum Einstellen von Artikeln finden wir verschiedene Ansätze:

- **eBay Trading API**: Diese API ist bereits etwas älter. Sie ist XML-basiert und bietet beispielsweise Funktionen wie **AddItem**.
- **eBay Inventory API**: Diese API ist moderner und verwendet JSON für den Datenaustausch.

Wir wollen uns in dieser Artikelreihe auf die aktuellen Techniken funktionieren und schauen uns daher die Variante an, welche die **eBay Inventory API** nutzt.

API-Katalog

Aufrufe von Rest-API haben wir in diesem Magazin bereits an einigen Beispielen demonstriert. Aber woher beziehen wir im Falle der eBay-API die wirklich wichtigen Informationen wie die folgenden:

- Welche Authentifizierungsmethode benötigen wir – anwendungs- oder benutzerorientiert?
- Wie lauten die Endpunkte für die gewünschten Aufrufe?
- Welche Parameter müssen wir noch übermitteln?
- Wie werden die JSON-Dokumente aufgebaut, die wir übermitteln?
- Und wie ist der Aufbau der JSON-Dokumente, die als Ergebnis zurückgeliefert werden?
- Welche Statuscodes werden von den verschiedenen Funktionen zurückgeliefert und was bedeuten diese?

Antworten auf viele dieser Fragen liefert der API-Katalog, den wir unter dem folgenden Link finden (alternativ suchen wir im Internet nach dem Suchbegriff **ebay api explorer**):

`https://developer.ebay.com/my/api_test_tool?index=0`

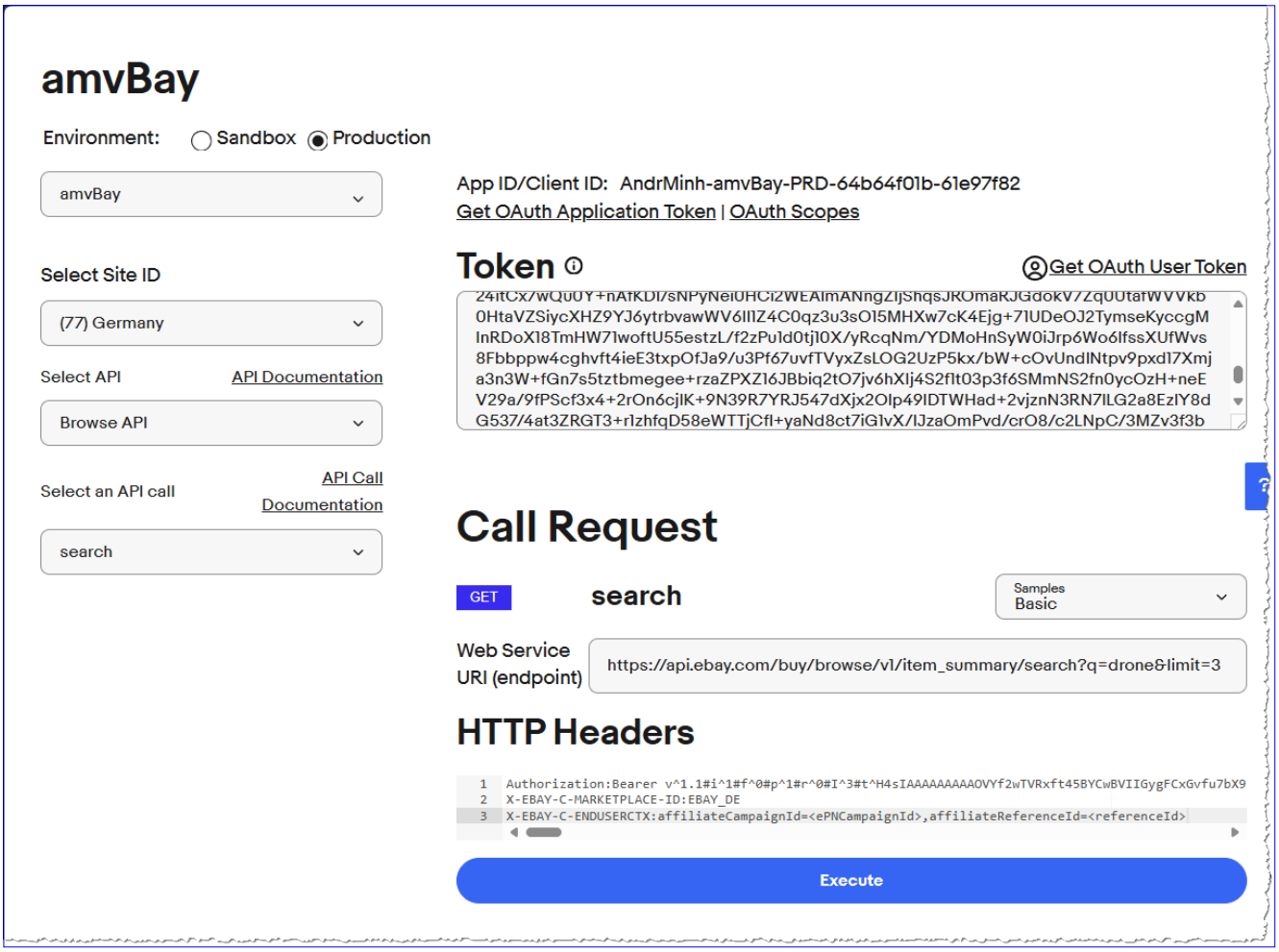

Bild 2: Aufrufen einer API-Funktion

Damit landen wir auf der **API Explorer**-Seite (siehe Bild 2). Hier sehen wir verschiedene Elemente.

Wichtig ist erst einmal, dass wir oben die richtige Umgebung auswählen, also **Sandbox** oder **Production**.

Danach wählen wir im linken Bereich die zu verwendende Anwendung aus, hier **amvBay**, und die eBay-Seite, die wir untersuchen wollen. Wir möchten auf der deutschen Seite suchen und wählen deshalb **(77) Germany** aus.

Danach entscheiden wir, welche API wir nutzen wollen. Wir wollen erst einmal Angebote durchsuchen, also wählen wir die **Browse API**.

Dies aktualisiert wiederum direkt die Auswahl unter **Select an API call**. Hier wählen wir die Funktion **search** aus.

Dadurch wird rechts das Feld **Web Service URI (endpoint)** gefüllt, in diesem Fall mit dem folgenden Beispielcode:

```
https://api.ebay.com/buy/browse/v1/item_summary/search?q=drone&limit=3
```

Dies sind die Bestandteile:

- Basis-URL **https://api.ebay.com/buy/browse/v1/**: Dies ist der Einstiegspunkt für die eBay Browse API,

die Informationen zu Artikeln bereitstellt, die auf eBay zum Verkauf stehen.

- Endpunkt **item_summary/search**: Dieser Endpunkt führt eine Suche nach Artikeln durch und liefert eine Zusammenfassung (**item summary**) der gefundenen Ergebnisse.

- Parameter **q=drone**: Dies ist der Suchbegriff. In diesem Fall wird nach Artikeln gesucht, die den Begriff **drone** enthalten. Es durchsucht Titel, Beschreibung und andere relevante Felder der Artikel.

- Anzahl der Ergebnisse **limit=3**: Dies legt fest, wie viele Ergebnisse pro Anfrage zurückgegeben werden sollen. In diesem Fall werden maximal drei Artikel zurückgegeben.

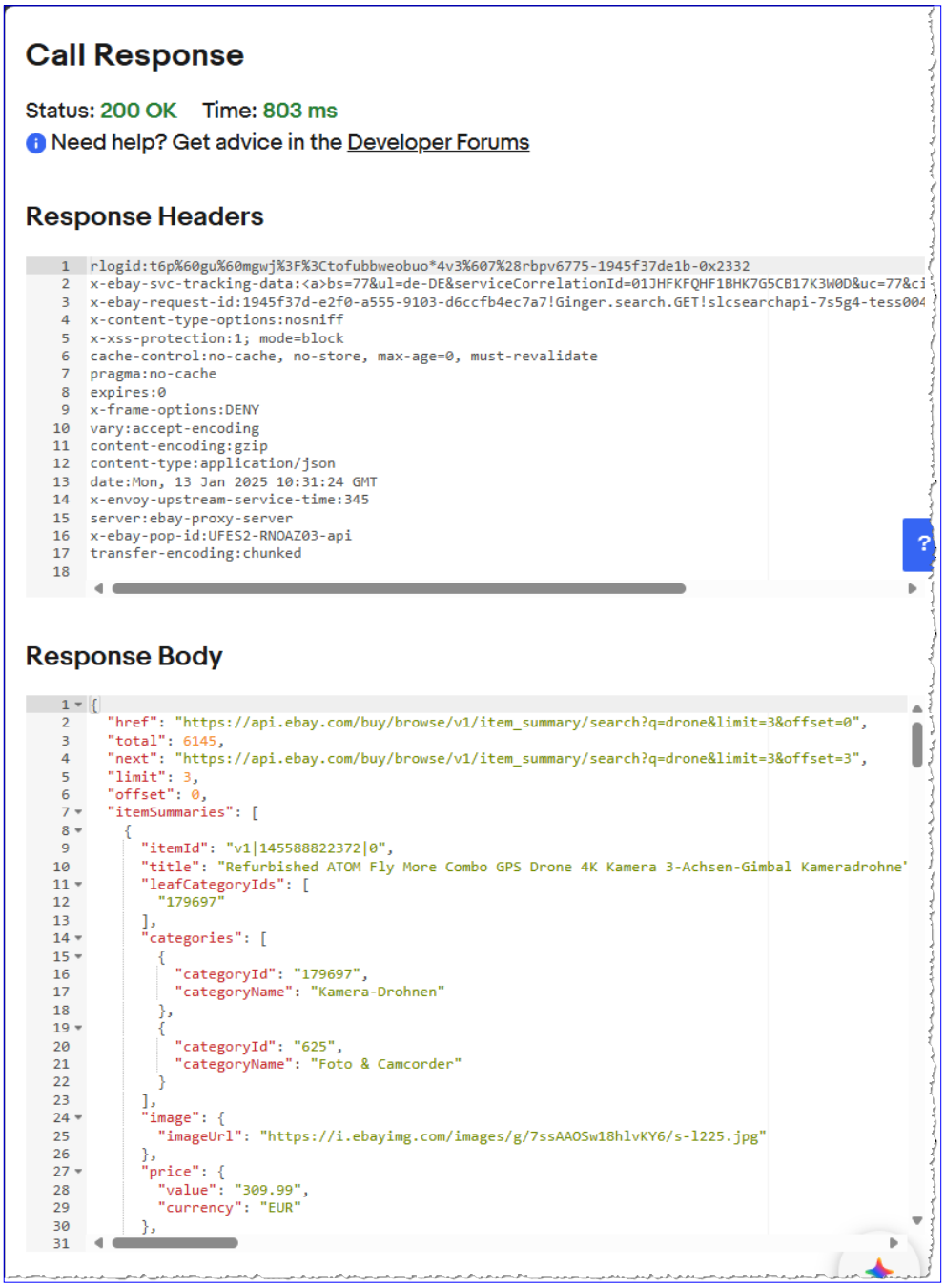

Bild 3: Ergebnis der Artikelsuche

Wichtig ist auch noch der Inhalt des kleinen Kästchens in der Nähe von Call Request, das zum Beispiel **GET**, **POST**, **PUT** oder **DELETE** enthalten kann – eine Information, die wir später für den Aufruf der Rest AP benötigen.

HTTP-Header

Darunter finden wir den Bereich **HTTP Headers**. Auch hier finden wir wertvolle Informationen, die wir gegebenenfalls in unserem eigenen Aufruf per VBA berücksichtigen müssen. Auch dort haben wir die

Möglichkeit, Header zu übersenden. Klicken wir nun auf **Execute**, erhalten wir im unteren Bereich das Ergebnis. Dieses sieht wie in Bild 3 aus.

Hier sehen wir im oberen Bereich die Response-Header, die für uns weniger interessant sind. Spannender ist das tatsächliche Ergebnis unter **Response Body**.

Hier finden wir im JSON-Format die Eigenschaften der gefundenen Artikel. Unser Ziel ist es, dieses Dokument per VBA auszulesen und es dann auslesen zu können. Das erledigen wir in den folgenden Abschnitten.

Voraussetzung: OAuth Application Token

Um auf die Api zuzugreifen, benötigen wir das sogenannte OAuth Application Token. Das holen wir uns, nachdem wir die im Artikel **eBay per VBA steuern: Zugangsdaten holen** (**www.vbentwickler.de/460**) beschriebenen Schritte zum Erstellen eines Entwicklerkontos und einer App durchgeführt haben, mit der VBA-Prozedur **GetEbayAppToken**. Diese rufen wir

```
Public Function GetItems(strResponse As String) As Boolean
    Dim objXMLHTTP As MSXML2.XMLHTTP60
    Dim strURL As String
    Dim strOAuthToken As String

    strURL = "https://api.ebay.com/buy/browse/v1/item_summary/search?q=drone&limit=3"

    strOAuthToken = "Bearer " & cStrAppToken

    Set objXMLHTTP = New MSXML2.XMLHTTP60
    objXMLHTTP.Open "GET", strURL, False

    objXMLHTTP.setRequestHeader "Authorization", strOAuthToken
    objXMLHTTP.setRequestHeader "X-EBAY-C-MARKETPLACE-ID", "EBAY_DE"
    objXMLHTTP.setRequestHeader "X-EBAY-C-ENDUSERCTX", _
        "ffiliateCampaignId=<ePNCampaignId>,affiliateReferenceId=<referenceId>"

    objXMLHTTP.Send

    strResponse = objXMLHTTP.responseText

    Select Case objXMLHTTP.status
        Case 200
            GetItems = True
        Case Else
            Debug.Print objXMLHTTP.status & vbCrLf & objXMLHTTP.responseText
    End Select

    Set objXMLHTTP = Nothing
End Function
```

Listing 1: Erster Aufruf der eBay-Rest API

wie folgt auf, um das Token direkt in der Registry zu speichern:

```
Public Sub GetEbayAppTokenInRegistry()
    Dim strAppToken As String
    strAppToken = GetEbayAppToken(GetAppSetting("AppID"), _
        GetAppSetting("CertID"), cStrAPIScope)
    SaveAppSetting "AppToken", strAppToken
End Sub
```

Voraussetzung ist, dass wir zuvor die übrigen notwendigen Informationen, also die **AppID** und die **CertID** sowie den **APIScope** wie im oben genannten Artikel ermittelt haben.

Erster Wurf: Aufruf testen

In der Prozedur **GetItems** haben wir einen ersten Anlauf zusammengestellt, um eine Liste von Artikeln abzurufen (siehe Listing 1). Wobei erster Anlauf untertrieben ist – wir haben im Vorfeld bereits einige Versuche unternommen, bis wir eine lauffähige Variante hatten.

Die Dokumentation des hier verwendeten Endpunkts **item_summary/search** findest Du hier:

```
https://developer.ebay.com/api-docs/buy/browse/resources/
item_summary/methods/search
```

Hier findest Du ausführlich alle möglichen Parameter und wie man sie einsetzt und es wird auch der komplette Output dieses Aufrufs erläutert.

Die Prozedur deklariert zunächst eine Variable des Typs **MSXML2.XMLHTTP60**. Um diese verwenden zu können, benötigen wir einen Verweis auf die Bibliothek **Microsoft XML, v6.0** hinzu.

In diesem Zuge addieren wir direkt noch einen Verweis auf Bibliothek **Microsoft Scripting Runtime** (siehe Bild 4).

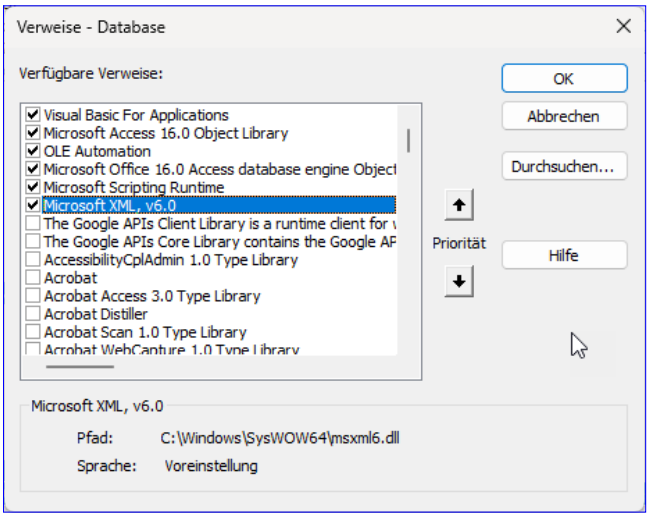

Bild 4: Verweise für XML und JSON

Letzteren benötigen wir für Funktionen für den Zugriff auf die von der Rest API zurückgelieferten JSON-Dokumente.

Wenn Du die Anwendung nachbauen möchtest, benötigst Du auf jeden Fall noch die folgenden beiden Module:

- **mdlJSON**: Enthält Funktionen zum Parsen von JSON-Dokumenten ein Objektmodell und zum Erzeugen von JSON-Dateien aus diesem Objektmodell.
- **mdlJSONDOM**: Enthält Funktionen, mit denen wir die Bezüge für den Zugriff auf die Elemente des Objektmodells für alle JSON-Dokumente leicht ausgeben und übernehmen können.

Außerdem ist auch das Modul **mdlZwischenablage** hilfreich. Dieses enthält eine Funktion, mit der wir den Inhalt einer Variablen in die Zwischenablage kopieren können.

Das ist hilfreich, wenn wir größere JSON-Dokumente von der Rest-API zurückbekommen, die wir nicht vollständig im Direktbereich ausgeben können, ohne

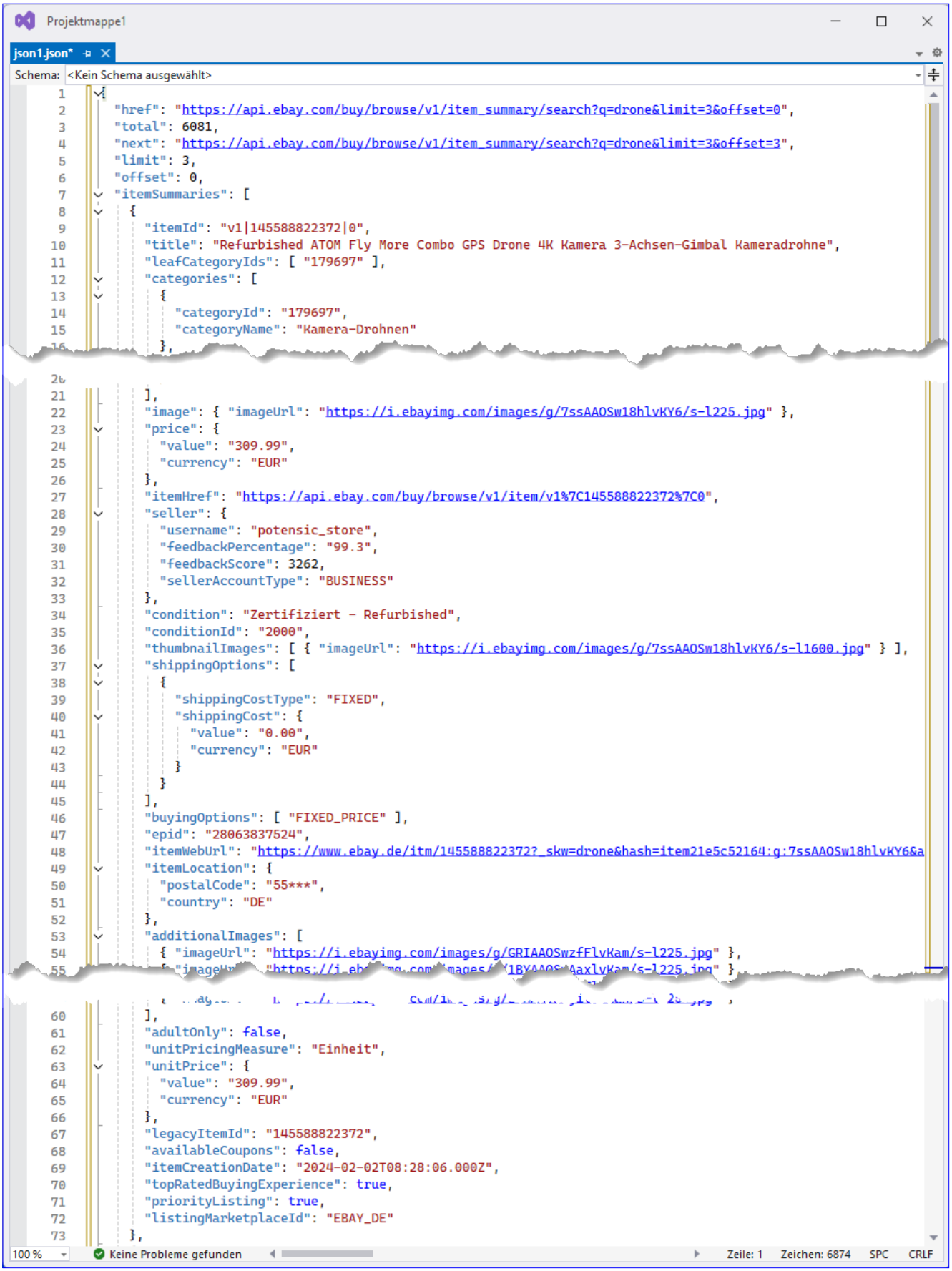

Bild 5: Ergebnis der Abfrage im JSON-Format

das der obere Teil gelöscht wird. Dann können wir solche Texte aus der Zwischenablage in einen Text- oder JSON-Editor kopieren, um sie dort zu analysieren.

Die Prozedur erstellt noch weitere **String**-Variablen. Dann fügt sie der Variablen **strUrl** die Adresse unserer Rest API-Funktion hinzu, die wir hier vorerst 1:1 aus dem API Explorer entnommen haben. Danach fügt sie das Schlüsselwort **Bearer** mit dem Anwendungstoken aus der Konstanten **cStrAppToken** zusammen.

Damit erstellt sie nun ein Objekt auf Basis des Typs **MSXML2.XMLHTTP60** und ruft dessen **Open**-Methode auf. Die Parameter sehen wie folgt aus:

- **bstrMethod**: Gibt die Zugriffsmethode an, zum Beispiel **GET, POST**, **PUT** oder **DELETE**

- **bstrURL**: Gibt die Adresse der Rest API-Funktion an.

- **varAsynch**: Gibt an, ob der Aufruf asynchron angegeben werden soll. Der Wert **False** legt fest, dass dieser synchron erfolgen soll, das heißt, die ausführende Prozedur wird erst fortgesetzt, wenn der Aufruf beendet ist.

Danach setzen wir Request-Header, die wir ebenfalls dem entsprechenden Bereich im API Explorer entnehmen können.

Diese teilen wir am Doppelpunkt auf und geben den linken Teil in Anführungszeichen für den Parameter **bstrHeader** an und den rechten als **bstrValue**. So übergeben wir das Token und auch weitere Informationen über den Aufruf wie beispielsweise den zu durchsuchenden Markt.

Danach senden wir die Anfrage schließlich mit der **Send**-Methode ab und erhalten verschiedene Ergebnisse, zum Beispiel:

- über die Eigenschaft **status** den Status der Anfrage, zum Beispiel mit dem Wert **200** im Erfolgsfall oder **404**, falls die Adresse aus **strUrl** nicht existiert und

- über die Eigenschaft **responseText**, die zumindest dann, wenn die URL korrekt war, einen Ergebnistext zurückliefert.

Geben wir zum Beispiel das falsche Token an, erhalten wir für **status** den Wert **401** und für **responseText** diesen Text:

```
{"errors":
  [
    {
      "errorId": 1001,
      "domain": "OAuth",
      "category": "REQUEST",
      "message": "Invalid access token",
      "longMessage": "Invalid access token. Check the
value of the Authorization HTTP request header." }
  ]
}
```

Im Erfolgsfall hingegen erhalten wir ein JSON-Dokument, das etwa wie in Bild 5 aussieht.

Hier ist der erste von drei Artikel aus dem Ergebnis der Abfrage abgebildet.

Wie aber wollen wir hier nur gezielt auf einzelne Elemente zugreifen?

Während das gelegentlich rein mit Zeichenkettenfunktionen erledigt wird, indem man beispielsweise nach dem Namen des gewünschten Elements sucht, ist das in größeren Dokumenten wie diesem hier keine angenehme Aufgabe.

Hier kommen unsere Prozeduren aus den beiden Modulen **mdlJSON** und **mdlJSONDOM** ins Spiel.

```
objJSON.Item("href"):https://api.ebay.com/buy/browse/v1/item_summary/search?q=drone&limit=3&offset=0
objJSON.Item("total"):6081
objJSON.Item("next"):https://api.ebay.com/buy/browse/v1/item_summary/search?q=drone&limit=3&offset=3
objJSON.Item("limit"):3
objJSON.Item("offset"):0
objJson.Item("itemSummaries").Item(1).Item("itemId"): v1|145588822372|0
objJson.Item("itemSummaries").Item(1).Item("title"):
    Refurbished ATOM Fly More Combo GPS Drone 4K Kamera 3-Achsen-Gimbal Kameradrohne
objJson.Item("itemSummaries").Item(1).Item("leafCategoryIds").Item(1): 179697
objJson.Item("itemSummaries").Item(1).Item("categories").Item(1).Item("categoryId"): 179697
objJson.Item("itemSummaries").Item(1).Item("categories").Item(1).Item("categoryName"): Kamera-Drohnen
objJson.Item("itemSummaries").Item(1).Item("categories").Item(2).Item("categoryId"): 625
objJson.Item("itemSummaries").Item(1).Item("categories").Item(2).Item("categoryName"): Foto & Camcorder
objJson.Item("itemSummaries").Item(1).Item("image").Item("imageUrl"):
    https://i.ebayimg.com/images/g/7ssAAOSw18hlvKY6/s-l225.jpg
objJson.Item("itemSummaries").Item(1).Item("price").Item("value"): 309.99
objJson.Item("itemSummaries").Item(1).Item("price").Item("currency"): EUR
objJson.Item("itemSummaries").Item(1).Item("itemHref"): https://api.ebay.com/buy/browse/v1/item/v1%7C145588822372%7C0
objJson.Item("itemSummaries").Item(1).Item("seller").Item("username"): potensic_store
objJson.Item("itemSummaries").Item(1).Item("seller").Item("feedbackPercentage"): 99.3
objJson.Item("itemSummaries").Item(1).Item("seller").Item("feedbackScore"): 3262
objJson.Item("itemSummaries").Item(1).Item("seller").Item("sellerAccountType"): BUSINESS
objJson.Item("itemSummaries").Item(1).Item("condition"): Zertifiziert - Refurbished
objJson.Item("itemSummaries").Item(1).Item("conditionId"): 2000
objJson.Item("itemSummaries").Item(1).Item("thumbnailImages").Item(1).Item("imageUrl"):
    https://i.ebayimg.com/images/g/7ssAAOSw18hlvKY6/s-l1600.jpg
objJson.Item("itemSummaries").Item(1).Item("shippingOptions").Item(1).Item("shippingCostType"): FIXED
objJson.Item("itemSummaries").Item(1).Item("shippingOptions").Item(1).Item("shippingCost").Item("value"): 0.00
objJson.Item("itemSummaries").Item(1).Item("shippingOptions").Item(1).Item("shippingCost").Item("currency"): EUR
objJson.Item("itemSummaries").Item(1).Item("buyingOptions").Item(1): FIXED_PRICE
objJson.Item("itemSummaries").Item(1).Item("epid"): 28063837524
objJson.Item("itemSummaries").Item(1).Item("itemWebUrl"):
    https://www.ebay.de/itm/145588822372?_skw=drone&hash=item21e5c52164:g:7ssAAOSw18hlvKY6&amdata=enc...
objJson.Item("itemSummaries").Item(1).Item("itemLocation").Item("postalCode"): 55***
objJson.Item("itemSummaries").Item(1).Item("itemLocation").Item("country"): DE
objJson.Item("itemSummaries").Item(1).Item("additionalImages").Item(1).Item("imageUrl"):
    https://i.ebayimg.com/images/g/GRIAAOSwzfFlvKam/s-l225.jpg
...
objJson.Item("itemSummaries").Item(1).Item("adultOnly"): Falsch
objJson.Item("itemSummaries").Item(1).Item("unitPricingMeasure"): Einheit
objJson.Item("itemSummaries").Item(1).Item("unitPrice").Item("value"): 309.99
objJson.Item("itemSummaries").Item(1).Item("unitPrice").Item("currency"): EUR
objJson.Item("itemSummaries").Item(1).Item("legacyItemId"): 145588822372
objJson.Item("itemSummaries").Item(1).Item("availableCoupons"): Falsch
objJson.Item("itemSummaries").Item(1).Item("itemCreationDate"): 2024-02-02T08:28:06.000Z
objJson.Item("itemSummaries").Item(1).Item("topRatedBuyingExperience"): Wahr
objJson.Item("itemSummaries").Item(1).Item("priorityListing"): Wahr
objJson.Item("itemSummaries").Item(1).Item("listingMarketplaceId"): EBAY_DE
objJson.Item("itemSummaries").Item(2).Item("itemId"): v1|285778132279|0
```

Listing 2: Ausdrücke für den Zugriff auf den Inhalt des JSON-Dokuments