HAUFE.
Ausgabe 04/2016

ACCESS
IM UNTERNEHMEN

XML-EXPORT MIT UND OHNE VBA

Exportieren Sie Ihre Tabellen mit und ohne VBA-Unterstützung, aber immer mit Bordmitteln, in das XML-Format (ab S. 2).

In diesem Heft:

EVERNOTE

Greifen Sie per VBA auf die Daten des beliebten Tools für das Erstellen und Bereitstellen von Notizen und Daten zu.

SEITE 62

XML-TRANSFORMATION

Erfahren Sie, wie XML-Dokumente mit der Sprache XSLT in die von Ihnen gewünschte Form bringen.

SEITE 27

SUCHEN IN XML-DATEN

Nutzen Sie XPath, um auf die Daten in XML-Dokumenten zuzugreifen oder diese zu suchen.

SEITE 51

Einmal XML mit allem

In der vorherigen Ausgabe haben wir Daten mit eBay ausgetauscht, in dieser Ausgabe greifen wir auf die Daten von Evernote zu, der beliebten Notizverwaltung. Immer ist XML im Spiel. Das ist ein Grund mehr, die Möglichkeiten unter die Lupe zu nehmen, die uns Access und VBA zu diesem Thema bieten. Herausgekommen ist ein Heft, dass sich komplett um dieses Thema dreht – aber sehen Sie selbst!

XML ist nicht nur eine Austauschformat, über das wir mit den meisten Webservices kommunizieren können. XML ist auch ein Thema, das Sie bewegt: In unserer aktuellsten Auswertung zeigte sich, dass unsere Beiträge zu diesem Thema ganz oben auf der Liste der beliebtesten Beiträge gelandet sind. Also wollen wir uns in dieser Ausgabe einmal genau mit diesem Thema befassen. Dabei sind einige hochinteressante Artikel herausgekommen.

Wir beginnen mit den Basics – das bedeutet, dass auch der Poweruser, der sich nicht mit VBA beschäftigen möchte, nicht zu kurz kommt. In diesem Fall schauen wir uns an, welche Möglichkeiten Ihnen Access für den Export der Daten aus den Tabellen Ihrer Datenbank bietet. Dafür steht natürlich ein Assistent zur Verfügung, der bereits einige Möglichkeiten für die Gestaltung der exportieren Datei bietet. Mehr dazu unter dem Titel **XML-Export ohne VBA** ab S. 2.

Ein ähnlicher Titel, ein ähnliches Thema – doch der Beitrag **XML-Export mit VBA** (ab S. 16) geht den Export von Tabellendaten in XML-Dokumente von einer ganz anderen Seite an. Hier beschreiben wir nämlich, welche Schritte Sie unter VBA erledigen müssen, um gleiche oder ähnliche Ergebnisse wie mit dem XML-Export-Assistenten zu erzielen. Eines vorweg: Haben Sie hier einmal die notwendigen Codezeilen geschrieben, reicht ein Mausklick für den Export, während der Assistent natürlich immer wieder neu bedient werden möchte.

Trotz der vielen Einstellungsmöglichkeiten sieht der Export von Daten im XML-Format nicht immer genau so aus, wie Sie sich ihn wünschen. Das ist aber noch lange kein Grund, den Export komplett per VBA zu programmieren und dabei eine Menge Zeit zu investieren. Stattdessen unterziehen wir das erste Resultat des Exports einer kleinen Transformation: Das heißt, dass wir in einer Sprache namens XSLT festlegen, wie genau wir die vorliegenden XML-Elemente umstrukturieren wollen und welche Elemente überhaupt im Zieldokument landen sollen. Wie das gelingt, erfahren Sie im Beitrag **XML-Dokumente transformieren mit XSLT** ab S. 27.

Beim Austausch von Daten im XML-Format etwa mit einem Webservice kann es auch vorkommen, dass Sie Bilder oder Dateien senden oder empfangen wollen. Diese werden dann üblicherweise in kodierter Form in das XML-Dokument eingebunden, dass auch noch die übrigen benötigten Informationen enthält. Das ist gar nicht so unkompliziert, aber mit den Funktionen des Beitrags **Dateien aus XML-Dokumenten speichern und lesen** ab S. 42 wird das zum Kinderspiel.

Beim Zugriff auf die Inhalte eines XML-Dokuments speziell mit VBA hat es eine entscheidende Bedeutung, welche Version die dazu verwendete XML-Bibliothek von Microsoft aufweist. Der Beitrag **XML-Zugriff per VBA: Welche Version?** zeigt, worauf Sie achten müssen und welche Version die richtige für Ihren Anwendungsfall ist.

Im Beitrag **XML-Zugriff per XPath** ab S. 51 kommen wir schließlich zu einem der wichtigsten Sprachen, wenn es um den programmgesteuerten Zugriff auf die Daten eines XML-Dokuments geht: XPath. Hier lernen Sie die gängigen Ausdrücke für den Zugriff auf verschiedene Elemente von XML-Dokumenten kennen.

Und schließlich wollen wir die gelernten Techniken ja auch noch in der Praxis anwenden. Dazu schauen wir uns die Notizzettelverwaltung **Evernote** an, mit der Sie Notizen nicht nur eingeben und verwalten, sondern auch auf beliebigen Entgeräten verfügbar machen. Wir zeigen Ihnen dabei, wie Sie Ihre Notizen von Evernote in eine Access-Datenbank exportieren und auch, wie Sie Notizen auf Basis von Daten einer Access-Tabelle zu Evernote hinzufügen.

Nun aber: Viel Spaß beim Lesen!

Ihr Michael Forster

XML-Export ohne VBA

Für den einen oder anderen Anwendungsfall benötigen Sie die Daten aus den Tabellen Ihrer Datenbank im XML-Format – zum Beispiel, um diese von einer anderen Anwendung aus einzulesen. Access stellt verschiedene Möglichkeiten für den Export von Daten im XML-Format zur Verfügung. Wir schauen uns an, wie dies über die Benutzeroberfläche gelingt und welche Möglichkeiten VBA für diesen Zweck bietet – diesmal ohne Nutzung externer Bibliotheken, also ausschließlich mit Bordmitteln.

XML ist ein wichtiges Austauschformat für Daten. Sehr viele Webservices bieten ihre Daten in diesem Format an oder nehmen diese so entgegen. Daher ist es wichtig zu wissen, welche Möglichkeiten Microsoft Access für den Export der Daten aus den Tabellen der Datenbank zur Verfügung stellt. In diesem Beitrag schauen wir uns daher die in Access eingebauten Möglichkeiten an, also solche, die ohne die Einbindung zusätzlicher Bibliotheken wie etwa **Microsoft XML, vX.0** auskommen. Dazu gehören sowohl die entsprechenden Export-Befehle der Benutzeroberfläche als auch einige VBA-Befehle der Access-Bibliothek.

Export per Benutzeroberfläche

Am einfachsten geht es natürlich über die Befehle der Benutzeroberfläche von Access. Wenn Sie im Ribbon zum **Tab**-Element **Externe Daten** wechseln, finden Sie dort im rechten Bereich eine Reihe von Befehlen für den Export in die verschiedenen Formate vor. Einer davon lautet **XML-Datei** (s. Bild 1).

Klicken Sie diesen Befehl beispielsweise an, während Sie die Tabelle **tblArtikel** (oder eine andere Tabelle) der aktuellen Datenbank im Navigationsbereich markiert haben, erscheint ein Dialog, der einen Dateinamen für die Export-Datei vorschlägt.

Dieser besteht aus dem **Eigene Dokumente**-Ordner des aktuellen Benutzerverzeichnisses und dem Namen der Tabelle mit der Dateiendung **.xml** (s. Bild 2). Diesen Pfad können Sie entweder durch direkte Eingabe per Tastatur

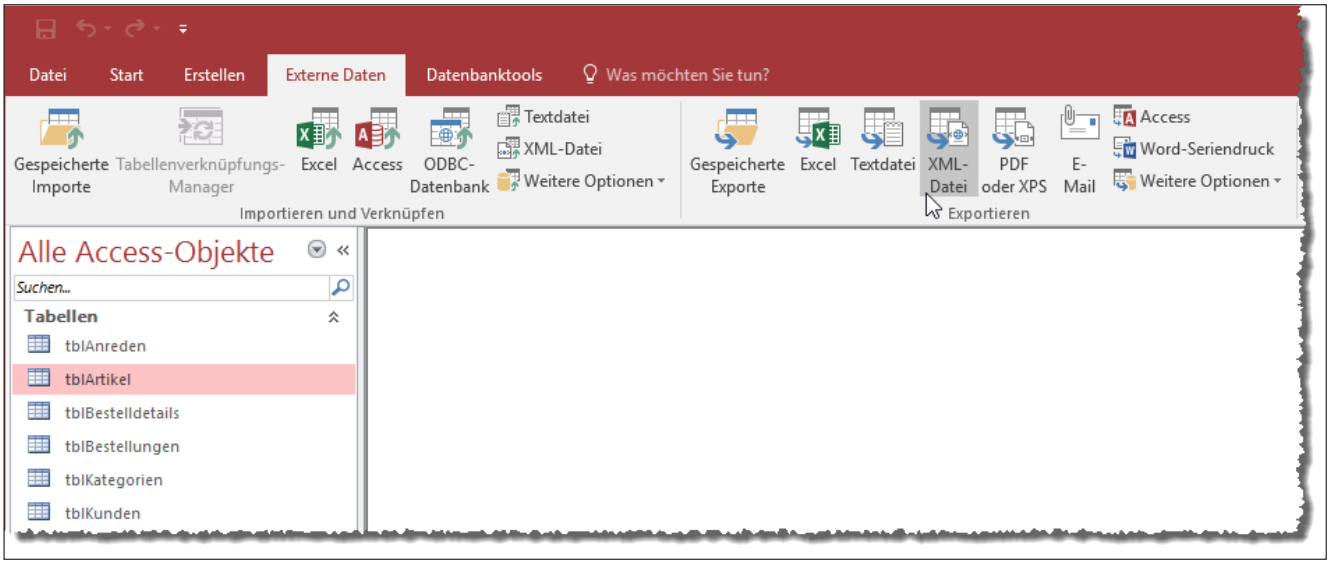

Bild 1: Starten des XML-Exports per Ribbon-Befehl

ändern oder aber über einen **Datei speichern**-Dialog, den Sie mit einem Klick auf **Durchsuchen...** öffnen.

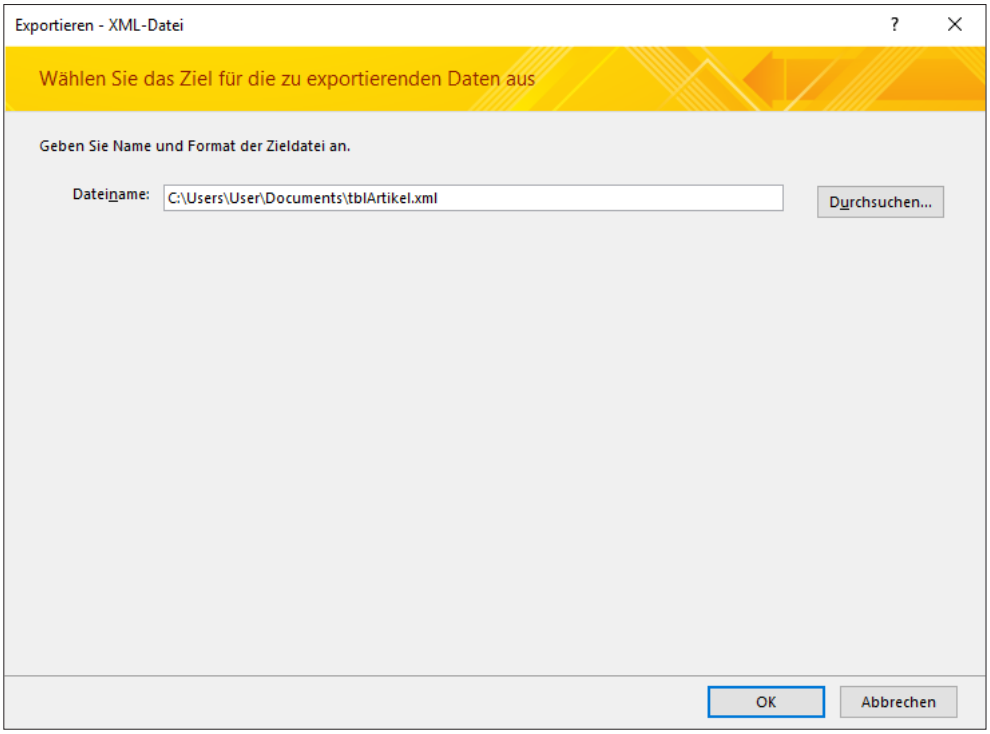

Bild 2: Erster Schritt des Export-Assistenten

Im nächsten Schritt erscheint der Dialog aus Bild 3. Hier legen Sie fest, welche Dateien angelegt werden sollen:

- **Daten (XML)**: XML-Datei mit den eigentlichen Daten
- **Schema der Daten (XSD)**: Schema, das zur Prüfung der Validität der im XML-Dokument enthaltenen Daten verwendet wird
- **Präsentation Ihrer Daten (XSL)**: Schema, das zur Umformung der Daten herangezogen werden kann

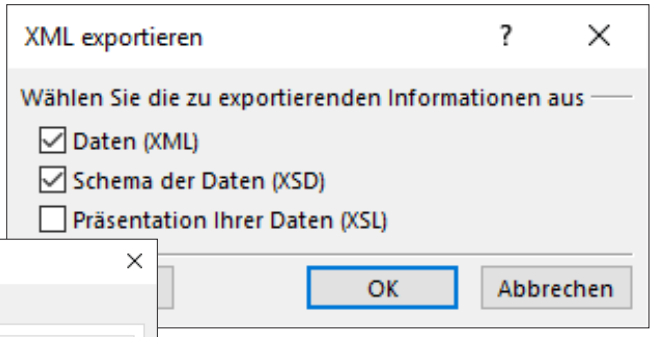

Bild 3: Angabe weiterer Informationen

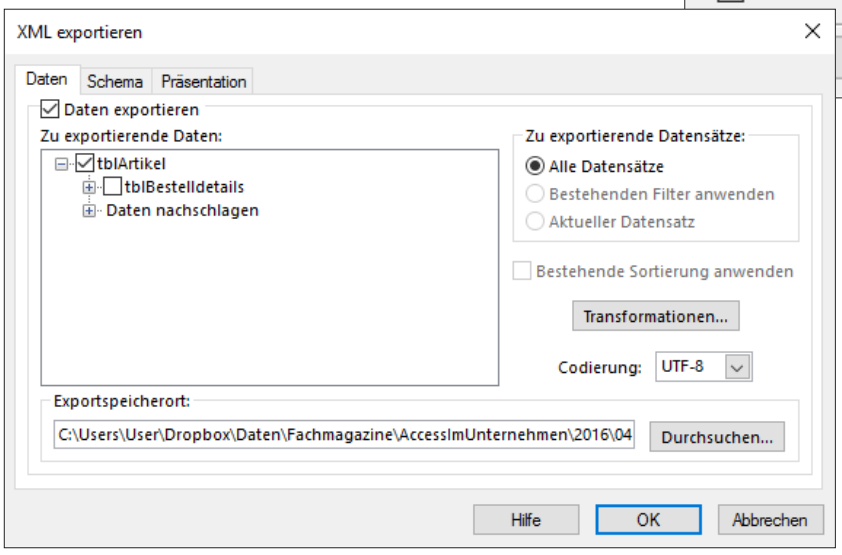

Bild 4: Weitere Optionen für den XML-Export

Um die letzten beiden wollen wir uns in diesem Beitrag nicht kümmern. Aber vielleicht liefert die Schaltfläche **Weitere...** noch interessante Optionen? Sie öffnet einen weiteren Dialog namens **XML exportieren**, der wie in Bild 4 aussieht.

Hier finden Sie für jede der drei in der vorherigen Abbildung möglichen Exportdateien einen Registerreiter mit weiteren

Einstellungen. Für uns ist nur die erste Registerkarte interessant, da wir ja keine XSL- oder XSD-Datei exportieren wollen, sondern nur die reine Daten-Datei.

Im TreeView-Steuerelement mit der Überschrift **Zu exportierende Daten** finden Sie ganz oben die Tabelle, die wir beim Aufrufen des Ribbon-Befehls **Externe Daten|Exportieren|XML-Datei** markiert hatten. Unterhalb davon finden wir zwei verschiedene Arten von Einträgen:

- Namen von Tabellen, in diesem Fall **tblBestelldetails**, und
- den Eintrag **Daten nachschlagen** mit weiteren untergeordneten Tabellennamen, zum Beispiel **tblKategorien** oder **tblLieferanten**.

Die unter **Daten nachschlagen** aufgeführten Tabellen sind solche Tabellen, die von der Haupttabelle **tblArtikel** aus per Fremdschlüsselfeld referenziert werden, aus der also Werte für den aktuellen Datensatz der Tabelle **tblArtikel** ausgewählt werden können.

Bei den direkt untergeordneten Tabellen handelt es sich wiederum um Tabellen, die selbst ein Nachschlagefeld zur Auswahl eines Datensatzes der Haupttabelle enthalten. Im Beispiel der Bestellverwaltung handelt es sich dabei um die Tabelle **tblBestelldetails**, welche über die beiden Fremdschlüsselfelder **ArtikelID** und **BestellungID** die Zuordnung von Artikeln zu einer Bestellung vornimmt.

Zu exportierende Datensätze filtern

Neben der Auswahl der einzuschließenden Daten finden Sie rechts einen Bereich namens **Zu exportierende Datensätze**, der die folgenden drei Optionen bereitstellt:

- **Alle Datensätze**
- **Bestehenden Filter anwenden**
- **Aktueller Datensatz**

Allerdings ist aktuell nur die erste Option aktiviert. Wie aber können wir die anderen beiden Optionen nutzen?

Aktuellen Datensatz exportieren

Die Option **Aktueller Datensatz** ist relativ einfach verfügbar: Dazu öffnen Sie lediglich zuerst die Tabelle mit den zu exportierenden Daten, in diesem Beispiel **tblArtikel**, und klicken dann im Ribbon auf den Befehl **Externe Daten|Exportieren|XML-Datei**. Nach der Angabe der Zieldatei und einem Klick auf die **OK**-Schaltfläche im

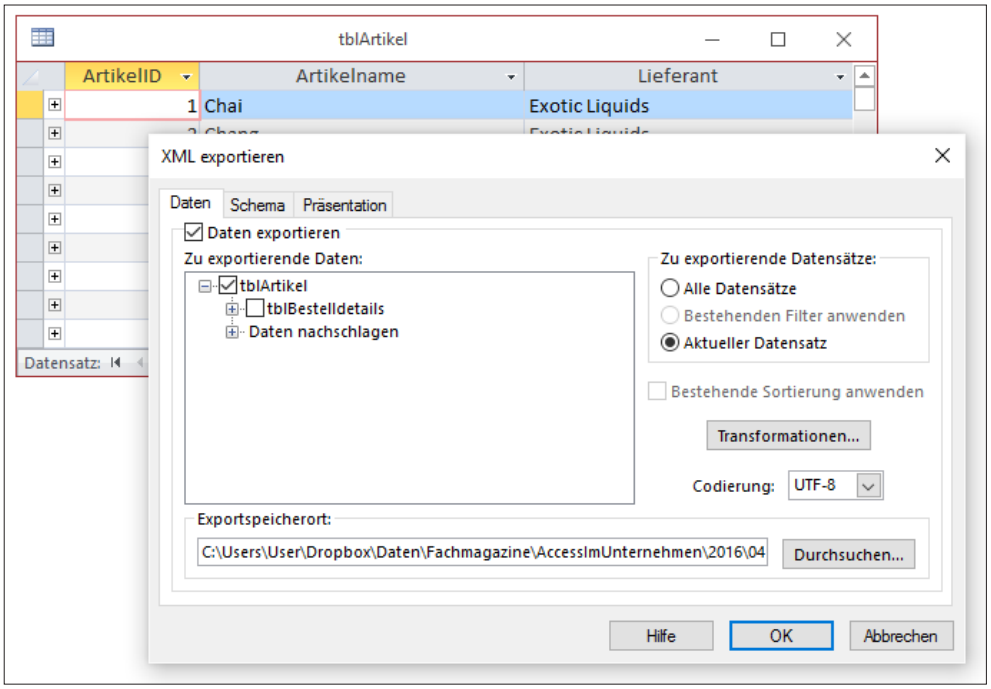

Bild 5: Export eines einzigen Datensatzes

Dialog **Exportieren - XML-Datei** sowie auf die Schaltfläche **Weitere...** des dann erscheinenden Dialogs **XML exportieren** taucht der Dialog mit den weiteren Optionen wie in Bild 5 auf.

Hier finden Sie nun die aktivierte Option **Aktueller Datensatz** vor, die wir nun auswählen. Der Export eines einzigen Datensatzes einer Tabelle sollte eine überschaubare XML-Datei ergeben, anhand derer wir uns einen ersten Überblick über das Ergebnis verschaffen können. Und wie erwartet sieht das Ergebnis wie folgt recht übersichtlich aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<dataroot xmlns:od="urn:schemas-microsoft-com:officedata"
generated="2016-07-02T19:01:14">
<tblArtikel>
<ArtikelID>1</ArtikelID>
<Artikelname>Chai</Artikelname>
<LieferantID>1</LieferantID>
<KategorieID>1</KategorieID>
<Liefereinheit>10 Kartons x 20 Beutel</Liefereinheit>
<Einzelpreis>9</Einzelpreis>
<Lagerbestand>39</Lagerbestand>
<BestellteEinheiten>0</BestellteEinheiten>
<Mindestbestand>10</Mindestbestand>
<Auslaufartikel>0</Auslaufartikel>
</tblArtikel>
</dataroot>
```

Wir erhalten also neben dem Element **dataroot** für den aktuellen Datensatz ein Element des Typs **tblArtikel**, welches für jeden Feldnamen der Tabelle ein untergeordnetes Element enthält, dessen Wert dem Feldwert entspricht.

Gefilterte Datensätze exportieren

Aber vielleicht möchten Sie nicht nur einen bestimmten oder alle Datensätze exportieren, sondern die aktuell per Filter festgelegten Datensätze. Dazu legen wir für die in der Datenblattansicht geöffnete Tabelle einen Filter fest, der zum Beispiel nur die Datensätze liefert, deren Artikelname mit dem Buchstaben **A** beginnt (s. Bild 6).

Wenn Sie nun den Export in das XML-Format wie in den vorherigen Anläufen starten, erscheinen im Dialog alle Optionen unter **Zu exportierende Datensätze** aktiviert (s. Bild 7). Sie können nun also auch einen Export durchführen, der nur die mit A beginnenden Artikel enthält.

Weitere Daten hinzufügen: Lookupdaten

Nun wollen wir Lookupdaten hinzufügen, im Falle der Tabelle **tblArtikel** also die Daten der Tabellen **tblKategorien** und **tblLieferanten**. Nun wird es interessant, denn wir können dies auf mindestens zwei Arten erledigen: indem wir einfach die entsprechenden Werte wie den Kategorienamen oder den Lieferantennamen anstelle der Werte der Felder **KategorieID** oder **LieferantID** exportieren oder indem wir jeweils die kompletten verknüpften Datensätze der beiden Tabellen **tblKategorien** und **tblLieferant** zum Datensatz der Tabelle **tblArtikel** hinzufügen. Für den ersten Fall legen wir einfach eine Abfrage an, welche alle Felder der Tabelle **tblArtikel** mit Ausnahme

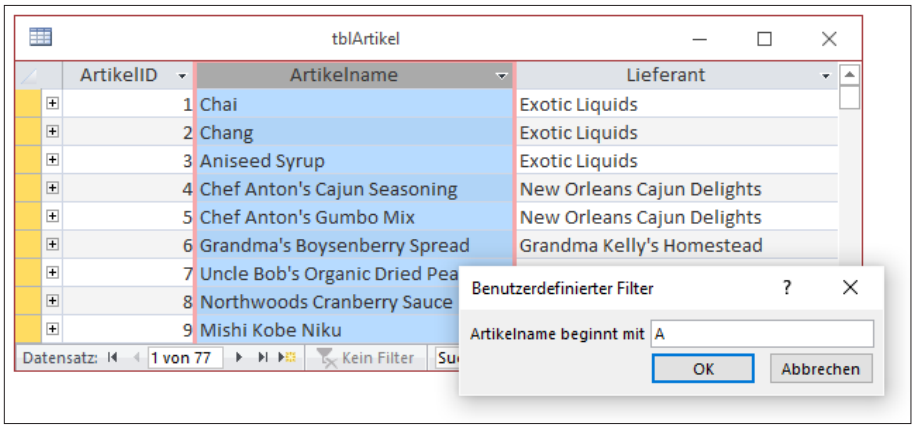

Bild 6: Filtern der Datensätze der Tabelle **tblArtikel**

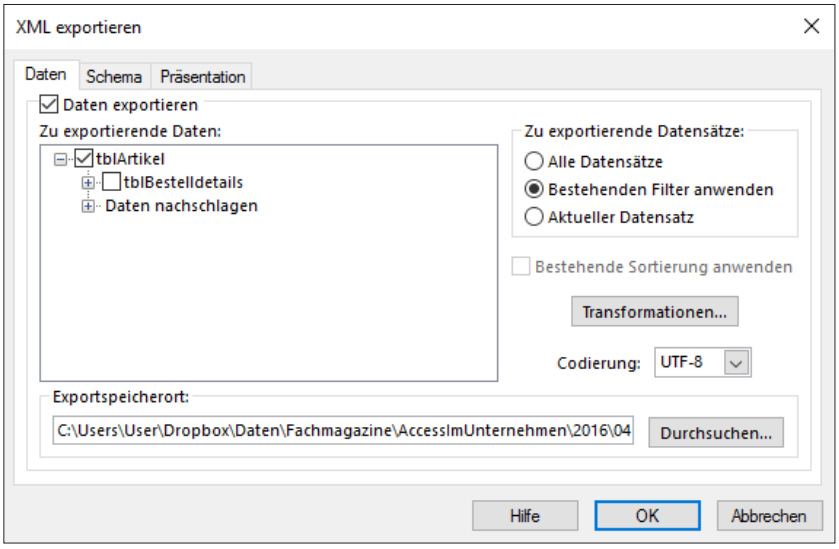

Bild 7: Ausgabe der Daten mit dem aktuell festgelegten Filter

von **KategorieID** und **LieferantID** enthält, dafür aber die Felder **Kategoriename** der Tabelle **tblKategorien** und **Firma** der Tabelle **tblLieferanten**. Die Abfrage sieht etwa wie in Bild 8 aus.

Um den Export zu starten, verwenden wir nun eine alternative Variante zum entsprechenden Ribbon-Eintrag: Diesmal klicken wir mit der rechten Maustaste auf den entsprechenden Eintrag im Navigationsbereich von Access, nämlich auf **qryArtikelMitKategorieUndLieferant**, und wählen aus dem Kontextmenü den Befehl **Exportieren|XML-Datei** aus (s. Bild 9).

Das Ergebnis sieht nun etwas anders aus, als wenn wir direkt die Tabelle **tblArtikel** exportieren. Diesmal erhalten wir statt der Elemente **KategorieID** und **LieferantID** mit den Fremdschlüsselwerten die Elemente **Kategoriename** und **Firma** mit den entsprechenden Bezeichnungen (s. Listing 1).

Für viele Fälle wird dies reichen, aber wenn Sie die Daten des XML-Dokuments beispielsweise in einer anderen Datenbank wieder einsetzen möchten, welche die gleiche Datenstruktur hat, benötigen Sie unter Umständen die Kategorien und Lieferanten inklusive Fremdschlüssel- und Primärschlüsselwerten.

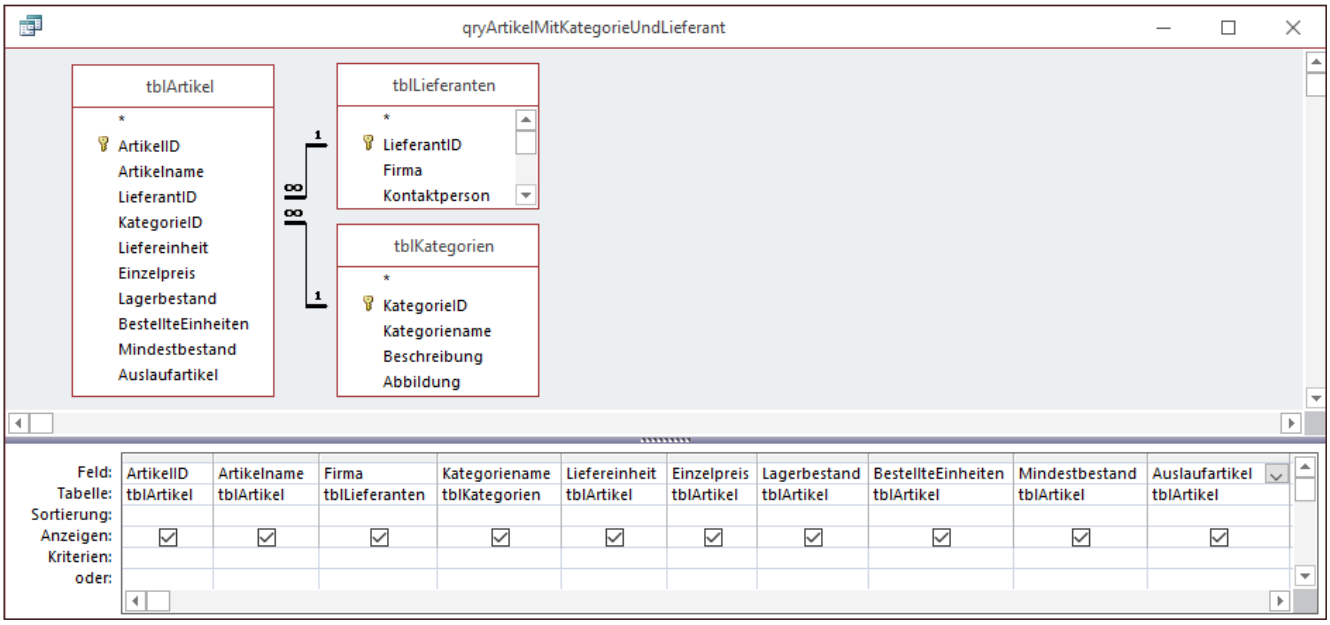

Bild 8: Abfrage für den Export der Daten

Daten verknüpfter Tabellen exportieren

Wenn Sie die Daten verknüpfter Tabellen exportieren wollen, gibt es ebenfalls verschiedene Möglichkeiten. Die erste exportiert die Daten in verschachtelter Form, und zwar so, dass in der ersten Ebene etwa die Daten der Tabelle **tblKategorien** landen und in untergeordneten Elementen jeweils die Datensätze der Tabelle **tblArtikel**. Hierbei ist zu beachten, dass zu jedem **Kategorien**-Element immer das komplette zugeordnete **Artikel**-Element angelegt wird. Diese Variante funktioniert nur, wenn Sie den Export ausgehend von einer Tabelle starten, welche das an der Beziehung beteiligte Primärschlüsselfeld enthält und wenn die untergeordnete Tabelle das Fremdschlüsselfeld beisteuert. Wenn Sie also etwa die Artikel als übergeordnete Elemente und darunter die Kategorien und Lieferanten ausgeben wollen, gelingt dies über den Assistenten nicht. Dies können Sie allerdings per VBA erledigen, wie Sie im Beitrag **XML-Export mit VBA** (**www.access-im-unternehmen.de/1046**) erfahren werden.

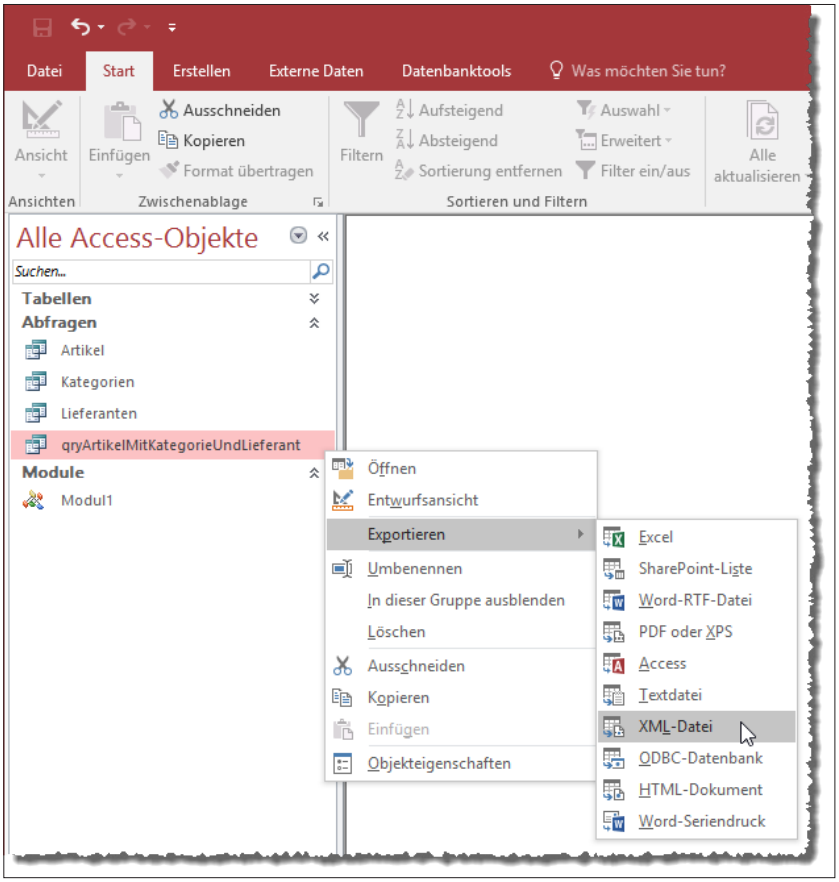

Bild 9: Starten des Exports per Kontextmenü

Die zweite exportiert einfach die Inhalte der betroffenen Tabellen in das XML-Dokument, und zwar so, dass zuerst alle Artikeldatensätze in Form entsprechender Elemente und dann alle Kategorie- und Lieferantendatensätze in die XML-Datei geschrie-

```
<?xml version="1.0" encoding="UTF-8"?>
<dataroot xmlns:od="urn:schemas-microsoft-com:officedata" generated="2016-07-03T12:38:01">
    <qryArtikelMitKategorieUndLieferant>
        <ArtikelID>1</ArtikelID>
        <Artikelname>Chai</Artikelname>
        <Firma>Exotic Liquids</Firma>
        <Kategoriename>Getränke</Kategoriename>
        <Liefereinheit>10 Kartons x 20 Beutel</Liefereinheit>
        <Einzelpreis>9</Einzelpreis>
        <Lagerbestand>39</Lagerbestand>
        <BestellteEinheiten>0</BestellteEinheiten>
        <Mindestbestand>10</Mindestbestand>
        <Auslaufartikel>0</Auslaufartikel>
    </qryArtikelMitKategorieUndLieferant>
    ...
</dataroot>
```

Listing 1: Export mit direkter Angabe der Kategorie und der Lieferanten-Firma

ben werden. Die Beziehung zueinander kann dann über die Fremdschlüsselfelder der Tabelle **tblArtikel** ermittelt werden, die ja ebenfalls in die **Artikel**-Elemente übernommen werden.

Diese Variante wird vom Assistenten automatisch gewählt, wenn die Tabelle, für die Sie den Export angestoßen haben, die Fremdschlüsselfelder zur Verknüpfung mit den Werten der Primärschlüsselfelder der verknüpften Tabellen enthält.

Wir schauen uns nun beide Varianten an.

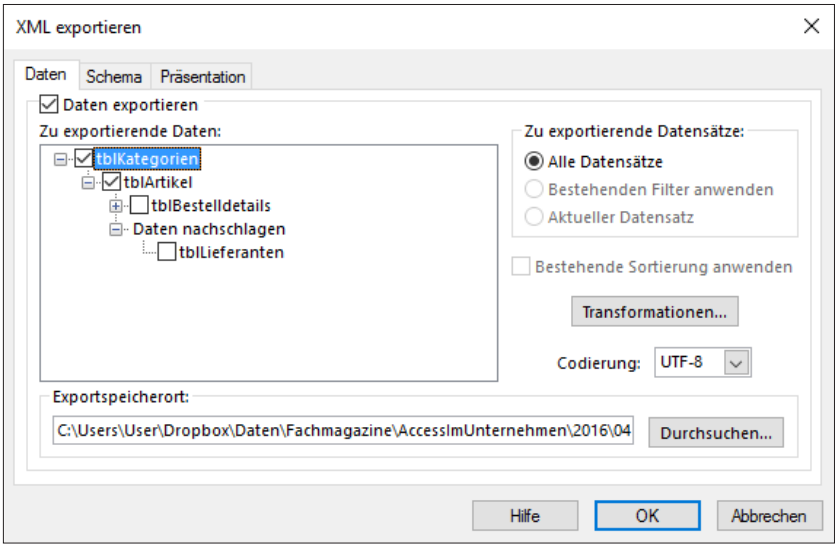

Bild 10: Konfiguration für den Export als verschachtelte Elemente

```
<?xml version="1.0" encoding="UTF-8"?>
<dataroot xmlns:od="urn:schemas-microsoft-com:officedata" generated="2016-07-03T13:22:52">
    <tblKategorien>
        <KategorieID>1</KategorieID>
        <Kategoriename>Getränke</Kategoriename>
        <Beschreibung>Alkoholfreie Getränke, Kaffee, Tee, Bier</Beschreibung>
        <Abbildung>FRwvAAIAAAANAA4AFAAhAP////9CaXRtYXAgSW1hZ2UA...</Abbildung>
        <tblArtikel>
            <ArtikelID>1</ArtikelID>
            <Artikelname>Chai</Artikelname>
            <LieferantID>1</LieferantID>
            <KategorieID>1</KategorieID>
            ...
        </tblArtikel>
        <tblArtikel>
            <ArtikelID>2</ArtikelID>
            <Artikelname>Chang</Artikelname>
            ...
        </tblArtikel>
        ...
    </tblKategorien>
    <tblKategorien>
        <KategorieID>2</KategorieID>
        <Kategoriename>Gewürze</Kategoriename>
        ...
    </tblKategorien>
    ...
</dataroot>
```

Listing 2: Export verschachtelter Daten

Daten verschachtelt speichern

Bei der ersten Variante markieren wir die Tabelle **tblKategorien** und starten dann den Export entweder über den entsprechenden Ribbon- oder Kontextmenü-Eintrag. In den erweiterten Optionen wählen wir nun die Tabelle **tblKategorien** sowie die Tabellen **tblArtikel** aus (s. Bild 10).

Das Ergebnis sehen Sie in Listing 2. Es gibt zu jedem Datensatz der Tabelle **tblKategorien** ein Element namens **tblKategorien**, das die Felder und Feldwerte als Unterelemente enthält – also etwa **KategorieID**, **Kategoriename** et cetera. Das Element **Abbildung** enthält den Binärcode

XML-Export mit VBA

Access stellt verschiedene Möglichkeiten für den Export von Daten im XML-Format zur Verfügung. Die Variante, mit der Sie Tabellen über die Benutzeroberfläche in das XML-Format exportieren können, können Sie natürlich auch per VBA nutzen – und zwar mit dem Befehl »ExportXML« des »Application«-Objekts. Dieser Beitrag zeigt, welcher Parameter welcher Option entspricht und welche zusätzlichen Features Sie per VBA erhalten.

Der Export von Daten aus Access heraus in ein XML-Dokument erfordert immer zumindest die Angabe einer Tabelle sowie der Zieldatei. Außerdem können Sie nicht nur Tabellen, sondern auch alle anderen Objekttypen von Access im XML-Format exportieren – also müssen Sie auch noch den Objekttyp angeben. Dementsprechend lautet die minimale Version eines Aufrufs der Methode **ExportXML** des **Application**-Objekts wie folgt:

```
Application.ExportXML acExportTable, "tblAnreden", _
                      CurrentProject.Path & "\tblAnreden.xml"
```

Der erste Parameter erwartet den Objekttyp, in diesem Fall **acExportTable** (bei einer Abfrage würden Sie etwa **acExportQuery** nutzen). Der zweite gibt den Namen des zu exportierenden Objekts an (hier **tblAnreden**) und der dritte den Pfad zu der zu erzeugenden Datei. Hier haben wir **CurrentProject.Path** zur Ermittlung des Verzeichnisses der aktuellen Datenbank genutzt, damit die XML-Datei direkt in diesem Verzeichnis angelegt wird – und zwar unter dem Namen **tblAnreden.xml**. Damit können Sie den ersten einfachen Export per Einzeiler erledigen und sich das Ergebnis ansehen:

```
<?xml version="1.0" encoding="UTF-8"?>
<dataroot ...>
  <tblAnreden>
    <AnredeID>1</AnredeID>
    <Anrede>Herr</Anrede>
  </tblAnreden>
  <tblAnreden>
    <AnredeID>2</AnredeID>
    <Anrede>Frau</Anrede>
  </tblAnreden>
</dataroot>
```

Mit diesem Einzeiler sparen Sie sich einige Mausklicks, die Sie bei der Benutzung des Export-Assistenten über die Benutzeroberfläche hätten ausführen müssen – zumindest, wenn Sie diese Anweisung einmal in eine Prozedur eingetragen haben und dann aufrufen.

Nun bietet der Assistent noch einige weitere Möglichkeiten, die wir uns auch schon im Beitrag **XML-Export ohne VBA** (**www.access-im-unternehmen.de/1045**) angesehen haben. Diese wollen wir nun auch mit VBA abbilden.

Zu exportierende Daten einschränken

In manchen Fällen möchten Sie vielleicht nicht alle Daten einer Tabelle exportieren, sondern nur eine Teilmenge. Bei Benutzung des Assistenten musste dafür die Tabelle mit den zu exportierenden Daten geöffnet sein.

Wenn Sie einen einzelnen Datensatz exportieren wollten, mussten Sie diesen zunächst markieren und dann den Assistenten starten, der nun die Option **Aktueller Datensatz** bereitstellte. Wenn Sie zuvor einen Filter für die Tabelle oder Abfrage festgelegt haben, konnten Sie auch mit der Option **Bestehenden Filter anwenden** arbeiten.

Nun sollen Sie vor dem Exportieren der Tabellendaten nicht erst die Tabelle öffnen und den Filter definieren. Dies erledigen Sie nämlich mit einem einfachen Parameter der **ExportXML**-Methode. Im Gegensatz zu den bereits

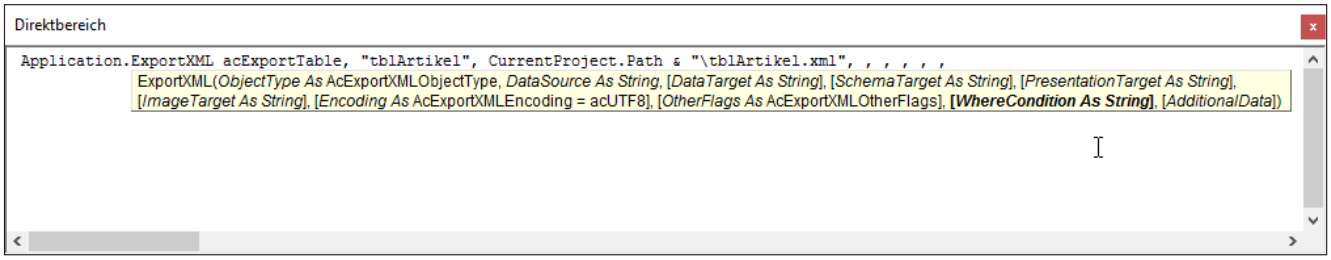

Bild 1: Eingabe der Parameter für den Export mit einer Bedingung

im vorherigen Beispiel vorgestellten drei Parametern befindet sich der nun gesuchte Parameter relativ weit hinten in der Parameterliste, sodass Sie entweder eine entsprechende Anzahl Kommata setzen, um zum gewünschten Parameter zu gelangen oder den Parameter benennen.

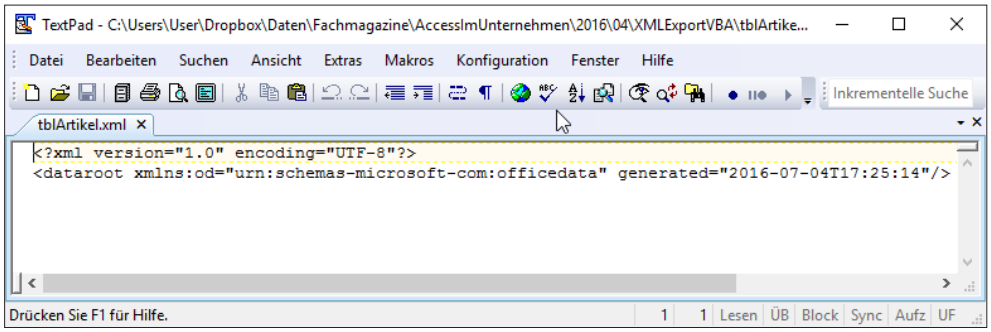

Bild 2: XML-Export ohne Daten

Wir wollen in diesem Fall alle Datensätze der Tabelle **tblArtikel** exportieren, deren Artikelname mit **A** beginnt.

Im ersten Anlauf fügen wir die entsprechende Anzahl Kommata ein, wobei uns IntelliSense wie in Bild 1 unterstützt. Der Befehl sieht dann wie folgt aus:

```
Application.ExportXML acExportTable, "tblArtikel", ,
CurrentProject.Path & "\tblArtikel.xml", , , , , ,
"Artikelname LIKE 'A*'"
```

Das Ergebnis überrascht: Es liefert nur den Kopf des Exportdokuments, aber keinerlei Artikeldaten (s. Bild 2). Eine kurze Prüfung der zu exportierenden Tabelle **tblArtikel** zeigt, dass dort durchaus Artikel vorliegen, deren **Artikelname** mit **A** beginnt. Ein Blick in die Online-Dokumentation liefert auch keine Hinweise auf einen Fehler in der Formulierung unserer Bedingung.

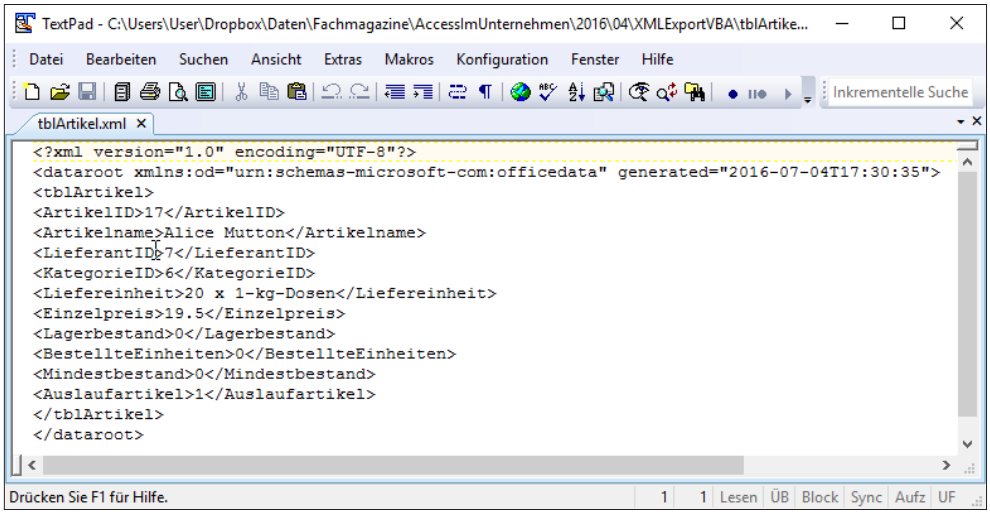

Bild 3: XML-Export mit Filter nach Artikelname

Also experimentieren wir einfach etwas herum und finden heraus, dass es mit kompletten Vergleichswerten funktioniert – also etwa mit folgendem Aufruf:

```
Application.ExportXML acExportTable, "tblArtikel", ⤷
      CurrentProject.Path & "\tblArtikel.xml", , , , , , ⤷
                        "Artikelname LIKE 'Alice Mutton'"
```

Wie Bild 3 zeigt, funktioniert das Setzen einer Bedingung also grundsätzlich.

Woran also hapert es bei Verwendung eines Platzhalters wie dem Sternchen (*)?

Da wir wissen, dass in anderen SQL-Dialekten durchaus auch mal andere Platzhalter-Zeichen zum Einsatz kommen, wobei statt des Sternchens beispielsweise das Prozentzeichen (%) verwendet wird, probieren wir es einfach einmal so aus:

```
Application.ExportXML acExportTable, "tblArtikel", ⤷
      CurrentProject.Path & "\tblArtikel.xml", , , , , , ⤷
                                "Artikelname LIKE 'A%'"
```

Und siehe da: Es funktioniert! Hier ist die Ausgabe für diese Anweisung mit den beiden betroffenen Artikel-Datensätzen:

```
<?xml version="1.0" encoding="UTF-8"?>
<dataroot ...>
  <tblArtikel>
    <ArtikelID>3</ArtikelID>
    <Artikelname>Aniseed Syrup</Artikelname>
    <LieferantID>1</LieferantID>
    <KategorieID>2</KategorieID>
    <Liefereinheit>12 x 550-ml-Flaschen</Liefereinheit>
    <Einzelpreis>5</Einzelpreis>
    <Lagerbestand>13</Lagerbestand>
    <BestellteEinheiten>70</BestellteEinheiten>
    <Mindestbestand>25</Mindestbestand>
    <Auslaufartikel>0</Auslaufartikel>
  </tblArtikel>
  <tblArtikel>
    <ArtikelID>17</ArtikelID>
    <Artikelname>Alice Mutton</Artikelname>
    <LieferantID>7</LieferantID>
    <KategorieID>6</KategorieID>
    <Liefereinheit>20 x 1-kg-Dosen</Liefereinheit>
    <Einzelpreis>19.5</Einzelpreis>
    <Lagerbestand>0</Lagerbestand>
    <BestellteEinheiten>0</BestellteEinheiten>
    <Mindestbestand>0</Mindestbestand>
    <Auslaufartikel>1</Auslaufartikel>
  </tblArtikel>
</dataroot>
```

Wenn Sie nicht mit den vielen Kommata durcheinanderkommen wollen, können Sie, wie oben erwähnt, mit einem benannten Parameter arbeiten. Der Aufruf sieht dann so aus:

```
Application.ExportXML acExportTable, ⤷
  "tblArtikel", CurrentProject.Path & "\tblArtikel.xml", ⤷
                    WhereCondition:="Artikelname LIKE 'A%'"
```

Sie können auch alle Parameter benennen:

```
Application.ExportXML ObjectType:=acExportTable, ⤷
    DataSource:="tblArtikel", ⤷
    DataTarget:=CurrentProject.Path & "\tblArtikel.xml", ⤷
    WhereCondition:="Artikelname LIKE 'A%'"
```

Parameter des Aufrufs

Einige Parameter haben Sie nun schon kennen gelernt, dennoch hier die Übersicht aller möglichen Parameter:

- **ObjectType**: Konstante für den Objekttyp, hier sinnvollerweise **acExportTable** oder **acExportQuery**. Andere Werte wie **acExportForm** oder **acExportReport** erscheinen auf den ersten Blick sinnlos, aber haben doch ihre Daseinsberechtigung: Sie exportieren damit nämlich nicht etwa eine Formular- oder Berichtsdefinition, sondern die darin angezeigten Daten.

- **DataSource**: Name der Datenquelle, also etwa der Tabelle oder der Abfrage.

- **DataTarget**: Pfad zur Zieldatei für den Export. Vorhandene Dateien werden ohne Warnung überschrieben.
- **SchemaTarget**: Ziel für die erstellte Schema-Datei.
- **PresentationTarget**: Ziel für eine Datei mit Informationen zur Präsentation der Daten.
- **ImageTarget**: Pfad für eventuell zu exportierende Bilder. Im Test konnten wir damit keine Bilder exportieren.

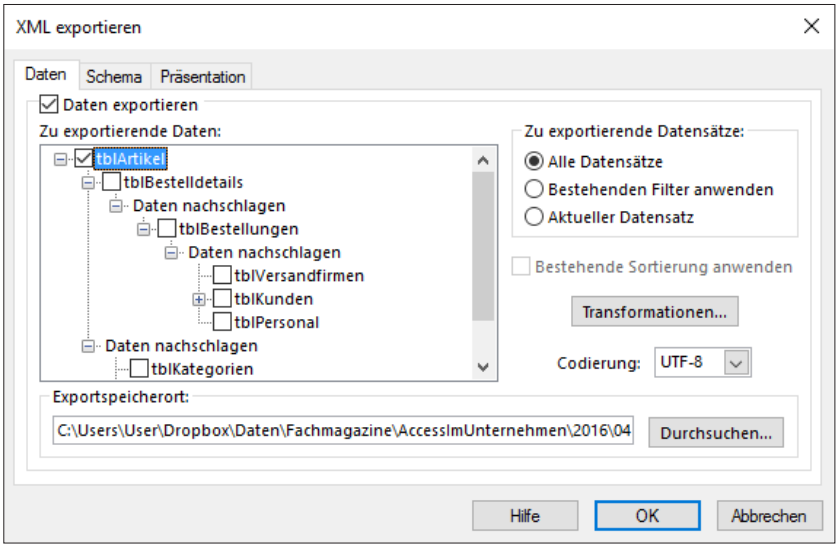

Bild 4: Auswahl der zu exportierenden Tabellen per TreeView und Kontrollkästchen

- **Encoding**: Konstante für die Kodierung des XML-Dokuments (**acUTF16** oder **acUTF32**)
- **OtherFlags**: Kombination aus keinem, einem oder mehreren der folgenden Konstanten (hier nur die wichtigsten – mehr in der Onlinehilfe): **acEmbedSchema** (**1**) – sorgt dafür, dass Schemainformationen in das XML-Dokument geschrieben werden, **acExcludePrimaryKeyAndIndexes** (**2**) – schließt Primärschlüssel und Indizes beim Export aus, **acExportAllTableAndFieldProperties** (**32**) – exportiert Eigenschaften von Tabellen und Feldern mit dem Schema.
- **WhereCondition**: Bedingung für die auszugebenden Datensätze
- **AdditionalData**: Angabe weiterer Tabellen, die exportiert werden sollen. Ermöglicht die Auswahl der Tabellen, wie es beim Assistenten per TreeView möglich ist (s. Bild 4). **AdditionalData** bietet allerdings noch mehr Möglichkeiten.

Daten aus mehreren Tabellen

Den Parameter **AdditionalData** schauen wir uns als Nächstes an. Onlinehilfe und Objektkatalog schweigen sich ebenso wie Intellisense über den Datentyp des zu übergebenden Wertes oder Objekts aus – hier taucht lediglich der Datentyp **Variant** auf, der ja verschiedene Typen aufnehmen kann. Schließlich hilft eine Suche nach dem Schlüsselwort **AdditionalData** im Objektkatalog weiter (s. Bild 5). Hier erfahren wir, dass es sich um eine eigene Klasse handelt, der wir wohl auch noch rekursiv

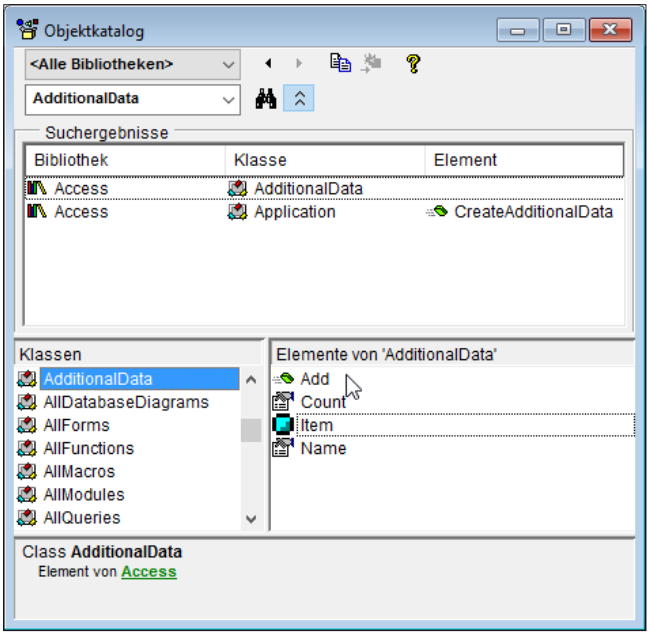

Bild 5: Der Objektkatalog liefert Informationen über den Datentyp von **AdditionalData**.

XML-Dokumente transformieren mit XSLT

Mit den eingebauten Funktionen für den Export von Daten aus Tabellen und Abfragen in das XML-Format können Sie bereits recht gute Ergebnisse erzielen. Natürlich können Sie aber nicht komplett steuern, wie das Zieldokument später aussehen wird. Je nach den Anforderungen der Anwendung, die das XML-Dokument weiterverarbeiten soll, sind noch Änderungen notwendig. Hier tritt die Transformation von XML-Dokumenten auf den Plan: Mit einer sogenannten .xslt-Datei legen Sie fest, wie ein Dokument in ein anderes umgeformt werden soll. Den vollständigen Vorgang steuern Sie dann per VBA-Prozedur. Dieser Beitrag liefert die Grundlagen der Transformation und die notwendigen VBA-Techniken.

Voraussetzungen

Wenn Sie mit den eingebauten Export-Funktionen einfach nur XML-Dokumente auf Basis von Tabellen oder Abfragen erstellen wollen, benötigen Sie dazu keine weiteren Bibliotheken. Auch eine Transformation eines exportierten XML-Dokuments über die entsprechende Funktion der Benutzeroberfläche (zum Beispiel über den Ribbon-Eintrag **Externe Daten|Exportieren|XML-Datei**) können Sie ohne weitere Hilfsmittel durchführen – Sie können einfach im Assistenten angeben, welche **.xslt**-Datei die Vorgaben für die Transformation enthält. Das erzeugte XML-Dokument wird dann nach dem Export automatisch auf Basis dieser Datei transformiert. Sollten Sie jedoch einen Export mit der Methode **ExportXML** des **Application**-Objekts durchführen wollen, können Sie die **.xslt**-Datei dort nicht etwa per Parameter angeben. Sie exportieren die Daten dort erst in ein XML-Dokument und führen dann die Transformation durch. Für diese Transformation benötigen Sie Objekte und Methoden der Bibliothek **Microsoft XML, vx.0**, wobei Sie die jeweils aktuellste Version dieser Bibliothek wählen sollten (s. Bild 1).

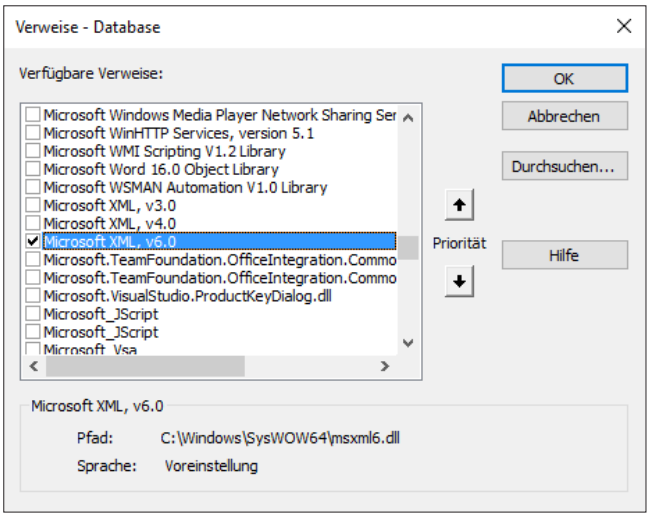

Bild 1: Verweis auf die Bibliothek **Microsoft XML, v6.0**

Transformations-Grundlagen

XML-Dokumente bestehen aus Daten und aus Elementen zur Strukturierung dieser Daten. Damit lassen sich beispielsweise die Daten aus verknüpften Tabellen einer Datenbank hierarchisch darstellen – zum Beispiel haben Sie eine Kategorien-Tabelle und eine Artikel-Tabelle, wo jedem Artikel eine Kategorie zugewiesen ist. In einem XML-Dokument könnten Sie nun die Kategorien und Artikel hierarchisch strukturiert speichern:

```
<?xml version="1.0" encoding="UTF-8"?>
<Kategorien>
  <Kategorie>
    <KategorieID>1</KategorieID>
    <Kategoriename>Kategorie 1</Kategoriename>
    <Artikel>
      <ArtikelID>1</ArtikelID>
      <Artikelname>Artikel 1</Artikelname>
    </Artikel>
    <Artikel>
      <ArtikelID>2</ArtikelID>
      <Artikelname>Artikel 2</Artikelname>
```

```
    </Artikel>
  </Kategorie>
  <Kategorie>
    <KategorieID>2</KategorieID>
    <Kategoriename>Kategorie 2</Kategoriename>
    <Artikel>
      <ArtikelID>3</ArtikelID>
      <Artikelname>Artikel 3</Artikelname>
    </Artikel>
    <Artikel>
      <ArtikelID>4</ArtikelID>
      <Artikelname>Artikel 4</Artikelname>
    </Artikel>
  </Kategorie>
</Kategorien>
```

Die Elemente aus solch einem XML-Dokument können Sie mit einer entsprechenden **.xslt**-Datei in beliebiger Form umstrukturieren, also transformieren.

Dazu sind nur wenige Schritte nötig:

- Sie benötigen ein Objekt des Typs **DOMDocument** (oder **DOMDocument60**, je nach verwendeter Typ-Bibliothek – in unserem Fall **Microsoft XML, v6.0** und **DOMDocument60**), das Sie mit dem Inhalt des zu transformierenden XML-Dokuments füllen.
- Ein weiteres Objekt des gleichen Typs füllen Sie mit dem Inhalt der **.xslt**-Datei.
- Schließlich brauchen Sie noch ein drittes **DOMDocument**-Objekt, in welchem die transformierte Datei landet.
- Für das erste **DOMDocument**-Objekt führen Sie die Methode **transformNodeToObject** aus, dem Sie das zweite und das dritte **DOMDocument**-Objekt als Parameter übergeben.

```
Public Function Transformieren(strQuelle As String, strXSLT As String, strZiel As String, _
        Optional strFehler As String) As Long
    Dim objQuelle As MSXML2.DOMDocument60
    Dim objXSLT As MSXML2.DOMDocument60
    Dim objZiel As MSXML2.DOMDocument60
    Set objQuelle = New MSXML2.DOMDocument60
    objQuelle.Load strQuelle
    If objQuelle.parseError = 0 Then
        Set objXSLT = New MSXML2.DOMDocument60
        objXSLT.Load strXSLT
        If objXSLT.parseError = 0 Then
            Set objZiel = New MSXML2.DOMDocument60
            objQuelle.transformNodeToObject objXSLT, objZiel
            objZiel.Save strZiel
        Else
            Transformieren = objXSLT.parseError.errorCode
            strFehler = ".xslt-datei: " & vbCrLf & strXSLT & vbCrLf & objXSLT.parseError.reason
        End If
    Else
        Transformieren = objQuelle.parseError.errorCode
        strFehler = "Quelldatei: " & vbCrLf & strQuelle & vbCrLf & objQuelle.parseError.reason
    End If
End Function
```

Listing 1: Prozedur für die einfache Transformation eines XML-Dokuments

Für diese Anweisungen haben wir eine einfache Prozedur geschrieben, der Sie die Pfade für die drei beteiligten XML-Dokumente per Parameter übergeben können. Diese Prozedur heißt **Transformieren** und sieht wie in Listing 1 aus. Mit dem ersten Parameter übergeben Sie den Pfad zu der zu transformierenden XML-Datei, mit dem zweiten den Pfad zur **.xslt**-Datei und mit dem dritten den Pfad, unter dem die transformierte Datei gespeichert werden soll.

Der vierte Parameter ist ein optionaler Rückgabeparameter, der von der Funktion mit einer Fehlermeldung gefüllt wird, wenn ein Fehler auftritt. Die Funktion deklariert dann die drei benötigten Objekte vom Typ **DOMDocument60**.

Dann erstellt sie das erste Objekt **objQuelle** mit der **New**-Anweisung und füllt es mit der **Load**-Methode. Die **Load**-Methode erwartet den Pfad zu einer XML-Datei, den wir mit dem Parameter **strQuelle** übergeben. Hierbei kann es geschehen, dass ein Fehler auftritt – beispielsweise, dass unter dem mit **strQuelle** angegebenen Pfad gar keine Datei gefunden werden kann. Tritt ein solcher Fehler auf, liefert die Eigenschaft **parseError** von **objQuelle** einen Wert ungleich **0**. Dies prüfen wir in einer **If...Then**-Bedingung, deren **Else**-Teil gegebenenfalls die Fehlermeldung in den Rückgabeparameter **strFehler** schreibt – samt Angabe der fehlerhaften Datei. Außerdem weist die Funktion dem Rückgabewert der Funktion die Fehlernummer zu.

Tritt kein Fehler auf, erstellt die Funktion das zweite Objekt **objXSLT** und füllt es mit dem Inhalt der mit **strXSLT** angegebenen **.xslt**-Datei – wieder unter Verwendung der **Load**-Methode. Auch hier eventuell auftretende Fehler werden entsprechend behandelt.

Ist bis hierher kein Fehler aufgetreten, erstellt die Prozedur das **DOMDocument60**-Objekt für das transformierte XML-Dokument. Die Transformation selbst erfolgt dann durch die Methode **transformNodeToObject** des Objekts **objQuelle**. Dieser übergeben wir Verweise auf die **DOMDocument60**-Objekte mit der **.xslt**-Datei und der Zieldatei als Parameter.

Nach erfolgter Transformation speichern wir den Inhalt des neu erzeugten und gefüllten XML-Dokuments aus **objZiel** mit der **Save**-Methode in der mit dem Parameter **strZiel** angegebenen XML-Datei.

Aufruf der Funktion »Transformieren«

Der Aufruf dieser Funktion kann, wenn Sie das zu transformierende XML-Dokument und das **.xslt**-Dokument bereits auf der Festplatte abgelegt haben, ganz einfach wie folgt geschehen:

```
Transformieren <Quelldokument>, <XSLT-Dokument>, ⤸
                                        <Zieldokument>
```

So erhalten Sie zwar keinen Zugriff auf eine eventuelle Fehlermeldung, aber es ist der schnellste Weg, um die Transformation durchzuführen, wenn die Dateien im Dateisystem liegen.

Wenn Sie sich die Funktionsweise inklusive Fehlermeldung ansehen möchten, können Sie die Methode aus Listing 2 nutzen – gemeinsam mit den Tabellen der Beispieldatenbank zu diesem Beitrag. Die Methode deklariert zunächst die benötigten Variablen. Dann exportiert sie ein XML-Dokument auf Basis der Tabellen **tblKategorien** und **tblArtikel**, wobei die Artikeldaten den Kategorie-Elementen untergeordnet werden sollen (wie dies im Detail funktioniert, lesen Sie im Beitrag **XML-Export mit VBA**, **www.access-im-unternehmen.de/1046**).

Der Export landet in der Datei **KategorienUndArtikel_Untransformiert.xml**. Die **.xslt**-Datei zu diesem Beispiel finden Sie in den Download-Dateien. Sie heißt **KategorienUndArtikel.xslt** und sollte sich im gleichen Verzeichnis wie die Datenbank befinden. Schließlich legt die Prozedur noch den Namen der Zieldatei fest, die unter **KategorienUndArtikel_Transformiert.xml** gespeichert werden soll. Die drei Variablen **strQuelle**, **strXSLT** und **strZiel** werden samt der Variablen **strFehler** für den optionalen Parameter an die Funktion **Transformieren** übergeben. Sollte hier einer der oben erläuterten Fehler auftreten, liefert

```
Public Sub TestTransformieren()
    Dim strQuelle As String
    Dim strXSLT As String
    Dim strZiel As String
    Dim strFehler As String
    Dim lngFehler As Long
    Dim objAdditionalData As AdditionalData
    Set objAdditionalData = Application.CreateAdditionalData
    objAdditionalData.Add "tblArtikel"
    strQuelle = CurrentProject.Path & "\KategorienUndArtikel_Untransformiert.xml"
    Application.ExportXML acExportTable, "tblKategorien", strQuelle, AdditionalData:=objAdditionalData
    strXSLT = CurrentProject.Path & "\KategorienUndArtikel.xslt"
    strZiel = CurrentProject.Path & "\KategorienUndArtikel_Transformiert.xml"
    lngFehler = Transformieren(strQuelle, strXSLT, strZiel, strFehler)
    If Not lngFehler = 0 Then
        MsgBox strFehler
    End If
End Sub
```

Listing 2: Aufruf der Funktion **Transformieren** mit Beispieldaten

diese einen Wert ungleich **0** zurück, was zur Ausgabe der mit **strFehler** zurückgegebenen Fehlermeldung per **MsgBox**-Anweisung führt. Anderenfalls finden Sie nun in der Datei **KategorienUndArtikel_Transformiert.xml** das transformierte XML-Dokument vor. Sie können sich das Beispiel vorab anhand der Beispieldaten anschauen, in den folgenden Abschnitten erläutern wir die einzelnen Elemente einer **.xslt**-Datei.

XSLT

XSLT ist die Sprache, mit der Transformationen von XML-Dateien in andere XML-Dateien oder auch HTML-Dateien durchgeführt werden. Dabei greifen Sie über eine spezielle weitere Sprache namens **XPath** auf das oder die gewünschten Elemente des zu transformierenden XML-Dokuments zu und überführen die kompletten Elemente oder auch nur deren Inhalt in das zu erstellende Dokument. Die Sprache XPath und ihre Anwendung mittels VBA beschreiben wir in einem weiteren Beitrag namens **VBA und XPath** (**www.access-im-unternehmen.de/1050**).

Wenn Sie schon einmal eine Webseite programmiert haben, die nicht nur aus reinem HTML besteht, sondern auch aus Skript-Elementen etwa auf Basis von PHP oder ASP/ASP.NET, haben Sie unbewusst bereits eine Vorstellung davon, wie XSLT ein neues Dokument auf Basis eines bestehenden Dokuments zusammensetzt. Ein XSLT-Dokument ist dabei ähnlich aufgebaut wie eine aus Skript- und HTML-Teilen bestehende Webseite.

Sie finden dort nämlich feste Zeichenketten, aber auch dynamische Elemente, mit denen etwa die Inhalte des zu transformierenden Dokuments ermittelt und ausgegeben werden.

XSLT deklarieren

Damit die **.xslt**-Datei korrekt interpretiert werden kann, teilen wir der jeweiligen Verarbeitungsinstanz (in unserem Beispiel etwa die Methode **transformNodeToObject**) mit einer entsprechenden Deklaration in der ersten Zeile mit, um was für einen Dokumenttyp es sich handelt. In diesem Fall soll die Datei mit der folgenden Zeile starten:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/TR/REC-html40">
```

Damit wird der offizielle Namespace des W3C-Konsortiums vorgegeben. Alle folgenden Zeilen, die XSLT-Befehle enthalten, starten mit **<xsl:** und werden mit einem XSLT-Schlüsselwort fortgesetzt. Dadurch können Sie die auszuführenden Elemente des **.xslt**-Dokuments von den statischen Elementen unterscheiden – ähnlich wie etwa beim einem PHP-Dokument, wo die PHP-Anweisungen in **<?php ... ?>**-Blöcken stehen.

Dieser Zeile stellen wir noch die folgende Zeile voran:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Das template-Element

Das Basis-Element einer **.xslt**-Datei ist das **template**-Element. Es enthält auch ein Attribut namens **match**. Mit **match** referenzieren Sie das Element eines XML-Dokuments, auf das sich die innerhalb des **template**-Elements befindlichen Elemente beziehen. Der Wert von **match** ist ein **XPath**-Ausdruck. XPath ist, wie oben bereits erwähnt wurde, die Sprache für den Zugriff auf die Elemente in einem XML-Dokument. Jede Menge Beispiele dazu finden Sie im Beitrag **VBA und XPath** (**www.access-im-unternehmen.de/1050**). Wenn Sie beispielsweise auf das Root-Element des Dokuments (also das oberste Element) zugreifen wollen, geben Sie für das Attribut **match** einen Schrägstrich an (**/**).

Es werden nur Informationen ausgegeben, die sich innerhalb eines **template**-Elements befinden. Sie können also Folgendes in die **.xslt**-Datei schreiben und es wird nichts ausgegeben:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns="http://www.w3.org/TR/REC-html40">
  <blabla>blub</blabla>
  <xsl:template match="/">
  </xsl:template>
</xsl:stylesheet>
```

Innerhalb des **template**-Elements befinden sich keine Daten, und das davor angegebene **blabla**-Element wird nicht ausgegeben, weil es sich nicht innerhalb eines **template**-Elements befindet. Das Ergebnisdokument ist folglich leer. Wenn Sie das **blabla**-Element innerhalb des **template**-Elements platzieren, wird es allerdings ausgegeben:

```
<xsl:stylesheet version="1.0" ...>
  <xsl:template match="/">
    <blabla>blub</blabla>
  </xsl:template>
</xsl:stylesheet>
```

Das heißt, dass Sie selbst eigene Elemente zur Ausgabe hinzufügen können, auch ohne dynamische **xsl:...**-Elemente innerhalb des **template**-Elements hinzuzufügen. Sie könnten also etwa die Grundstruktur des Dokuments anlegen:

```
<xsl:stylesheet version="1.0" ...>
  <xsl:template match="/">
    <Bestellverwaltung>
    </Bestellverwaltung>
  </xsl:template>
</xsl:stylesheet>
```

Dies liefert die folgende Ausgabe:

```
<?xml version="1.0" encoding="UTF-16"?>
<Bestellverwaltung ...></Bestellverwaltung>
```

Zeilenumbruch herstellen

XML-Dokumente haben den Vorteil, dass sie sowohl maschinell erfasst werden können also auch durch das menschliche Auge in den meisten Fällen gut verarbeitet werden können. Dies fällt jedoch umso leichter, wenn der Inhalt des Dokuments einigermaßen strukturiert ausgegeben wird – also mit Zeilenumbrüchen und Einrückungen. Das vorherige Beispiel enthält keine Zeilenumbrüche, was bei dem Hauptelement **<Bestellverwaltung>** jedoch hilfreich wäre, da ja dazwischen noch einige weitere

Informationen eingefügt werden sollen. Also fügen wir dazwischen einen Zeilenumbruch hinzu, was wir mit dem Element **<xsl:text>
</xsl:text>** erledigen:

```
<xsl:stylesheet version="1.0" ...>
  <xsl:template match="/">
    <Bestellverwaltung>
        <xsl:text>&#xa;</xsl:text>
    </Bestellverwaltung>
  </xsl:template>
</xsl:stylesheet>
```

Damit erhalten wir nun im Zieldokument:

```
<?xml version="1.0" encoding="UTF-16"?>
<Bestellverwaltung xmlns="http://www.w3.org/TR/REC-html40">
</Bestellverwaltung>
```

Das ist viel besser – darauf können wir aufbauen! Das Element **<xsl:text>** schließt übrigens auszugebenden und zu interpretierenden Text ein. Würden Sie den Ausdruck **
** genau wie etwa **<Bestellverwaltung>** einfach in die **.xslt**-Datei schreiben, würde dies nicht korrekt als Zeilenumbruch interpretiert werden. Deshalb schließen Sie Platzhalter für ASCII-Zeichen wie **Chr(10)**, hier als **
** definiert (**a** ist hexadezimal und steht für **10**, **x** füllt die zweistellige Hexadezimalzahl auf) in das **<xsl:text>**-Element ein.

Kommentare

Wenn Sie während der Erstellung eines **.xslt**-Dokuments Bereiche auskommentieren wollen, finden Sie dazu ein eigenes Element. Dieses heißt **comment** und wird beispielsweise wie folgt eingesetzt:

```
<xsl:comment>
... auszukommentierender Bereich
</xsl:comment>
```

Daten aus dem Originaldokument ausgeben

Nun wollen wir endlich auf die Daten in unserem Ausgangsdokument zugreifen, das wir transformieren wollen.

Bereits jetzt wird offensichtlich, dass es eher eine Neuerstellung eines Dokuments ist als eine Transformation, denn wir müssen wohl für jedes einzelne gewünschte Element festlegen, ob und wo wir es platzieren wollen.

Wenn Sie nur den Inhalt eines bestimmten Elements des Ausgangsdokuments ausgeben wollen, verwenden Sie dazu das **value-of**-Element. Dieses erwartet mit dem **select**-Attribut die Angabe des betroffenen Elements, das Sie wiederum mit einem XPath-Ausdruck definieren.

Wir möchten einfach den Namen der ersten Kategorie in unserem Ausgangsdokument ermitteln. Damit wir wissen, von welchem Aufbau wir beim Auslesen des Dokuments reden, haben wir dieses auszugsweise in Listing 3 abgebildet. Dieses Dokument wird mit der Prozedur **TestTransformieren** aus den beiden Tabellen **tblKategorien** und **tblArtikel** der Beispieldatenbank erzeugt, die wir weiter oben vorgestellt haben.

Wir möchten also nun auf den Inhalt des Elements **Kategoriename** unterhalb von **dataroot** und **tblKategorien** zugreifen. Dazu fügen wir unserem Dokument nun einfach eine Zeile mit dem **value-of**-Element und der Angabe des gesuchten Elements, also **dataroot/tblKategorien/Kategoriename** hinzu:

```
...
<Bestellverwaltung>
  <xsl:text>&#xa;</xsl:text>
  <xsl:value-of select="dataroot/tblKategorien/Kategoriename"/>
  <xsl:text>&#xa;</xsl:text>
</Bestellverwaltung>
...
```

Dies liefert nun ein XML-Dokument mit folgendem Inhalt:

```
<?xml version="1.0" encoding="UTF-16"?>
<Bestellverwaltung xmlns="http://www.w3.org/TR/REC-html40">
Getränke
</Bestellverwaltung>
```

Oh – das ist zwar ein gültiges XML-Dokument, aber wir wollen die Kategorie natürlich in ein eigenes Element stecken.

Dazu fügen wir einfach ein paar statische Elemente zum **.xslt**-Dokument hinzu, sodass wir sowohl ein **Kategorie**- als auch ein **Kategoriename**-Element erhalten:

```
<?xml version="1.0" encoding="UTF-8"?>
<dataroot xmlns:od="urn:schemas-microsoft-com:officedata" generated="2016-07-09T10:30:28">
  <tblKategorien>
    <KategorieID>1</KategorieID>
    <Kategoriename>Getränke</Kategoriename>
    <Beschreibung>Alkoholfreie Getränke, Kaffee, Tee, Bier</Beschreibung>
    <Abbildung>...</Abbildung>
    <tblArtikel>
      <ArtikelID>1</ArtikelID>
      <Artikelname>Chai</Artikelname>
      <LieferantID>1</LieferantID>
      <KategorieID>1</KategorieID>
      <Liefereinheit>10 Kartons x 20 Beutel</Liefereinheit>
      <Einzelpreis>9</Einzelpreis>
      <Lagerbestand>39</Lagerbestand>
      <BestellteEinheiten>0</BestellteEinheiten>
      <Mindestbestand>10</Mindestbestand>
      <Auslaufartikel>0</Auslaufartikel>
    </tblArtikel>
    <tblArtikel>
      <ArtikelID>2</ArtikelID>
      <Artikelname>Chang</Artikelname>
      ...
    </tblArtikel>
    ...
  </tblKategorien>
  <tblKategorien>
    <KategorieID>2</KategorieID>
    <Kategoriename>Gewürze</Kategoriename>
    <Beschreibung>Süße und saure Soßen, Gewürze</Beschreibung>
    <Abbildung>...</Abbildung>
    <tblArtikel>
      <ArtikelID>3</ArtikelID>
      <Artikelname>Aniseed Syrup</Artikelname>
      ...
    </tblArtikel>
    ...
  </tblKategorien>
  ...
</dataroot>
```

Listing 3: Ausgangsdokument für unsere Experimente

Dateien aus XML-Dokumenten speichern und lesen

Webservices können nicht nur Texte und Zahlen liefern oder entgegennehmen, sondern auch etwa mit Bildern oder PDF-Dateien arbeiten. So könnten Sie einen Webservice nutzen, der Bilder bearbeitet oder umwandelt, oder mit einem, der ein PDF für eine bestimmte Dienstleistung liefert und dazu in einen XML-Response verpackt – zum Beispiel eine Versandmarke et cetera. Für solche Fälle kann es hilfreich sein zu wissen, wie Sie die endlosen Zahlenkombinationen aus dem XML-Dokument in eine Datei umwandeln – und umgekehrt.

Dateien aus XML-Dokumenten exportieren

Wenn Sie gelegentlich Dateien im XML-Format verarbeiten, kommt dort sicher auch mal eine Base64-kodierte Datei vor. Eine einfache XML-Datei mit dem Root-Element **file**, das zwei weitere Elemente namens **filename** (zum Speichern des Dateinamens) und **data** (mit dem kodierten Inhalt einer PDF-Datei) enthält, sieht beispielsweise wie folgt aus – nur je nach Dateigröße viel länger:

```
<?xml version="1.0"?>
<file>
  <filename>Beispieldatei.pdf</filename>
  <data>JVBERi0xLjQNJeLjz9MNCjk3NCAwIG9iag08PC9MaW5lYXJpeI
IFsgMTA5OCA3MjddPj4NZW5kb2JqDSAgICAg...</data>
</file>
```

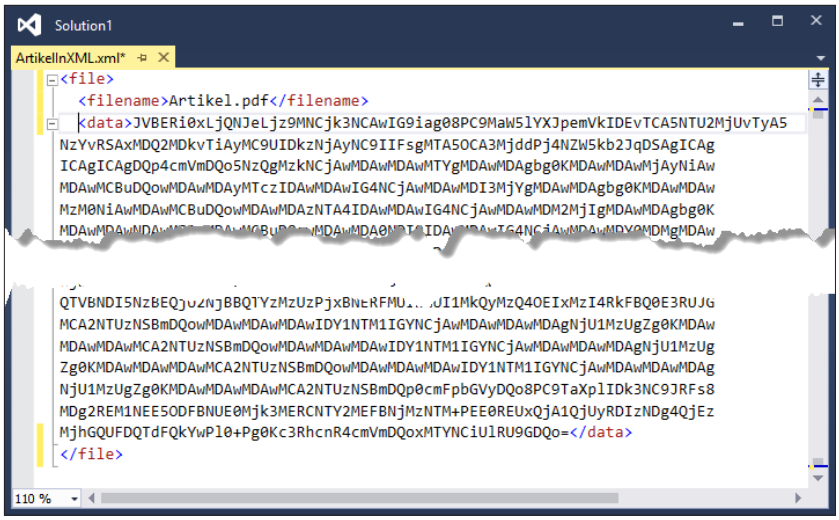

Bild 1: Beispiel für eine in einem XML-Dokument gespeicherte Datei

Einen besseren Eindruck vom Umfang einer solchen Datei erhalten Sie durch einen Blick auf Bild 1. Die hier abgeschnittene Darstellung besteht tatsächlich aus über 17.000 Zeilen!

```
Public Sub XMLElementInDatei()
    Dim objDocument As MSXML2.DOMDocument60
    Dim strXMLDatei As String
    Dim strZieldatei As String
    Dim objData As MSXML2.IXMLDOMElement
    Set objDocument = New MSXML2.DOMDocument60
    strXMLDatei = CurrentProject.Path & "\ArtikelInXML.xml"
    objDocument.Load strXMLDatei
    strZieldatei = CurrentProject.Path & "\" & objDocument.selectSingleNode("//filename").nodeTypedValue
    Set objData = objDocument.selectSingleNode("//data")
    WriteFileFromBytes strZieldatei, DecodeBase64(objData.nodeTypedValue)
End Sub
```

Listing 1: Auslesen einer Datei aus einem XML-Dokument in das Filesystem

Und dieses Durcheinander wollen wir nun in eine Datei umwandeln? Aber natürlich! Mehr dazu gleich.

Voraussetzungen

Für die Beispiele aus diesem Beitrag (und auch, wenn Sie diese in Ihrer eigenen Datenbank nutzen möchten) benötigen Sie zwei zusätzliche Verweise, nämlich auf die Bibliothek **Microsoft XML, v6.0** und **Microsoft ActiveX Data Objects 6.1 Library** (s. Bild 2).

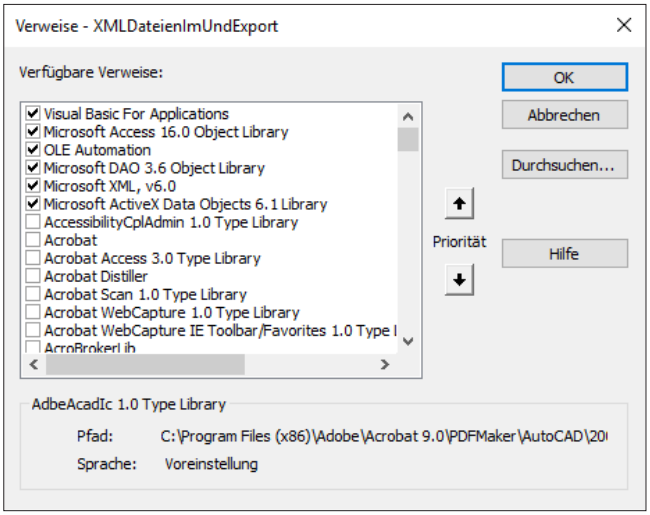

Bild 2: Verweise der Beispieldatenbank

Datei auslesen

Die Steuerung übernimmt die Prozedur aus Listing 1, in der Sie auch festlegen, welche XML-Datei als Quelle dient, aus welchem Element der Name der zu erzeugenden Datei und aus welchem Element der kodierte Dateiinhalt kommt. Die Prozedur erstellt ein neues **DOMDocument**-Objekt und lädt den Inhalt der Datei **ArtikelInXML.xml**, deren Inhalt Sie oben bereits kennen gelernt haben, in dieses Objekt. Die nächste Anweisung trägt das Verzeichnis der aktuellen Datenbank gefolgt von einem Backslash (\) und dem aus dem Element **filename** ermittelten Dateinamen, hier **Beispieldatei.pdf**, in die Variable **strZieldatei** ein.

Mit der **IXMLDOMElement**-Variablen **objBild** referenzieren wir den Inhalt des Elements **data** unserer Zieldatei. Schließlich rufen wir gleich zwei Hilfsfunktionen in einer Zeile auf: Die Funktion **WriteFileFromBytes** erwartet den Namen der zu erstellenden Datei sowie ein Byte-Array mit dem zu schreibenden Dateiinhalt als Parameter und soll die Datei in das Dateisytem schreiben. Das Byte-Array steuert mit dem zweiten Parameter die Funktion **DecodeBase64** bei, welche mit dem Inhalt des Elements **Data** aus der Variablen **objData** versorgt wird.

Data-Element dekodieren

Schauen wir uns als Erstes die Funktion **DecodeBase64** an. Diese erwartet einen String als Parameter, in diesem Fall den Inhalt des **data**-Elements unserer XML-Datei. Und jetzt kommt der Clou: Wir nutzen keinen selbst programmierten Algorithmus, um die Dekodierung vorzunehmen, sondern wir erstellen ein XML-Dokument, fügen diesem ein neues Element namens **tmp** hinzu, stellen den Datentyp auf **bin.base64** ein und füllen es über seine **Text**-Eigenschaft mit dem Wert aus dem Parameter **strBase64**. Die XML-Bibliothek erledigt

```
Private Function DecodeBase64(strBase64 As String) As Byte()
    Dim objDocument As MSXML2.DOMDocument60
    Dim objElement As MSXML2.IXMLDOMElement
    Set objDocument = New MSXML2.DOMDocument60
    Set objElement = objDocument.createElement("tmp")
    objElement.DataType = "bin.base64"
    objElement.Text = strBase64
    DecodeBase64 = objElement.nodeTypedValue
End Function
```

Listing 2: Dekodieren von Base64 in ein Byte-Array

```
Private Sub WriteFileFromBytes(strFile As String, byt() As Byte)
    Dim objStream As ADODB.Stream
    Set objStream = New ADODB.Stream
    objStream.Type = adTypeBinary
    objStream.Open
    objStream.Write byt()
    objStream.SaveToFile strFile, adSaveCreateOverWrite
End Sub
```

Listing 3: Schreiben eines Byte-Arrays in eine Datei

die Umwandlung dann automatisch für uns, sodass wir über die Eigenschaft **nodeTypedValue** des Elements auf das resultierende Byte-Array zugreifen können. Diesen Wert weisen wir dem Rückgabewert **DecodeBase64** der Funktion zu, der als **Byte()** deklariert ist (s. Listing 2).

Byte-Array in Datei schreiben

Damit erhalten wir den zweiten Parameter für die Funktion **WriteFileFromBytes** (s. Listing 3). Der erste Parameter erwartet den Pfad für die zu erstellende Datei, der zweite das soeben mit der Funktion **DecodeBase64** erstellte Byte-Array. Die Prozedur nutzt wieder eine externe Bibliothek, und zwar **Microsoft ActiveX Data Objects 6.1 Library**. Davon deklariert sie zunächst ein **ADODB.Stream**-Objekt und erstellt eine neue Instanz dieses Objekts. Den Typ des Objekts stellt die Funktion auf **adTypeBinary** ein. Sie

```
Public Sub DateiInXMLElement()
    Dim objDocument As MSXML2.DOMDocument60
    Dim objRoot As MSXML2.IXMLDOMElement
    Dim strDatei As String
    Dim strData As String
    Dim strQuelle As String
    Dim objElement As MSXML2.IXMLDOMElement
    Set objDocument = New MSXML2.DOMDocument60
    strDatei = CurrentProject.Path & "\XMLMitPDFDatei.xml"
    strQuelle = CurrentProject.Path & "\Artikel.pdf"
    Set objRoot = objDocument.createElement("file")
    objDocument.appendChild objRoot
    Set objElement = objDocument.createElement("filename")
    objElement.nodeTypedValue = Mid(strQuelle, _
        InStrRev(strQuelle, "\") + 1)
    objRoot.appendChild objElement
    Set objElement = objDocument.createElement("data")
    objRoot.appendChild objElement
    strData = EncodeBase64(ReadBytesFromFile(strQuelle))
    objElement.nodeTypedValue = strData
    objDocument.Save CurrentProject.Path & "\ArtikelInXML.xml"
End Sub
```

Listing 4: Übertragen einer Datei in ein Element eines XML-Dokuments

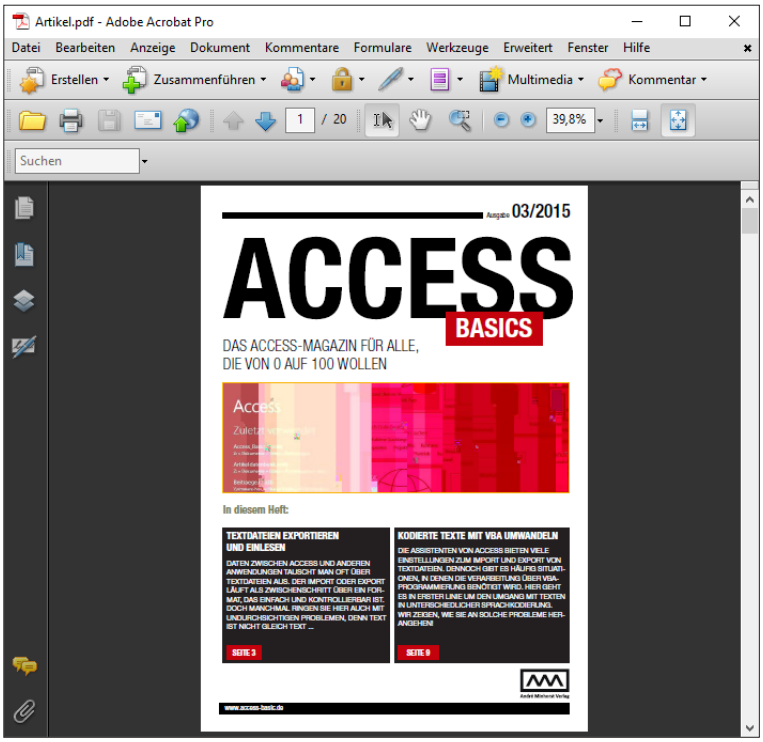

Bild 3: Es klappt – die exportierten XML-Daten ergeben tatsächlich eine lesbare Datei.

öffnet das Objekt mit der **Open**-Methode und schreibt dann das mit dem zweiten Parameter **byt()** übergebene Byte-Array hinein. Das Stream-Objekt bietet direkt eine **SaveToFile**-Methode an, mit der wir den Inhalt in die mit dem ersten Parameter übergebene Datei schreiben können. Mit dem Wert des zweiten Parameters legen wir fest, dass die Datei neu erstellt werden oder eine bestehende Datei überschreiben soll.

Es hat funktioniert – die Daten aus dem XML-Dokument ergeben tatsächlich eine lesbare Datei, hier im PDF-Format (s. Bild 3).

Datei in ein XML-Dokument schreiben

Natürlich wollen wir uns auch den umgekehrten Fall ansehen. Dabei soll eine auf der Festplatte liegende Datei, beispielsweise eine PDF-Datei, in das **data**-Element eines XML-Dokuments

XML-Zugriff per VBA: Welche Version?

Beim Umgang mit XML-Dokumenten per VBA gibt es ein paar kleine Dinge, die Sie beachten müssen. Eines davon ist die Version der Bibliothek Microsoft XML, vx.0. In diesem Beitrag zeigen wir die für den Einsatz mit VBA relevanten Unterschiede auf. Dabei erfahren Sie, welche der beiden populärsten Versionen, nämlich die Version 3.0 und die Version 6.0, die für Ihren Anwendungszweck optimale Version ist.

XML-Bibliotheks-Versionen

Bei der Wahl des Verweises auf eine der verschiedenen Versionen der Bibliothek **Microsoft XML, vx.0** gibt es verschiedene Ideen: Der eine nimmt die Version 3.0, weil er auf Altes vertraut und auf Nummer sicher gehen will, die Mindestvoraussetzungen zu erfüllen. Der andere wählt die Version 6.0, weil er gern die neuesten und umfangreichsten Bibliotheken nutzt (s. Bild 1). Die Versionen dazwischen werden recht selten genutzt, daher konzentrieren wir uns auf die beiden Versionen 3.0 und 6.0.

Wir wollen uns in diesem Beitrag auf die Unterschiede beschränken, welche die bei unserer täglichen Arbeit auftauchenden Elemente der beiden Bibliotheken betreffen. Da finden wir zunächst das **DOMDocument**-Objekt, also das Objekt, mit dem wir XML-Dokumente aus Dateien oder String-Variablen einlesen und auf diese zugreifen können.

Den wichtigsten Unterschied decken Sie bereits auf, wenn Sie einen Blick in die Objektkataloge der beiden Bibliotheken werfen. Nach dem Setzen eines Verweises auf die Bibliothek **Microsoft XML, v3.0** finden Sie etwa die Objekte aus Bild 2 vor. Hier tauchen gleich drei **DOMDocument**-Elemente auf: **DOMDocument**, **DOMDocument26** und **DOMDocument30**.

Ein Blick in die Bibliothek **Microsoft XML, v6.0** liefert eine aufgeräumtere Ansicht: Hier finden Sie nur ein **DOMDocument**-Objekt vor, nämlich **DOMDocument60** (s. Bild 3).

Sie können übrigens nur einen von beiden Verweisen in einem VBA-Projekt setzen: Wenn bereits die Version 3.0 referenziert ist und Sie fügen die Version 6.0 hinzu, erhalten Sie die Fehlermeldung aus Bild 4 – und umgekehrt. Mit der Version 4.0 gelingt dies auch nicht.

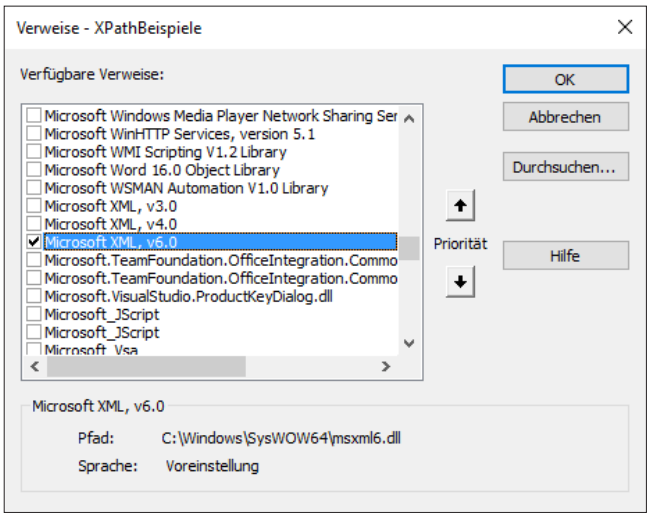

Bild 1: Hinzufügen des Verweises auf die XML-Bibliothek

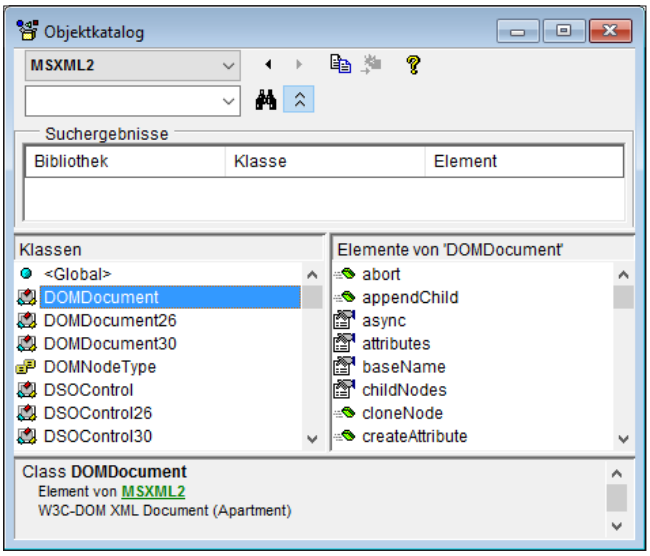

Bild 2: Objektkatalog mit der Bibliothek **Microsoft XML, v3.0**

Einer der wichtigsten Unterschiede zwischen den beiden Bibliotheken ist, dass die Klassen für die **DOMDocument**-Objekte auf keinen Fall gleich benannt sind und Sie dementsprechend, wenn Sie aus irgendwelchen Gründen etwa die Bibliothek **Microsoft XML, v3.0** durch die Bibliothek **Microsoft XML, v6.0** ersetzen, auf jedem Fall auch eine Menge Deklarationen und Zuweisungen ändern müssen.

Angekommen, Ihre zuvor mit **Microsoft XML, v3.0**, enthält folgenden Code:

```
Dim objDocument As MSXML2.DOMDocument
Set objDocument = New MSXML2.DOMDocument
```

Dann müssen Sie diesen unter der Version 6.0 wie folgt ändern:

```
Dim objDocument As MSXML2.DOMDocument60
Set objDocument = New MSXML2.DOMDocument60
```

Gegebenenfalls sind solche Anweisungen über das VBA-Projekt verstreut, sodass eine Menge Arbeit entsteht, die Sie aber natürlich per Suchen und Ersetzen-Funktion einfach erledigen können.

3.0 oder 6.0?

Sollten Sie nun die Version 3.0 oder 6.0 verwenden? Ein etwas älterer Blog-Beitrag eines Mitglieds des XML-Teams von Microsoft besagt, dass diese beiden Versionen zumindest gegenüber den Versionen 4.0 und 5.0 bevorzugt gepflegt werden. 3.0 wird gepflegt, weil es in sehr vielen älteren Anwendungen genutzt wird, 6.0, weil es die aktuellste Version ist. Den Blogbeitrag finden Sie hier:

https://blogs.msdn.microsoft.com/xmlteam/2006/10/23/using-the-right-version-of-msxml-in-internet-explorer/

Einer der wichtigsten Unterschiede zwischen den beiden Versionen 3.0 und 6.0 ist, dass die Version 6.0 im Gegensatz zur Version 3.0 einige Sicherheitsfeatures standardmäßig aktiviert hat.

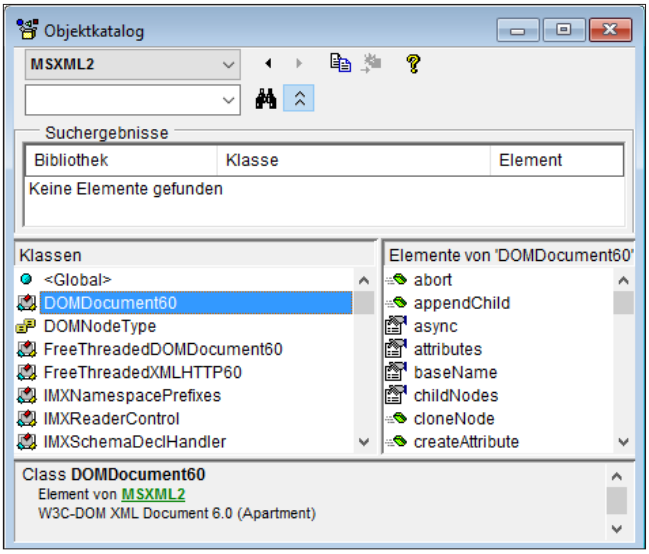

Bild 3: Objektkatalog mit der Bibliothek **Microsoft XML, v3.0**

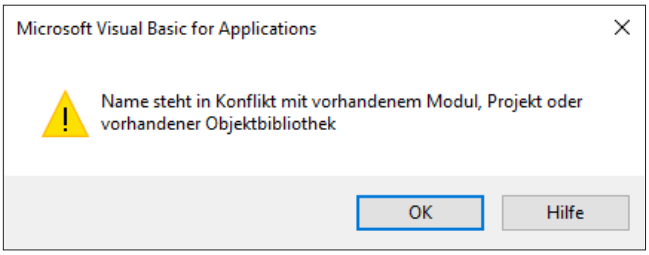

Bild 4: Fehler beim Setzen von Verweisen auf mehrere Microsoft XML-Bibliotheken

Ein weiterer wichtiger Unterschied liegt in der Abfragesprache, die bei Verwendung der beiden Befehle **selectNodes** und **selectSingleNodes** verwendet wird. Unter der Version 3.0 ist standardmäßig **XSLPattern** voreingestellt, unter Version 6.0 standardmäßig **XPath**.

Die aktuell verwendete Abfrage testen Sie mit einer kleinen Prozedur:

```
Public Sub TestAbfragesprache()
    Dim objDocument As MSXML2.DOMDocument
    Set objDocument = New MSXML2.DOMDocument
    Debug.Print objDocument.getProperty("SelectionLanguage")
End Sub
```

Unter der Version 3.0 liefert dies den Wert **XSLPattern**.

XML-Zugriff per XPath

XML-Dokumente erscheinen je nach Größe auf den ersten Blick oft unübersichtlich und mächtig. Wie soll man hier die gewünschten Daten extrahieren – und das auch noch programmgesteuert per VBA? Beispielsweise, um Informationen aus einem XML-Dokument in eine Access-Tabelle zu übertragen? Dafür steht die Abfragesprache XPath zur Verfügung. Sie erlaubt es, mit verschiedenen Ausdrucken gezielt auf Elemente mit bestimmten Namen oder Eigenschaften zuzugreifen. Dieser Beitrag zeigt anhand einiger Beispiele, wie Sie XPath unter Access/VBA einsetzen.

Voraussetzungen

Um die folgenden Beispiele auszuführen, benötigen Sie eine Beispieldatenbank mit einem Verweis auf die XML-Bibliothek von Microsoft. Dazu öffnen Sie den Verweise-Dialog (VBA-Editor, Menüeintrag **Extras|Verweise**) und wählen dort den Eintrag **Microsoft XML, v3.0** aus (s. Bild 1). Mit **Microsoft XML, v6.0** gab es bei der Erstellung der Beispiele Probleme, da hier einige Methoden nicht die erwarteten Ergebnisse lieferten.

Beispieldokument

Um per VBA auf den Inhalt eines XML-Dokuments zugreifen zu können, benötigen Sie zunächst ein solches. Unseres heißt **KategorienUndArtikel.xml** und sollte sich im gleichen Verzeichnis wie die Beispieldatenbank befinden, da die Zugriffe auf diese Datei dahingehend ausgerichtet sind. Der Inhalt dieser XML-Datei sieht wie in Listing 1 aus.

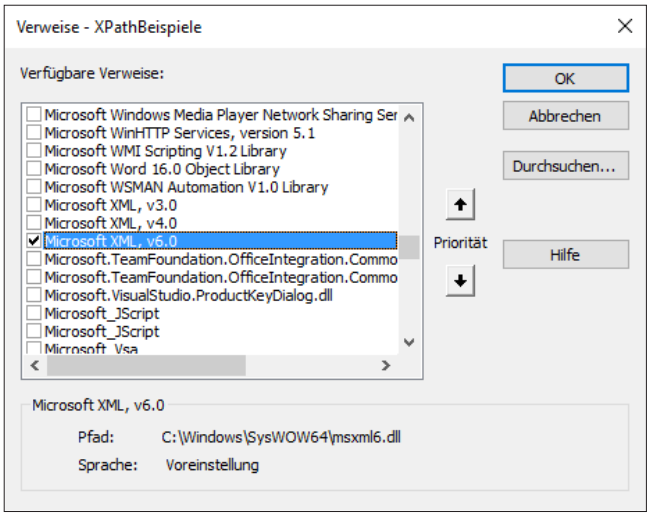

Bild 1: Hinzufügen des Verweises auf die XML-Bibliothek

Um per XPath per VBA auf ein XML-Dokument zugreifen zu können, müssen Sie dieses zunächst in den Speicher laden, beziehungsweise es mit einem geeigneten Objekt referenzieren. Dies erledigen Sie mit dem Code, wie er etwa in der folgenden Prozedur enthalten ist:

```
Public Sub DokumentLaden()
    Dim strDatei As String
    Dim objXML As MSXML2.DOMDocument
    strDatei = _
        CurrentProject.Path & "\KategorienUndArtikel.xml"
    Set objXML = New MSXML2.DOMDocument
    objXML.Load strDatei
    If Not Len(objXML.XML) = 0 Then
        Debug.Print objXML.XML
    Else
        Debug.Print objXML.parseError.errorCode, _
                                    objXML.parseError.reason
    End If
End Sub
```

Die Prozedur erstellt ein neues Objekt des Typs **DOMDocument** und verwendet die **Load**-Methode, um die angegebene Datei in das Objekt zu laden. Gelingt dies, sollte die Länge der über die Eigenschaft **XML** zu ermittelnde Zeichenkette, also der Inhalt des Dokuments, größer als **0** sein. In diesem Fall soll die Prozedur den Inhalt des XML-Dokuments im Direktbereich des VBA-Editors ausgeben.

Anderenfalls ist etwas beim Einlesen schiefgelaufen. Dann soll die Fehlernummer samt der Fehlerbeschreibung im Direktbereich erscheinen.

XPath per VBA nutzen

Um die Abfragesprache XPath von VBA aus nutzen zu können, gibt es zwei Funktionen. Die erste heißt **selectSingleNode** und erwartet einen **XPath**-Ausdruck als Parameter. Sie liefert ein einziges **Node**-Element als Ergebnis zurück, sofern die Abfrage ein Ergebnis hat.

Die zweite Funktion heißt **selectNodes** und liefert eine Auflistung des Typs **DOMSelection** zurück. Sie kann kein, ein oder mehrere Elemente enthalten, die wiederum den Typ **DOMDocument** für das Dokument-Objekt, **IXMLDOMProcessingInstruction** für das **<?xml...?>**-Element oder **IXMLDOMElement** für die übrigen Elemente aufweisen.

Üblicherweise werden Sie aber mit den Elementen des Typs **IXMLDOMElement** arbeiten, ein Zugriff auf das **DOMDocument**-Objekt oder das **IXMLDOMProcessingInstruction**-Objekt ist selten in Zusammenhang mit dem Zugriff per XPath nötig.

Unsere Beispielprozedur für das Erstellen eines XML-Dokuments und das Füllen dieses Objekts aus einer XML-

```
<?xml version="1.0" encoding="UTF-16"?>
<Bestellverwaltung xmlns="http://www.w3.org/TR/REC-html40">
  <Kategorie KategorieID="1">
    <Kategoriename>Getränke</Kategoriename>
    <Beschreibung>Alkoholfreie Getränke, Kaffee, Tee, Bier</Beschreibung>
    <Artikel ArtikelID="1">
      <Artikelname>Chai</Artikelname>
      <Einzelpreis>EUR 9.00</Einzelpreis>
    </Artikel>
    <Artikel ArtikelID="2">
      <Artikelname>Chang</Artikelname>
      <Einzelpreis>EUR 9.50</Einzelpreis>
    </Artikel>
    ..
  </Kategorie>
  <Kategorie KategorieID="2">
    <Kategoriename>Gewürze</Kategoriename>
    <Beschreibung>Süße und saure Soßen, Gewürze</Beschreibung>
    <Artikel ArtikelID="3">
      <Artikelname>Aniseed Syrup</Artikelname>
      <Einzelpreis>EUR 5.00</Einzelpreis>
    </Artikel>
    <Artikel ArtikelID="4">
      <Artikelname>Chef Anton's Cajun Seasoning</Artikelname>
      <Einzelpreis>EUR 11.00</Einzelpreis>
    </Artikel>
  ...
  </Kategorie>
  ...
</Bestellverwaltung>
```

Listing 1: XML-Dokument für die Beispiele dieses Beitrags

Datei haben wir etwas abgewandelt, damit wir damit mit einer Anweisung innerhalb unserer Beispielprozeduren auf das XML-Dokument zugreifen können (s. Listing 2).

Auf das Root-Element zugreifen

Ein Beispiel für den Zugriff auf ein einzelnes XML-Element sieht danach wie folgt aus:

```
Public Sub RootelementHolen()
    Dim objElement As MSXML2.IXMLDOMElement
    Dim objDocument As MSXML2.DOMDocument
    Set objDocument = GetDocument
    Set objElement = _
        objDocument.selectSingleNode("Bestellverwaltung")
    Debug.Print objElement.XML
End Sub
```

Dies liest das Element **Bestellverwaltung** samt allen untergeordneten Elementen ein. Wenn wir wie in obiger Beispielprozedur den Inhalt der XML-Eigenschaft im Direktbereich ausgeben, erhalten wir also fast das komplette Dokument – mit Ausnahme der Formatinformationen in der **<?xml...?>**-Zeile.

Dies gelingt aber auch nur über den XPath-Ausdruck **Bestellverwaltung**, weil wir die **selectSingleNode**-Funktion für das **DOMDocument**-Objekt aufrufen und das Bestellverwaltung-Objekt diesem direkt untergeordnet ist.

Wir könnten also nicht etwa auf das erste **Kategorie**-Objekt zugreifen, indem wir einfach folgenden Ausdruck nutzen:

```
Set objElement = objDocument.selectSingleNode("Kategorie")
```

Wenn wir dies versuchen, erhalten wir eine Fehlermeldung wie in Bild 2. Die Anweisung **Set objElement...** löst zwar noch keinen Fehler aus. Aber **objElement** wird hier nicht gefüllt und der folgende Zugriff auf eine Eigenschaft von **objElement** führt dann zum Fehler.

Auf ein direktes Unterelement des Root-Elements zugreifen

Mit dem Namen eines Elements allein können Sie also nur auf ein Element zugreifen, wenn sich dieses direkt unterhalb des Objekts befindet, für das Sie die **SelectSingleNode**-Methode aufrufen.

Dafür müssten Sie zuerst das Root-Element **Bestellverwaltung** per **IXMLDomElement**-Variable referenzieren und könnten dann von dort aus auf das **Kategorie**-Element zugreifen:

```
Dim objBestellverwaltung As MSXML2.IXMLDOMElement
Dim objKategorie As MSXML2.IXMLDOMElement
Dim objDocument As MSXML2.DOMDocument
Set objDocument = GetDocument
```

```
Public Function GetDocument() As MSXML2.DOMDocument
    Dim strDatei As String
    Dim objXML As MSXML2.DOMDocument
    strDatei = CurrentProject.Path & "\KategorienUndArtikel.xml"
    Set objXML = New MSXML2.DOMDocument
    objXML.Load strDatei
    If Not Len(objXML.XML) = 0 Then
        Set GetDocument = objXML
    Else
        MsgBox "Fehler " & objXML.parseError.errorCode & ": " & objXML.parseError.reason
    End If
End Function
```

Listing 2: Hilfsfunktion, um ein gefülltes **DOMDocument**-Element zu holen

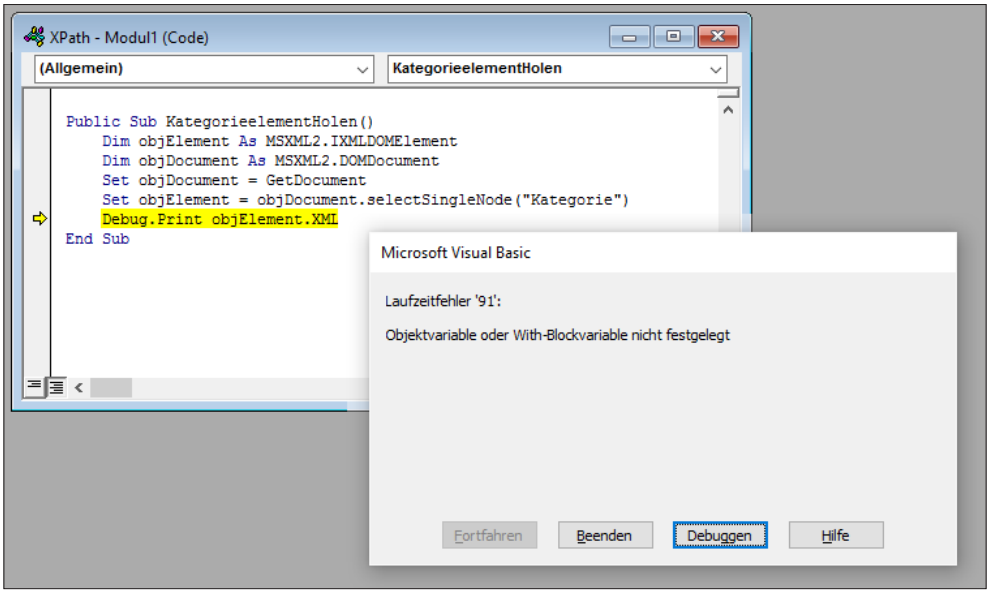

Bild 2: Fehler beim Zugriff auf ein XML-Element

```
Set objBestellverwaltung = _
        objDocument.selectSingleNode("Bestellverwaltung")
Set objKategorie = objBestellverwaltung._
                           selectSingleNode("Kategorie")
Debug.Print objKategorie.XML
```

Dies gibt den Inhalt des ersten **Kategorie**-Elements aus.

An dieser Stelle ist es wichtig zu erwähnen, dass die **SelectSingleNode** immer das erste Element liefert, das dem angegebenen Ausdruck entspricht. Während es nur ein **Bestellverwaltung**-Element gibt, befinden sich darunter allerdings gleich sieben **Kategorie**-Elemente. Davon liefert **SelectSingleNode** dann das erste.

Auf mehrere Elemente zugreifen

Das ist ein guter Anlass, die Funktion **selectNodes** vorzustellen. Sie liefert nicht nur ein einziges Element zurück, sondern kann auch einmal kein oder mehrere Elemente zurückgeben. Diese kommen immer in einer Auflistung vom Typ **DOMSelection**.

Wenn Sie die gefundenen Elemente mit der **For Each**-Schleife durchlaufen wollen, definieren Sie wie im folgenden Beispiel sowohl ein Objekt namens **objKategorie** mit dem Typ **IXMLDOMElement** als Laufvariable sowie eines für die Auflistung namens **objKategorien** mit dem Typ **IXMLDOMSelection**.

Dann referenzieren Sie wieder das Root-Element und nutzen dann dessen **selectNodes**-Funktion, um alle untergeordneten **Kategorie**-Elemente zu ermitteln. Diese landen dann im Auflistungsobjekt **objKategorien**. Dieses können wir dann per **For Each**-Schleife mit der Laufvariablen **objKategorie** durchlaufen. Innerhalb der Schleife geben wir wieder den Inhalt der XML-Eigenschaft aus:

```
Public Sub KategorienHolen()
    Dim objBestellverwaltung As MSXML2.IXMLDOMElement
    Dim objKategorie As MSXML2.IXMLDOMElement
    Dim objKategorien As MSXML2.IXMLDOMSelection
    Dim objDocument As MSXML2.DOMDocument
    Set objDocument = GetDocument
    Set objBestellverwaltung = _
         objDocument.selectSingleNode("Bestellverwaltung")
    Set objKategorien = _
            objBestellverwaltung.selectNodes("Kategorie")
    For Each objKategorie In objKategorien
        Debug.Print objKategorie.XML
    Next objKategorie
End Sub
```

Da die Ausgabe alle **Kategorie**-Elemente umfasst, die selbst jeweils einige **Artikel**-Elemente enthalten, sprengt die Ausgabe das Direktfenster. Also geben wir etwas weniger Umfangreiches aus, indem wir die **Debug.Print**-

Anweisung wie folgt ersetzen und damit gleich noch einen einfachen XPath-Ausdruck nutzen:

```
Debug.Print objKategorie.selectSingleNode(_
                        "Kategoriename").nodeTypedValue
```

Da wir in diesem Fall nicht einfach den Inhalt der Eigenschaft **XML** ausgeben wollen, sondern den Inhalt des Elements **Kategoriename** selbst, verwenden wir die Eigenschaft **nodeTypedValue**.

Die gefundenen Elemente können Sie auch per **For... Next**-Schleife durchlaufen. Dann nutzen Sie die **length**-Eigenschaft der **IXMLDOMSelection**-Auflistung zur Bestimmung der Anzahl der Elemente.

Über die **Item()**-Eigenschaft greifen Sie dann auf das jeweilige Element zu, wobei der Index 0-basiert ist, was für den Wertebereich der Schleife berücksichtigt werden muss:

```
Dim i As Integer
...
Set objKategorien = _
    objBestellverwaltung.selectNodes("Kategorie")
For i = 0 To objKategorien.length - 1
    Debug.Print objKategorien.Item(i).selectSingleNode(_
        "Kategoriename").nodeTypedValue
Next i
```

Beispiele für XPath-Ausdrücke

Nachdem Sie nun erfahren haben, wo und wie Sie überhaupt XPath-Ausdrücke einsetzen können (wobei wir ja immer nur den Namen eines Elements als Ausdruck angegeben haben), wollen wir uns nun die wichtigsten Beispiele für XPath-Ausdrücke ansehen.

Dabei ist immer der Kontext wichtig, also von welchem Element des XML-Dokuments aus Sie das betroffene Element referenzieren wollen. Für die ersten Beispiele gehen wir jeweils vom Root-Element aus, also vom Element **Bestellverwaltung**.

Zugriff auf Enkel-Elemente

Wenn Sie direkt vom **DOMDocument**-Objekt auf die **Kategorie**-Elemente unterhalb von **Bestellverwaltung** zugreifen wollen, gelangen Sie so dorthin:

```
Set objKategorien = _
    objDocument.selectNodes("Bestellverwaltung/Kategorie")
For i = 0 To objKategorien.length - 1
    Debug.Print objKategorien.Item(i).selectSingleNode(_
                        "Kategoriename").nodeTypedValue
Next i
```

Alle Elemente mit bestimmtem Namen

Wenn Sie alle Elemente eines Dokuments mit einem bestimmten Namen finden wollen, können Sie den Ausdruck **//Kategorie** verwenden:

```
Set objKategorien = objDocument.selectNodes("//Kategorie")
```

Vorsicht, wenn sich auch in anderen Ebenen noch Elemente des gleichen Namens befinden – die werden dann auch berücksichtigt.

Wenn Sie beispielsweise alle Elemente namens **Artikelname** ermitteln wollen, die sich unterhalb des Elements **Artikel** befinden, gelingt dies so:

```
Set objArtikel = _
          objDocument.SelectNodes("//Artikel/Artikelname")
```

Alle Elemente

Sollten Sie überhaupt alle Elemente eines Dokuments durchlaufen wollen, dann nutzen Sie den XPath-Ausdruck **//***. Die Elemente werden dann von oben nach unten durchlaufen:

```
Dim objObjects As MSXML2.IXMLDOMSelection
Dim objDocument As MSXML2.DOMDocument
Dim i As Integer
Set objDocument = GetDocument
Set objObjects = objDocument.selectNodes("//*")
```

Evernote und Access

Evernote ist ein sehr praktisches Tool, um Informationen zu sammeln und auf allen möglichen Endgeräten verfügbar zu machen. Schreiben Sie etwa eine Notiz in der Evernote-App auf dem Smartphone, können Sie diese später am Desktop-Rechner weiternutzen. Oder Sie speichern Rechnungen vom heimischen Rechner und können diese, wenn Sie mal etwas umtauschen wollen, wieder auf dem Smartphone abrufen. All dies wird natürlich noch interessanter, wenn Sie auch noch per Access-Datenbank auf die gespeicherten Daten zugreifen oder diese VBA-gesteuert erweitern können.

Die Nutzungsmöglichkeiten von Evernote sind wirklich sehr umfangreich. Ich habe beispielsweise einen ScanSnap-Scanner von Fujitsu auf dem Schreibtisch stehen, in den ich ein oder mehrere Blätter Papier einlege und den Inhalt per Knopfdruck einscanne, in ein PDF-Dokument konvertiere und automatisch in Evernote speichere.

Evernote klinkt sich auch per Hilfstool in die verfügbaren Internetbrowser ein und bietet eine Schaltfläche an, mit der sich die Inhalte von Webseiten per Mausklick in Evernote speichern lassen. Oder Sie ziehen Dateien aus dem Dateisystem in einen der Evernote-Ordner, um diese überall verfügbar zu machen, wo Sie Evernote nutzen.

Evernote selbst kommt unter Windows als Anwendung, die links eine Ordnerstruktur anzeigt, in der Mitte die gespeicherten Notizen und rechts den Inhalt der aktuell markierten Notiz (siehe Bild 1).

In diesem Beispiel sehen Sie einen Ordner mit einigen Ausgaben von Access im Unternehmen, deren Inhalt sich ganz einfach anzeigen und auch durchsuchen lässt

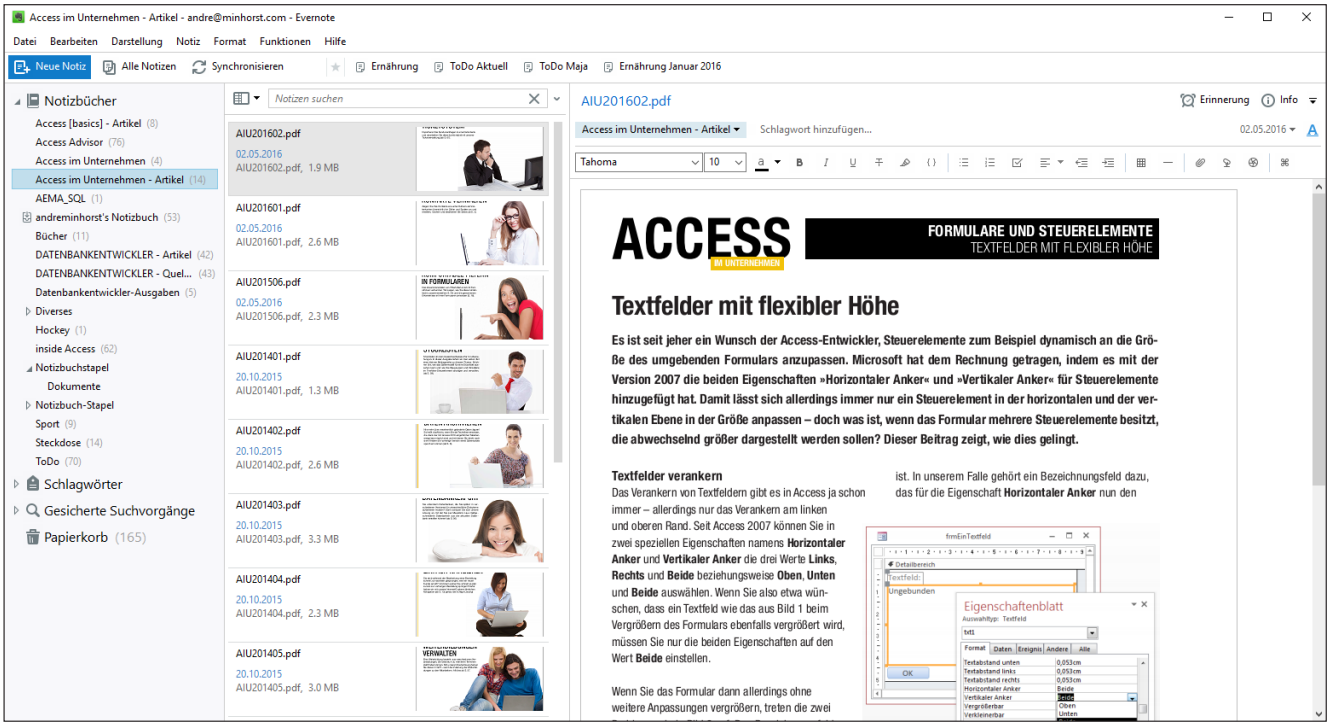

Bild 1: Einrichten des API-Zugriffs

(speichern von PDFs nur mit einer bestimmten Version möglich, siehe weiter unten). Wenn Sie mit Evernote arbeiten möchten, benötigen Sie einen entsprechenden Benutzerzugang, den Sie auf **www.evernote.com** anlegen können. Zur Registrierung geben Sie dort E-Mail-Adresse und Kennwort an und klicken auf **Kostenlos registrieren**.

Verschiedene Varianten

Danach haben Sie die Qual der Wahl: Es gibt nämlich verschiedene Varianten von Evernote, wobei die **BASIC**-Variante kostenlos ist

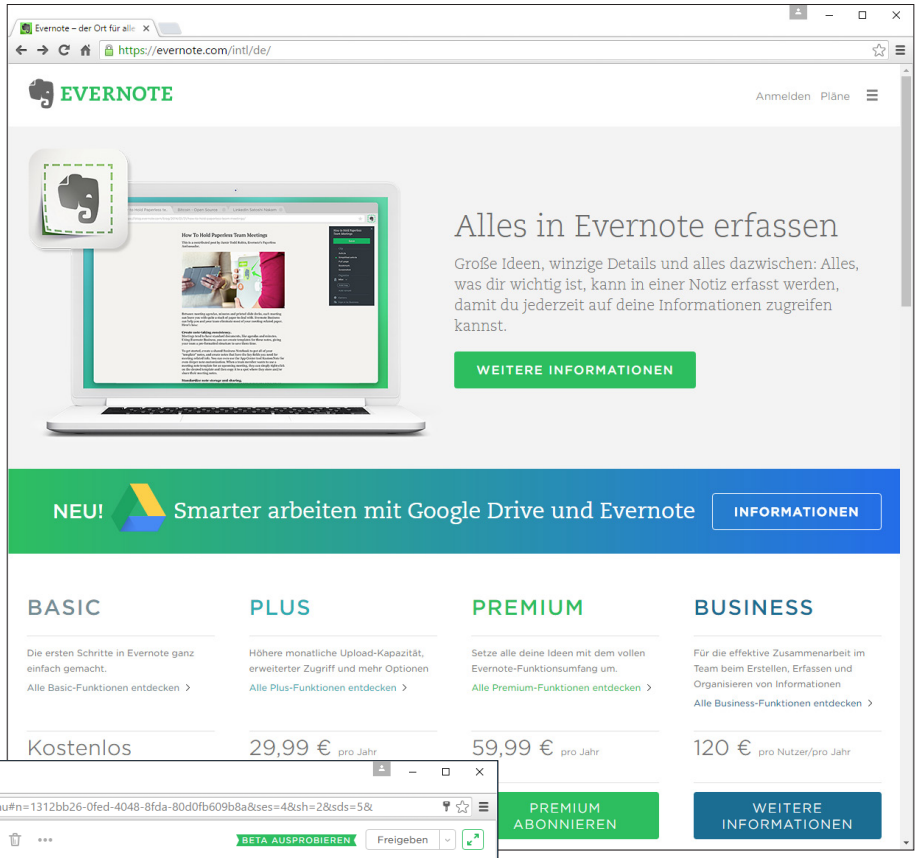

Bild 2: Verschiedene Varianten

(s. Bild 2). Die folgenden Varianten **PLUS**, **PREMIUM** und **BUSINESS** enthalten jeweils mehr Speicherplatz und Funktionen. Ich selbst nutze die **PREMIUM**-Version, weil es die kleinste Version ist, die das Speichern und Durchsuchen von PDF- und Office-Dokumenten erlaubt. Zum Ausprobieren können Sie allerdings auch erst einmal die **BASIC**-Variante wählen.

Bezüglich des Speicherplatzes gibt es hier eine Besonderheit: Es gibt keine Begrenzung des eigentlichen Speicherplatzes, sondern nur der Menge der monatlich hochgeladenen Daten.

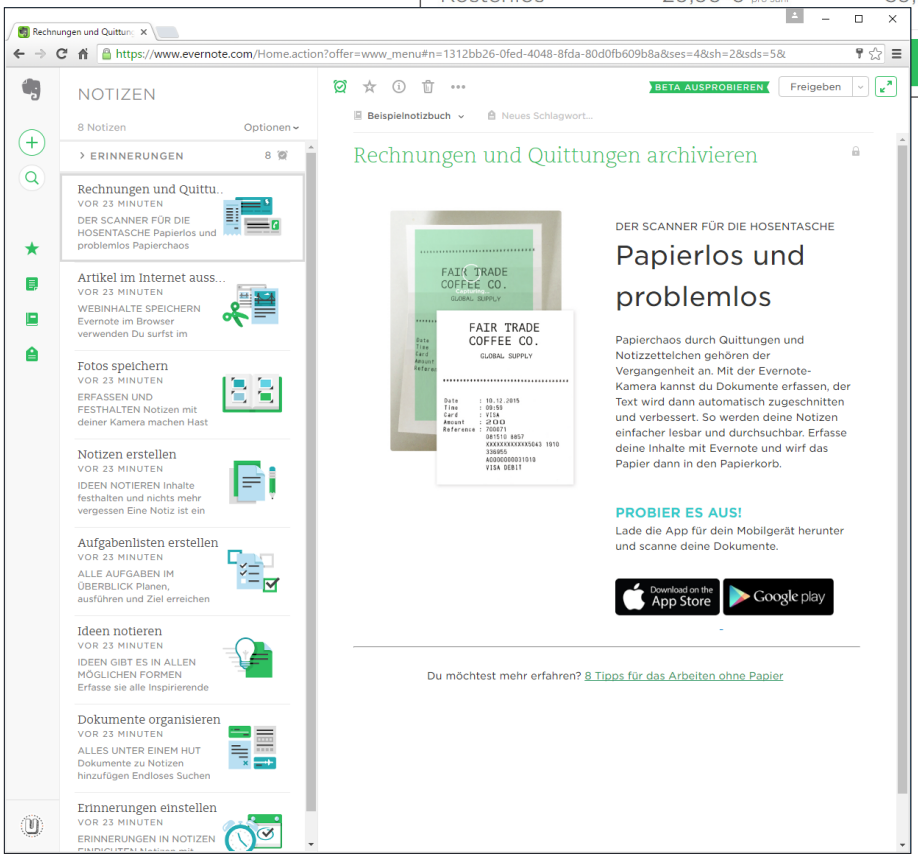

Bild 3: Zugriff auf Evernote per Browser

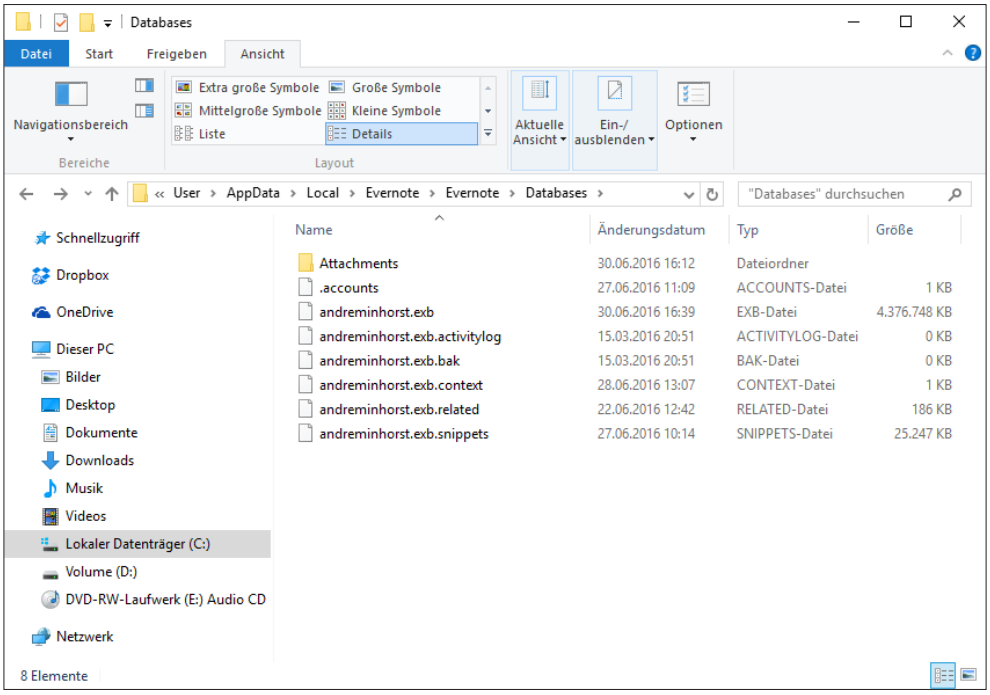

Bild 4: Windows-Verzeichnis mit den Evernote-Dateien

Webbrowser-Version

Nach der Anmeldung können Sie direkt über den Webbrowser mit der Eingabe von Notizen beginnen (s. Bild 3). Hier finden Sie gleich einige Beispielnotizen vor, die gängige Szenarien beschreiben.

Desktop-Version

Richtig interessant wird es allerdings, wenn Sie sich die Evernote-Software für den Windows-Desktop herunterladen. Den Download finden Sie unter **https://evernote.com/intl/de/download/**. Dies lädt eine **.exe**-Datei auf Ihren Rechner, welche die Installationsdateien für Evernote enthält.

Wo ist nun der Unterschied zwischen der Desktop-Version und der Version, die Sie über den Webbrowser von überall aus aufrufen können? Zunächst bietet die Desktop-Software naturgemäß eine komfortablere Benutzeroberfläche mit einfacher zu bedienenden Funktionen. Ein weiterer wichtiger Unterschied ist jedoch, dass die Daten nun nicht mehr nur auf dem Webserver von Evernote gespeichert werden, sondern auch lokal auf der Festplatte. Das heißt, dass Sie hier unabhängig von einer Internetverbindung arbeiten können.

Außerdem müssen Sie sich nicht darauf verlassen, dass Evernote Ihre Daten zuverlässig sichert, sondern können selbst für ein Backup sorgen. Dazu sichern Sie einfach die Dateien im Verzeichnis **C:\Users\<Benutzername>\AppData\Local\Evernote\Evernote\Databases** (s. Bild 4). Die innerhalb von Notizen gespeicherten Dateien wie etwa PDF- oder Office-Dokumente finden Sie außerdem nochmals als Kopie im Unterverzeichnis **Attachments**. Sollten Sie also größere Mengen Dokumente etwa via Scanner im PDF-Format in Evernote gespeichert haben, müssen Sie die Dateien noch nicht einmal aus den Evernote-Datenbanken extrahieren, sondern können direkt auf die Dateien zugreifen.

Struktur unter Evernote

Evernote bietet zwar die Möglichkeit an, Notizen in Verzeichnissen zu kategorisieren. Aber es gibt nur maximal zwei Verzeichnisebenen, was dem einen oder anderen etwas schmalbrüstig vorkommen mag. Letztlich kann man sich aber mit dieser Einschränkung anfreunden.

In den Verzeichnissen und Unterverzeichnissen können Sie beliebig viele Notizen anlegen. Zusätzlich können Sie Notizen aber mit Schlagwörtern versehen, was eine wesentlich mächtigere Markierung als die Unterbringung in einem Ordner ist: Die Ordnerstruktur erlaubt schließlich nur die Angabe eines übergeordneten Ordners je Notiz, während Sie jeder Notiz kein, ein oder mehrere Schlag-

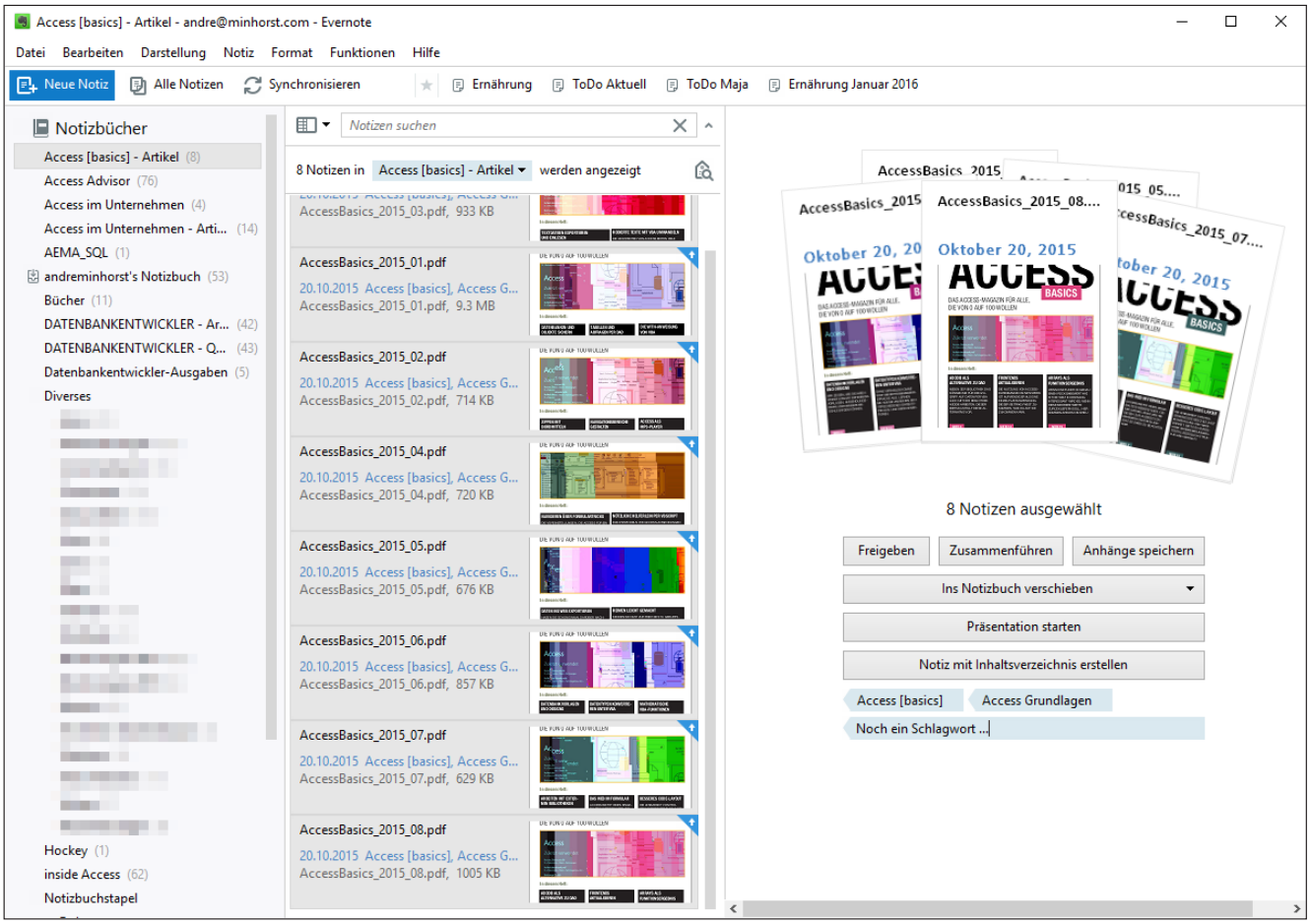

Bild 5: Vergabe mehrerer Schlagwörter für verschiedene gleichzeitig markierte Notizen

wörter zuordnen können. So lassen sich beispielsweise Artikel zum Thema Access mit dem Schlagwort **Access** versehen, aber gleichzeitig auch mit dem Schlagwort **VBA**.

Bild 5 zeigt ein Beispiel für einige Objekte, die mit verschiedenen Schlagwörtern versehen wurden.

Im Grunde genommen handelt es sich bei der Struktur unter Evernote noch nicht einmal um zwei vollwertige Ordnerebenen. Die erste Ebene kann nämlich nur Notizbücher enthalten, aber keine Notizen. Stattdessen zeigt ein Ordner der ersten Ebene, ein sogenannter Notizbuchstapel, nur die in untergeordneten Ordnern/Notizbüchern enthaltenen Notizen an. Einen Notizbuchstapel erstellen Sie, indem Sie ein Notizbuch auf ein anderes Notizbuch ziehen, wobei diese beiden Notizbücher dem neuen Notizbuchstapel untergeordnet werden. Diesen Notizbuchstapel können Sie nun nach Wunsch benennen.

Wer suchet, der findet

Wer große Mengen an Informationen in einer Datenbank wie der von Evernote speichert, möchte natürlich auch schnell über eine entsprechende Suchfunktion an die gewünschte Information gelangen.

Die Suchfunktion hat sich in meiner Praxis als großartig erwiesen. Sie liefert in Sekundenbruchteilen alle relevanten Notizen. Dabei geben Sie alle benötigten Suchbegriffe

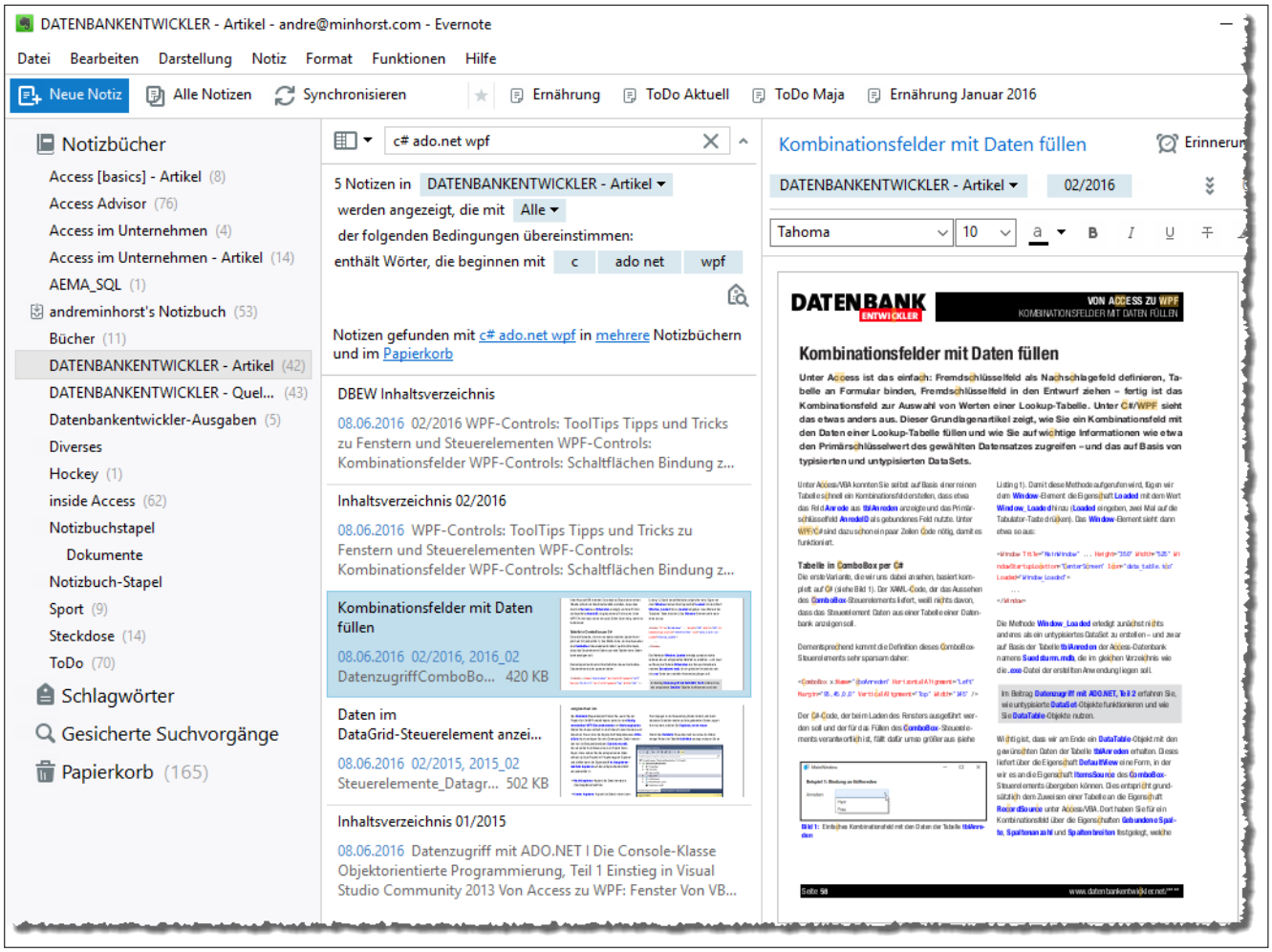

Bild 6: Die Suchfunktion liefert alle Ergebnisse für die mit »Und« verknüpften Suchbegriffe.

in das Suchfenster ein. Als Ergebnis liefert Evernote eine Liste aller Notizen, in denen alle Suchbegiffe vorkommen.

Der Clou ist, dass die Ergebnisliste nach der Eingabe eines jeden Zeichens aktualisiert wird, sodass Sie nicht immer noch auf eine Suchen-Schaltfläche klicken oder die Eingabetaste betätigen müssen (s. Bild 6).

Wenn Sie nach einer Phrase suchen möchten, fassen Sie die einzelnen Wörter einfach in Anführungszeichen ein. Außerdem können Sie einige spezielle Suchfunktionen nutzen, indem Sie folgende Operatoren voranstellen:

- **intitle**: Suchen im Titel der Notiz
- **notebook**: Name des Notizbuchs beziehungsweise des Ordners, in dem gesucht werden soll
- **any**: **Oder**-Verknüpfung der Suchbegriffe statt der standardmäßig verwendeten **Und**-Verknüpfung
- **tag**: Zeigt nur Notizen mit dem nachfolgend angegebenen Schlagwort an.
- **-tag**: Zeigt nur Notizen ohne das angegebene Schlagwort an.
- **resource**: Suche nach bestimmten Medientypen, zum Beispiel **resource:application/pdf** für PDF-Dokumente