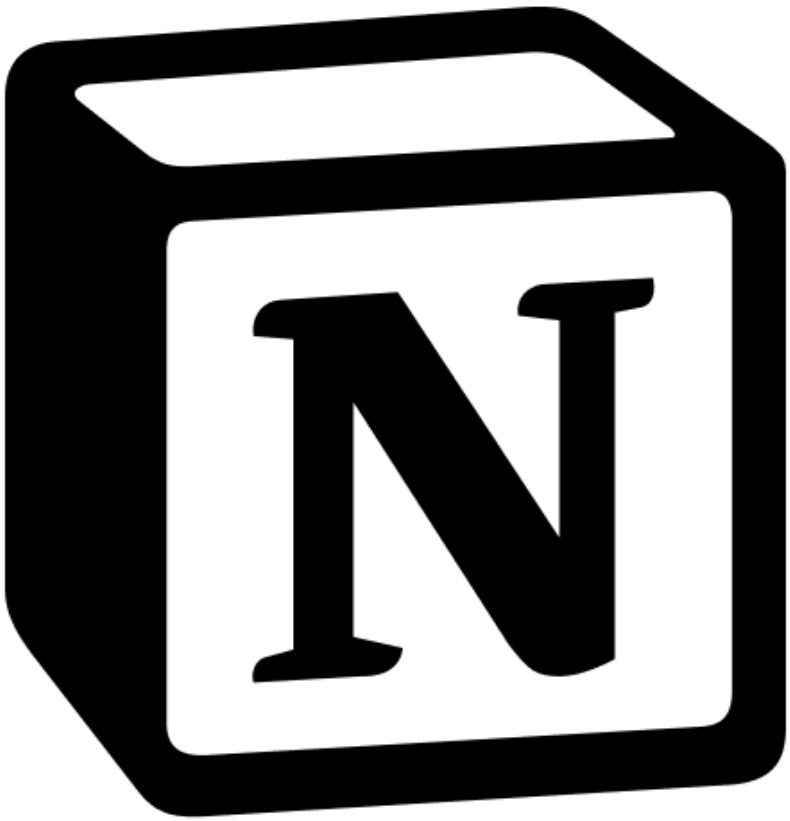
HAUFE.
Ausgabe 06/2022

ACCESS IM UNTERNEHMEN

PRODUKTIVITÄT MIT NOTION

Die neue Produktivitäts-App Notion soll viele Aufgaben vereinfachen. Noch besser klappt das, wenn wir von Access aus per VBA darauf zugreifen können (ab Seite 48).



In diesem Heft:

Produktivität mit Notion

Es gibt mittlerweile unzählige Blogs, YouTube-Kanäle, Bücher, Tools et cetera rund um das Thema Produktivität. Das Ziel ist, die täglichen Abläufe zu optimieren, weniger Arbeit gleichzeitig auf dem Schreibtisch zu haben, Aufgaben nach Themen zu sortieren und abzuarbeiten und die Zeitfenster für diese Aufgaben vorzudefinieren. Gerade für Access-Entwickler gibt es da viele nützliche Ansätze. Ein praktisches Tool ist Notion. Dabei handelt es sich um ein webbasiertes Tool, mit dem man Listen, Datenbanken oder Kalender verwalten kann. Aber was hilft schon ein Tool, auf das wir nicht mit Access zugreifen können? Genau das erledigen wir in einer Beitragsreihe, die in dieser Ausgabe startet.

Notion ist in einer einfachen Version kostenlos erhältlich und bietet sich deshalb zum Ausprobieren an. Hat man einmal begonnen, seine Projekte, Aufgaben und andere Daten in die Notion-Datenbanken zu schreiben und die verschiedenen Ansichten zu ihrer Darstellung zu nutzen, zum Beispiel als Liste mit Filter- und Sortiermöglichkeiten, in einem Kanban-Board oder, wenn die Daten eine Datumsangabe enthalten, auch in einem Kalender, will man nicht mehr davon weg. Einen kleinen Einblick in die Möglichkeiten bietet der Beitrag **Produktivität mit Notion steigern** ab Seite 48.

Doch wie eingangs erwähnt, wollen wir als Access- und VBA-Entwickler wissen, wie wir programmgesteuert auf die in Notion hinterlegten Daten zugreifen und diese gegebenenfalls ändern oder sogar neue Daten anlegen können. Damit können wir dann nicht nur die Daten aus Access-Tabellen, sondern gegebenenfalls auch Termine, Kontakte et cetera aus Outlook mit Notion synchronisieren. Wie wir per VBA aus einer Access-Datenbank auf Notion zugreifen, stellen wir im Beitrag **Mit Access auf Notion zugreifen** ab Seite 61 vor.

In zwei weiteren Beiträgen schauen wir uns das Ribbon von Access genauer an. Der erste heißt **Kontextabhängige tab-Elemente im Ribbon** (ab Seite 2). Hier untersuchen wir die sogenannten kontextabhängigen Tabs. Diese werden im Kontext mit bestimmten Objekttypen eingeblendet. So erscheint beispielsweise das Tab mit der Beschriftung **Tabelle Felder** erst, wenn der Benutzer eine Tabelle in der Datenblattansicht öffnet. Wir schauen uns an, welche kontextabhängigen Tabs es gibt und wie wir diese einblenden, ausblenden oder erweitern können.

Das gleiche Thema greifen wir nochmals im folgenden Beitrag **Ribbontab beim Öffnen eines Formulars anzeigen** ab Seite 9 auf. Hier stellen wir die verschiedenen Möglichkeiten vor, um Ribbonbefehle nur dann einzublenden, wenn der Benutzer ein bestimmtes Formular geöffnet hat.

Der Beitrag **Dynamische Bereichshöhe im Endlosformular** zeigt ab Seite 17, wie wir die Höhe mehrerer gleichzeitig angezeigter Bereiche im Endlosformular an die Höhe des Formulars anpassen können – so, dass eine bestimmte Anzahl Datensätze erscheinen.

Unsere Beitragsreihe zur Rechnungserstellung setzen wir mit den beiden Beiträgen **Rechnungsverwaltung: Kundenübersicht mit Suche** (ab Seite 21) und **Kunden nach bestellten Produkten filtern** (ab Seite 31) fort.

Schließlich stellen wir im Beitrag **Prüfen, ob Datenbank geöffnet ist** ab Seite 43 eine Funktion vor, mit der wir sichergehen können, dass wir exklusiv auf eine Datenbankdatei zugreifen können.

Und unter **Dateien per VBA öffnen** zeigen wir ab Seite 45, wie Sie Dateien mit der dafür geeigneten Anwendung starten können.

Jetzt aber viel Spaß beim Lesen und Ausprobieren!

Ihr André Minhorst

Kontextabhängige tab-Elemente im Ribbon

Wenn Sie schon einmal benutzerdefinierte Ribbondefinitionen in einer Ihrer Anwendungen eingesetzt haben, kennen Sie vielleicht auch schon die kontextabhängigen Tabs, die man mit ein paar Extra-Elementen definiert und die gemeinsam mit dem jeweils zugeteilten Formular angezeigt werden. Die Besonderheit ist, dass diese kontextabhängigen tab-Elemente, auf Englisch Contextual Tabs, optisch etwas anders angezeigt werden und zusätzlich zu den aktuell angezeigten Ribbon-Tabs erscheinen. Es gibt jedoch nicht nur kontextabhängige Tabs für Formulare und Berichte, sondern auch noch weitere, die beispielsweise in der Entwurfsansicht verschiedener Elemente erscheinen oder in der Datenblattansicht. In diesem Beitrag schauen wir uns an, welche es gibt und wie wir diese Tabs selbst erweitern oder anpassen können.

Im Beitrag **Ribbontab beim Öffnen eines Formulars anzeigen** (**www.access-im-unternehmen.de/1391**) schauen wir uns an, wie wir bei Anzeige eines Formulars ein benutzerdefiniertes, kontextabhängiges Ribbon-Tab einblenden können.

Dort nutzen wir unter anderem die Möglichkeit, ein neues Tab im Ribbon als kontextabhängiges **tab**-Element zu platzieren. Dazu fügen wir unter dem **ribbon**-Element nicht direkt das **tabs**-Element und darin die **tab**-Elemente ein, sondern schalten noch ein **contextualTabs**-Element und ein **tabSet**-Element dazwischen.

Dieses **tabSet**-Element können wir nicht mit einem individuellen Wert für das Attribut **id** versehen, sondern es gibt nur das Attribut **idMso**. Für dieses hinterlegen wir dort den Wert **TabSetFormReportExtensibility**.

Erst darunter können wir dann die **tab**-Elemente und die untergeordneten Elemente anlegen, die dann in einem optisch hervorgehobenen **tab**-Element angezeigt werden – allerdings nur, wenn die Ribbondefinition aktiv ist und ein Formular oder Bericht in der Formular- oder Berichtsansicht angezeigt wird.

Kontextabhängige tab-Elemente nur für Formulare?

Dieses kontextabhängige Element fügen wir normalerweise einer benutzerdefinierten Ribbondefinition hinzu, die über die Eigenschaft **Name des Menübands** einem Formular zugewiesen wird.

Dann erscheint das kontextabhängige **tab**-Element auch nur, wenn das Formular angezeigt wird.

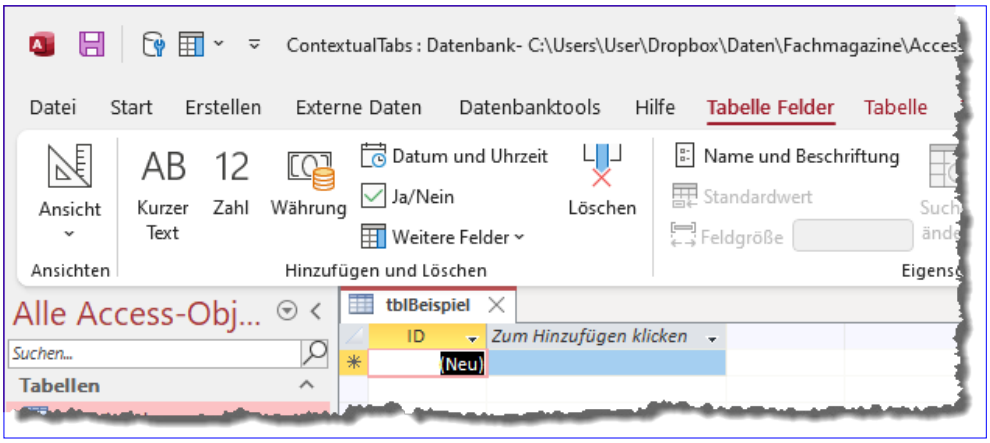


Bild 1: Kontextabhängige Tabs bei Anzeige einer Tabelle in der Datenblattansicht

Eingebaute tabSet-Elemente

Neben den kontextabhängigen **tabSet**-Elementen gibt es auch einige eingebaute **tabSet**-Elemente, die zum Beispiel beim Anzeigen verschiedener Datenbankobjekte rechts neben den Standard-Tabs erscheinen.

Ein Beispiel ist die Anzeige einer Tabelle in der Datenblattansicht (siehe Bild 1).

Benutzerdefinierte tabSet-Elemente

Wenn man an benutzerdefinierte kontextabhängige **tab**-Elemente denkt, fällt einem zuerst das für Formulare und Berichte ein. Wir können aber auch für alle anderen Objekte in verschiedenen Ansichten kontextabhängige Erweiterungen hinzufügen oder auch die vorhandenen Erweiterungen anpassen. Wir schauen uns zuerst einmal an, wie wir den eingebauten **tabSet**-Elementen eigene **tab**-Elemente hinzufügen können.

Eingebaute tabSet-Elemente erweitern

Bevor wir die eingebauten **tabSet**-Elemente um eigene **tab**-Elemente mit entsprechenden **group**-Elementen und darin enthaltenen Steuerelementen ergänzen können, müssen wir zuerst einmal die Werte kennen, die wir für das Attribut **idMso** angeben müssen, um die **tabSet**-Elemente zu referenzieren.

Deshalb schauen wir uns diese zuerst an. Dabei fangen wir mit den aktuell, also mit den Access-Versionen 2016 und 2019 kompatiblen **idMso**-Werten an:

- **TabSetTableToolsDatasheet**: Wird mit der Datenblattansicht einer Tabelle eingeblendet und enthält die beiden **tab**-Elemente **Tabelle Felder** und **Tabelle**.
- **TabSetTableToolsDesign**: Wird mit der Entwurfsansicht einer Tabelle eingeblendet und bietet das **tab**-Element **Tabellenentwurf** an.
- **TabSetQueryTools**: Wird mit der Entwurfsansicht und der SQL-Ansicht von Abfragen angezeigt und enthält das **tabSet**-Element namens **Abfrageentwurf**.
- **TabSetFormToolsLayout**: Wird mit der Layoutansicht von Formularen eingeblendet und zeigt die **tabSet**-Elemente **Entwurf des Formularlayouts**, **Anordnen** und **Format** an.
- **TabSetFormTools**: Erscheint mit der Entwurfsansicht eines Formulars und enthält die **tabSet**-Elemente **Formularentwurf**, **Anordnen** und **Format**.
- **TabSetFormDatasheet**: Wird mit der Datenblattansicht von Formularen angezeigt und liefert das **tabSet**-Element **Formulardatenblatt**.

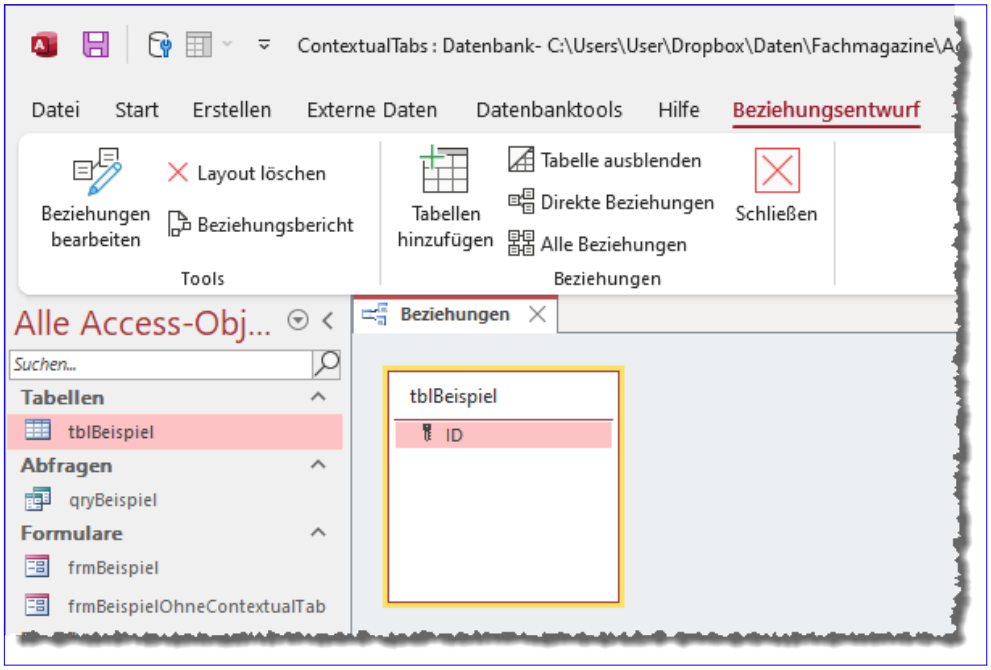


Bild 2: Contextual Tab für das Beziehungsfenster

- **TabSetReportToolsLayout**: Wird mit der Layoutansicht von Berichten aktiviert und enthält die **tabSet**-Elemente **Berichtslayoutentwurf**, **Anordnen**, **Format** und **Seite einrichten**.

- **TabSetReportTools**: Erscheint mit der Entwurfsansicht von Berichten und zeigt die **tabSet**-Elemente **Berichtsentwurf**, **Anordnen**, **Format** und **Seite einrichten** an.

- **TabSetMacroTools**: Wird mit der Entwurfsansicht eines Makros angezeigt und zeigt das **tabSet**-Element **Makroentwurf** an.

- **TabSetRelationshipTools**: Wird mit dem Beziehungsfenster eingeblendet und stellt das **tabSet**-Element Beziehungsentwurf bereit (siehe Bild 2).

Das **tabSet**-Element **TabSetFormReportExtensibility** ist ein Sonderfall. Eigentlich dachten wir, es würde nur angezeigt werden, wenn es in einer Ribbondefinition definiert ist, die einem Formular oder einem Bericht zugeordnet ist. Natürlich haben wir ausprobiert, dieses in einer Ribbondefinition anzulegen, die wir der Eigenschaft **Name des Menübands** der Anwendung selbst in den Access-Optionen zugewiesen haben. Haben wir dann ein Formular in der Formularansicht geöffnet, ist dieses jedoch nicht erschienen.

Wenn wir allerdings ein **tabSet** des Typs **TabSetFormReportExtensibility** für das Anwendungsribbon definieren und auch für ein Formular und das Formular dann anzeigen, dann erscheinen beide benutzerdefinierte **tabSet**-Elemente.

Es gibt noch einige weitere **idMso**-Werte für **tabSet**-Elemente, die für ältere Access-Versionen noch funktionieren und die vielleicht für den einen oder anderen Leser noch interessant sind. Zunächst die Elemente für die Anzeige von Tabellen in den Pivot-Ansichten, die schon länger nicht mehr verfügbar sind:

- **TabSetPivotTableAccess**

- **TabSetPivotChartAccess**

Noch länger fehlt die Möglichkeit, Access-Projekte (**.adp**) zu verwenden. Dennoch der Vollständigkeit halber hier die **idMso**-Werte für die verschiedenen Ansichten der Elemente, die es nur in **.adp**-Datenbanken gibt:

- **TabSetAdpFunctionAndViewTools**

- **TabSetAdpStoredProcedure**

- **TabSetAdpSqlStatement**

- **TabSetAdpDiagram**

Erweitern um benutzerdefinierte tab-Elemente

Wenn wir die eingebauten **tabSet**-Elemente um eigene tab-Elemente erweitern wollen, um den verschiedenen Ansichten von Tabellen, Abfragen, Formularen, Berichten, Makros und dem Beziehungsfenster neue Steuerelemente hinzuzufügen, benötigen wir also jeweils das **contextualTabs**-Element, darin das entsprechende **tabSet**-Element mit einer der oben angegebenen **idMso**-Werte und darin die gewünschten **tab**-, **group**- und Steuerelemente.

In Listing 1 finden Sie Beispiele für alle **tabSet**-Elemente, denen wir jeweils ein **tab**-Element hinzugefügt haben. Wenn Sie diese Ribbondefinition mit dem Wert **Main** im Feld **RibbonName** in der Tabelle **USysRibbons** speichern, die Anwendung neu starten, den Wert **Main** für die Eigenschaft **Name des Menübands** in den Access-Optionen einstellen und die Anwendung nochmals neu starten, wird diese Definition angewendet.

Es erscheint dann beispielsweise bei der Anzeige einer Tabelle in der Entwurfsansicht ein zusätzliches **tab**-Element wie in Bild 3. Das ist nicht nur hilfreich bei der Identifikation der einzelnen **tabSet**-Elemente, sondern kann auch

Ribbontab beim Öffnen eines Formulars anzeigen

Sie kennen sicher die Ribbontabs, die erscheinen, wenn Sie bestimmte Objekte in Access öffnen. Wenn Sie eine neue Tabelle anlegen, erscheint beispielsweise ein Tab namens »Tabellenentwurf«. Wechseln Sie zur Datenblattansicht der Tabelle, erscheinen die Tabs »Tabelle Felder« und »Tabelle«. Die Gemeinsamkeiten dieser Elemente sind, dass diese sich optisch ein wenig von den links davon befindlichen Tabs unterscheiden. Wie genau, hängt von der jeweils verwendeten Access-Version ab. In diesem Beitrag schauen wir uns an, wie wir überhaupt Ribbons mit einem Formular einblenden und dieses aktivieren und wie wir kontextabhängige Ribbons programmieren können.

Verschiedene Möglichkeiten

Wir stellen in diesem Beitrag die folgenden Varianten für das Anzeigen eines Ribbontabs beim Öffnen eines Formulars vor:

- Einfaches zusätzliches Tab, das mit dem Öffnen des Formulars erscheint, aber nicht aktiviert wird
- Tab, das alle anderen Elemente ausblendet und deshalb den Fokus erhält
- Tab, das als kontextabhängiges Tab ausgelegt ist und direkt mit dem Anzeigen des Formulars erscheint und aktiviert wird – aber nur beim ersten Aufruf des Formulars
- Zusätzliches Tab, das mit dem Formular erscheint und auch direkt aktiviert wird, und zwar bei jedem Öffnen des Formulars erneut.
- Die dort verwendete Technik wenden wir dann auch noch auf ein kontextabhängiges Tab an, damit dieses nicht nur beim ersten Anzeigen aktiviert wird, sondern bei jedem Öffnen des Formulars.

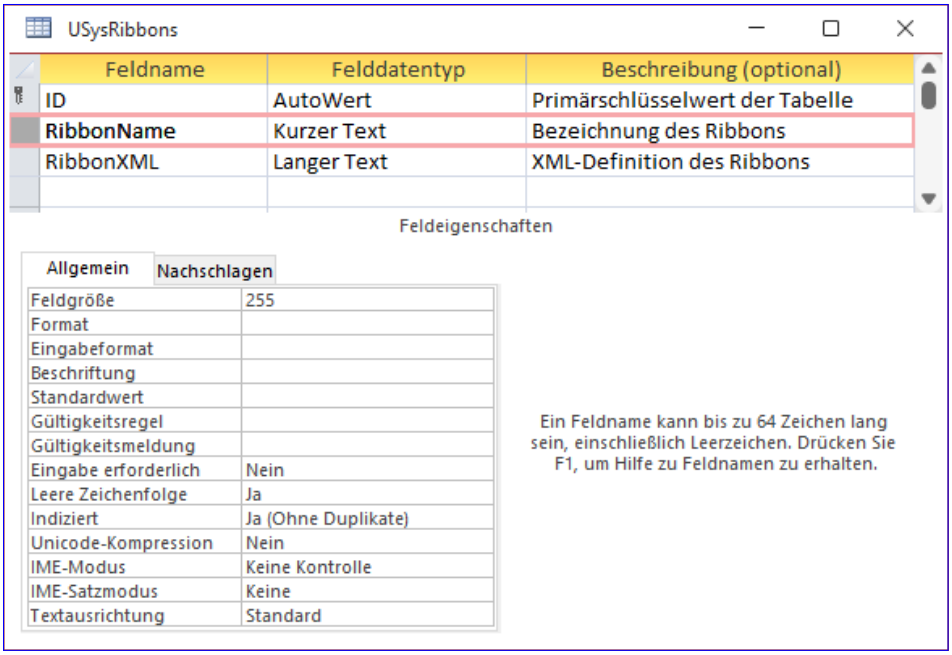


Bild 1: Tabelle zum Speichern der Ribbondefinitionen

Vorbereitung für den Einsatz von Ribbons

Bevor wir uns die Eigenarten der verschiedenen Ribbondefinitionen ansehen, schaffen wir die Voraussetzungen für die Anzeige von Ribbons. Dazu benötigen wir als Erstes eine Tabelle namens **USysRibbons**, welche die Ribbondefinitionen samt Bezeichnung speichert. Diese enthält die Felder **ID**, **RibbonName** (mit eindeutigem Index, damit jede Bezeichnung nur einmal vergeben wird) und **RibbonXML** und sieht im Entwurf wie in Bild 1 aus. Nach dem Speichern verschwindet diese Tabelle direkt, da die Bezeichnung **USys...** wie **MSys...** Systemobjekten vorbe-

halten ist, die nicht standardmäßig im Navigationsbereich angezeigt werden.

Außerdem benötigen wir ein Modul namens **mdlRibbons**, in das wir den Code der durch das Ribbon ausgelösten Ereignisse schreiben sowie ein Modul namens **mdlRibbonImages**. Dieses können Sie aus der Beispieldatenbank herauskopieren – sie ist bereits mit einigen Routinen gefüllt, welche die Anzeige von Bildern im Ribbon ermöglichen. Um Fehler in den Ribbondefinitionen zu erkennen, aktivieren wir die Option **Fehler von Benutzeroberflächen-Add-Ins anzeigen** in den Access-Optionen unter **Clienteinstellungen**.

Definition des Anwendungsribbons, das die Formulare öffnet

Als Erstes erstellen wir eine Ribbondefinition, die beim Öffnen der Anwendung erscheint und Schaltflächen enthält, mit denen wir die Formulare der nachfolgend beschriebenen Beispiele öffnen können. Diesen weisen wir dann jeweils eine Ribbondefinition zu, die mit dem Öffnen des Formulars angewendet wird.

Diese Definition sieht wie in Listing 1 aus und damit sie für die Anwendung verfügbar ist, fügen wir einen neuen Datensatz zur Tabelle **USysRibbons** hinzu, der im Feld **RibbonName** den Wert **Main** enthält und im Feld **RibbonXML** das XML-Dokument.

Im Ribbon-Dokument definieren wir ein **Ribbon**-Element, das durch den Wert **True** für das Attribut **startFromScratch** die eingebauten Elemente ausblendet – so ist es übersichtlicher. Außerdem enthält es fünf **button**-Elemente, die alle jeweils mit dem Attribut **onAction** ausgestattet sind.

Ribbon als Anwendungsribbon festlegen

Damit die Anwendung dieses Ribbon beim Öffnen anzeigt, müssen wir dieses als Anwendungsribbon festlegen. Damit dies überhaupt möglich ist, muss Access dieses

```
<?xml version="1.0"?>
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui" loadImage="loadImage">
  <ribbon startFromScratch="true">
    <tabs>
      <tab id="tabAnwendung" label="Anwendung">
        <group id="grpFormulare" label="Formulare">
          <button image="table_selection_row" label="Kundendetails" id="btnKundendetails" onAction="onAction"
            size="large"/>
          <button image="form" label="Kundendetails StartFromScratch"
            id="btnKundendetailsStartFromScratch" onAction="onAction" size="large"/>
          <button image="table_selection_all" label="Kundendetails Contextual" id="btnKundendetailsContextual"
            onAction="onAction" size="large"/>
          <button image="tables" label="Kundendetails mit Fokus" id="btnKundendetailsFokus" onAction="onAction"
            size="large"/>
          <button image="table_selection_cell" label="Kundendetails Contextual Fokus"
            id="btnKundendetailsContextualFocus" onAction="onAction" size="large"/>
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

Listing 1: Definition des Anwendungsribbons

Ribbon erst einmal als vorhanden erkennen. Das einfache Hinzufügen eines Datensatzes zur Tabelle **USysRibbons** reicht dazu nicht aus, da die Inhalte nur beim Öffnen der Access-Datenbank eingelesen werden. Daher schließen wir die Datenbankanwendung und öffnen diese erneut. Anschließend können wir die Ribbondefinition in den Access-Optionen als Anwendungsribbon einstellen. Dazu wählen wir dort im Bereich **Aktuelle Datenbank** unter **Menüband- und Symbolleistenoptionen** für die Eigenschaft **Name des Menübands** den Wert **Main** aus (siehe Bild 2).

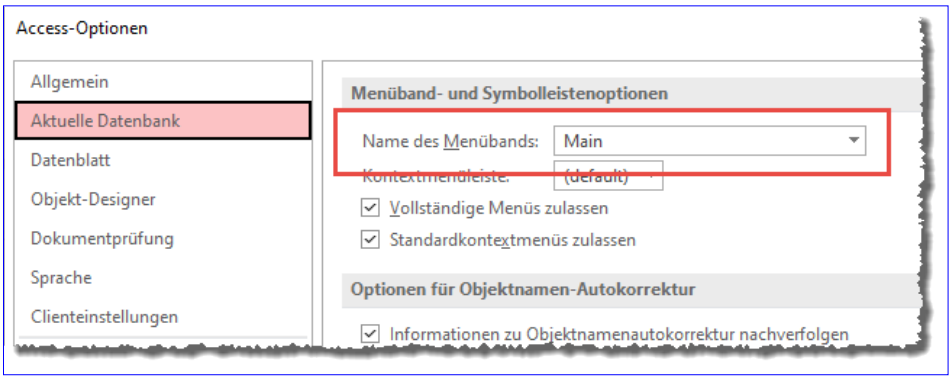


Bild 2: Einstellen des Anwendungsribbons

Damit die Anwendung dieses Ribbon anzeigt, müssen wir sie nun nochmals schließen und erneut öffnen. Erst dann prüft Access beim Öffnen, ob die Eigenschaft **Name des Menübands** einen Wert enthält und wendet dann die in der Tabelle **USysRibbons** gespeicherte Ribbondefinition an. Das Ergebnis sieht dann schließlich wie in Bild 3 aus.

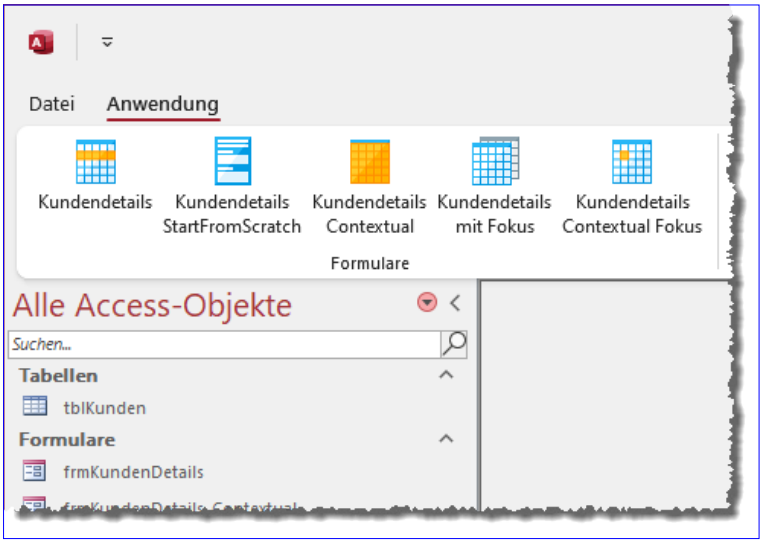


Bild 3: Das von uns definierte Anwendungsribbon

Die onAction-Prozedur

Beim Anklicken einer der Ribbon-Schaltflächen, die mit dem **onAction**-Attribut versehen sind, rufen diese die Prozedur **onAction** auf, die wir im Modul **mdlRibbons** wie folgt hinterlegt haben:

```
Sub onAction(control As IRibbonControl)
    Select Case control.ID
        Case "btnKundenDetails"
            DoCmd.OpenForm "frmKundenDetails"
        Case "btnKundenDetailsStartFromScratch"
            DoCmd.OpenForm _
                "frmKundenDetails_StartFromScratch"
        Case "btnKundendetailsContextual"
            DoCmd.OpenForm "frmKundendetails_Contextual"
        Case "btnKundendetailsFokus"
            DoCmd.OpenForm "frmKundendetails_Fokus"
        Case "btnKundendetailsContextualFocus"
            DoCmd.OpenForm _
                "frmKundendetails_Contextual_Fokus"
        Case Else
            Debug.Print control.ID
    End Select
End Sub
```

Die Prozedur prüft jeweils, welches **button**-Element diese aufgerufen hat und öffnet im jeweiligen **Case**-Zweig der **Select Case**-Anweisung das entsprechende Formular.

Dynamische Bereichshöhe im Endlosformular

Ein Leser stellte mir neulich die Frage, ob und wie man die Höhe der einzelnen Bereiche im Endlosformular dynamisch so einstellen kann, dass beispielsweise immer drei Datensätze angezeigt werden – auch, wenn der Benutzer die Höhe des Endlosformulars während der Anzeige ändert. Dieser Beitrag zeigt, wie das möglich ist. Dabei behelfen wir uns mit einer Ereignisprozedur, die immer beim Ändern der Größe eines Formulars ausgelöst wird – und eines kleinen Tricks, der wegen der enthaltenen Steuerelemente nötig wurde.

Aufgabenstellung

Zum besseren Verständnis hilft der Screenshot aus Bild 1. Wenn der Benutzer die Höhe des Formulars ändert, dann sollen die drei angezeigten Detailbereiche ihre Höhe automatisch so ändern, dass sie immer jeweils 1/3 des verfügbaren Platzes einnehmen.

1/3 deshalb, weil wir uns für das Beispiel in diesem Beitrag dazu entschieden haben, dass das Endlosformular immer drei Datensätze anzeigen soll (außer natürlich, es enthält weniger als drei Datensätze oder der Benutzer scrollt ganz nach unten).

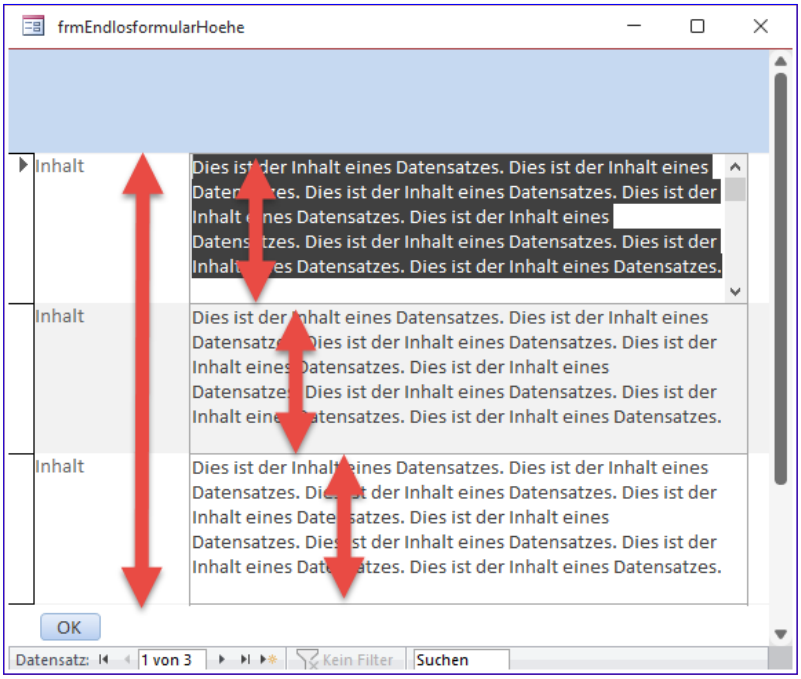


Bild 1: So soll die Höhe der Detailbereiche angepasst werden, wenn der Benutzer die Höhe des Formulars ändert.

Beispielformular

Als Beispiel verwenden wir das Formular **frmEndlosformularHoehe**. Es verwendet die Tabelle **tblInhalte** als Datensatzquelle. Diese Tabelle enthält nur die beiden Felder **ID** und **Inhalt** (Felddatentyp **Langer Text**).

Wir ziehen nun das Feld **Inhalt** aus der Feldliste in den Detailbereich und passen Höhe von Detailbereich und Textfeld einander an. Außerdem stellen wir die Eigenschaft **Standardansicht** auf **Endlosformular** ein und aktivieren die in Endlosformularen häufig sichtbaren Bereiche **Formularkopf** und **Formularfuß** (siehe Bild 2).

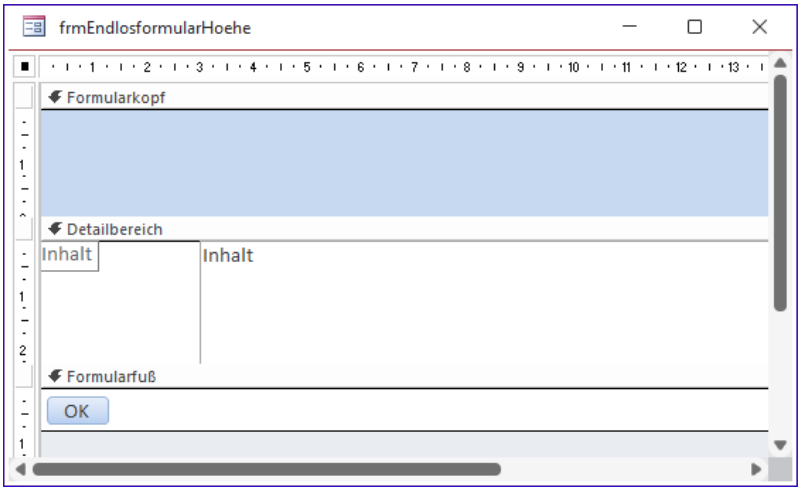


Bild 2: Entwurf des Formulars

Umsetzung mit dem Ereignis »Bei Größenänderung«

Wer schon ein wenig mit Formularen programmiert hat, der weiß: Wir werden vermutlich mit dem Ereignis **Bei Größenänderung** arbeiten, um das gewünschte Ergebnis zu erreichen.

Und wir benötigen einige Kenntnisse über die Eigenschaften, mit denen wir die Höhen verschiedener Bereiche des Formulars und des Formulars selbst ermitteln können.

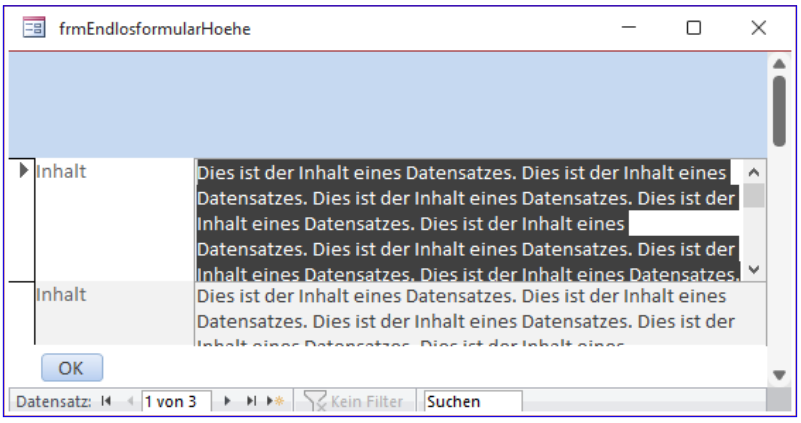


Bild 3: Die Detailbereiche werden nur bis zur Entwurfshöhe verkleinert.

Höhe der verschiedenen Bereiche

Man mag vermuten, dass man die Höhe eines Formulars mit einer Eigenschaft wie **Me.Height** ermitteln kann. Weit gefehlt: Diese Eigenschaft bietet das Formular gar nicht an. Die **Height**-Eigenschaft finden wir allerdings für die verschiedenen Bereiche des Formulars vor – für den Detailbereich, den Formularkopf und den Formularfuß beispielsweise. Wenn wir allerdings ermitteln wollen, wie hoch der Detailbereich sein darf, damit genau drei Datensätze im Bereich zwischen Formularkopf und Formularfuß sichtbar sind, benötigen wir eine Information über die gesamte Höhe des Formulars.

Ein genauerer Blick in das Objektmodell liefert schnell die Eigenschaft **InsideHeight**. **InsideHeight** liefert genau die Höhe des sichtbaren Bereichs innerhalb des Formularrahmens. Damit können wir die Höhe des Detailbereichs für drei Datensätze nach der Formel **(Gesamthöhe – Formularkopf – Formularfuß) / 3** berechnen.

In einem ersten, naiven Ansatz könnten wir also die folgende Anweisung in die Ereignisprozedur schreiben, die durch das Ereignis **Bei Größenänderung** ausgelöst wird:

```
Private Sub Form_Resize()
    Me.Detailbereich.Height = (Me.InsideHeight - _
        Me.Formularkopf.Height - Me.Formularfuß.Height) / 3
End Sub
```

Damit kommen wir schon erstaunlich weit. Gleich beim Öffnen zeigt das Formular drei Detailbereiche an, die genau den Platz zwischen Formularkopf und Formularfuß einnehmen. Wenn wir die Höhe des Formulars vergrößern, passt sich die Höhe der Detailbereiche automatisch an. Wenn wir die Höhe des Formulars allerdings so weit verkleinern, dass die im Entwurf eingestellte Höhe der Detailbereiche unterschritten wird, verkleinert die Prozedur die Höhe der Detailbereiche nicht mehr (siehe Bild 3).

Warum ist das der Fall? Der Grund ist schnell gefunden: Die enthaltenen Textfelder ändern ihre Höhe nicht analog zu der Höhe der Detailbereiche. Auch die Einstellung der Eigenschaft **Horizontaler Anker** auf den Wert **Beide** bewirkt keine Änderung. Die erste alternative Idee hierzu ist, die Höhe des Textfeldes **Inhalt** einfach auf die gleiche Höhe einzustellen wie den Detailbereich:

```
Private Sub Form_Resize()
    Me.Detailbereich.Height = (Me.InsideHeight - _
        Me.Formularkopf.Height - Me.Formularfuß.Height) / 3
    Me!Inhalt.Height = Me.Detailbereich.Height
End Sub
```

Das klappt auch, solange man die Höhe des Formulars nur vergrößert. Sobald wir die Höhe verkleinern, tut sich nichts mehr – weder die Höhe des Detailbereichs noch die des Textfeldes ändert sich. Auch dafür gibt es einen guten Grund. Wir versuchen hier, zuerst die Höhe des Detailbe-

Rechnungsverwaltung: Kundenübersicht mit Suche

Wenn ein Kunde anruft, möchten Sie schnell den entsprechenden Kundendatensatz auf dem Bildschirm haben. Dazu stellen wir im vorliegenden Beitrag ein Formular samt Unterformular zusammen, mit denen die gewünschten Daten schnell ermittelt werden können. Im Hauptformular bieten wir einige Suchfunktionen an, im Unterformular liefern wir die den Suchkriterien entsprechenden Daten in der Datenblattansicht. Außerdem soll das Formular die Möglichkeit bieten, den gefundenen Kundendatensatz im Detailformular zu öffnen, damit wir auch noch die Bestellungen des Kunden einsehen können.

Unterformular für die Kundenliste

Bevor wir beginnen, das Hauptformular zu programmieren, legen wir zunächst das Unterformular an. Dann können wir dieses gleich im Anschluss direkt zum Hauptformular hinzufügen.

Das neue Unterformular soll **sfmKundenuebersicht** heißen und standardmäßig die Daten der Tabelle **tblKunden** anzeigen. Deshalb stellen wir seine Eigenschaft **Datensatzquelle** auf diese Tabelle ein. Anschließend können wir alle Felder der Feldliste in den Detailbereich des Formulars ziehen. Gegebenenfalls lassen wir das Primärschlüsselfeld **ID** weg, da dieses nur zum Herstellen von Beziehungen dient und keine geschäftliche Funktion hat.

Damit das Formular die Daten in der Datenblattansicht anzeigt, legen wir für die Eigenschaft **Standardansicht** noch den Wert **Datenblatt** fest (siehe Bild 1). Damit können wir die Arbeiten am Unterformular bereits beenden und dieses schließen.

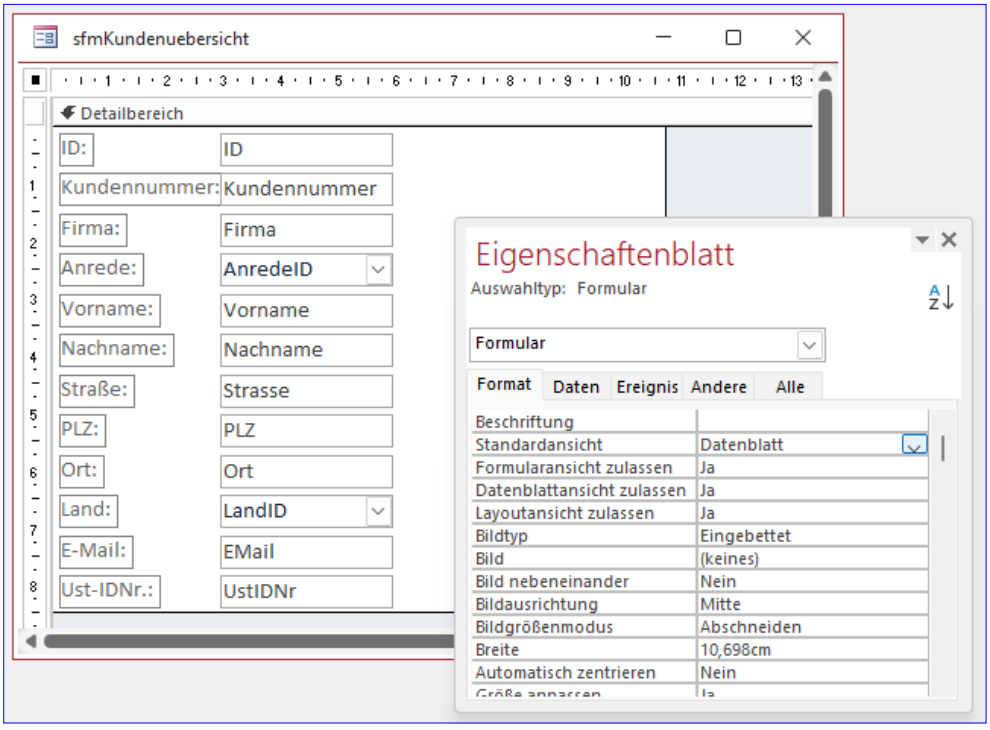


Bild 1: Entwurf des Unterformulars **sfmKundenuebersicht**

Hauptformular erstellen

Anschließend erstellen wir das Hauptformular und speichern dieses direkt unter dem Namen **frmKundenuebersicht**. Diesem weisen wir keine Datensatzquelle zu, weil es selbst keine Daten anzeigen soll – diese liefert allein das Unterformular. Im Hauptformular wollen wir nur Steuerelemente zum Durchsuchen der Kundendaten und zum Öffnen von Detaildatensätzen bereitstellen. Daher benötigen wir im Hauptformular auch nicht die Steuerelemente zum Navigieren in Datensätzen und stellen daher

einige Eigenschaften ein. Die Eigenschaften **Navigationsschaltflächen**, **Datensatzmarkierer**, **Bildlaufleisten** und **Trennlinien** setzen wir auf den Wert **Nein**, die Eigenschaft **Automatisch zentrieren** auf **Ja**.

Dann ziehen wir aus dem Navigationsbereich das Unterformular **sfmKundenuebersicht** in den Detailbereich des Formularentwurfs von **frmKundenuebersicht**. Das Bezeichnungsfeld des Unterformular-Steuerelements entfernen wir, da wir ja wissen, dass dieses Kundendaten anzeigt.

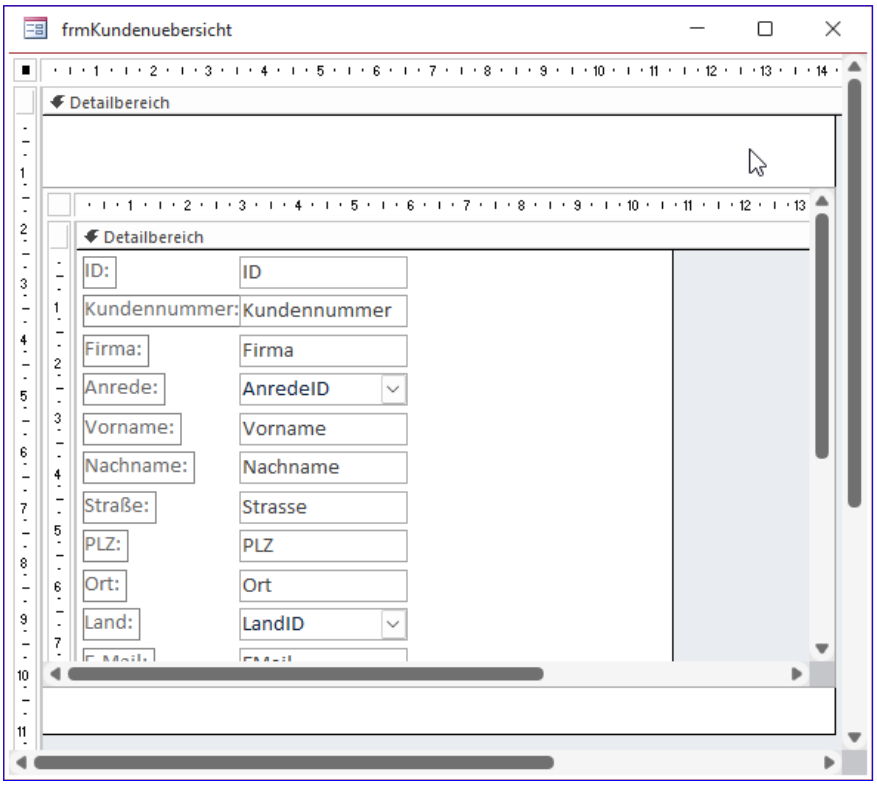


Bild 2: Haupt- und Unterformular im Entwurf

Damit das Unterformular beim Vergrößern des Hauptformulars ebenfalls vergrößert wird, stellen wir seine Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** jeweils auf **Beide** ein.

Wenn wir oben und unten ein wenig Platz für die Steuerelemente zum Suchen und zum Aufrufen der Anzeige der Kundendetails lassen, sieht der Entwurf nun wie in Bild 2 aus.

Steuerelemente zum Filtern der Kunden hinzufügen

Damit kommen wir zu den Steuerelementen, mit denen wir die Filterkriterien für die anzuzeigenden Kunden eingeben wollen. Diese platzieren wir wie in Bild 3 im Bereich über dem Unterformular. Die Steuerelemente heißen:

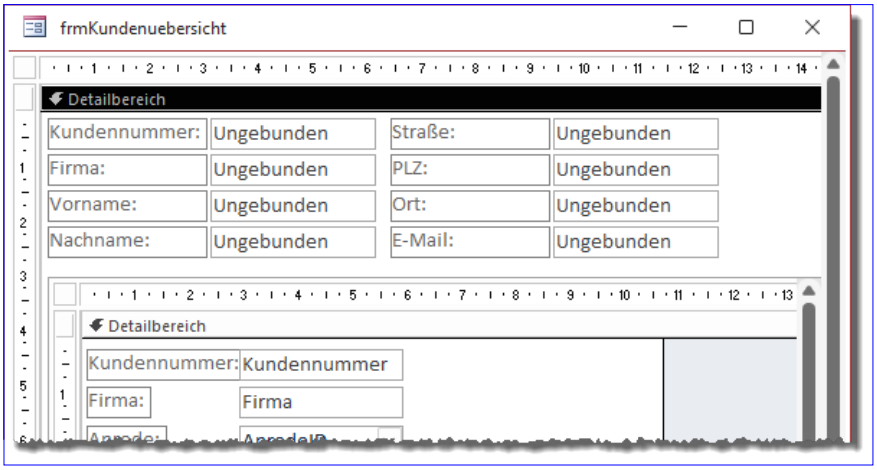


Bild 3: Hinzufügen der Steuerelemente zum Filtern der Kundendatensätze

- **txtFilterKundennummer**
- **txtFilterFirma**
- **txtFilterVorname**
- **txtFilterNachname**
- **txtFilterStrasse**
- **txtFilterPLZ**
- **txtFilterOrt**

- **txtFilterEMail**

Das Filtern wollen wir einfach und schnell realisieren: Jede Eingabe in eines der Filterfelder soll unmittelbar die gefilterten Daten anzeigen. Außerdem soll immer im kompletten zu filternden Feld nach dem im jeweiligen Filterfeld eingegebenen Text gesucht werden. Sprich: Wenn der Benutzer im Feld **txtFilterKundennummer** den Wert **1** eingibt, sollen alle Datensätze angezeigt werden, die an beliebiger Stelle im Feld **Kundennummer** den Wert **1** enthalten.

Das heißt auch, dass wir für jedes der Steuerelemente zur Eingabe der Filterkriterien das Ereignis **Bei Änderung** implementieren müssen. Mit diesem rufen wir dann jeweils eine weitere Prozedur auf, welche die jeweiligen Werte der Filter-Textfelder ausliest und ein entsprechendes SQL-Kriterium zusammenstellt.

Unterschiedliche Eigenschaften zum Ermitteln der Vergleichskriterien

Das Problem dabei ist, dass wir für die Inhalte der Filter-Textfelder auf unterschiedliche Eigenschaften zugreifen müssen. Normalerweise greifen wir einfach auf die **Value**-Eigenschaft zu. Diese können wir allerdings nicht nutzen, wenn wir während der Eingabe auf den Inhalt im aktuellen Textfeld zugreifen wollen. Der Grund ist, dass die **Value**-Eigenschaft erst mit dem aktuell im Textfeld enthaltenen Wert gefüllt wird, wenn der Benutzer die Eingabe bestätigt – beispielsweise, indem er das Textfeld verlässt und den Fokus auf das nächste Textfeld verschiebt. In diesem Fall lässt sich der aktuell im Textfeld dargestellte Text nur mit der Eigenschaft **Text** auslesen.

Wir müssen also zumindest für das aktuelle Textfeld auf die **Text**-Eigenschaft zugreifen und können nicht die **Value**-Eigenschaft nutzen. Aber warum ist das überhaupt ein Problem? Können wir nicht einfach auf den Inhalt aller Felder über die **Text**-Eigenschaft zugreifen? Das wiederum gelingt nicht, weil man nur für das Textfeld auf die **Text**-Eigenschaft zugreifen kann, das aktuell den Fokus hat.

Wir müssen also den Inhalt des aktuell bearbeiteten Textfeldes mit der **Text**- und den der übrigen Textfelder mit der **Value**-Eigenschaft abfragen und prüfen, ob diese Kriteriumsausdrücke enthalten. Wobei die **Value**-Eigenschaft wiederum die Standardeigenschaft der **TextBox**-Klasse ist, weshalb wir diese gar nicht angeben müssen. Statt **Me!txtFilterKundennummer.Value** können wir einfach **Me!txtFilterKundennummer** schreiben.

Prozedur zum Ermitteln des Filterkriteriums für das Unterformular

Wir legen also zunächst für jedes der Filter-Textfelder eine Ereignisprozedur für das Ereignis **Bei Änderung** an. Diese enthält für alle Filter-Textfelder den Aufruf einer weiteren Prozedur namens **KundenFiltern** – wie hier am Beispiel der Prozedur **txtFilterEMail** zu sehen:

```
Private Sub txtFilterEMail_Change()
    KundenFiltern
End Sub
```

Einfache Variante zum Zusammensetzen des Filterkriterums

Die Prozedur zum Zusammenstellen eines Filterausdrucks für das Unterformular können wir geradlinig und einfach aufbauen oder auch auf den ersten Blick etwas komplizierter, dafür aber flexibler. Wir schauen uns beide Varianten an. Die erste finden Sie in Listing 1.

Diese Version verwendet eine Variable namens **strFilter**, um den Filterausdruck darin zusammenzustellen. Sie prüft für jedes der Filter-Textfelder, ob es sich dabei um das aktuelle Steuerelement handelt, also das Steuerelement, das aktuell den Fokus enthält. Das dient der Unterscheidung, ob wir die **Text**- oder die **Value**-Eigenschaft zum Ermitteln des jeweils in dem Steuerelement enthaltenen Textes verwenden müssen.

Um herauszufinden, ob wir es bei dem aktuellen Steuerelement mit dem aktiven Steuerelement zu tun haben, vergleichen wir dieses mit dem Verweis auf das Steuer-

```
Private Sub KundenFiltern()
    Dim strFilter As String
    If Me!txtFilterKundennummer Is Me.ActiveControl Then
        If Not Len(Me!txtFilterKundennummer.Text) = 0 Then
            strFilter = strFilter & " AND Kundennummer LIKE '*" & Me!txtFilterKundennummer.Text & "*'"
        End If
    Else
        If Not Len(Me!txtFilterKundennummer.Value) = 0 Then
            strFilter = strFilter & " AND Kundennummer LIKE '*" & Me!txtFilterKundennummer.Value & "*'"
        End If
    End If
    If Me!txtFilterFirma Is Me.ActiveControl Then
        If Not Len(Me!txtFilterFirma.Text) = 0 Then
            strFilter = strFilter & " AND Firma LIKE '*" & Me!txtFilterFirma.Text & "*'"
        End If
    Else
        If Not Len(Me!txtFilterFirma.Value) = 0 Then
            strFilter = strFilter & " AND txtFilterFirma LIKE '*" & Me!txtFilterFirma.Value & "*'"
        End If
    End If
    '...
    If Not Len(strFilter) = 0 Then
        strFilter = Mid(strFilter, 5)
        With Me!sfmKundenuebersicht.Form
            .Filter = strFilter
            .FilterOn = True
        End With
    Else
        Me!sfmKundenuebersicht.Form.Filter = ""
    End If
End Sub
```

Listing 1: Prozedur zum Einstellen des Filters für das Unterformular, einfache Version

element, das wir mit der Eigenschaft **ActiveControl** des Formulars ermitteln können. Sind beide gleich, handelt es sich um das aktive Steuerelement, sonst nicht. Ist das Steuerelement das aktive Steuerelement und ist dieses nicht leer, lesen wir den Inhalt mit der **Text**-Eigenschaft und stellen damit einen Ausdruck wie den folgenden zusammen:

```
AND Kundennummer LIKE '*Vergleichsausdruck*'
```

Vergleichsausdruck entspricht dabei dem Inhalt des jeweiligen Textfeldes. Falls es sich bei dem aktuell untersuchten Steuerelement nicht um das aktive Steuerelement handelt, verwenden wir für den gleichen Ausdruck den Inhalt der **Value**-Eigenschaft des Textfeldes.

Das Problem ist, dass wir in dieser Version für jedes Textfeld die folgenden Zeilen, hier beispielhaft am Textfeld **txtFilterKundennummer** gezeigt, in der Prozedur anlegen müssen:

```
If Me!txtFilterKundennummer Is Me.ActiveControl Then
    If Not Len(Me!txtFilterKundennummer.Text) = 0 Then
        strFilter = strFilter & " AND Kundennummer 7
```

Kunden nach bestellten Produkten filtern

Kunden nach bestellten Produkten kann jeder filtern, der sich ein wenig mit dem Abfrageentwurf beschäftigt hat. Etwas aufwendiger ist es schon, ein Formular zu erstellen, das verschiedene Möglichkeiten zum Filtern von Kunden nach den bestellten Produkten bietet. Hier wollen wir beispielsweise ein Produkt auswählen, sodass direkt alle Kunden in einer Liste angezeigt werden, die dieses Produkt bestellt haben. Oder wir gehen noch einen Schritt weiter und wollen Kunden anzeigen, die mindestens eines von mehreren Produkten geordert haben. Um dann vielleicht noch solche Kunden auszuschließen, die bereits ein bestimmtes anderes Produkt besitzen. Also auf ins Abenteuer!

Warum Kunden nach Produkten filtern?

Bevor wir uns an die Arbeit dieses recht aufwendigen Unterfangens machen, wollen wir uns überlegen, wozu wir das Ergebnis überhaupt nutzen können. Mir als Shopbetreiber fällt da direkt ein, Kunden, die ein bestimmtes Produkt erworben haben, über eine neue Version dieses Produkts zu informieren. Oder man möchte dem Kunden mitteilen, dass eine Lizenz ausläuft, damit er diese verlängern kann. Vielleicht wollen wir auch einfach Kunden, die Produkt A bestellt haben, eine Empfehlung zu dem dazu passenden Produkt B geben. Dabei wollen wir dann natürlich nur Kunden anschreiben, die Produkt B noch nicht bestellt haben. Sie sehen: Es lohnt sich, die Kunden nach den bestellten Produkten selektieren zu können.

Voraussetzungen

Als Basis für die Ermittlung von Kunden nach den bestellten Produkten verwenden wir die Beispieldatenbank, die wir bereits in der Beitragsreihe zur Rechnungsverwaltung vorgestellt haben. Den ersten Teil dieser Beitragsreihe finden Sie übrigens unter dem Titel **Rechnungsverwaltung: Datenmodell** (**www.access-im-unternehmen.de/1385**).

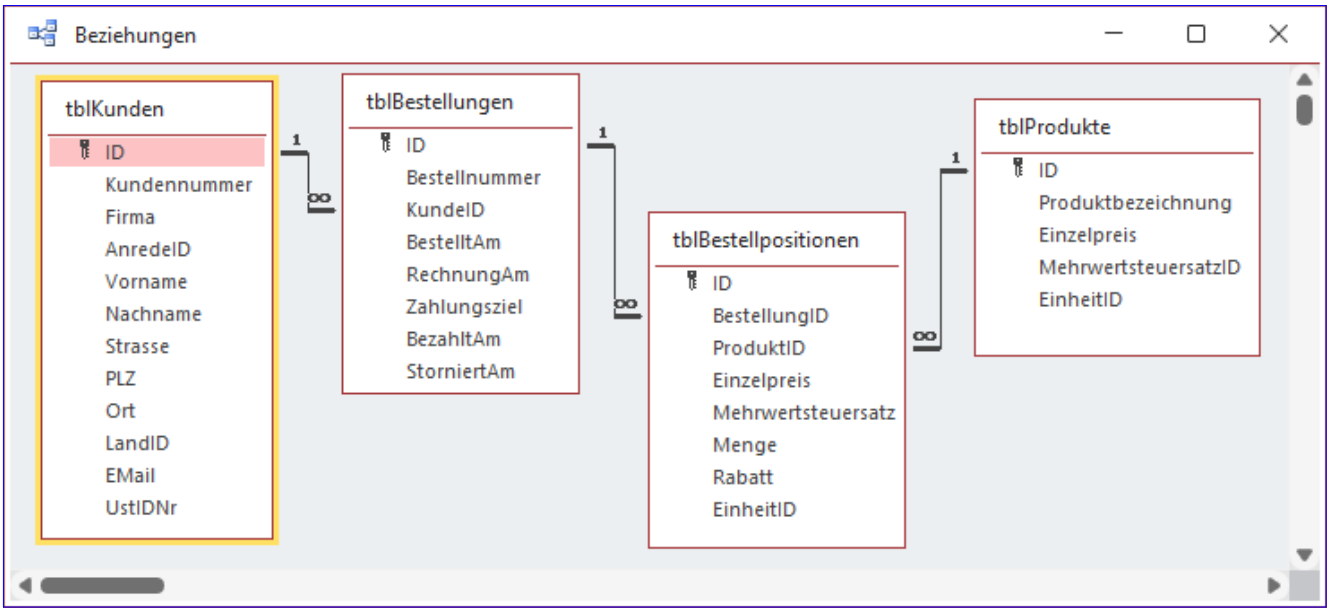


Bild 1: Tabellen, die am Filtern von Kunden nach Produkten beteiligt sind

Der relevante Teil des Datenmodells dieser Datenbank ist in Bild 1 abgebildet.

Geplante Funktionen für die Lösung

Wir wollen mit dem Formular zum Filtern von Kunden nach bestellten Produkten möglichst flexibel arbeiten können.

Also schauen wir uns zunächst an, welche Filtermöglichkeiten wir abbilden wollen:

- Filtern nach Kunden, die ein bestimmtes Produkt bestellt haben
- Filtern nach Kunden, die eines von mehreren Produkten bestellt haben (Oder-Verknüpfung)
- Filtern nach Kunden, die ein bestimmtes Produkt nicht bestellt haben
- Filtern nach Kunden, die bestimmte Produkte nicht bestellt haben (Und-Verknüpfung)

Die ersten beiden Filtermöglichkeiten sollen außerdem per Und-Verknüpfung mit den letzten beiden Möglichkeiten kombiniert werden.

Einfache Auswahl der Produkte, nach denen gefiltert werden soll

Wir wollen zwei Listenfelder im Formular anzeigen, die unterschiedliche Produkte enthalten:

- Das erste Listenfeld soll die Produkte anzeigen, von denen der Kunde mindestens eines bestellt hat.
- Das zweite Listenfeld soll die Produkte anzeigen, die der Kunde nicht bestellt hat.

Um die beiden Listenfelder zu füllen, bieten wir ein drittes Listenfeld an, das alle Produkte enthält. Aus diesem wählen wir dann die Produkte aus, die den beiden zuvor genannten Listenfeldern hinzugefügt werden sollen.

Um aus diesem Listenfeld komfortabel die gewünschten Produkte auswählen zu können, wollen wir über diesem Listenfeld noch ein Textfeld einfügen, mit dem wir die angezeigten Produkte nach dem eingegebenen Text filtern können.

Unter den Listenfeldern zur Auswahl der zu filternden Produkte fügen wir schließlich noch ein Unterformular ein, das die Kunden anzeigt, welche die den Listenfeldern hinzugefügten Produkte entweder bestellt haben – oder auch nicht.

Unterformular zur Anzeige der Kunden

Als Erstes legen wir das Unterformular **sfmKundenNachProduktenFiltern** an. Dieses soll die Kunden anzeigen, die den im Hauptformular festgelegten Kriterien entsprechen. Beim Öffnen des Formulars jedoch soll es zunächst alle Kunden anzeigen. Deshalb stellen wir die Eigenschaft **Datensatzquelle** auf eine Abfrage ein, welche die Tabelle **tblKunden** als Datenherkunft verwendet und davon die Felder **KundeID**, **AnredeID**, **Nachname**, **Vorname** und **EMail** anzeigt, sortiert nach den Feldern **Nachname** und **Vorname** (siehe Bild 2).

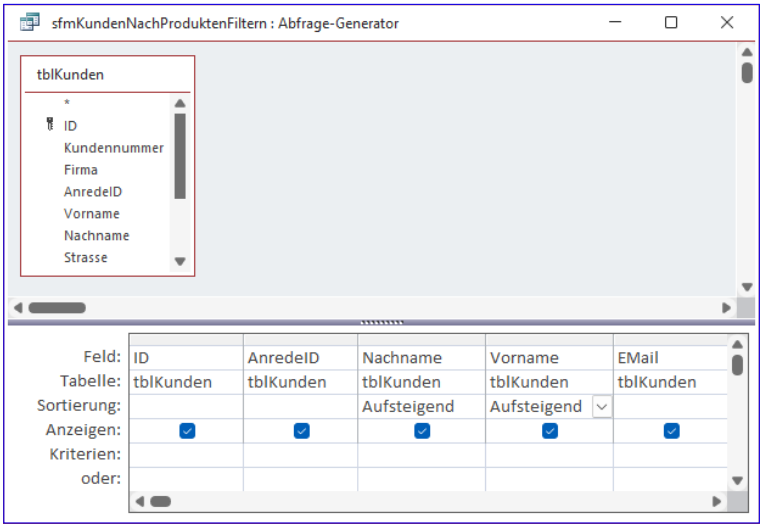


Bild 2: Abfrage für das Unterformular

Anschließend ziehen wir alle Felder dieser Abfrage aus der Feldliste in den Entwurf des Formulars (siehe Bild 3).

Damit dieses Formular seine Daten in der Datenblattansicht anzeigt, stellen wir seine Eigenschaft **Standardansicht** auf den Wert **Datenblatt** ein. Nun schließen und speichern wir das Unterformular.

Hauptformular zum Filtern der Kunden

Dann legen wir ein weiteres neues Formular namens **frmKundenNachProduktenFiltern** an und fügen diesem folgende Steuerelemente hinzu:

- Textfeld **txtProduktfilter**
- Listenfeld **lstAlleProdukte**
- Listenfeld **lstBestellteProdukte**
- Listenfeld **lstNichtBestellteProdukte**

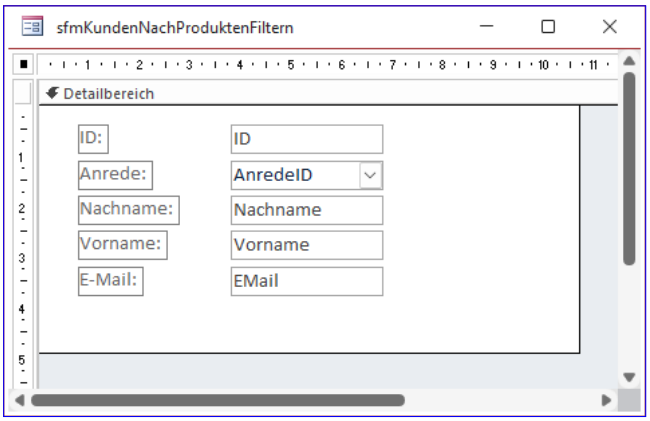


Bild 3: Entwurf des Unterformulars

- Schaltfläche **cmdZuBestelltenProdukten**
- Schaltfläche **cmdZuNichtBestelltenProdukten**

Außerdem ziehen wir aus dem Navigationsbereich das Formular **sfmKundenNachProduktenFiltern** in das Hauptformular. Die Steuerelemente ordnen wir dabei wie in Bild 4 an.

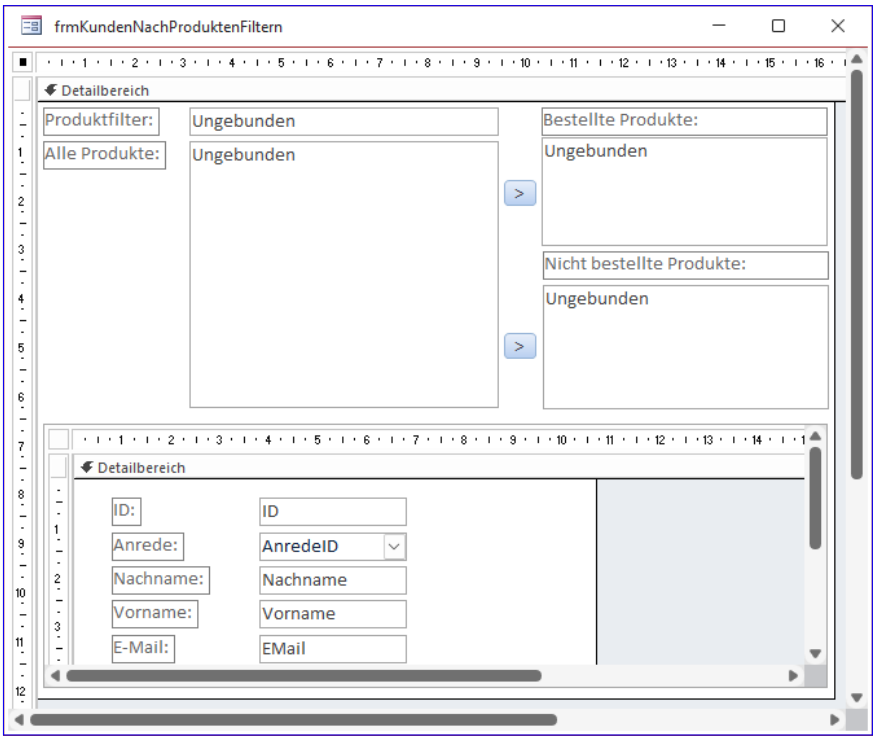


Bild 4: Anordnung der Steuerelemente im Hauptformular

Listenfelder mit Daten füllen

Das Listenfeld **lstAlleProdukte** soll alle Produkte anzeigen mit Ausnahme derer, die bereits zu einem der Listenfelder **lstBestellteProdukte** oder **lstNichtBestellteProdukte** hinzugefügt wurden.

Das Listenfeld **lstBestellteProdukte** soll alle Produkte anzeigen, die nach Markierung im Listenfeld **lstAlleProdukte** mit der Schaltfläche **cmdZuBestelltenProdukten** hinzugefügt wurden.

Das Listenfeld **lstNichtBestellteProdukte** soll analog alle Produkte anzeigen, die nach Markierung im Listenfeld **lstAlleProdukte** mit der Schaltfläche **cmdZuNichtBestelltenProdukten** hinzugefügt wurden.

Nun müssen wir allerdings noch festlegen, wie wir die Produkte markieren, die in einem der beiden Listenfelder **lstBestellteProdukte** oder **lstNichtBestellteProdukte** angezeigt werden.

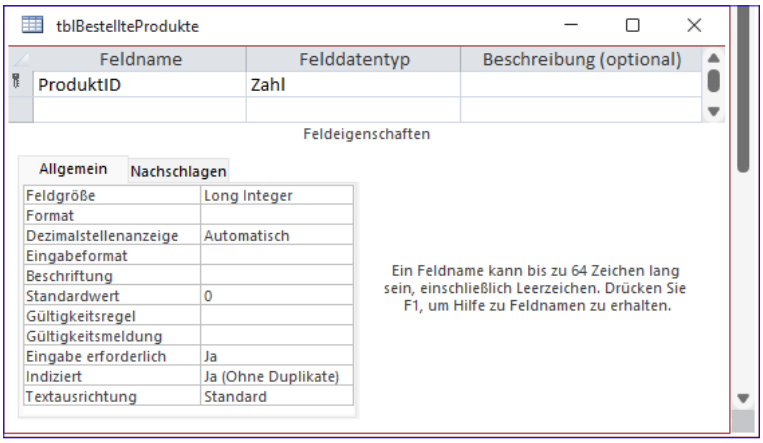


Bild 5: Entwurf der Tabelle **tblBestellteProdukte**

Tabellen zum Speichern der ein- und auszuschließenden Produkte

Wir könnten die beiden Listenfelder einfach mit dem Wert **Wertliste** für die Eigenschaft **Herkunftsart** ausstatten und die Produkte als String-Liste der Eigenschaft **Datensatzherkunft** hinzufügen. Wenn wir die Produkte in diesen beiden Listen allerdings jederzeit nach dem Produktnamen sortiert anzeigen wollen, haben wir mehr Programmieraufwand. Und da wir in Access arbeiten, legen wir schnell zwei Tabellen namens **tblBestellte-Produkte** und **tblNichtBestellteProdukte** an. Die erste sieht in der Entwurfsansicht wie in Bild 5 aus und enthält lediglich ein Zahlenfeld zum Speichern der **ProduktID**-Werte, nach denen die Kunden gefiltert werden sollen. Dieses Feld haben wir als Primärschlüsselfeld definiert, jedoch logischerweise nicht mit dem Datentyp **Autowert**. Die zweite unterscheidet sich nur durch den Namen von der ersten Tabelle. Zeitspar-Tipp: Einfach die erste Tabelle kopieren und unter dem Namen **tblNichtBestellteProdukte** speichern.

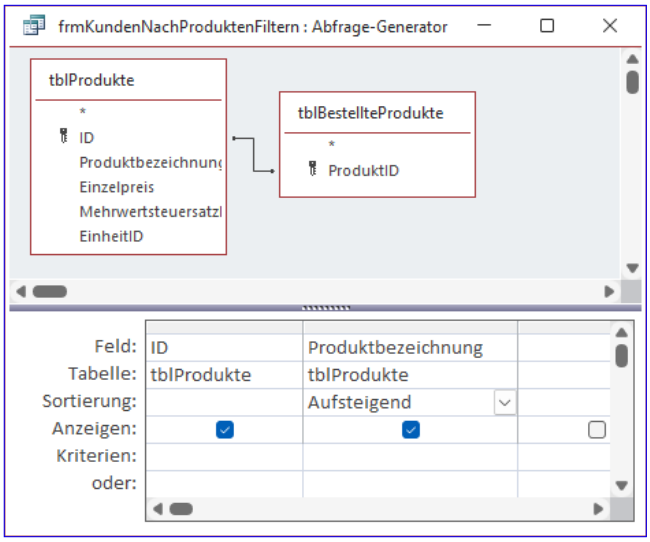


Bild 6: Datensatzherkunft des Listenfeldes **lstBestellteProdukte**

Datensatzherkunft der Listenfelder lstBestellteProdukte und lstNichtBestellteProdukte

Die Datensatzherkunft der beiden rechten Listenfelder können wir nun bereits einstellen. Die für das Listenfeld **lstBestellteProdukte** gestalten wir wie in Bild 6. Sie soll alle Datensätze der Tabelle **tblProdukte** anzeigen, deren Wert im Feld **ProduktID** in der Tabelle **tblBestellteProdukte** gespeichert ist. Nachdem Sie die beiden Tabellen zum Abfrageentwurf hinzugefügt haben, müssen Sie die Beziehung zwischen den Feldern **ID** der Tabelle **tblProdukte** und **ProduktID** der Tabelle **tblBestellteProdukte** manuell hinzufügen. Diese Beziehung wollen wir nicht im Datenbankfenster festlegen.

Die Datensatzherkunft für das Listenfeld **lstNichtBestellteProdukte** erstellen wir analog, diesmal verwenden wir jedoch die Tabelle **tblNichtBestellteProdukte** als zweite Tabelle.

Weitere Einstellungen für die Listenfelder

Für alle drei Listenfelder nehmen wir nun die **Format**-Einstellungen vor. Hier legen wir für die Eigenschaft **Spaltenanzahl** den Wert **2** und für **Spaltenbreiten** den Wert **0cm** fest. So zeigen die Listenfelder immer nur die Produktbezeichnung an, aber nicht den Wert der gebundenen Spalte **ProduktID**.

Daten des Listenfeldes lstAlleProdukte

Der Name dieses Listenfeldes ist eigentlich irreführend, denn es zeigt nur direkt nach dem Öffnen des Formulars alle Produkte an. Sobald der Benutzer über das Textfeld **txtProduktfilter** einen Filter eingegeben hat oder ein Produkt in eines der rechten Listenfelder verschoben hat, zeigt es nicht mehr alle Produkte an.

Es soll dann nur noch die Produkte anzeigen, die nach dem Filtern und Entfernen der bereits in einem der übrigen Listenfeldern enthaltenen Produkte übrig bleiben.

Zuerst aber soll dieses Listenfeld einfach alle Produkte anzeigen, die in der Tabelle **tblProdukte** enthalten sind – und zwar nach dem Alphabet sortiert. Dazu weisen wir der Eigenschaft **Datensatzherkunft** des Listenfeldes die Abfrage aus Bild 7 zu.

Die enthaltenen Daten sollen anschließend nach verschiedenen Aktionen aktualisiert werden. Bei den Aktionen handelt es sich um die folgenden:

- Filtern über das Textfeld **txtProduktfilter** nach dem Auslösen des Ereignisses **Bei Änderung**
- Hinzufügen eines Produkts zu den Listenfeldern **lstBestellteProdukte** und **lstNichtBestellteProdukte mit den Schaltflächen cmdZuBestelltenProdukten** und **cmdZuNichtBestelltenProdukten**
- Entfernen eines Produkts aus den Listenfeldern **lstBestellteProdukte** und **lstNichtBestellteProdukte** per Doppelklick auf einen der Einträge der Listenfelder

Deshalb rufen wir von all den durch die oben beschriebenen Aktionen Prozeduren auf, welche die Datensatzherkunft des Listenfeldes **lstAlleProdukte** aktualisiert (und später auch die Liste der Kunden).

Diese Prozeduren sollen **ProdukteAktualisieren** und **KundenAktualisieren** heißen.

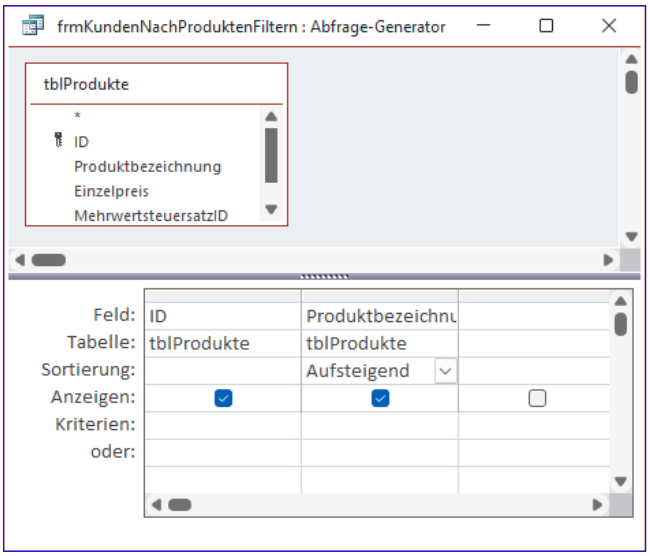


Bild 7: Datensatzherkunft des Listenfeldes **lstAlleProdukte**

Prozeduren zum Aktualisieren von Produkten und Kunden

Die Prozeduren **ProdukteAktualisieren** und **KundenAktualisieren** sollen den im Textfeld **txtProduktfilter** eingegebenen Suchbegriff berücksichtigen und zusätzlich über die in die beiden Tabellen **tblBestellteProdukte** und **tblNichtBestellteProdukte** gespeicherten Einträge die noch verfügbaren Produkte im Listenfeld **lstAlleProdukte** anzeigen. Schließlich soll noch die Anzeige der Kunden passend zu den in den beiden Tabellen gespeicherten Daten aktualisiert werden.

Da diese Prozeduren sowohl durch das Ereignis **Bei Änderung** des Textfeldes **txtProduktfilter** ausgelöst werden, kann als auch durch das Hinzufügen oder Entfernen von Einträgen zu den Tabellen **tblBestellteProdukte** und **tblNichtBestellteProdukte**, haben wir ein kleines Problem: Beim Aufruf über das Textfeld können wir auf den aktuellen Inhalt des Textfeldes nur über die Eigenschaft **Text** des Textfeldes zugreifen. Beim Aufruf über eines der anderen Ereignisse müssen wir die **Value**-Eigenschaft des Textfeldes verwenden.

Wir können das auf verschiedene Arten lösen. Eine ist, den Wert des Textfeldes jeweils in einer Variablen zu

speichern, auf die wir dann von der Prozedur **ProdukteAktualisieren** aus zugreifen. Die zweite ist, dass wir den Inhalt der Variablen **Text** beim Aufruf vom Textfeld aus als Parameter übergeben und beim Aufruf von den Schaltflächen beziehungsweise Listenfeldern den Wert über die Eigenschaft **Value** ermitteln und dann ebenfalls per Parameter übergeben.

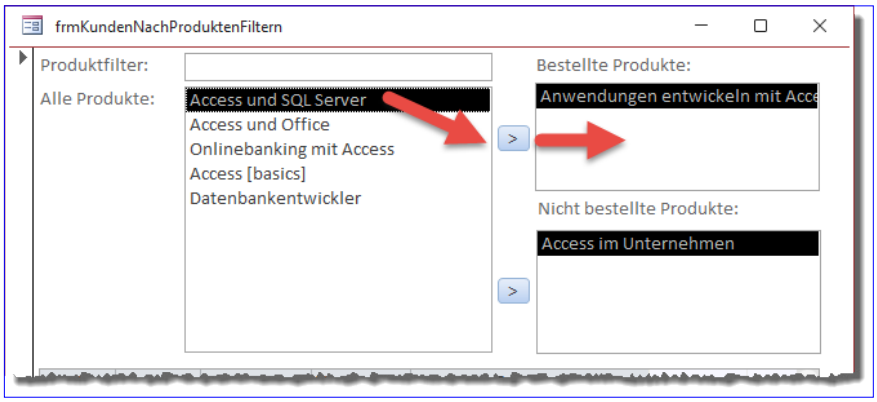


Bild 8: Hinzufügen eines Produkts zu den bestellten Produkten

Da Variablen unter Umständen ihren Wert verlieren können, nutzen wir die letztere Variante.

Nach Produktname filtern

Die Eingabe beziehungsweise Änderung des Inhalts des Textfeldes **txtProduktfilter** soll die im Listenfeld **lstAlleProdukte** enthaltenen Datensätze filtern. Dazu hinterlegen wir für die Ereigniseigenschaft **Bei Änderung** die folgende Prozedur:

```
Private Sub txtProduktfilter_Change()
    ProdukteAktualisieren Me!txtProduktfilter.Text
End Sub
```

Diese ruft lediglich eine weitere Prozedur namens **ProdukteAktualisieren** auf und übergibt dieser den aktuellen Inhalt des Textfeldes. Diese Prozedur beschreiben wir weiter unten. Erst schauen wir uns weitere Aufrufe dieser Prozedur an.

Hinzufügen von Produkten als einschließende Kriterien für den Kundenfilter

Wenn der Benutzer wie in Bild 8 einen der Einträge des Listenfeldes **lstAlleProdukte** markiert und dann auf die Schaltfläche **cmdZuBestelltenProdukten** klickt, löst dies das Ereignis **Beim Klicken** der Schaltfläche aus.

Die Prozedur gestalten wir wie in Listing 1. Sie prüft zunächst, ob überhaupt ein Eintrag im Listenfeld **lstAlleProdukte** markiert ist. Falls ja, erstellt sie eine Referenz auf das aktuelle **Database**-Objekt und führt damit eine **INSERT INTO**-Aktionsabfrage aus.

Diese soll den Primärschlüsselwert des aktuell im Listenfeld markierten Eintrags als neuen Datensatz in die Tabelle **tblBestellteProdukte** einfügen. Anschließend erfolgt hier ebenfalls ein Aufruf der bereits weiter oben erwähnten Prozedur **ProdukteAktualisieren**, diesmal mit dem über

```
Private Sub cmdZuBestelltenProdukten_Click()
    Dim db As DAO.Database
    If Not IsNull(Me!lstAlleProdukte) Then
        Set db = CurrentDb
        db.Execute "INSERT INTO tblBestellteProdukte(ProduktID) VALUES(" & Me!lstAlleProdukte & ")", dbFailOnError
        ProdukteAktualisieren Nz(Me!txtProduktfilter)
        KundenAktualisieren
    End If
End Sub
```

Listing 1: Produkt zum Listenfeld der als Kriterien einzuschließenden Produkte hinzufügen

Prüfen, ob Datenbank geöffnet ist

In einem früheren Beitrag haben wir mit einer Funktion geprüft, ob eine Datenbank geöffnet ist. Diese war jedoch nicht in jedem Fall zuverlässig – also liefern wir eine neue Version für eine solche Funktion. In dieser neuen Funktion versuchen wir, die Datenbank exklusiv zu öffnen. Das gelingt nur, wenn diese aktuell nicht geöffnet ist. Mehr dazu im vorliegenden Beitrag!

In der Funktion namens **IsDatabaseOpen**, die wir im Beitrag **Kunde zu einer E-Mail öffnen** (**www.access-im-unternehmen.de/1291**) vorgestellt haben, prüfen wir nur, ob sich im gleichen Verzeichnis der Datenbank auch eine **.laccdb**-Datei befindet. Diese wird normalerweise erstellt, wenn die Datenbank geöffnet ist.

Carsten Gromberg hat uns darauf hingewiesen, dass diese Funktion nicht immer das korrekte Ergebnis liefert und eine verbesserte Version der Funktion beigesteuert.

Dabei verwenden wir einen zuverlässigeren Weg, um zu prüfen, ob eine Datenbank geöffnet ist. Dabei versuchen wir, die Datenbank exklusiv zu öffnen. Das ist nur möglich, wenn die Datenbank bisher gar nicht geöffnet ist.

Die neue Version der Funktion **IsDatabaseOpen** finden Sie in Listing 1. Die Funktion erwartet den Pfad zu der zu untersuchenden Datenbankdatei sowie einen **Boolean**-Parameter, der angibt, ob bei bereits geöffneter Datenbank eine Meldung ausgegeben werden soll.

Die Funktion erstellt ein Objekt auf Basis der verborgenen Klasse **PrivDBEngine**, die im Gegensatz zu **DBEngine** eine neue Session außerhalb der Session der aktuellen Datenbank öffnet und somit unabhängig ist. Warum das wichtig ist, schauen wir uns weiter unten an.

Nach dem Erstellen von **objEngine** deaktiviert die Funktion zunächst die eingebaute Fehlerbehandlung. Somit kann sie die nachfolgende **OpenDatabase**-Methode des

```
Public Function IsDatabaseOpen(ByVal strDBName As String, _
                               Optional ByVal bolShowMessage As Boolean) As Boolean
    Dim objEngine   As DAO.PrivDBEngine
    Set objEngine = New DAO.PrivDBEngine
    On Error Resume Next
    objEngine.OpenDatabase strDBName, True, True
    If Not Err.Number = 0 Then
        IsDatabaseOpen = True
    End If
    If IsDatabaseOpen And bolShowMessage Then
        MsgBox strDBName & vbCrLf _
            & "kann nicht exklusiv geöffnet werden!" & vbCrLf & vbCrLf _
            & "Fehler: " & CStr(Err.Number) & vbCrLf _
            & Err.Description, vbExclamation
    End If
    Set objEngine = Nothing
End Function
```

Listing 1: Funktion zum Prüfen, ob eine Datenbank bereits geöffnet ist

Dateien per VBA öffnen

Es gibt viele Gelegenheiten, zu denen man gern eine Datei per VBA öffnen möchte. Ein gutes Beispiel ist ein soeben auf Basis eines Berichts erstelltes PDF-Dokument. Doch der Standardumfang von VBA liefert keine Möglichkeit, diese Aufgabe zu erledigen. Und tatsächlich ist das Anzeigen einer Datei nicht trivial, zumindest dann nicht, wenn wir vorher noch nicht wissen, welchen Dateityp die Datei hat und mit welcher Anwendung diese geöffnet werden soll. Allerdings weiß Windows ja auch meistens, mit welcher Anwendung eine Datei geöffnet werden soll, wenn wir diese im Windows Explorer doppelt anklicken. Also muss es einen Weg geben, diese Aufgabe per Code zu erledigen. Und die Lösung ist eine API-Funktion namens ShellExecute.

Beispieltabelle mit Dateiinformationen

Zu Beispielzwecken haben wir einer Datenbank eine Tabelle namens **tblDateien** hinzugefügt. Diese sieht wie in Bild 1 aus und enthält neben einer Bezeichnung den Namen von Dateien. Auf Pfadangaben haben wir aufgrund der besseren Nutzbarkeit der Beispieldatenbank verzichtet und gehen davon aus, dass die angegebenen Dateien sich im gleichen Verzeichnis wie die Datenbank befinden. Auf diese Weise können wir in den folgenden Beispielen per Code unter Zuhilfenahme von **CurrentProject.Path** auf die Dateien zugreifen.

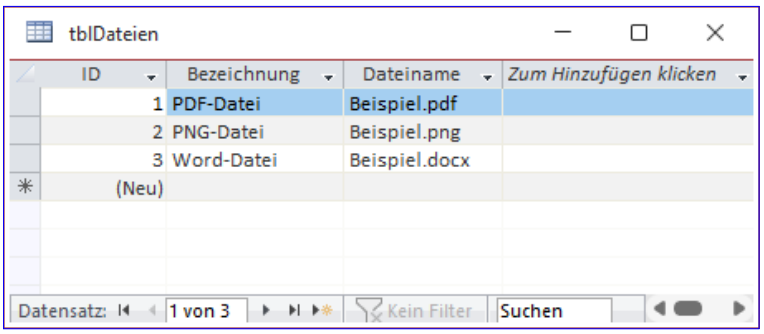


Bild 1: Tabelle mit Dateiangaben

Formular zum Anzeigen der Beispieldateien

Um die Funktion zum Anzeigen der Dateien einfach aufrufen zu können, erstellen wir ein Formular, das die Daten der Tabelle **tblDateien** anzeigt und eine Schaltfläche bereitstellt, mit der wir die Datei anzeigen können (siehe Bild 2). Diese Schaltfläche heißt **cmdDateiOeffnen**. Gleich fügen wir der Schaltfläche den Code zum Öffnen der Datei aus dem Feld **Dateiname** hinzu.

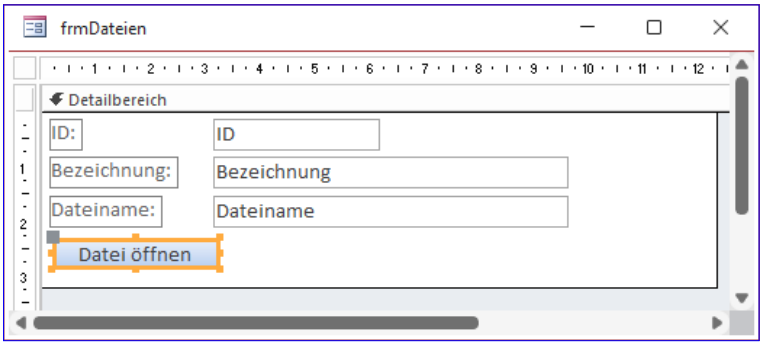


Bild 2: Entwurf eines Formulars zur Anzeige von Dateien in der jeweiligen Anwendung

API-Funktion zum Anzeigen von Dateien

Für das Anzeigen von Dateien benötigen wir die API-Funktion **ShellExecuteA**. Diese deklarieren wir wie in Listing 1 in zweifacher Ausführung, einmal für Access-Versionen, die ein älteres VBA verwenden, und einmal für die aktuellen Fassungen. Die Deklaration platzieren wir oben in einem neuen Standardmodul namens **mdlDateienAnzeigen**. Der Unterschied besteht in der Verwendung des Schlüsselworts **PtrSafe** sowie dem Datentyp **LongPtr** für den Parameter **hWnd** bei der Version für aktuelles VBA.

```
#If VBA7 Then
    Public Declare PtrSafe Function ShellExecuteA Lib "Shell32" (ByVal hWnd As LongPtr, ByVal lpOperation As String, _
      ByVal lpFile As String, ByVal lpParameters As String, ByVal lpDirectory As String, ByVal nCmdShow As Long) As Long
#Else
    Public Declare Function ShellExecuteA Lib "Shell32" (ByVal hWnd As Long, ByVal lpOperation As String, ByVal _
       lpFile As String, ByVal lpParameters As String, ByVal lpDirectory As String, ByVal nCmdShow As Long) As Long
#End If
```

Listing 1: Deklaration der API-Funktion ShellExecute für 32-Bit und 64-Bit-VBA

Die Funktion erwartet einige Parameter, die wir uns hier anschauen:

- **hWnd**: Erwartet ein Handle auf das aufrufende Fenster. Hier können wir den Wert **0** übergeben.
- **lpOperation**: Erwartet eine Zeichenkette mit der auszuführenden Operation. Für unsere Aufgabe, eine Datei anzuzeigen, verwenden wir hier den Wert **Open**.
- **lpFile**: Nimmt den Pfad zu der anzuzeigenden Datei entgegen.
- **lpParameters**: Parameter, den wir hier nicht benötigen und deshalb mit **vbNullString** füllen
- **lpDirectory**: Parameter, den wir hier nicht benötigen und deshalb mit **vbNullString** füllen
- **nCmdShow**: Erwartet eine Konstante für den Anzeigemodus.

Die möglichen Werte für **nCmdShow** lauten:

- **SW_HIDE** (**0**): Öffnet die Datei im versteckten Modus.
- **SW_MAXIMIZE** (**3**): Öffnet das Fenster mit der Datei im maximierten Zustand.
- **SW_MINIMIZE** (**6**): Zeigt das Fenster minimiert an.
- **SW_NORMAL** (**1**): Aktiviert das Fenster beim Öffnen.
- **SW_SHOW** (**5**): Einfache Anzeige.
- **SW_RESTORE** (**9**): Stellt die Fenstergröße wieder her.
- **SW_SHOWMAXIMIZED** (**3**): Zeigt das Fenster an und maximiert es.
- **SW_SHOWMINIMIZED** (**2**): Zeigt das Fenster an und minimiert es.
- **SW_SHOWMINNOACTIVE** (**7**): Minimiert das Fenster und aktiviert es nicht
- **SW_SHOWNA** (**8**): Zeigt das Fenster an, aber aktiviert es nicht.
- **SW_SHOWNOACTIVATE** (**4**): Zeigt das Fenster an, ohne es zu aktivieren.
- **SW_SHOWNORMAL** (**1**): Zeigt das Fenster und aktiviert es.

Damit wir die Konstanten statt der Zahlenwerte angeben können, deklarieren wir diese oberhalb der API-Deklaration im gleichen Modul:

```
Public Const SW_HIDE = 0
Public Const SW_MAXIMIZE = 3
Public Const SW_MINIMIZE = 6
Public Const SW_NORMAL = 1
Public Const SW_SHOW = 5
Public Const SW_RESTORE = 9
Public Const SW_SHOWMAXIMIZED = 3
```

Produktivität mit Notion steigern

Gelegentlich gönnen wir von Access im Unternehmen uns einen Ausflug zu einer anderen Anwendung. In diesem Fall geht es um Notion, einer modernen Produktivitätsapp. Eigentlich ist gar keine große Rechtfertigung notwendig, denn die meisten Leser dieses Magazins arbeiten vermutlich produktiv mit Access und sind auf eine entsprechend strukturierte Arbeitsweise angewiesen. Diese unterstützen Tools wie Notion, denn sie erlauben eine Ablage aller möglichen Informationen in strukturierter Form. Außerdem können Sie damit beispielsweise Aufgaben auflisten und abarbeiten und beliebige andere Daten damit verwalten – in entsprechend kostenpflichtigen Versionen sogar im Team. Und es gibt noch einen wesentlichen Grund, darüber in diesem Magazin zu berichten: Wir können nämlich über die API auf die in Notion abgelegten Daten zugreifen und diese auch von einer Access-Datenbank aus befüllen. Doch dies soll Thema eines anderen Beitrags sein – hier schauen wir uns erst einmal die grundlegenden Funktionen von Notion an.

Notion ist eine Anwendung, die Sie sowohl im Webbrowser als auch über eine eigene App für die verschiedenen mobilen Endgeräte und Betriebssysteme bedienen können. Das macht es interessant, denn während wir zwar nur am Windows-Rechner an unseren Datenbanken programmieren können, so kommen zumindest mir doch immer wieder Ideen zu meinen Projekten, während ich gerade nicht an meinem Arbeitsplatz sitze. Dann kann ich mir jedoch schnell eine entsprechende Notiz in meiner Notion-App auf dem Smartphone machen, die ich dann später am Rechner wieder vorfinden und umsetzen kann.

Start mit Notion

Der Einstieg gelingt sehr schnell über die Webseite **www.notion.so** (siehe Bild 1). Hier findet sich der Button mit der Aufschrift **Try Notion free**, der einen zum Sign up-Bildschirm leitet.

Hier kann man sich aussuchen, ob man mit E-Mail oder einem bereits vorhandenen Konto von Google oder Apple einsteigen möchte.

Nach dem Anlegen eines neuen Accounts gibt man noch ein paar weitere Informationen wie den Namen und das Kennwort an. Außerdem gibt es noch ein paar Fragen zum Einsatzzweck von Notion. Anschließend legt man fest, ob man Notion mit einem Team nutzen möchte oder nur

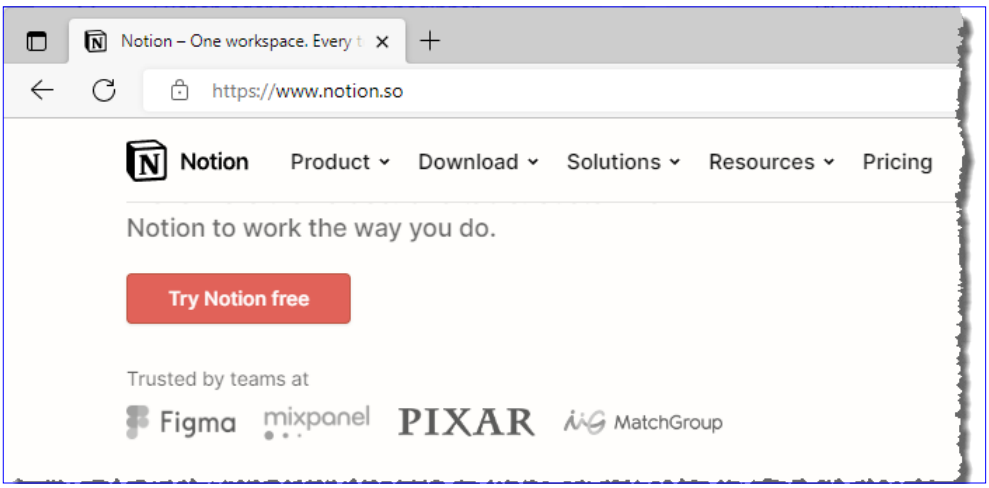


Bild 1: Einstieg in Notion

für sich allein. Wir wählen letztere Variante, weil nur diese ein kostenloses Konto bietet.

Gleich danach landen wir auf der Startseite, wo Notion ein paar Templates zur Orientierung für uns angelegt hat. Wer gern von Null beginnt, kann diese auch ablehnen (siehe Bild 2).

Die **Getting Started**-Seite bietet beispielsweise gleich einmal eine ToDo-Liste an und die Einträge in der Übersicht links bieten die Möglichkeit, andere Formate für Inhalte in Notion anzusehen.

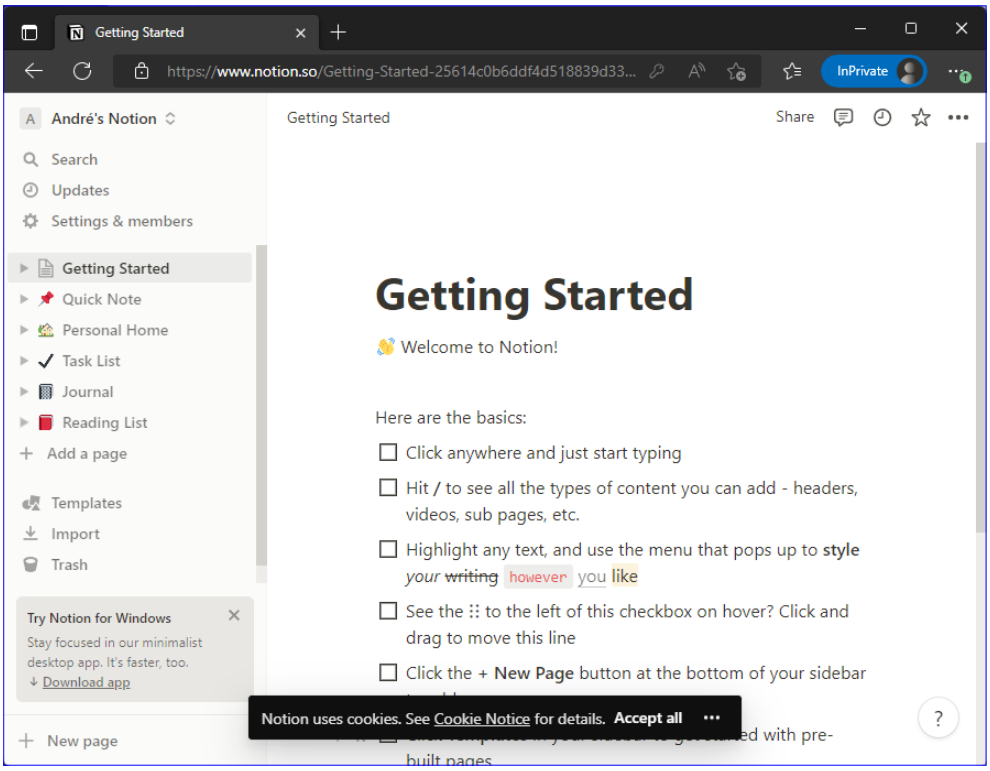


Bild 2: Die Getting-Started-Seite von Notion

Neue Seite anlegen

Wer gleich mit einer neuen, leeren Seite starten möchte, nutzt dazu den Eintrag **Add a page** im linken Menü. Auf der neuen, mit dem Titel **Untitled** versehenen Seite stehen dem User alle Möglichkeiten offen (siehe Bild 3). Als Erstes geben wir hier einen Titel ein, beispielsweise **ToDo-Liste**.

Wer Spaß an optischen Elementen hat, wird begeistert sein: Wir haben die Möglichkeit, ein Icon sowie ein Hintergrundbild als

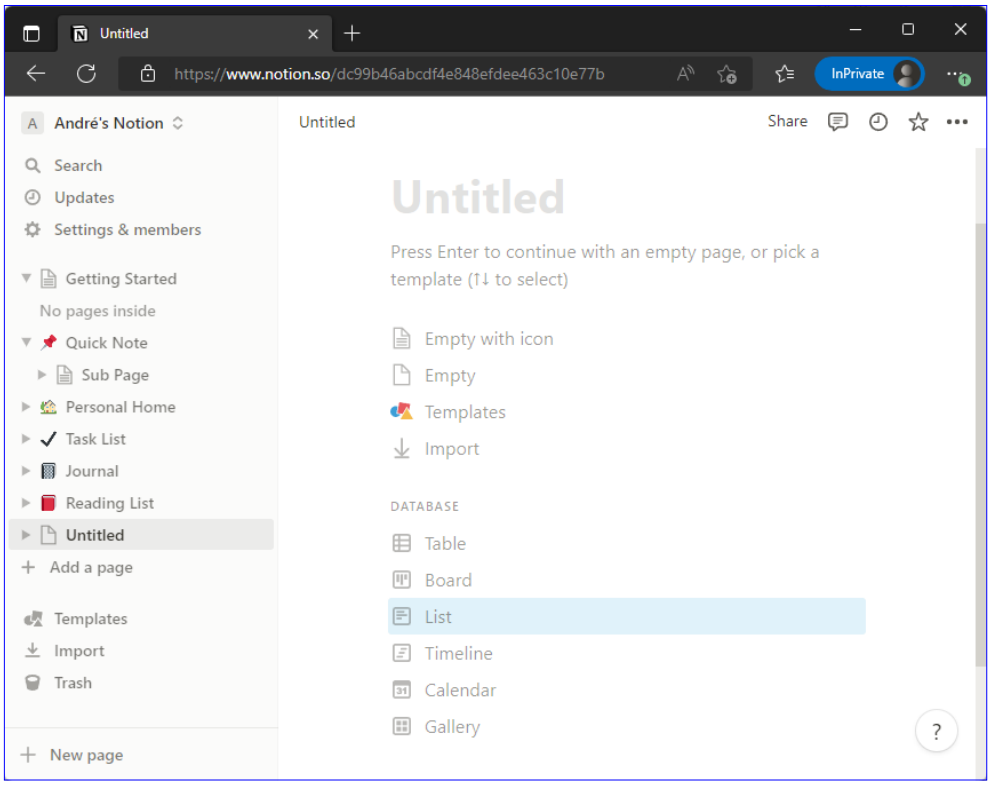


Bild 3: Eine neue, leere Seite

Seitenheader festzulegen. Die dazu notwendigen Befehle tauchen auf, wenn man wie in Bild 4 mit der Maus über den Titel fährt.

Ein paar Mausklicks später haben wir unsere ToDo-Seite mit den von Notion bereitgestellten Elementen optisch aufgewertet (siehe Bild 5).

Elemente zur Page hinzufügen

Um nun tatsächlich Inhalte hinzuzufügen, brauchen wir einfach nur etwas dort einzugeben, wo jetzt noch **Type '/' for commands** steht. Hier können wir nun einfach einen Text eingeben, oder wir fügen eines der vorgefertigten Elemente hinzu. Das gelingt am schnellsten durch die Eingabe des Schrägstrichs (*/*).

Dies zeigt wie in Bild 6 alle möglichen Formate und Elemente an, die wir an dieser Stelle einfügen können. Von normalem Text über ToDo-Listen und verschiedene Überschriftenebenen bis hin zu komplexeren Elementen wie Tabellen, verschiedene Listen, Zitate, Trennstriche, Links, Callouts, Formeln et cetera.

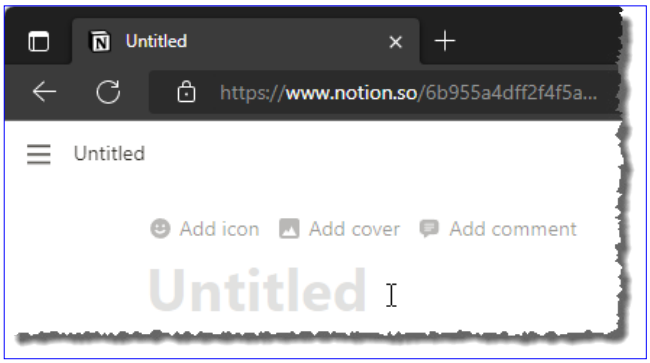


Bild 4: Hinzufügen von Icon, Cover und Comment

Außerdem gibt es noch sogenannte **Databases**. Hier war ich als Access-Entwickler zuerst einigermaßen verwirrt. Es ist aber schnell klar geworden, dass es sich nicht um Datenbanken, sondern um das, was wir unter Access unter einer Tabelle verstehen, handelt.

Vielleicht nennt Notion diese Tabellen **Database** und nicht **Table**, um eine Abgrenzung zu den Tabellen unter Excel zu erreichen. Der Unterschied besteht darin, dass eine Notion-Database ein oder mehrere Spalten enthält, für die feste Datentypen festgelegt werden müssen.

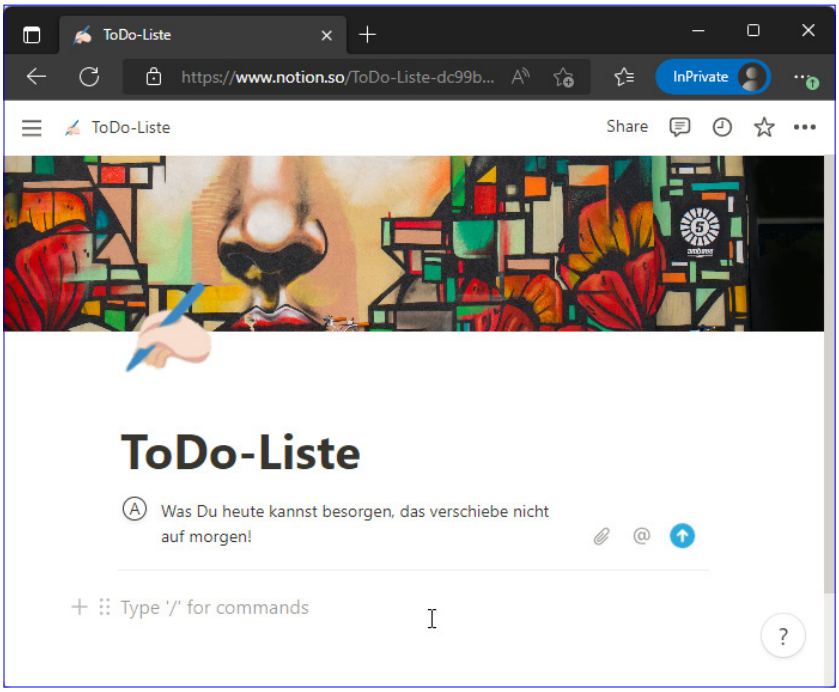


Bild 5: Eine optisch ansprechende ToDo-Liste – allerdings noch ohne ToDos.

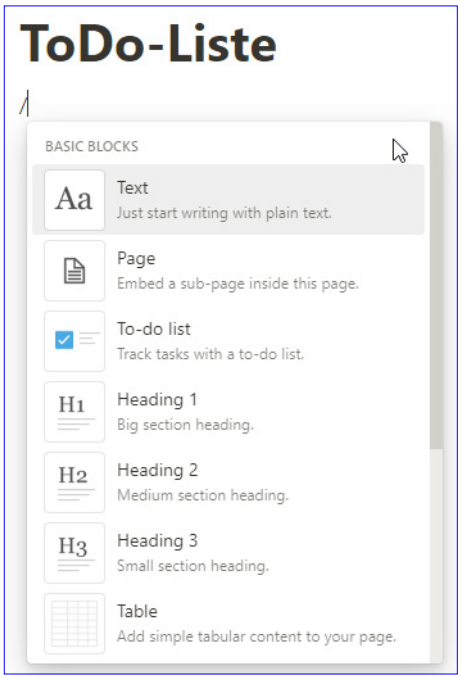


Bild 6: Auswahl der verfügbaren Elemente

Warum sollte man mit einer Notion-Database arbeiten und nicht einfach eine Access-Tabelle verwenden? Während eine Access-Tabelle nur eine Ansicht mit Daten bietet, nämlich die Datenblatt-Ansicht, liefern Notion-Databases gleich eine ganze Reihe von Ansichten, die alle je nach Anwendungszweck sehr praktisch sein können. Dazu kommen wir jedoch später.

Dazu kommen dann noch Bilder, Videos, Audios oder Dateien und man kann auch Code als Code formatiert unterbringen. Die Möglichkeiten sind wirklich umfassend und es ergibt Sinn, hier einfach einmal ein wenig herumzuspielen oder auch nur je nach der aktuellen Anforderung die entsprechenden Elemente zu nutzen.

Das Anlegen einfacher Elemente wie Überschriften, Text, Auflistungen et cetera ist relativ einfach, daher gehen wir direkt mal zum Anlegen und Verlinken von Seiten.

ToDo-Liste anlegen

Wir haben zwar nun eine Seite namens **ToDo-Liste** angelegt, aber darin noch keine ToDo-Liste erstellt. Dazu geben wir nun das Schrägstrich-Zeichen gefolgt von den ersten Buchstaben des Wortes **ToDo** ein, was den Eintrag **To-do list** aktiviert (siehe Bild 7).

Das anschließende Betätigen der Eingabetaste sorgt dafür, dass die ToDo-Liste mit einem ersten Element angelegt wird. Danach können wir schon starten, Einträge hinzuzufügen. Mit der Eingabetaste legen wir den nächsten Eintrag an. Die Tabulator-Taste ordnet den aktuellen Eintrag dem vorherigen unter, mit **Umschalt + Tab** holen wir den Eintrag wieder auf die übergeordnete Ebene (siehe Bild 8).

Weitere Elemente auf der gleichen Seite

Notion legt uns nicht auf ein Element pro Seite fest. Wir können beliebig viele verschiedene Elemente auf einer Seite ablegen. Vor oder hinter der ToDo-Liste passen also Texte, Überschriften und alle anderen verfügbaren Elemente, und natürlich können wir diese auch ohne ToDo-Liste anlegen.

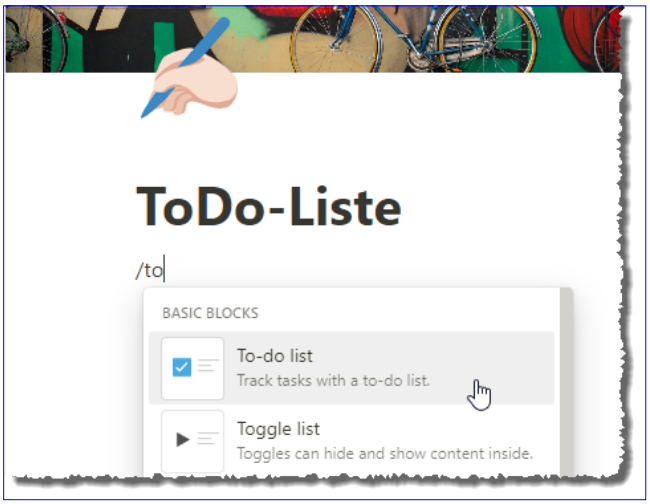


Bild 7: Hinzufügen der eigentlichen ToDo-Liste

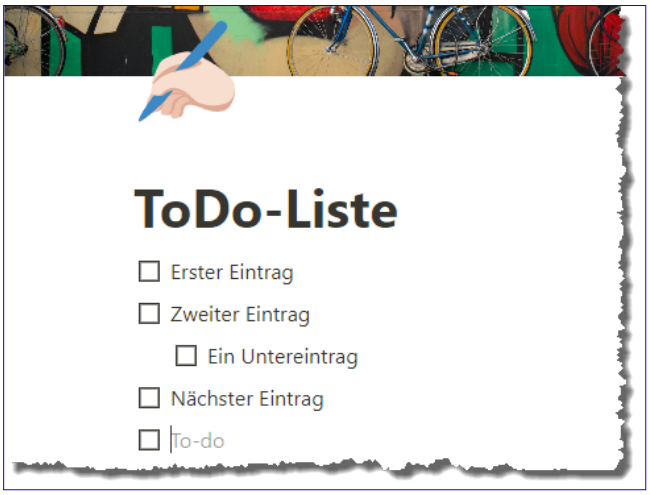


Bild 8: Einträge einer ToDo-Liste

Seiten in anderen Seiten verlinken

Diese ToDo-Liste könnte vielleicht der Mittelpunkt Ihres Notion-Workspaces sein, aber vermutlich wird es weitere Seiten geben. Auch wenn die ToDo-Liste nun in der Navigation im linken Bereich ansteuerbar ist, so möchten Sie vielleicht doch eine Übersichtsseite erstellen, die zu Beginn erscheint und Links zu den wichtigsten Seiten Ihres Workspace in Notion enthält.

Und, was noch wichtiger ist: Mit der nachfolgend vorgestellten Technik können Sie nicht nur in einer Übersichtsseite, sondern von überall auf andere Seiten verweisen. Dazu verwendet man nach dem Eingeben des Schräg-

strichs den Eintrag **Link to page**. Dies öffnet eine Liste aller bereits vorhandenen Seiten, aus denen wir die gewünschte auswählen (siehe Bild 9).

Der neue Link zu einer anderen Seite erscheint wie in Bild 10.

Klicken wir nun auf diesen Link, wechseln wir direkt zu der verlinkten Seite. Diese enthält im oberen Bereich nun die Angabe **1 backlink**, was bedeutet, dass es eine Seite gibt, die auf diese Seite per Link verweist. Klicken wir diesen Eintrag an, erscheint eine Liste der referenzierenden Seiten, auf die wir über diesen Weg zugreifen können (siehe Bild 11).

Mit Notion-Databases arbeiten

Richtig spannend wird es, wenn wir mit den sogenannten Databases arbeiten, was eigentlich Tabellen sind. Es gibt auch das **Table**-Objekt in Notion. Das ist aber nur eine einfache Tabelle ohne jegliche Funktion – es ist also kein Filtern oder Sortieren darin möglich, sondern nur die Darstellung von Werten in Tabellenform.

Um eine neue Database hinzuzufügen, erstellen wir eine neue Seite namens **Artikelübersicht**. Die Database können wir gar nicht explizit erzeugen, sondern wir legen eine der verfügbaren Views für eine Database an. Dazu nutzen wir wieder den Schrägstrich plus die Auswahl eines der Einträge unterhalb von **Database**, in diesem Fall **Table View** (siehe Bild 12).

Als Erstes legen wir die Datenquelle fest. Dazu können wir eine der Beispieldatenquellen verwenden, die mit dem Workspace automatisch angelegt wurden, oder wir klicken auf **New Database**, um eine neue Database hinzuzufügen (siehe Bild 13).

Dies legt eine neue, leere Tabelle an und öffnet diese in der gewählten Ansicht, in diesem Fall **Table** (siehe Bild 14). Statt **Untitled** geben wir hier als Erstes einen Database-Namen ein, zum Beispiel **Artikeldatenbank**.

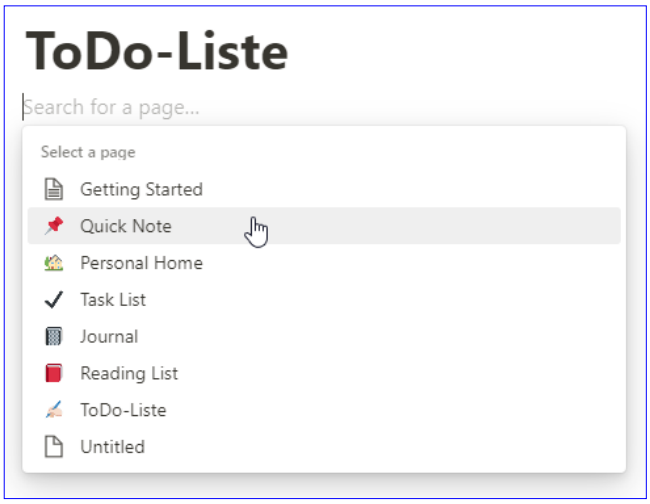


Bild 9: Hinzufügen eines Links zu einer anderen Seite

Bild 10: Link zu einer anderen Seite

Bild 11: Backlink zur verlinkenden Seite

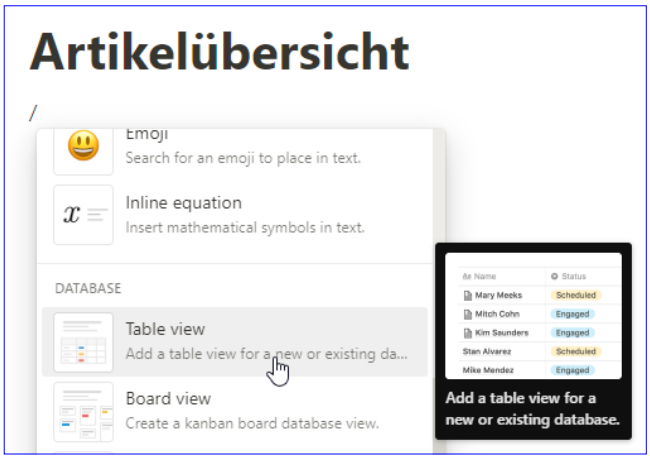


Bild 12: Hinzufügen einer Database in der **Table View**

Mit Access auf Notion zugreifen

Notion ist die Produktivitätsapp der Stunde, wenn es um Verwaltung von Listen, Projekten, Teams und vieles mehr geht. Eigentlich sind die Möglichkeiten nur durch die Phantasie begrenzt. Logisch, dass wir uns ansehen wollen, ob man die Daten, die man in Notion angelegt hat, auch von Access aus einlesen kann oder ob man sogar Daten von Access aus nach Notion verschieben kann.

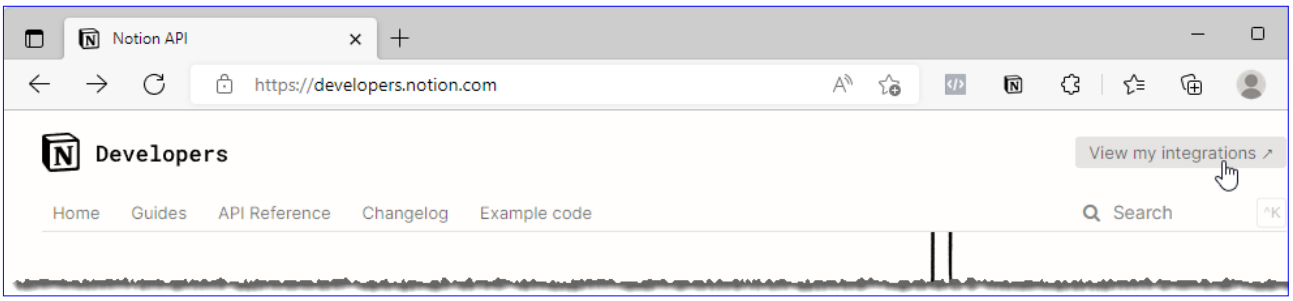


Bild 1: Von der Startseite des Entwicklerbereichs geht es gleich weiter zu den Integrationen.

In einem weiteren Beitrag namens **Produktivität mit Notion steigern** (**www.access-im-unternehmen.de/1402**) zeigen wir die Grundlagen der App **Notion**. Im vorliegenden Beitrag schauen wir uns an, wie wir die dort angelegten Daten von einer Access-Anwendung aus auslesen und sogar schreiben können. Damit erleichtern wir uns einige Arbeiten, zum Beispiel das Anlegen größerer Mengen von Terminen, das Eintragen von Daten aus den Tabellen einer Datenbank et cetera.

Wir gehen davon aus, dass Sie einen Notion-Account angelegt haben. Mit diesem ist auch die Möglichkeit verbunden, sogenannte **Integrations** anzulegen. Dazu wechseln wir zur folgenden Webseite:

```
https://developers.notion.com
```

Dort finden wir oben links einen Link mit dem Text **View my integrations** (siehe Bild 1).

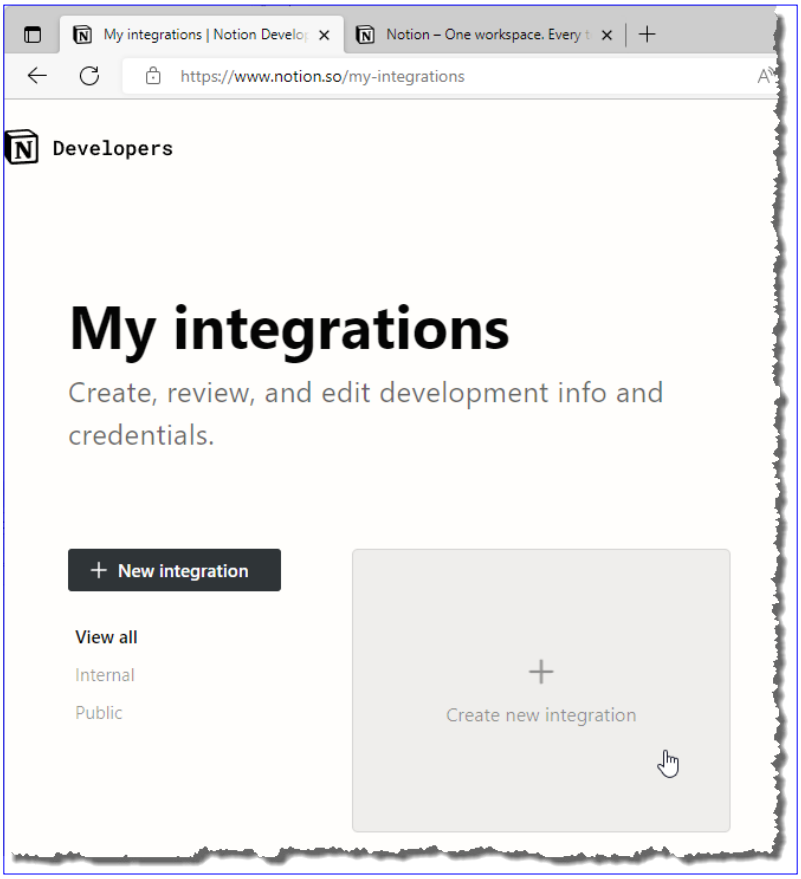


Bild 2: Anlegen einer neuen Integration

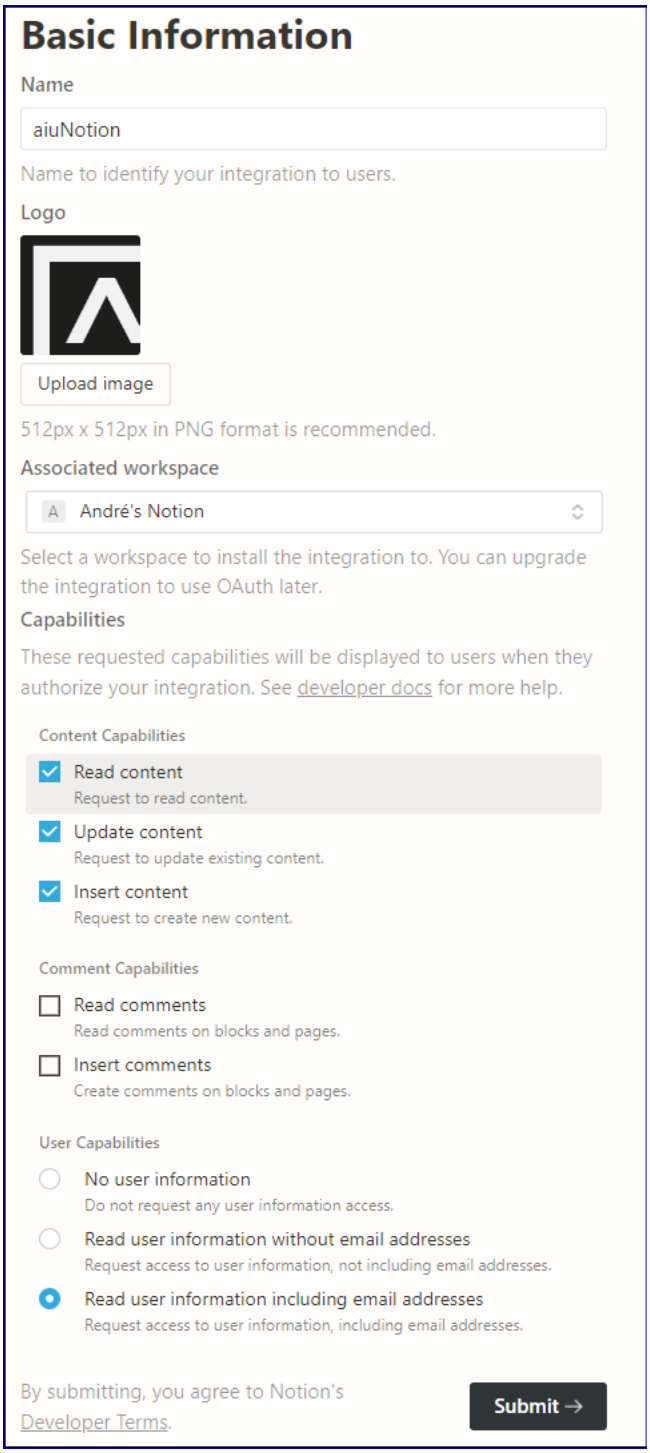


Bild 3: Festlegen der Eigenschaften der Integration

Klicken wir diesen an, landen wir direkt auf der Seite **My integrations**, wo wir mit einem Klick auf die Schaltfläche **Create new integration** eine neue Integration anlegen können (siehe Bild 2).

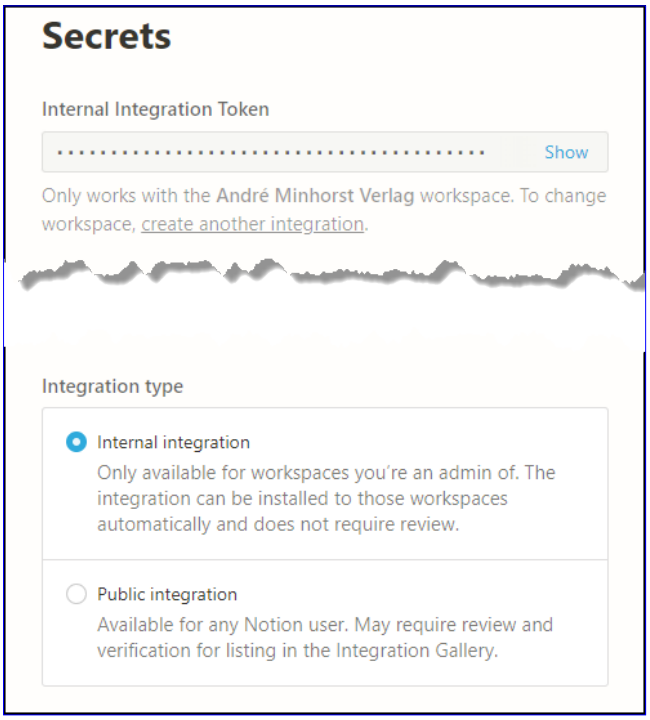


Bild 4: Wichtige Information nach dem Anlegen der Integration

Neue Notion-Integration anlegen

Auf der folgenden Seite fragt Notion einige Informationen über die anzulegende Integration ab. Dazu gehören der Name, ein Logo, ein assoziierter Workspace und die Rechte, die der Integration erteilt werden sollen.

Wir legen die Daten wie in Bild 3 fest. Wenn Sie gerade einen neuen Notion-Account angelegt haben, ist die Zusammenstellung der Daten recht einfach – das Feld **Associated Workspace** bietet dann nur einen einzigen Eintrag an.

Nach dem Klick auf die Schaltfläche **Submit** erscheint eine weitere Seite, auf der wir eine wichtige Information vorfinden: den **Internal Integration Token**. Das ist unsere Eintrittskarte für den VBA-gesteuerten Zugriff auf unsere Notion-Daten (siehe Bild 4). Diesen Token können wir direkt einmal in die Zwischenablage kopieren, wir werden

diesen gleich benötigen. Auf dieser Seite behalten wir die Option **Internal integration** bei, da wir davon ausgehen, dass wir erst einmal nur eine Integration für den Eigengebrauch anlegen wollen. Außerdem fasst die Seite nochmals die zuvor festgelegten Einstellungen zusammen.

Damit sind die Arbeiten auf der Seite von Notion bereits abgeschlossen – wir benötigen hier nur den Token, um diesen als Schlüssel für den Zugriff auf die Daten in unserem Notion-Workspace zu verwenden.

Zur Sicherheit schauen wir noch unter **My integrations** nach und finden dort die neu angelegte Integration vor (siehe Bild 5).

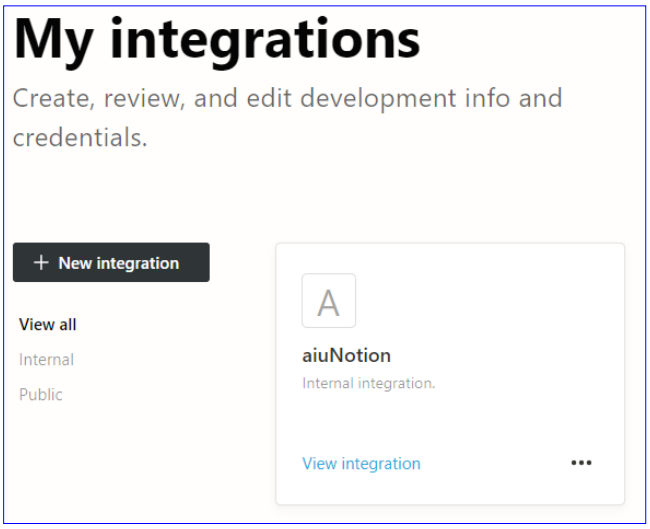


Bild 5: Die neu angelegte Integration

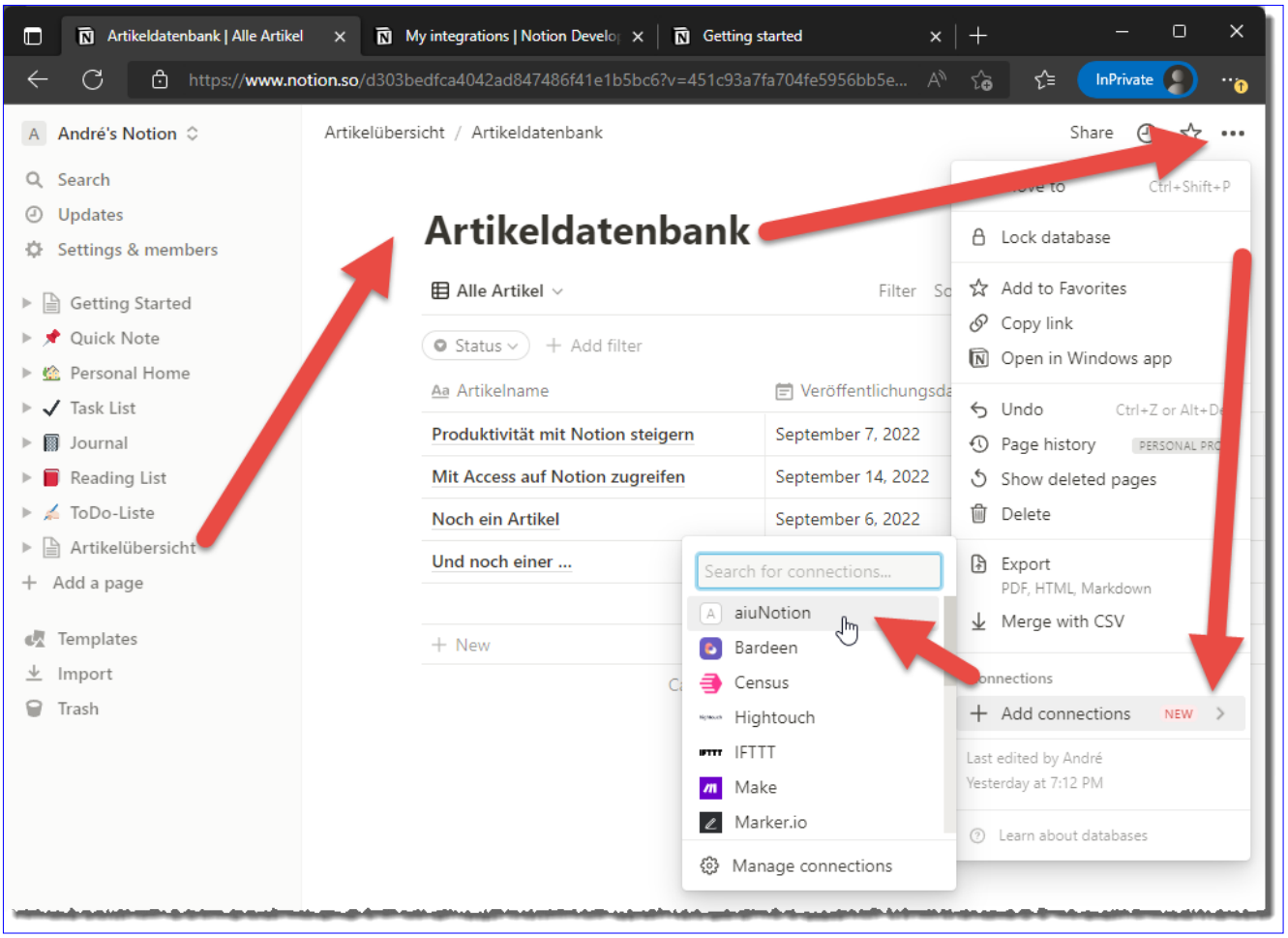


Bild 6: Anlegen einer Verbindung zu einer Notion-Datenbank

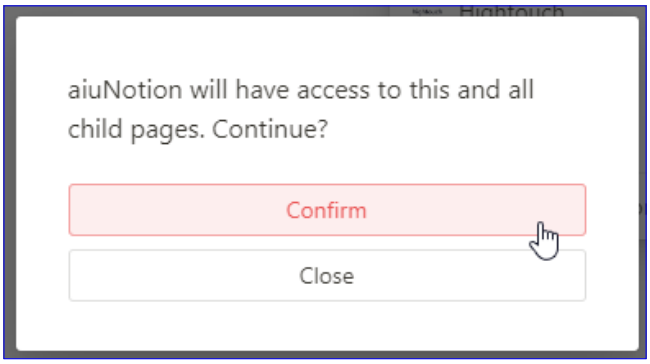


Bild 7: Bestätigen der Freigabe für die Integration

Elemente für die Integration mit VBA freigeben

Bevor wir per VBA auf Datenbanken, Seiten et cetera in Notion zugreifen können, müssen wir diese freigeben beziehungsweise mit unserer Integration teilen. Das gelingt jedoch mit wenigen Schritten.

In meinem Notion-Workspace habe ich eine Datenbank namens **Artikelübersicht** erstellt, mit der ich meine Artikel verwalten möchte – siehe **Produktivität mit Notion steigern** (**www.access-im-unternehmen.de/1402**). Wenn ich diese über den Eintrag in der Seitenleiste aktiviere (entweder in der App oder im Browser), erscheint die entsprechende Datenbank im Hauptbereich.

Nun klicken wir rechts oben auf die drei Punkte, was ein Kontextmenü öffnet. Hier finden wir unten den Eintrag **Connections|Add connections**.

Klicken wir diesen an, sollten wir bereits den Namen unserer Integration in der Liste vorfinden (siehe Bild 6).

Danach erscheint eine Meldung, die wir mit **Confirm** bestätigen (siehe Bild 7).

Ab dann finden wir im soeben geöffneten Kontextmenü unter **Connections** einen neuen Eintrag mit dem Namen unserer Integration. Diese Verbindung können Sie über den einzigen Befehl des Untermenüs dieses Eintrags wieder trennen.

Jede Datenbank einzelnen freigeben

Falls Sie im Laufe der Zeit weitere Datenbanken zu Notion hinzufügen, tauchen diese nicht automatisch auf, wenn Sie, wie nachfolgend beschrieben, die Datenbanken ihres Workspaces auslesen.

Sie müssen jede Datenbank, auf die Ihre Integration zugreifen können soll, einzeln wie oben beschreiben freigeben.

API-Befehle herausfinden

Die Notion-API enthält eine ganze Reihe Befehle, mit denen die meisten der Aktionen durchgeführt werden können, die Sie sonst über die Benutzeroberfläche erledigen würden.

Eine Übersicht der API finden Sie unter dem folgenden Link:

```
https://developers.notion.com/reference/intro
```

Hier gibt es beispielsweise die folgenden API-Befehle:

- **search**: Mit diesem Befehl, angehängt an die URL **https://api.notion.com/v1/**, lesen Sie, wenn ohne Parameter verwendet, alle **database**- oder **page**-Elemente ein, die für die Integration freigegeben wurden. Das enthält also auch die **page**-Elemente, die innerhalb eines **database**-Elements angelegt wurden. Übergeben wir zusätzlich weitere Elemente wie beispielsweise das **query**-Element per Header, können wir auch **database**- oder **page**-Elemente nach dem Titel ermitteln – ein Beispiel finden Sie weiter unten.

- **databases**: Hängen wir den Befehl **databases** an die URL **https://api.notion.com/v1/** an, müssen wir noch die ID der Datenbank folgen lassen. Diese ID erhalten wir beispielsweise über den obigen **search**-Befehl. Außerdem folgt noch **/query**, sodass die URL beispielsweise **https://api.notion.com/v1/databases/d303bedf-ca40-42ad-8474-86f41e1b5bc6/query**

```
Public Function HTTPRequest(strURL As String, strAuthorization As String, Optional strMethod As String = "POST", _
        Optional strContentType As String = "application/json", Optional strVersion As String = "2022-06-28", _
        Optional strData As String, Optional strResponse As String) As Integer
    Dim objHTTP As ServerXMLHTTP60
    Set objHTTP = New MSXML2.ServerXMLHTTP60
    With objHTTP
        .Open strMethod, strURL, False
        .setRequestHeader "Accept", strContentType
        .setRequestHeader "Content-Type", strContentType
        .setRequestHeader "Authorization", strAuthorization
        .setRequestHeader "Notion-Version", strVersion
        .send strData
        strResponse = .responseText
        HTTPRequest = .status
    End With
End Function
```

Listing 1: Prozedur zum Ausführen von HTTP-Anfragen

lautet. Auch hier kann man per Header noch Filterausdrücke übergeben, mit denen nur den Kriterien entsprechende **page**-Elemente ermittelt werden.

- **pages**: Ebenso wie auf **database**-Elemente können wir auch auf **page**-Elemente zugreifen. Dazu müssen wir zuvor die ID der jeweiligen Seite ermittelt haben, was wir beispielsweise mit dem obigen **databases**-Befehl erreichen.

Funktion zum Ausführen einer Anfrage an die Notion-API

Wir definieren als Nächstes eine allgemeine Funktion für das Absenden einer HTTP-Anfrage an die Notion-API (siehe Listing 1). Diese erwartet die folgenden Parameter:

- **strURL**: Aufzurufender API-Endpoint, beispielsweise **https://api.notion.com/v1/search**
- **strMethod**: Optionaler Parameter für die Methode des Aufrufs, also **POST** (Standard), **GET**, **PUT** oder **DELETE**.
- **strAuthorization**: Parameter zum Übergeben des Ausdrucks aus **Bearer** und **Token**
- **strContentType**: Optionaler Parameter zur Übergabe des Inhaltstyps, standardmäßig **application/json**.
- **strVersion**: Optionaler Parameter zur Übergabe der zu verwendenden API-Version, standardmäßig **2022-06-28**.
- **strData**: Optionaler Parameter zur Übergabe zusätzlicher Daten, in diesem Fall im JSON-Format
- **strResponse**: Optionaler Parameter zum Zurückgeben des Ergebnisses des API-Aufrufs

Bevor wir in die Beschreibung der Funktion einsteigen, fügen wir noch einen Verweis auf die Bibliothek **Microsoft XML, v6.0** zum Projekt hinzu. Das erledigen wir im **Verweise**-Dialog des VBA-Editors, den wir mit dem Menübefehl **Extras|Verweise** öffnen (siehe Bild 8).

Die Prozedur erstellt ein neues Objekt des Typs **ServerXMLHTTP60** und ruft dessen **Open**-Methode auf, um die Methode (**POST**) und die URL der API zu übergeben.

Der dritte Parameter gibt mit dem Wert **False** an, dass der Aufruf nicht asynchron gestartet werden soll.

Danach weist die Funktion einige Header hinzu. **Accept** und **Content-Type** erhalten den Wert aus **strContentType**, also standardmäßig **application/json**. Den Header **Authorization** füllen wir mit dem Wert des Parameters **strAuthorization**. Schließlich weisen wir **Notion-Version** den Wert aus **strVersion**, typischerweise **2022-06-28**, zu.

Mit der **send**-Methode schicken wir noch den Inhalt von **strData**. strData nutzen wir, wenn wir zusätzliche Daten zu den mit der URL gesendeten Daten übergeben müssen.

Wenn wir nur Informationen zu verschiedenen Elementen von Notion abrufen wollen wie zu einem **database**- oder **page**-Objekt, brauchen wir nur eine ID per Parameter zu übergeben. Wenn wir jedoch neue **database**- oder **page**-Objekte hinzufügen wollen, dann müssen wir die Eigenschaften dieser Objekte übergeben.

Diese stellen wir wiederum in einem JSON-Dokument zusammen und übergeben dieses mit dem Parameter **strData**. Die **send**-Methode startet auch den eigentlichen Aufruf der API-Funktion von Notion. Das Ergebnis finden wir anschließend in verschiedenen Eigenschaften. **status** liefert einen Zahlenwert wie beispielsweise **200** für einen erfolgreichen Aufruf.

Diesen Wert gibt die Funktion als Ergebnis zurück. Das eigentliche Ergebnis, also beispielsweise Informationen über die Notion-Datenbanken, liefert die Eigenschaft **responseText**, deren Wert wir mit dem Parameter **strResponse** an die aufrufende Instanz zurückschicken.

Konstanten definieren

Im Kopf des Moduls **mdlNotion** speichern wir einige Werte in Konstanten. Diese sehen wie folgt aus:

```
Const cStrURLSearch As String = _
    "https://api.notion.com/v1/search"
Const cStrNotionKey As String = _
    "secret_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
Const cStrContentType As String = "application/json"
Const cStrVersion As String = "2022-06-28"
```

Alle Datenbanken abfragen

Unser erster Aufruf der Funktion **HTTPRequest** soll alle Datenbanken abfragen, die wir in unserem Notion-Workspace angelegt haben. Bevor wir die notwendige Funktion zum Einlesen der Datenbanken per JSON beschreiben, schauen wir uns die Prozedur an, mit der wir einen Test der Funktion starten.

Diese ruft die Funktion **GetJSON** auf und übergibt dieser den zu verwendenden Link (aus **cStrURLSearch**), die Art des Zugriffs (hier **POST**) sowie eine leere **String**-Variable, die mit dem JSON-Dokument gefüllt werden soll. Wenn die Funktion den Wert **True** zurückliefert, enthält die Variable **strResponse** das JSON-Dokument mit den gewünschten Daten. Falls nicht, enthält es die entsprechende Fehlermeldung:

```
Public Sub Test_SEARCH()
    Dim strResponse As String
```

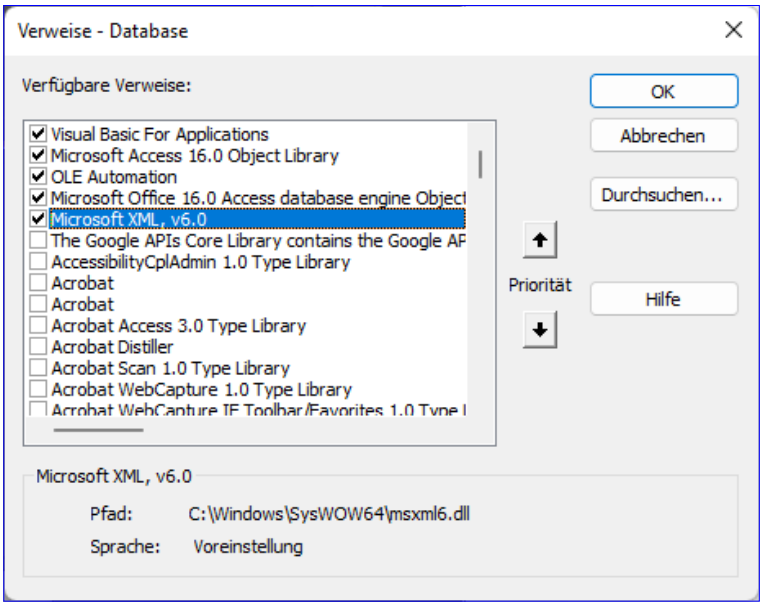


Bild 8: Notwendiger Verweis auf die Bibliothek **Microsoft XML, v6.0**