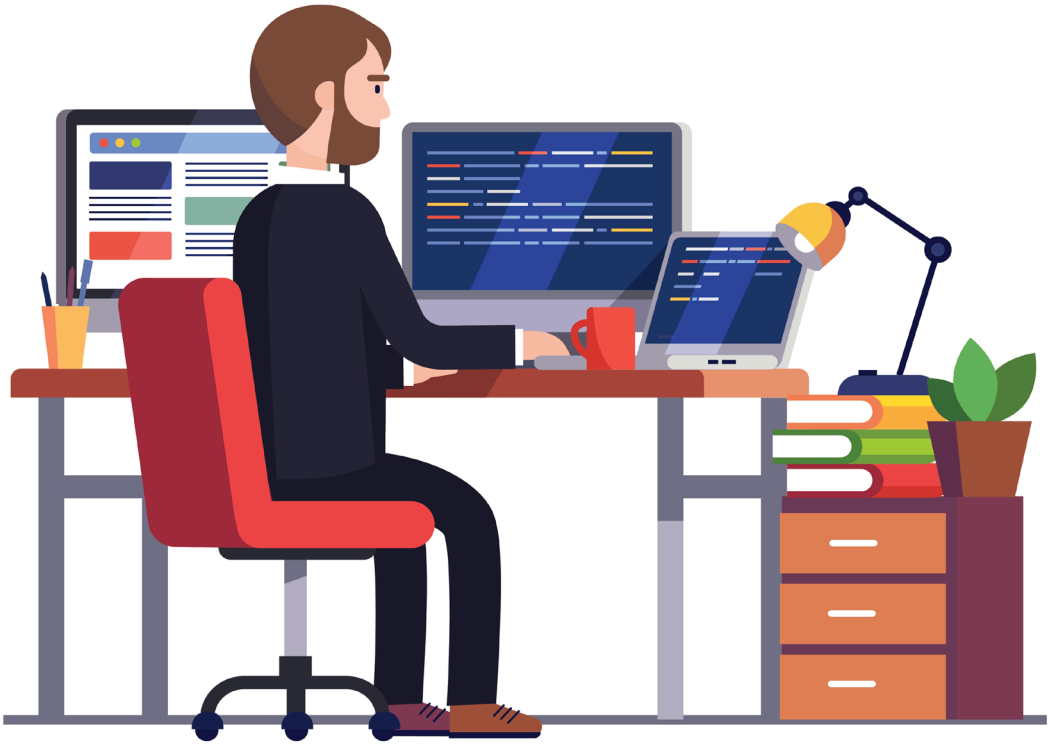
AUSGABE 02/2023

VISUAL BASIC ENTWICKLER

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



IN DIESEM HEFT:

EXCEL ERWEITERN

Lerne, wie Du benutzerdefinierte Funktionen als Formeln verwendest und wie Du eigene Prozeduren über das Ribbon verfügbar machst.

SEITE 4

OPENAI UND VBA

Stelle per VBA Fragen an die künstliche Intelligenz und werte sie aus. Nutze dazu unsere Lösung für das Analysieren von JSON-Dateien!

SEITE 12

EINFACHE .NET-ANWENDUNGEN

Programmiere oft wiederkehrende Aufgaben mit .NET und erstelle eine .exe-Datei daraus, die Du überall nutzen kannst.

SEITE 24

André Minhorst Verlag

Coole Lösungen mit .NET

Ein Magazin, das sich »Visual Basic Entwickler« nennt, kann sich nicht nur auf die Vorstellung von Themen rund um die Office-Programmierung konzentrieren. Deshalb eröffnen wir in dieser Ausgabe ein neues Feld, nämlich die Programmierung mit Visual Basic in der .NET-Variante. Damit können wir auch Office-Lösungen programmieren, zum Beispiel COM-DLLs oder COM-Add-Ins. Aber .NET bietet viel mehr. Vielleicht benötigst Du einmal eine alleinstehende .exe-Anwendung, mit der Du regelmäßige Aufgaben außerhalb von Office erledigen möchtest – beispielsweise um Dateien automatisiert hin- und herzuschieben? Wie das gelingt, zeigen wir in der aktuellen Ausgabe.

Damit Du schon nach der Lektüre weniger Seiten in der Lage bist, selbst Anwendungen mit .NET zu erstellen, haben wir das Wichtigste in den folgenden Artikel beschrieben – und auch noch zwei kleine Lösungen beigefügt, die zeigen, was mit .NET möglich ist. Der erste Artikel heißt **Standalone-Apps mit .NET programmieren** (ab Seite 24). Hier erfährst Du, wie Du Deine erste .NET-Anwendung mit Benutzeroberfläche erstellst. Da die hier verwendete Technik namens WPF sich wesentlich von den in den Office-Anwendungen verwendeten unterscheidet, gehen wir etwas genauer darauf ein, wie Du damit einfache Benutzeroberflächen erstellen kannst.

Gerade wenn Du neue Anwendungen programmierst oder wenn Du diese später häufiger nutzt, möchtest Du einmal gewählte Werte beim nächsten Start automatisch wieder vorfinden, damit Du sie nicht jedes Mal erneut eingeben musst. Wie Du solche Werte speichern kannst, erläutern wir im Artikel **Anwendungsdaten speichern per VB.NET** ab Seite 32.

Um diese Werte direkt zu speichern, wenn Du sie in ein Steuerelement eingegeben hast, benötigst Du ein Ereignis, das direkt nach dem Abschluss der Eingabe ausgelöst wird. Ein solches bieten die WPF-Steuerelemente aber nicht. Die Philosophie von WPF bietet allerdings alternative Techniken, die wir im Artikel **AfterUpdate für WPF-Steuerelemente mit VB.NET** (ab Seite 37) vorstellen.

Schließlich kommen wir zu zwei Lösungen, die wir mit den hier vorgestellten Programmiertechniken realisieren konnten. Die erste findest Du ab Seite 44 unter dem Titel **QR-Codes per .NET-Anwendung erstellen**. Hier zeigen wir, wir Du eine Bibliothek in ein Projekt einbindest, mit de Du QR-Codes erstellen und speichern kannst. In der zweiten Lösung erfährst Du, wie Du PDF-Dokumente aufteilen und wieder zusammenfügen kannst. Damit schaffst Du die Grundlage, größere Dokumente nach bestimmten Kriterien aufzuteilen oder mehrere Dokumente, die ein größeres Dokument ergeben sollen, zu verbinden. Mehr dazu unter dem Titel **PDFs aufteilen und zusammenfügen mit .NET** ab Seite 53.

Auch zum Thema Excel hält diese Ausgabe Know-how bereit: Du lernst, wie Du eigene VBA-Funktionen als Formeln im Tabellenblatt nutzen kannst (**Excel: Benutzerdefinierte Funktionen per Add-In**, ab Seite 4) und wie Du eigene Prozeduren über eine Schaltfläche im Ribbon verfügbar machst (**Excel: Add-In mit Ribbon-Button erstellen**, ab Seite 8).

Und schließlich zeigen wir noch, wie Du mit VBA auf die künstliche Intelligenz von OpenAI zugreifen kannst (**OpenAI mit VBA**, ab Seite 12) und dabei praktische Tools für das Analysieren von JSON-Dateien nutzt (**Mit JSON arbeiten**, ab Seite 19).

Nun viel Spaß beim Lesen!

Dein André Minhorst

Excel: Benutzerdefinierte Funktionen per Add-In

Benutzerdefinierte VBA-Funktionen lassen sich leicht zu einem Excel-Workbook hinzufügen. Sie sind dann aber normalerweise nur in dem entsprechenden Workbook verfügbar. Was aber, wenn Du richtig coole Funktionen entwickelt hast, die Du nicht nur in einem Workbook nutzen möchtest, sondern in verschiedenen Dateien – und Du hast keine Lust, den VBA-Code immer wieder in das VBA-Projekt neuer Workbooks zu kopieren? In diesem Fall gibt es gute Nachrichten: Excel bietet den Dateityp Excel-Add-In an. Darin kannst Funktionen definieren, die immer verfügbar sind.

UDF oder Userdefined Functions

Was wir in der Überschrift **Benutzerdefinierte Funktionen** genannt haben, findest Du auf den englischen Excel-Seiten unter der Bezeichnung **Userdefined Functions**, kurz **UDF**. Wir verwenden in diesem und auch in anderen Artikeln jedoch die deutsche Bezeichnung **Benutzerdefinierte Funktion**.

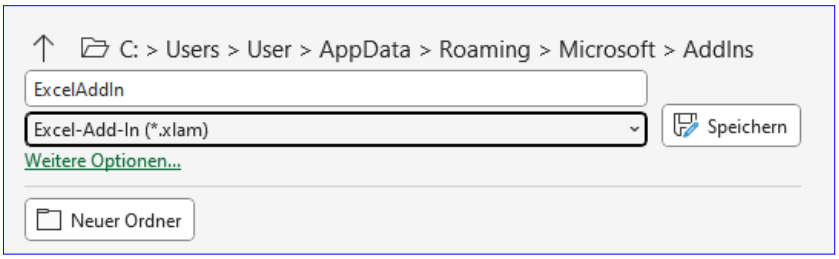


Bild 1: Speichern als Excel-Add-In

Anlegen eines Excel-Add-Ins

Das Erstellen eines Add-Ins besteht erst einmal in einem ganz kleinen Schritt: Dem Speichern einer neuen,

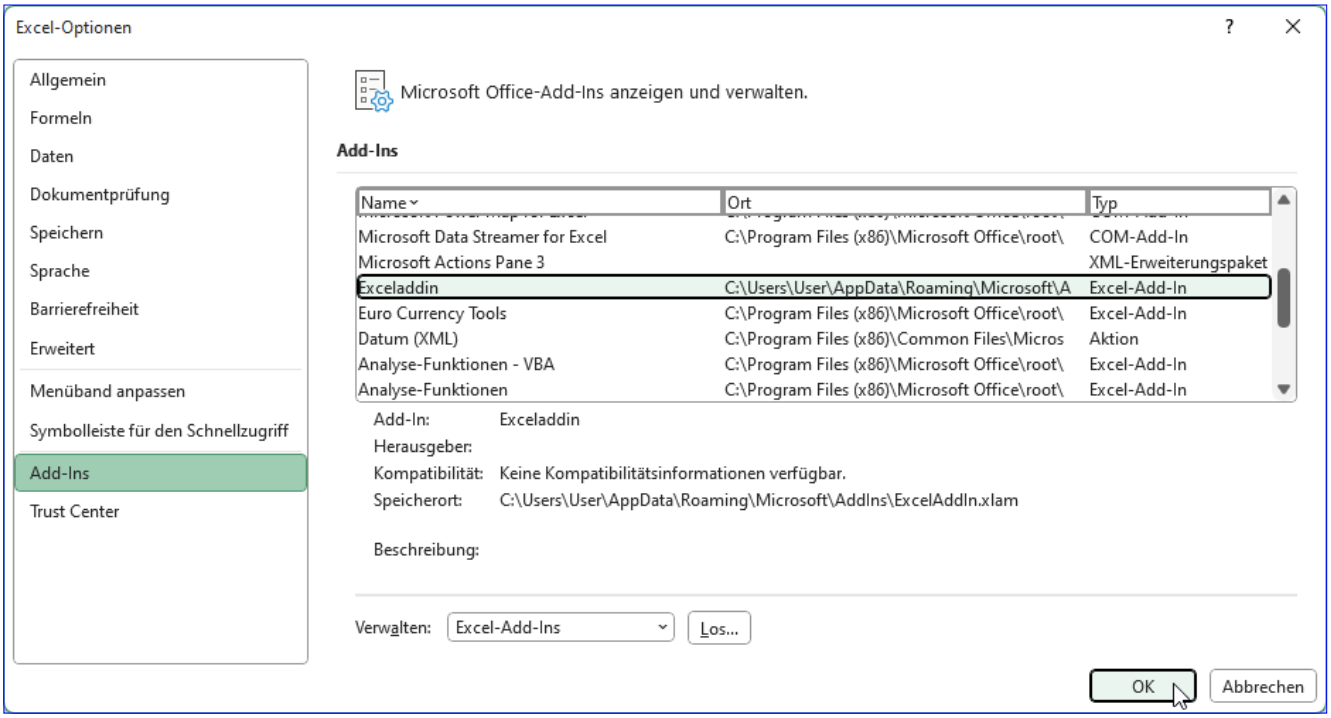


Bild 2: Übersicht der Office-Add-Ins

leeren Excel-Datei (oder auch einer Datei, die bereits die gewünschten Funktionen enthält) im Format **Excel-Add-In (.xlam)**. Dieses Format findest Du, wenn Du die Datei mit dem Ribbonbefehl **Datei|Speichern** unter den Bereich **Speichern unter** im Backstage-Bereich öffnest und dort nach der Auswahl des entsprechenden Dateityps auf **Speichern** klickst (siehe Bild 1).

Als Speicherort wird dabei automatisch der Add-In-Ordner für den aktuellen Benutzer gewählt, der sich im Verzeichnis **C:\Users\[Benutzername]\AppData\Roaming\Microsoft\AddIns** befindet.

Ob das Anlegen funktioniert hat, und das Excel-Add-In verfügbar ist, findest Du in den Excel-Optionen heraus. Dazu klickst Du auf **Datei|Optionen** und wechselst im nun erscheinenden Dialog **Excel-Optionen** auf den Bereich **Add-Ins**. Hier sehen wir schon den Eintrag für unser Add-In (siehe Bild 2).

Wählen wir dann unten unter Verwalten den Eintrag **Excel-Add-Ins** aus und klicken auf **Los...**, erscheint ein Dialog, der nur die Excel-Add-Ins anzeigt (siehe Bild 3). Auch hier finden wir unser Add-In vor. Klicken wir es dort an, sehen wir allerdings keinen Beschreibungstext wie bei den übrigen Add-Ins der Liste. Aber keine Sorge – darum kümmern wir uns später.

Add-In verbinden

Es kann sein, dass das Add-In im Dialog **Add-Ins** mit einem leeren Kontrollkästchen angezeigt wird. In diesem Fall ist es noch nicht verbunden. Setze einen Haken in das Kontrollkästchen und Du kannst das Add-In nutzen – und zwar in allen Excel-Workbooks, die Du auf diesem Rechner öffnest.

Testfunktion im Excel-Add-In anlegen

Wir starten mit einer einfachen Testfunktion. Diese soll einfach nur den Text **Test** zurückgeben. Dazu legen wir in der Datei **ExcelAddIn.xlam** ein neues VBA-Modul an:

- Öffne den VBA-Editor mit **Alt + F11**.
- Betätige den Menübefehl **Einfügen|Modul**.
- Ändere den Namen des neuen Moduls in **mdlAddIn**.
- Füge eine Funktion wie in Bild 4 hinzu.

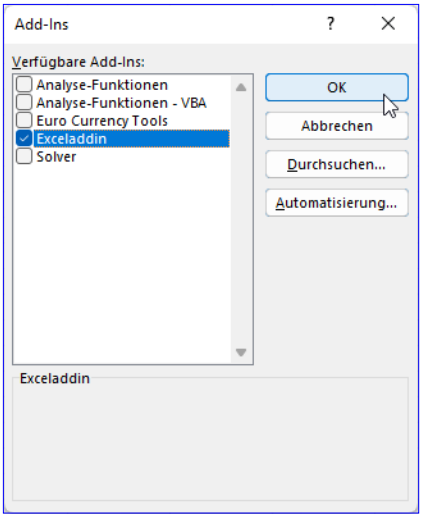


Bild 3: Der Dialog **Add-Ins**

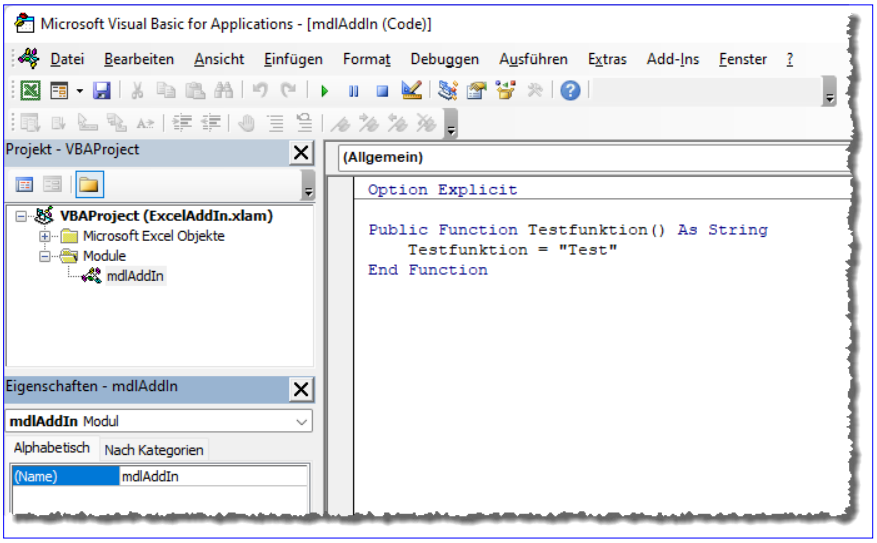


Bild 4: Die Funktion des Add-Ins

Excel: Add-In mit Ribbon-Button erstellen

Wir können Excel nicht nur um benutzerdefinierte Funktionen erweitern, die wir in eine Add-In-Datei schreiben und die dann überall als Formel verfügbar sind. Wir können auch Funktionen hinzufügen, mit denen wir die Anwendung um selbst programmierte Abläufe erweitern. In diesem Artikel zeigen wir, wie Du einem Excel-Add-In ein eigenes Ribbontab hinzufügst sowie eine Funktion, die durch einen Button des Ribbontabs ausgelöst wird.

Wenn Du nicht nur eigene Funktionen benötigst, die Du als Formeln in Excel nutzen möchtest, sondern auch noch VBA-Prozeduren programmiert hast, die Du gern in allen Excel-Workbooks auf Deinem Rechner einsetzen willst, findest Du in diesem Artikel die benötigten Vorgehensweise. Wir werden:

- eine Excel-Add-In-Datei erstellen,
- diese verfügbar machen,
- der Datei ein Ribbon mit einer Schaltfläche und einem Icon hinzufügen sowie
- eine Prozedur anlegen, die durch diese Schaltfläche ausgelöst wird.

Excel-Add-In-Datei erstellen

Das Erstellen der Add-In-Datei ist in wenigen Sekunden erledigt:

- Lege ein neues Excel-Workbook an.
- Speichere es unter dem Namen **ExcelAddInMitRibbon.xlam** als **Excel-Add-In (*.xlam)**.

Fertig! Die neue Add-In-Datei wird automatisch in das Verzeichnis **C:\Users\[Benutzername]\AppData\Roaming\Microsoft\AddIns** verschoben.

Excel-Add-In verfügbar machen

Um das Excel-Add-In in Excel auf dem aktuellen Rechner verfügbar zu machen, nehmen wir eine Einstellungen in den Optionen von Excel vor:

- Klicke im Ribbon auf **Datei|Optionen**.
- Wechsle zum Bereich **Add-Ins**.
- Klicke unter **Verwalten** für den Eintrag **Excel-Add-Ins** auf die Schaltfläche **Los...**

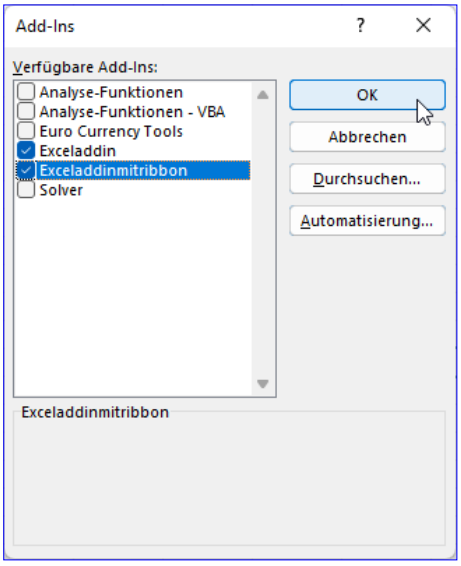


Bild 1: Aktivieren des Add-Ins

- Suche den Eintrag **Exceladdinmitribbon** aus und setze einen Haken vor diesen Eintrag (siehe Bild 1) und schließe den Dialog und die Optionen wieder.

Ribbon hinzufügen

Nun schließen wir die Add-In-Datei und öffnen ein Tool namens **Office RibbonX Editor** – mehr dazu siehe im Artikel **Ribbons in Office-Dokumenten** (**www.vbentwickler.de/329**).

- Hier öffnen wir zuerst die frisch erstellte Excel-Add-In-Datei.
- Danach erscheint im linken Bereich des **Office RibbonX Editors** ein Eintrag mit dem Namen der Daten.

- Diesen klicken wir mit der rechten Maustaste an und wählen den Befehl **Office 2010+ Custom UI-Abschnitt einfügen** aus (siehe Bild 2).
- Dies fügt ein Unterelement namens **customUI14.xml** hinzu. Dieses klicken wir doppelt an.
- Im nun erscheinenden neuen Bereich auf der rechten Seite fügen wir den Code aus Listing 1 ein.
- Das Ergebnis sieht nun wie in Bild 3 aus.
- Nun klicken wir noch auf die Schaltfläche **VBA-Methoden generieren**. Dies öffnet den Dialog aus Bild 4. Die hier angezeigte Prozedur kopieren wir in die Zwischenablage.

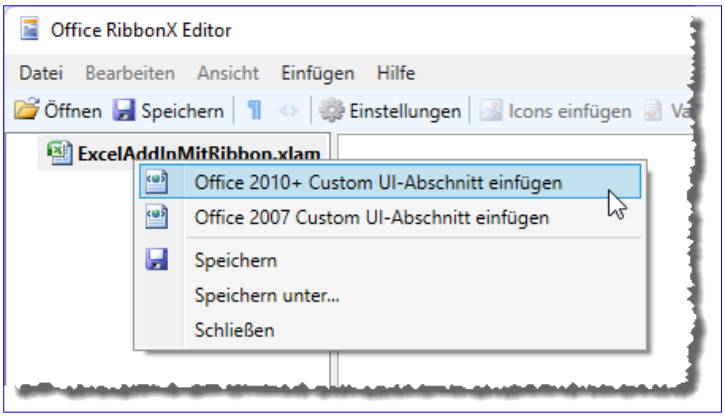
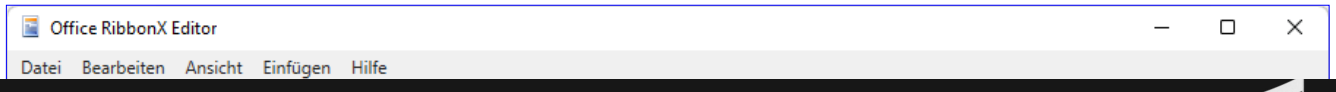


Bild 2: Hinzufügen eines Ribbons zum Excel-Add-In

```
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
    <ribbon>
        <tabs>
            <tab id="tabAmvFunktionen" label="amvFunktionen">
                <group id="grpBeispiele" label="Beispiele" >
                    <button id="btnBeispiel" label="Beispiel" onAction="onAction"/>
                </group>
            </tab>
        </tabs>
    </ribbon>
</customUI>
```

Listing 1: Ribbon-Definition für ein einfaches Ribbon



OpenAI mit VBA

Schlagwörter wie OpenAI oder ChatGPT beherrschen die Schlagzeilen der Welt. Wir wollen nicht entscheiden, ob es gut oder schlecht ist, ob es Dir den Job wegnimmt oder ob es eine super Unterstützung ist, sondern wir zeigen in diesem Artikel einfach, wie Du es per VBA steuern und in eigenen Anwendungen nutzen kannst. Dabei greifen wir auf die Rest API von OpenAI zu und ermöglichen, dass Du die Antworten auf die mit der Anwendung eingegebene Frage nutzen kannst.

Ob ChatGPT, OpenAI et cetera nun nützlich für uns Entwickler sind oder nicht: Wir können per Rest API darauf zugreifen, also ist es auf jeden Fall interessant! Den Einstieg findest Du unter folgendem Link:

```
https://openai.com/api/
```

Entweder Du hast schon einen Zugang, dann meldest Du Dich dort unter **Log In** an, oder Du registrierst Dich auf der Seite mit dem Button **Sign up**. Dazu benötigst Du einen Google- oder Microsoft-Account oder Du registrierst Dich neu mit Deiner E-Mail-Adresse. Im letzteren Fall klickst Du noch den Bestätigungslink in der an Dich versendeten E-Mail an, um die Registrierung abzuschließen. Schließlich fragt OpenAI noch einige Informationen wie Name, Telefonnummer et cetera ab. Die Telefonnummer muss mit dem an diese Nummer gesendeten Code bestätigt werden.

Auf der Plattform findest Du anschließend oben rechts die Möglichkeit, auf Deinen Account zuzugreifen und dort finden wir auch den Befehl **View API Keys**. (siehe Bild 1). Während wir Abfragen über die Webseite ohne Weiteres absenden können, weil wir dort angemeldet sind, benötigen wir für den Zugriff über die Rest API einen API Key, um uns zu authentifizieren.

Warum sollen wir uns überhaupt anmelden und können OpenAI nicht anonym nutzen? Weil es kostenpflichtig ist. Die aktuellen Preise für die Nutzung der verschiedenen APIs finden wir auf der folgenden Seite:

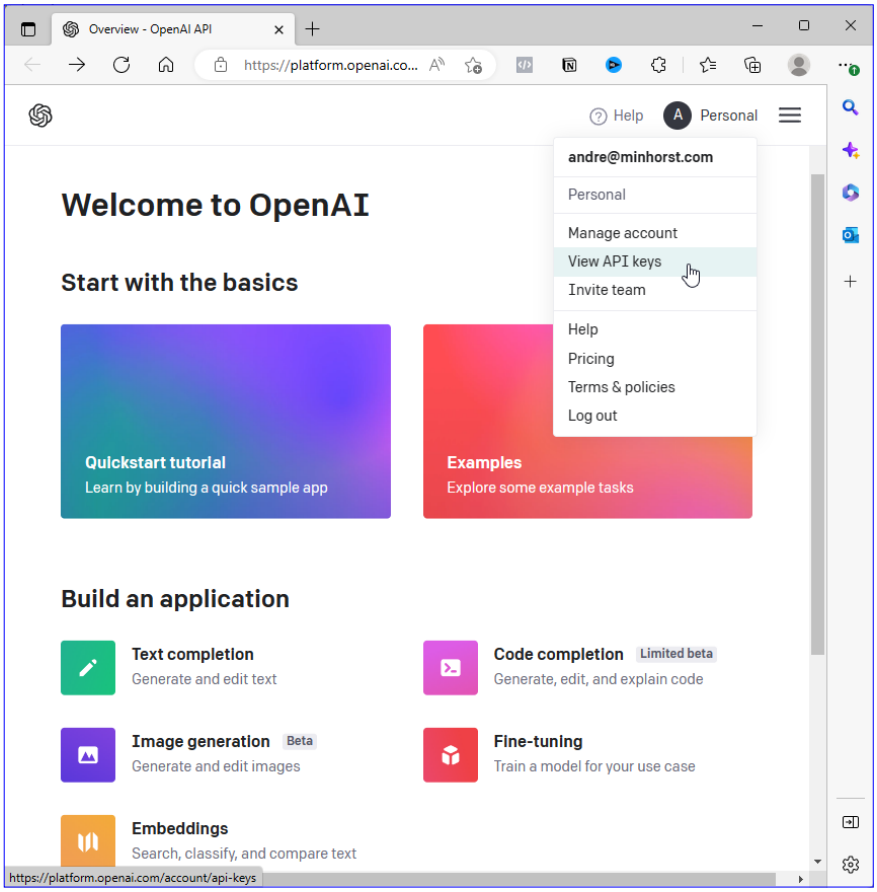


Bild 1: Von OpenAI zum API Key

```
https://openai.com/api/pricing/
```

Allerdings werden wir nicht sofort zur Kasse gebeten, sondern bekommen Zugriffe im Wert von 18$ geschenkt, die wir innerhalb von drei Monaten ab unserer Registrierung für zum Ausprobieren einsetzen können. Genug, um einige Experimente von unserer VBA-Anwendung aus durchzuführen.

API Key holen

Wir benötigen also einen API Key, und den holen wir uns unter dem folgenden Link:

```
https://platform.openai.com/account/api-keys
```

Hier klicken wir einfach auf **Create new secret key** (siehe Bild 2). Den damit gewonnenen Key speichern wir in der Zwischenablage und fügen ihn dann als Konstante in ein VBA-Standardmodul ein:

```
Const cstrAPIkey As String = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
```

Erste Abfrage an OpenAI

Danach geht es auch direkt ans Werk. Was wir programmieren wollen, ist eine allgemein nutzbare VBA-Funktion, die wir in jede beliebige Office-Anwendung einbauen und von dort aus aufrufen können. Ein einfacher Aufruf soll wie folgt aussehen:

```
Debug.Print OpenAI("Wie kann ich OpenAI in Microsoft Office nutzen?")
```

Die Antwort von OpenAI lautet:

```
OpenAI kann nicht direkt in Microsoft Office genutzt werden, da es sich dabei um zwei verschiedene Programme handelt. Allerdings können Sie die Technologien, die OpenAI entwickelt hat, in Microsoft Office verwenden. Zum Beispiel können Sie Microsoft Word mit einer OpenAI-Künstlichen Intelligenz (AI) erweitern, um automatisierten Text zu schreiben. Auch Microsoft Excel kann mit OpenAI-Technologien für maschinelles Lernen, Statistiken und Prognosen erweitert werden.
```

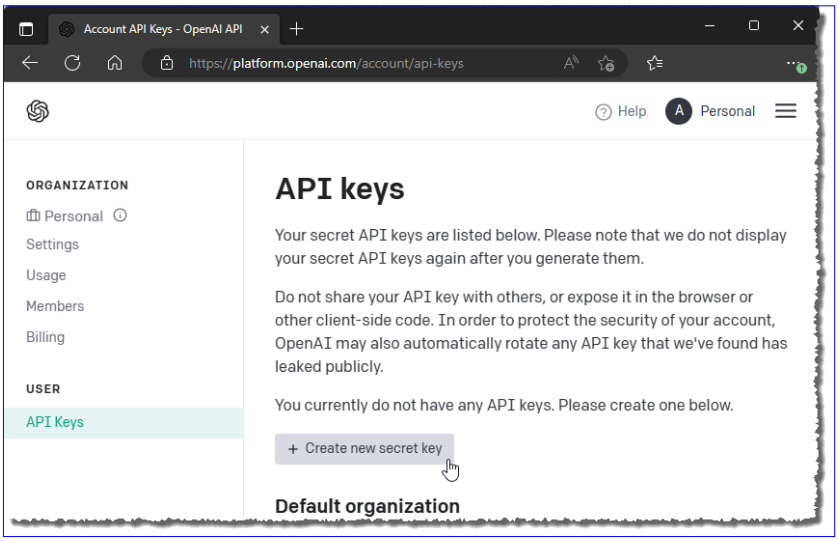


Bild 2: API key holen

Bis zu diesem Ergebnis fehlen uns allerdings noch ein paar Zeilen Code.

Die Dokumentation der OpenAI-API

Die Endpoints der Rest API und ihre Beschreibung finden wir unter dem folgenden Link:

```
https://platform.openai.com/docs/api-reference/introduction
```

Uns interessiert hier speziell die Completions-API, mit der wir im Playground interessante Ergebnisse erhalten haben. Wenn Du erstmal einige Experimente direkt mit OpenAI durchführen möchtest, kannst Du das im Playground unter der folgenden Adresse tun:

```
https://platform.openai.com/playground
```

Hier kannst Du beispielsweise eine Frage wie die folgende stellen:

Schreibe eine VBA-Prozedur, mit der ich alle Datensätze einer Tabelle namens tblKunden mit den Feldern KundeID, Vorname und Nachname durchlaufen kann.

Die Antwort findest Du in Bild 3 – das ist wirklich beeindruckend gemessen daran, welchen Aufwand man als Anfänger sonst hätte, passenden Code im Internet zu suchen! Der Code ist nicht perfekt, aber man könnte ihn direkt in ein VBA-Modul übernehmen und ausprobieren. Auf der rechten Seite findest Du einige Parameter, die wir auch bei Verwendung der Rest API nutzen können – und die in der Dokumentation genauer beschrieben werden.

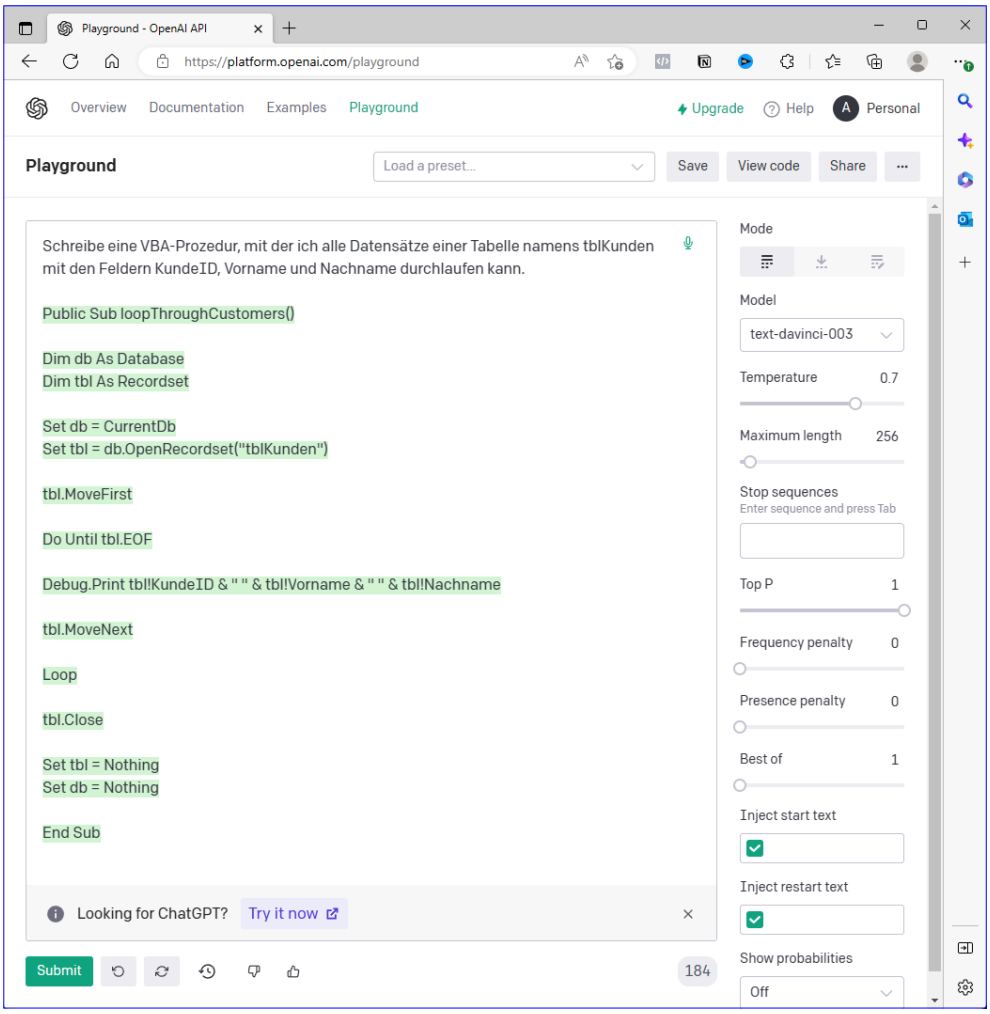


Bild 3: Beispiel für das Ermitteln von Code per OpenAI

Bestandteile des Aufrufs der Rest API von OpenAI

Funktion zum Zusammenstellen des Aufrufs und zum Parsen des Ergebnisses

Mit JSON arbeiten

JSON heißt JavaScript Object Notation und ist wie XML ein Format, mit dem Daten strukturiert gespeichert werden können. Als Visual Basic-Entwickler hat man üblicherweise nicht viele Berührungspunkte mit dieser Notation. Wenn man jedoch gelegentlich mit Webservices beziehungsweise Rest APIs arbeitet, findest der Datenaustausch entweder mit XML oder JSON statt. Während es für den Zugriff auf den Inhalt von XML-Dokumenten die »Microsoft XML, vX.0«-Bibliothek gibt, ist man bei JSON auf Lösungen von Drittherstellern angewiesen. In diesem Fall nutzen wir eine Bibliothek von Tim Hall und eine Erweiterung im Eigenbau, mit der wir relativ einfach auf die Daten in JSON-Dokumenten zugreifen können.

Wer mit Rest APIs wie mit dem aktuell gefragten Webservice von OpenAI arbeitet, bekommt von diesen Antworten entweder im XML- oder im JSON-Format. Wie Du die Rest API von OpenAI ansprichst, um die dahinter stehende, sogenannte »künstliche Intelligenz« zu nutzen, erklären wir im Artikel **OpenAI mit VBA** (**www.vbentwickler.de/355**). Das Ergebnis eines solchen Aufrufs sieht wie in Listing 1 aus.

Ein solches Ergebnis können wir in eine **String**-Variable einfügen und dann mithilfe von Zeichenketten-Funktionen die gewünschten Informationen heraus-

```
{
   "id":"cmpl-6kCY6I3xWbEUljecYvr9nKV5GsY3s",
   "object":"text_completion",
   "created":1676469346,
   "model":"text-davinci-003",
   "choices":[
      {
         "text":"\n\nLeider gibt es keine native OpenAI-Integration in Microsoft Office. Wenn Sie OpenAI jedoch in
            Microsoft Office verwenden möchten, können Sie versuchen, eine Lösung zu implementieren, die OpenAI-
            APIs verwendet, um die Anwendungen in Ihrer Office-Suite zu erweitern. Solche Lösungen beinhalten
            normalerweise die Entwicklung eigener Apps und Add-Ins, die OpenAI-APIs verwenden, um bestimmte
            Funktionen in Microsoft Office bereitzustellen.",
         "index":0,
         "logprobs":null,
         "finish_reason":"stop"
      }
   ],
   "usage":{
      "prompt_tokens":14,
      "completion_tokens":146,
      "total_tokens":160
   }
}
```

Listing 1: Beispiel für eine JSON-Datei

```
Public Sub JSONPerZeichenkette()
    Dim strJSON As String
    Dim strText As String
    Dim lngStart As Long
    Dim lngEnde As Long
    strJSON = strJSON & "{""id"":""cmpl-6kCY6I3xWbEUljecYvr9nKV5GsY3s"",""object"":""text_completion"","
        & """created"":1676469346,""model"":""text-davinci-003"",""choices"":[{""text"":""\n\nLeider gibt es "
        & "keine native OpenAI-Integration in Microsoft Office. ..."",""index"":0,""logprobs"":null,"
        & """finish_reason"":""stop""}],""usage"":{""prompt_tokens"":14,""completion_tokens"":146,"
        & """total_tokens"":160}}" & vbCrLf
    lngStart = InStr(1, strJSON, """text"":""")
    If Not lngStart = 0 Then
        lngStart = lngStart + Len("""text"":""")
        lngEnde = InStr(lngStart, strJSON, """,")
        If Not lngEnde = 0 Then
            strText = Mid(strJSON, lngStart, lngEnde - lngStart)
            Debug.Print strText
        End If
    End If
End Sub
```

Listing 2: Parsen auf herkömmliche Art mit Zeichenkettenfunktionen

filtern. Wenn wir für das JSON-Dokument aus dem Beispiel etwa den Inhalt des Elements **text** erhalten wollen, könnten wir wie folgt vorgehen – und hier gehen wir davon aus, dass das JSON-Dokument nicht so schön formatiert daherkommt wie im abgebildeten Listing, sondern ohne Zeilenumbrüche und ohne Einrückungen. Genau genommen wären die formatierenden Elemente beim Auslesen von Daten per VBA auch eher hinderlich (siehe Listing 2). Hier haben wir das zu untersuchende JSON-Dokument in die Variable

dingung der **If...Then**-Bedingung erfüllt. Da wir aber eigentlich die Startposition des Inhalts von **text** haben wollen, zählen wir noch die Länge der Zeichenkette, die dem gesuchten Text vorausgeht, hinzu. Nun benötigen wir noch die Endposition des einzulesenden Textes. Dazu gehen wir davon aus, dass dieser durch ein Anführungszeichen mit einem folgenden Komma abgeschlossen wird. Die Position dieser Fundstelle ermitteln wir wieder mit **InStr**, diesmal allerdings mit der Position aus **lngStart** als Startposition. Finden wir

Standalone-Apps mit .NET programmieren

Nicht immer möchte man Erweiterungen gezielt für eine Office-Anwendung programmieren. Gelegentlich fallen Aufgaben an, die man zwar mit einer der Office-Anwendungen erledigen könnte, aber dazu benötigt man auch immer die jeweilige Office-Anwendung und ein passendes Dokument wie ein Excel-Workbook oder eine Access-Datenbank. Und gerade bei Access ist eine der meist gestellten Fragen: Wie kann ich die Datenbank in eine .exe-Datei umwandeln? Die Antwort lautet: Gar nicht. Wenn es aber allein darum geht, Aufgaben zu erledigen, die nicht unbedingt mit Office zusammenhängen, dann könnte man auch schnell eine .NET-App programmieren. Die hat den Vorteil, dass man erstens viel mehr Steuerelemente nutzen kann, zweitens viel mehr Bibliotheken zur Verfügung hat und drittens eine .exe erstellen kann, die man sogar noch weitergeben kann. In diesem Artikel schauen wir uns die Grundlagen für die Erstellung einer einfachen .exe-Datei mit .NET an.

Aufgaben für eine .NET-App

Aufgaben für solche Anwendungen gibt es wie Sand am Meer. Damit der Aufwand zum Erstellen einer solchen Anwendung gerechtfertigt ist, sollte die Aufgabe nicht nur einmal vorkommen, sonst könnte man diese auch von Hand erledigen. Es sollte sich also um eine Aufgabe handeln, die entweder sehr umfangreich ist und dennoch sich wiederholende Schritte enthält oder eine, die man regelmäßig erledigen muss und für die man die Anwendung immer wieder nutzen kann.

Hier sind ein paar Beispiele:

- Dokumente umbenennen und kopieren oder verschieben, zum Beispiel um regelmäßige Eingänge in den Download-Ordner in andere Ordner zu übertragen
- E-Mails von Outlook auslesen und in bestimmten Verzeichnissen speichern
- Seiten aus PDF-Dokumenten extrahieren oder zu neuen Dokumenten zusammensetzen – Beispiel siehe **PDFs aufteilen und zusammenfügen mit .NET** (**www.vbentwickler.de/356**)
- PDF-Dokumente analysieren (Beispiel siehe **PDF-Dokumente analysieren mit VB.NET** (**www.vbentwickler.de/357**)
- Alle möglichen Aufgaben, die mit dem Abrufen oder Hochladen von Daten an REST APIs beziehungsweise Webservices zu tun haben – .NET bietet hier für alle möglichen Dienste Bibliotheken in sogenannten NuGet-Paketen an
- Daten aus verschiedenen Quellen einlesen und weiterverarbeiten
- Automation von Abläufen wie Sichern von Dateien, Senden von E-Mails oder Ausführen von Skripten

Dir fallen sicher noch weitere Anwendungsfälle ein. Diese kannst Du gern als Thema für weitere Artikel vorschlagen – schreibe dazu einfach eine E-Mail an **andre@minhorst.com**.

Werkzeuge

Wir stellen in diesem Artikel die Vorgehensweise und einige Grundlagen zum Erstellen von **.exe**-Programmen mit Benutzeroberfläche vor. Die Benutzerober-

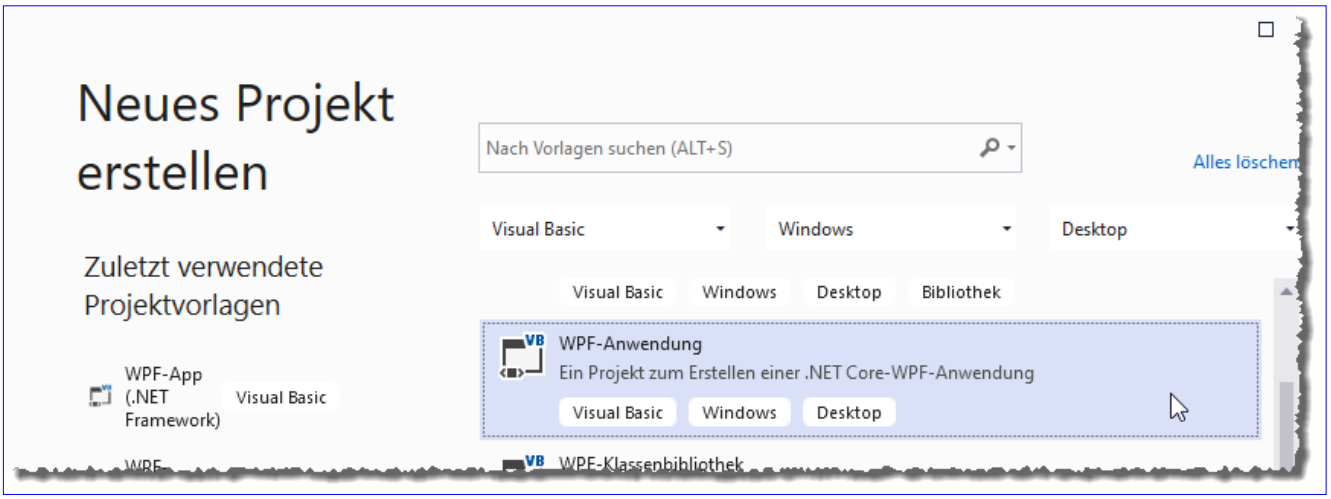


Bild 1: Erstellen eines neuen Projekts in Visual Studio

fläche erstellen wir mit WPF. Als Programmiersprache verwenden wir Visual Basic in der .NET-Variante.

Neues Projekt erstellen

Um ein für unsere Anforderungen geeignetes Projekt zu erstellen, rufen wir in Visual Studio den Menübefehl **Datei|Neu|Projekt...** auf. Im nun folgenden Dialog **Neues Projekt erstellen** können wir mit den Auswahlfeldern die Projekttypen einschränken. Wir wollen uns auf Visual Basic als Sprache, Windows als Betriebssystem und Desktop als Ziel konzentrieren und finden dann in der Liste den Eintrag **WPF-Anwendung – Ein Projekt zum Erstellen einer .NET Core-WPF-Anwendung** (siehe Bild 1). Es gibt auch noch den Typ **WPF-App (.NET Framework)**. Die Unterschiede liegen im Wesentlichen darin, dass die .NET-Core-Variante moderner ist und grundsätzlich plattformübergreifend genutzt werden kann. Wie es bei moderneren Versionen üblich ist, findet man jedoch nicht alle Bibliotheken, die auch für ältere Projektvorlagen verfügbar sind. Da wir aktuell nur Beispiele für die Windows-Plattform programmieren wollen, reicht also der Typ **WPF-App (.NET Framework)** aus – daher verwenden wir diesen Typ.

Dass Microsoft diesen Projekttyp nicht mehr weiterentwickelt, sollte man ebenfalls berücksichtigen. Hier spielt der Zeitfaktor eine Rolle: Wenn das Tool nur über einen begrenzten Zeitraum eingesetzt werden soll, kann man für das .NET Framework programmieren. Wenn die Anwendung jedoch etwas zukunftssicherer sein soll, ist die Programmierung für .NET Core sinnvoller.

Projekt konfigurieren

Der folgende Schritt zeigt die wesentlichen Merkmale des Projekts an, die für die Erstellung nötig sind (siehe Bild 2). Hier geben wir zuerst einen Projektnamen an, in diesem Fall schlicht Beispiel-App. Unter **Ort** legen wir das Verzeichnis fest, in dem die Projektdateien gespeichert werden sollen. In der Regel benötigen wir eine neue Projektmappe, weshalb wir diese Einstellung beibehalten. Es kann sein, dass wir zwei oder mehr Projekte in einer Projektmappe verwalten wollen, dann würden wir hier den Wert **Hinzufügen** wählen. Falls wir eine Projektmappe anlegen wollen, geben wir für diese auch einen Namen an. Da wir in den meisten Fällen ein Projekt in einer Projektmappe verwalten, verwenden wir den gleichen Namen wie für das Projekt.

Beim Typ **WPF-App (.NET Framework)** wählen wir im letzten Schritt noch das gewünschte Zielframework aus – das ist beim Erstellen eines Projekts für .NET

Core nicht nötig. Hier ist die Auswahl der aktuellsten Version sinnvoll, es sei denn, der Zielrechner verwendet ein älteres Framework und kann nicht aktualisiert werden.

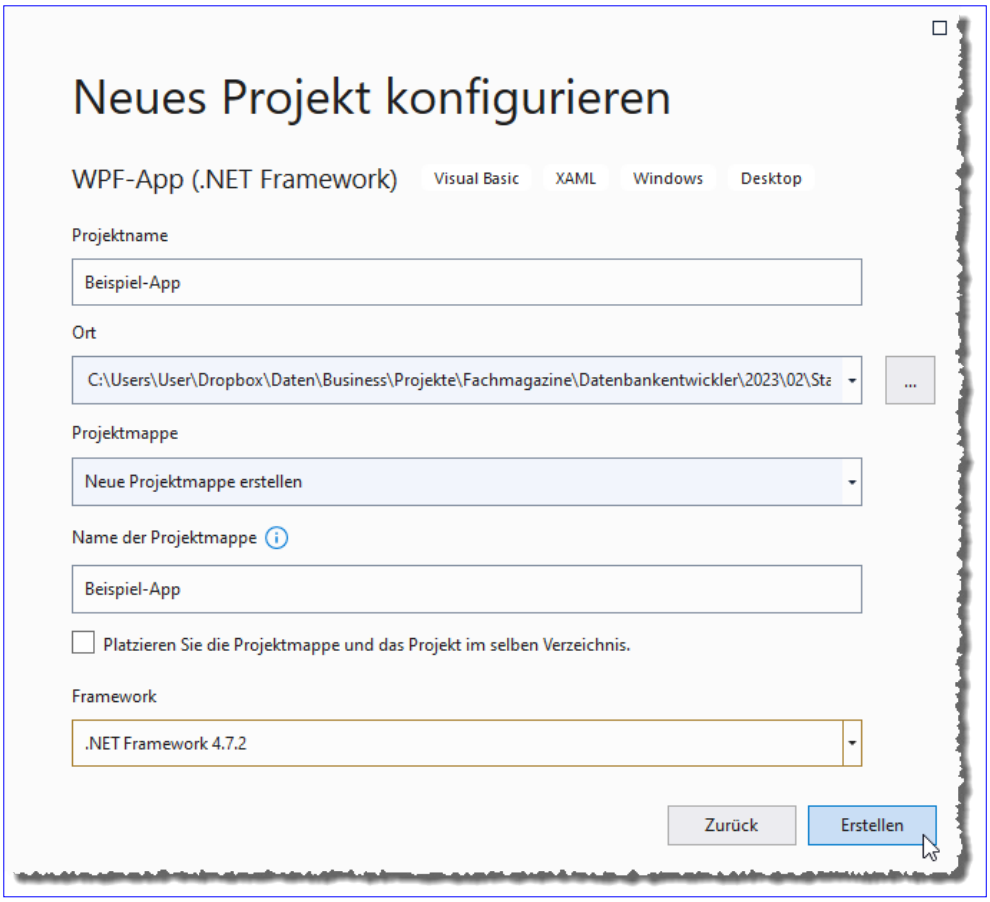


Bild 2: Eingabe der Projekteigenschaften

Grundelemente von Visual Studio, die wir kennen müssen

Danach erscheint Visual Studio mit einigen Elementen (siehe Bild 3). Bei einem WPF-Projekt gehört dazu immer ein erstes WPF-Fenster namens **MainWindow.xaml**. Dieses stellt, wenn wir die Anwendung nun starten würden, die Benutzeroberfläche der Anwendung. Hier können wir Steuerelemente anlegen, diese mit Inhalten versehen und Methoden für die verschiedenen Ereignisse definieren. Letzteres erledigen wir im sogenannten Code behind-Modul, dass prinzipiell wie ein Klassenmodul für ein Access-Formular betrachtet werden kann. Es nimmt den Code auf, der direkt durch die Ereignisse des Fensters oder seines Steuerelemente aufgerufen wird.

Das Element **MainWindow.xaml** ist standardmäßig nach dem Erstellen geöffnet und enthält zwei Bereiche:

- Im oberen Bereich sehen wir den Entwurf der Benutzeroberfläche.
- Im unteren Bereich sehen wir den Code, der diese Benutzeroberfläche definiert.

Die beiden Bereiche aktualisieren sich gegenseitig – wenn Du ein Steuerelement aus der Toolbox hinzufügst, wird dafür ein entsprechender Eintrag im unteren Bereich angelegt. Und wenn Du per Code ein Steuerelement hinzufügst oder seine Eigenschaften änderst, wirken sich diese Änderungen direkt auf den Entwurf der Benutzeroberfläche aus.

Nicht sichtbar ist hier der Code, der durch die Elemente der Benutzeroberfläche ausgelöst wird. Diesen blenden wir über den Projektmappen-Explorer ein. Dieser wird standardmäßig im rechten Bereich angezeigt und listet die bisherigen Elemente der Projektmappe und des Projekts auf. Das für uns aktuell wichtigste Element ist **MainWindow.xaml**. Um den Code zu diesem Element anzuzeigen, klicken wir auf den nach rechts zeigenden Pfeil links von diesem Eintrag und erwei-

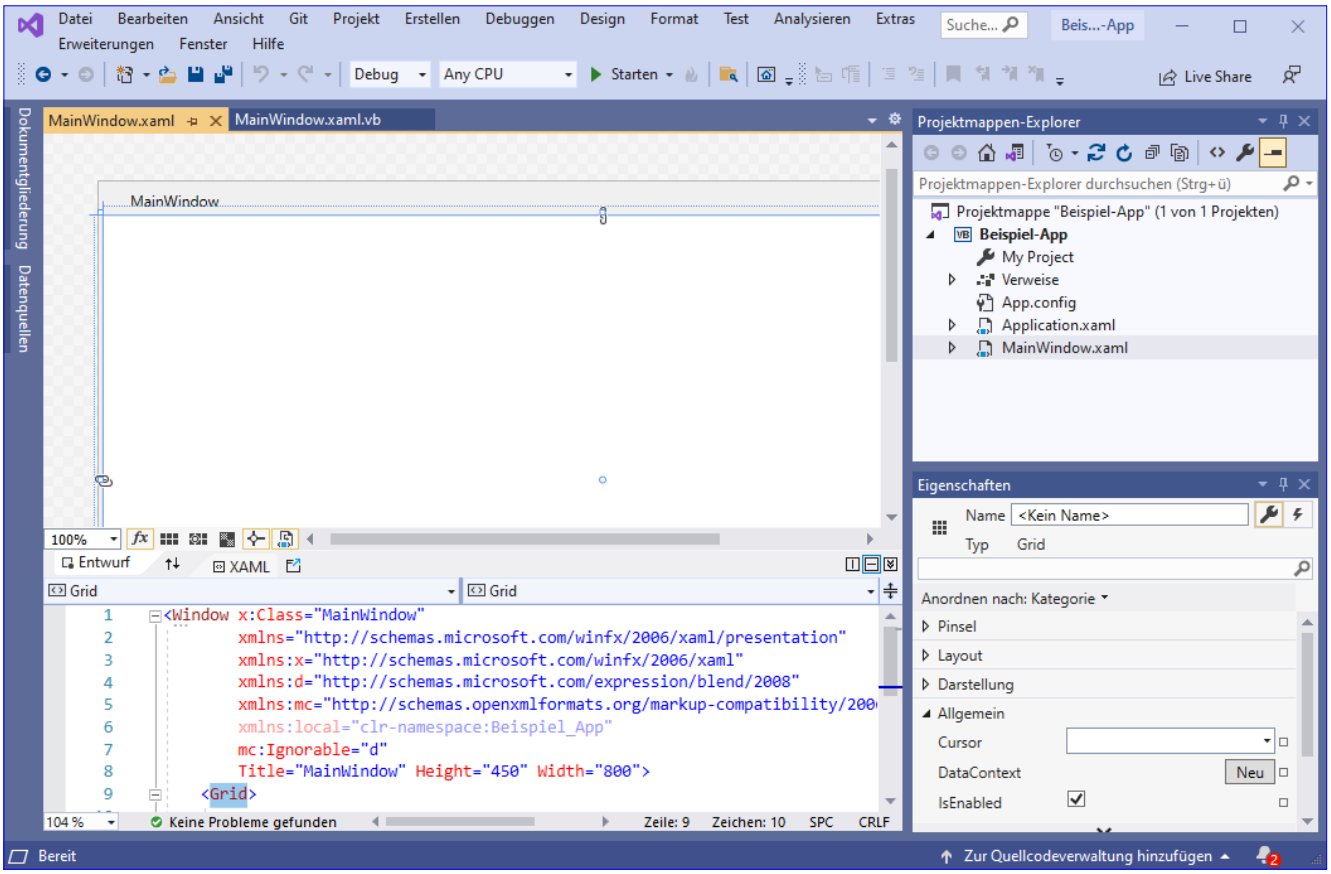


Bild 3: Die Elemente von Visual Studio

tern die Anzeige damit so, dass nun ein weiteres Element namens **MainWindow.xaml.vb** erscheint. Mit einem Doppelklick auf diesen Eintrag zeigen wir das Codefenster mit dem Code behind-Modul an, das aktuell nur das Element zur Definition der Klasse enthält

Schaltfläche hinzufügen

Damit beginnen wir gleich mit dem Programmieren einfacher Funktionen. Wir wollen eine Schaltfläche hinzufügen. Dazu haben wir zwei Möglichkeiten. Für den Anfang ist es am einfachsten, einfach mit

Anwendungsdaten speichern per VB.NET

Wenn man wie im Artikel »Standalone-Apps mit .NET programmieren« beschrieben kleine Hilfsprogramme erstellt, kommt es vor, dass man dort Daten eingibt, die auch nach dem Schließen und dem erneuten Öffnen der Anwendung noch erhalten sein sollen. Wenn Du in einer solchen Anwendung beispielsweise immer wieder Daten aus dem gleichen Verzeichnis verarbeiten möchtest, willst Du das Verzeichnis nicht jedes Mal erneut auswählen. Man kann damit auch andere Daten wie Verbindungszeichenfolgen, Benutzernamen für Logins oder ganz allgemein Optionen speichern. All dies könnte man auch in eine Datenbank schreiben, aber wenn die Menge der Daten überschaubar ist, erhalten wir mit der in diesem Artikel vorgestellten Lösung eine wesentlich leichtgewichtigere Alternative.

Als Entwickler, der sich viel mit Microsoft Access beschäftigt, würde ich mir wünschen, dass diese Anwendung sich zum Beispiel beim Auswählen von Dateien für den Import das Verzeichnis merken würde, das ich beim letzten Mal verwendet habe. Leider ist das nicht der Fall, und dieses Verhalten trifft man leider immer wieder an verschiedenen Stellen an.

Wenn ich selbst Beispiele oder Lösungen programmiere, bei denen Dateien für den Import oder Export von Daten festgelegt werden müssen, baue ich daher allein aus Faulheit rechtzeitig eine Möglichkeit ein, damit ein einmal ausgewähltes Verzeichnis gespeichert und beim nächsten Öffnen der Anwendung nicht nochmals ausgewählt werden muss.

Während dies in Access logischerweise mit einer Optionentabelle geschieht, die man schnell selbst anlegt, ist dies bei einer .NET-Standalone-Anwendung wie in **Standalone-Apps mit .NET programmieren** (**www.vbentwickler.de/358**) beschrieben recht aufwendig – zumindest, wenn diese Anwendung nicht ohnehin eine Datenbank verwendet. Allerdings gibt es eine Alternative, nämlich die Datei **App.config**. Diese dient speziell zum Speichern von Konfigurationsdateien, und welche Daten man darin speichert, ist dem Entwickler selbst überlassen. Dieser Artikel zeigt, wie Du Daten aus Textfeldern beim Schließen der Anwendung speicherst und diese beim erneuten Öffnen wiederherstellst.

Dabei gehen wir wie folgt vor:

- Erstellen einer Anwendung mit Optionen
- Definieren der Einstellungen
- Erstellen einer Methode zum Speichern der Einstellung
- Erstellen einer Methode zum Wiederherstellen der Einstellung

Erstellen einer Anwendung mit Optionen

Als Erstes legen wir ein neues VB.NET-Projekt wie in **Standalone-Apps mit .NET programmieren** (**www.vbentwickler.de/358**) beschrieben an. Diese enthält Steuerelemente zum Eingeben von Daten – zum Beispiel Textfelder zur Eingabe von Texten und Zahlen, ein Kontrollkästchen und ein Auswahlfeld. Der Entwurf sieht wie in Bild 1 aus. Außerdem finden wir zwei Schaltflächen vor, mit denen wir die Daten speichern und wiederherstellen können.

Die Definition der Steuerelemente lautet in gekürzter Form wie folgt:

```
<Label>Text:</Label>
<TextBox x:Name="txtOptionText" ... />
<Label>Zahl:</Label>
<TextBox x:Name="txtOptionZahl" ... />
<Label>Ja/Nein:</Label>
<CheckBox x:Name="chkOptionJaNein" .../>
<Label>Auswahl:</Label>
<ComboBox x:Name="cboAuswahl">
    <ComboBoxItem>Wert 1</ComboBoxItem>
    <ComboBoxItem>Wert 2</ComboBoxItem>
    <ComboBoxItem>Wert 3</ComboBoxItem>
</ComboBox>
<StackPanel Orientation="Horizontal" >
    <Button x:Name="btnSpeichern"
        Click="btnSpeichern_Click">Speichern</Button>
    <Button x:Name="btnWiederherstellen"
        Click="btnWiederherstellen_Click">
        Wiederherstellen</Button>
</StackPanel>
```

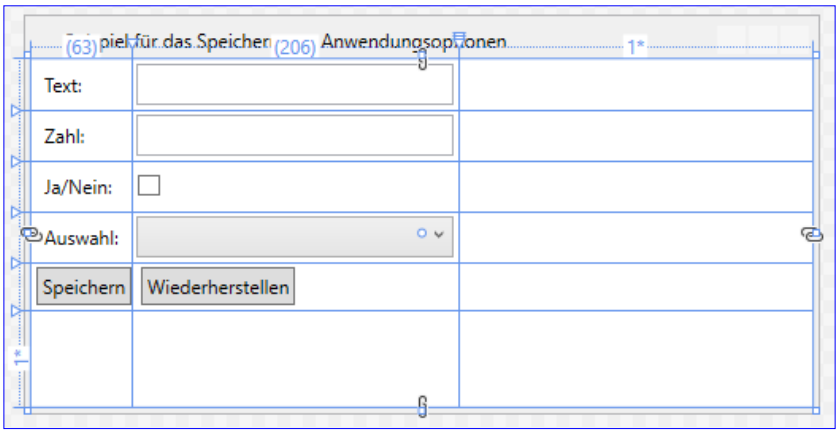


Bild 1: Fenster mit den Beispielsteuerelementen

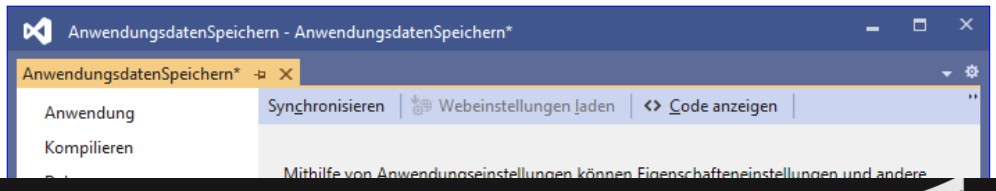
Definieren der Anwendungseinstellungen

Nun definieren wir die Einstellungen für die Werte, die wir in die Steuerelemente eingeben wollen. Dazu klicken wir im Projektmappen-Explorer doppelt auf den Eintrag **My Project**.

Dies blendet den Bereich mit den Anwendungseinstellungen ein. Für uns ist die Einstellungen sollen für den jeweiligen Benutzer gespeichert werden.

Für einige Datentypen hinterlegt der Dialog automatisch Standardwerte – zum Beispiel **0** bei der Einstellung **Zahl**.

Diese kann man beibehalten oder auch leeren. Auch kann man Standardwerte für solche Einträge hinterlegen, die in der Spalte **Wert** noch keinen Eintrag aufweisen.



AfterUpdate für WPF-Steuerelemente mit VB.NET

Wenn man einfache Anwendungen wie im Artikel »Standalone-Apps mit .NET programmieren« erstellt, stößt man relativ schnell an Grenzen. Eine davon sind die verfügbaren Ereignisse für Steuerelemente. Wer einmal mit Access gearbeitet hat, weiß, dass es für jedes Steuerelement ein Ereignis wie »Nach Aktualisierung« gibt. Unter WPF ist das nicht der Fall, was an der zugrunde liegenden Philosophie liegt. Diese lautet, dass Benutzeroberfläche und Anwendungslogik so weit wie möglich getrennt werden sollen. In diesem Artikel zeigen wir anhand eines Beispiels, wie sich dies in der Praxis auswirkt.

Im Artikel **Standalone-Apps mit .NET programmieren** (**www.vbentwickler.de/358**) haben wir grundlegend gezeigt, wie man **.exe**-Dateien zum Erledigen kleiner oder auch größerer Aufgaben mit Visual Studio programmieren kann. In einem weiteren Artikel namens **Anwendungsdaten speichern per VB.NET** (**www.vbentwickler.de/359**) haben wir darauf aufbauend eine kleine Anwendung erzeugt, mit der wir die Daten in Textfeldern und anderen Steuerelementen als Anwendungskonfigurationsdaten speichern können.

Allerdings haben wir diese immer nur komplett entweder nach einem Klick auf eine dafür vorgesehene **Speichern**-Schaltfläche oder beim Schließen der Anwendung gespeichert.

Dort haben wir auch festgestellt, dass es unter WPF nicht für alle Steuerelemente Ereignisse gibt, die nach der Aktualisierung des Inhalts ausgelöst werden. Bei einem **TextBox**-Element können wir zwar das **TextChanged**-Ereignis nutzen, das nach der Eingabe eines jeden Zeichens ausgelöst wird, aber den Inhalt nach jeder nicht durch die Eingabetaste bestätigten Änderung zu speichern, wäre doch etwas übertrieben.

Also schauen wir uns im vorliegenden Artikel an, wie man solche Aufgaben unter Berücksichtigung der Philosophie von WPF löst. Und dabei steigen wir nicht in die Tiefe ein – das würde bedeuten, das **MVVM**-Entwurfsmuster zu erläutern (**Model-View-ViewModel**).

Wir wollen nun die Daten der Steuerelemente ohne Ereignisse der Steuerelemente selbst und nur mit Datenbindung in der Konfigurationsdatei speichern und diese beim Starten der Anwendung wiederherstellen.

Elemente bei der WPF-Datenbindung

Bei der WPF-Datenbindung benötigen wir die folgenden Elemente:

- Ein Steuerelement, dessen Eigenschaft, über die der Inhalt festgelegt wird, an eine Eigenschaft des Code behind-Moduls gebunden wird.
- Eine Eigenschaft im Code behind-Modul, die ausgelesen und beschrieben werden kann.
- Die Angabe, an welche Klasse das XAML-Fenster gebunden wird und woher es seine Daten beziehen soll – in unserem Fall die Code behind-Klasse.

Nachfolgend schauen wir uns die dazu notwendigen Schritte an. Dabei gehen wir davon aus, dass Du bereits ein neues Projekt des Typs **WPF-App (.NET Framework)** erstellt hat und das Fenster **MainWindow.xaml** in der Entwurfsansicht angezeigt wird.

Außerdem kannst Du bereits einmal im Projektmappen-Explorer den Eintrag **MainWindow.xaml** erweitern, sodass Du hier die Code behind-Datei **MainWindow.xaml.vb** siehst. Diese öffnen wir per Doppelklick.

TextBox mit Datenbindung anlegen

Dem Fenster **MainWindow.xaml** fügen wir nun ein erstes Steuerelement namens **txtVorname** hinzu.

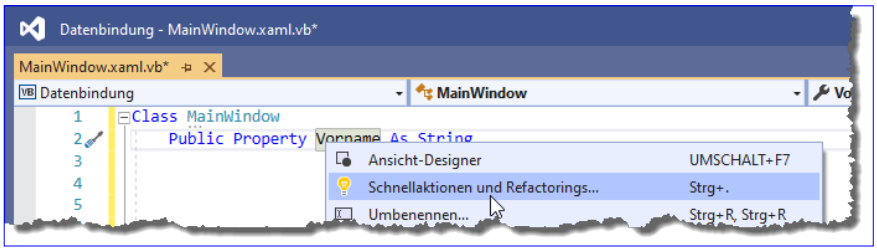


Bild 1: Aufrufen der Schnellaktionen

Im Beispielprojekt haben wir dieses in eine Zeile des Grids eingebettet – mehr dazu im Artikel **Standalone-Apps mit .NET programmieren** (**www.vbentwickler.de/358**).

Der Code für dieses Element sieht derzeit so aus:

```
<TextBox x:Name="txtVorname" Grid.Column="1"
Width="200"></TextBox>
```

Damit haben wir nun ein **TextBox**-Element, in das wir Text eintragen können – aber es geschieht nichts weiter damit.

Eigenschaft für die Datenbindung hinzufügen

Damit die eingegebenen Daten weiterverarbeitet werden können oder wir das Textfeld aus dem Code heraus füllen können, wenden wir uns nun der Code behind-Datei **MainWindow.xaml.vb** zu. Hier fügen wir nun eine öffentliche Eigenschaft hinzu, an die wir später das Steuerelement binden wollen.

Um Schreibarbeit zu sparen, legen wir die Eigenschaft zunächst wie folgt an:

```
Public Property Vorname As String
```

Dann klicken wir mit der rechten Maustaste auf diese Zeile und wählen im Kontextmenü den Eintrag **Schnellaktionen und Refactorings...** aus (siehe Bild 1).

Dies zeigt ein Popupmenü an, das ganz unten den Eintrag **In vollständige Eigenschaft konvertieren** anbietet. Fahren wir mit der Maus über diesen Eintrag, liefert dieser eine Vorschau der durch diesen Befehl angestoßenen Codeänderung (siehe Bild 2).

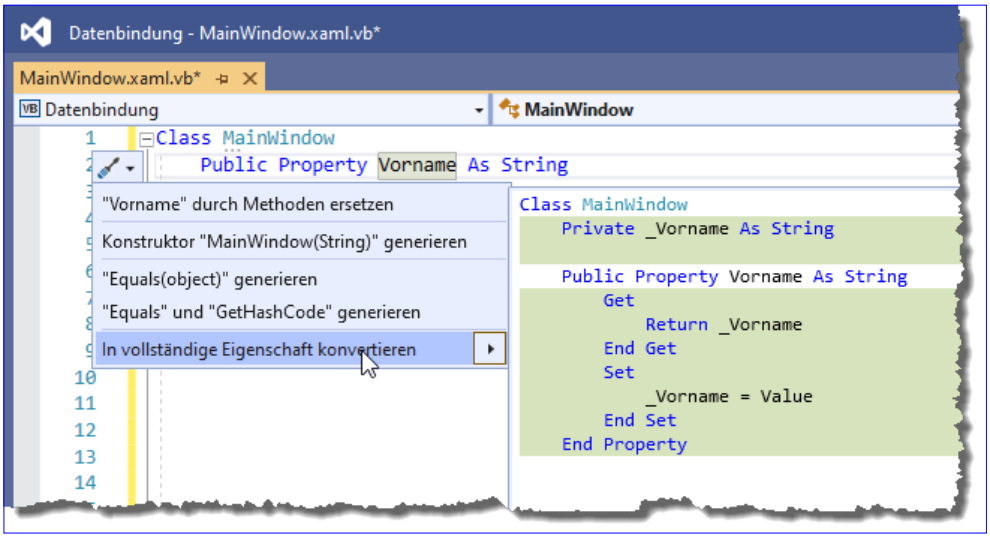


Bild 2: Anzeige der gewünschten Schnellaktion

Klicken wir diese Option an, erhalten wir das Ergebnis aus Bild 3. Was haben wir nun davon? Die Code behind-Klasse enthält nun eine Eigenschaft, die wir per Code füllen können oder an die wir unser Steuerelement **txtVorname** binden können. Letzteres führt dazu, dass bei einer Änderung des Inhalts von **txtVorname** der Inhalt über die öf-

fentliche Eigenschaft **Vorname** in die private Variable **_Vorname** geschrieben wird. Diese wiederum können wir über die öffentliche Eigenschaft auslesen.

Der Vorteil ist jedoch: Wenn wir das Steuerelement an diese Eigenschaft binden und dann den Wert des Steuerelements über die Benutzeroberfläche ändern, wird automatisch der Wert der öffentlichen Eigenschaft geändert.

Dabei durchlaufen wir in diesem Fall die Anweisungen zwischen **Set** und **End Set**. Hier können wir nun Code einfügen, der den neuen Wert im Steuerelement weiterverarbeitet – zum Beispiel, um diesen zu speichern. Doch zunächst benötigen wir die Datenbindung.

Steuerelement an Eigenschaft binden

Damit das Steuerelement an die Eigenschaft gebunden wird, sind zwei Schritte nötig. Einer davon ist, in der Definition des Steuerelements anzugeben, dass es an die öffentliche Eigenschaft gebunden werden soll. Das erledigen wir, indem wir die Definition des Steuerelements wie folgt erweitern:

```
<TextBox x:Name="txtVorname" ...
    Text="{Binding Vorname}"></TextBox>
```

Wir weisen also der Eigenschaft des Steuerelements, welches den anzuzeigenden Text enthält, den Wert

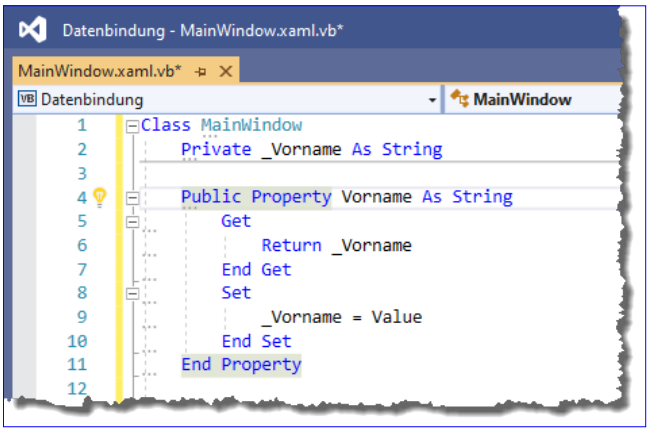
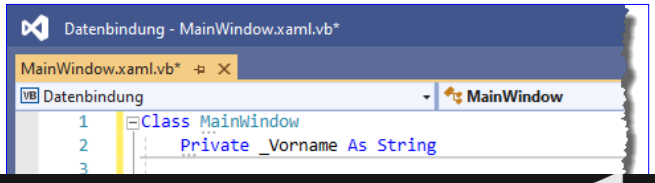


Bild 3: Vollständige öffentliche Eigenschaft

Methoden aus auch Parameter übergeben, aber das ist an dieser Stelle nicht nötig.

Um die Konstruktor-Methode anzulegen, klicken wir mit der rechten Maustaste in einen leeren Bereich im Code behind-Modul und wählen dort den Eintrag **Konstruktor generieren...** aus (siehe Bild 4).

Es erscheint der Dialog aus Bild 5, der einige Member anzeigt, die aktuell noch ausgewählt sind. Bleiben diese ausgewählt, werden entsprechende Parameter zur Konstruktor-Methode hinzugefügt und den jeweiligen



QR-Codes per .NET-Anwendung erstellen

QR-Codes sind praktische Helfer in der heutigen Zeit der Smartphones und ihren Kameras. Man kann damit wichtige Informationen so codieren, dass man diese mit der Kamera erkennen und weiterverarbeiten kann – standardmäßig so, dass der Code eine URL zu einer Webseite erhält und man diese dann im Browser des Smartphones aufrufen und die Inhalte konsumieren kann. Dieser Artikel zeigt, wie Du QR-Codes mit .NET-Anwendung erstellen kannst. Dreh- und Angelpunkt ist dabei ein NuGet-Paket, das die Funktionen für die Erstellung von QR-Codes auf Basis verschiedener Informationen bereitstellt. Der Artikel zeigt, wie wir dieses Paket in ein .NET-Projekt einbinden und es dann nutzen können.

QR-Codes und ihre Anwendung

Kennst Du eigentlich die Bedeutung der Abkürzung QR? Ich habe sie lustigerweise zum ersten Mal bewusst wahrgenommen, als ich für diesen Artikel recherchiert habe. QR steht für Quick Response.

Für QR-Codes gibt es mittlerweile sehr viele Anwendungsfälle. Hier sind nur einige von ihnen:

- Produktinformationen: QR-Codes können auf Produktverpackungen platziert werden, um Kunden schnell und einfach auf weitere Informationen wie Produktbeschreibungen, Kundenbewertungen oder Anleitungen zu verweisen.
- Marketing und Werbung: QR-Codes können in Marketing- und Werbekampagnen eingesetzt werden, um Kunden auf spezielle Angebote, Wettbewerbe oder Veranstaltungen aufmerksam zu machen. Sie können auch verwendet werden, um Kunden auf eine Landingpage oder eine mobile App zu leiten.
- Eventmanagement: QR-Codes können auf Eintrittskarten oder Veranstaltungsbroschüren platziert werden, um Besuchern den Zugang zu zusätzlichen Informationen, Updates und Planungstools zu ermöglichen.
- Mobile Zahlungen: QR-Codes können als Zahlungsoption für mobile Zahlungsanwendungen verschiedener Banken verwendet werden.
- Identifikation und Authentifizierung: QR-Codes können für die Identifikation von Personen oder als Bestätigung für den Zugang zu bestimmten Orten oder Veranstaltungen verwendet werden.
- Gesundheitswesen: QR-Codes können für die schnelle Übertragung von Patientendaten und medizinischen Informationen verwendet werden.
- Tourismus und Reisen: QR-Codes können für die Bereitstellung von Informationen zu touristischen Attraktionen, Transportmöglichkeiten und Unterkünften verwendet werden.

QR-Codes in .NET

Im Beispiel wollen wir eine WPF-Anwendung nutzen, um Texte eingeben und daraus QR-Codes zu erzeugen.

Dazu erstellen wir eine neue Anwendung auf Basis der Vorlage **WPF-App (.NET Framework)** – siehe Bild 1.

Nach dem Erstellen des Projekts erscheint das Hauptfenster von Visual Studio .NET und zeigt die WPF-Seite im Entwurf und mit dem Beschreibungscode im

XAML-Format an. Diese Seite wird nach dem Starten der Anwendung angezeigt. Deshalb platzieren wir hier die Steuerelemente für die Daten, auf deren Basis der QR-Code erzeugt wird.

Hier passen wir den Code im unteren Bereich so an, dass wir später den Entwurf aus Bild 2 erhalten.

Dazu verwenden wir den Code aus Listing 1. Im oberen Teil passen wir gegenüber der Vorlage die Höhe und die Breite des **Window**-Elements so an, dass die Eigenschaften **Height** und **Width** jeweils den Wert **550** erhalten.

Für das darunter liegende **Grid**-Element legen wir ein Raster fest, das aus drei Spalten und vier Zeilen besteht.

Das erledigen wir für die Spalten mit dem Element **Grid.ColumnDefinitions**, dem wir drei **ColumnDefinition**-Elemente unterordnen, die jeweils die Breite der Spalten angeben – die ersten beiden sollen an die enthaltenen Steuerelemente angepasst werden (**Width="Auto"**),

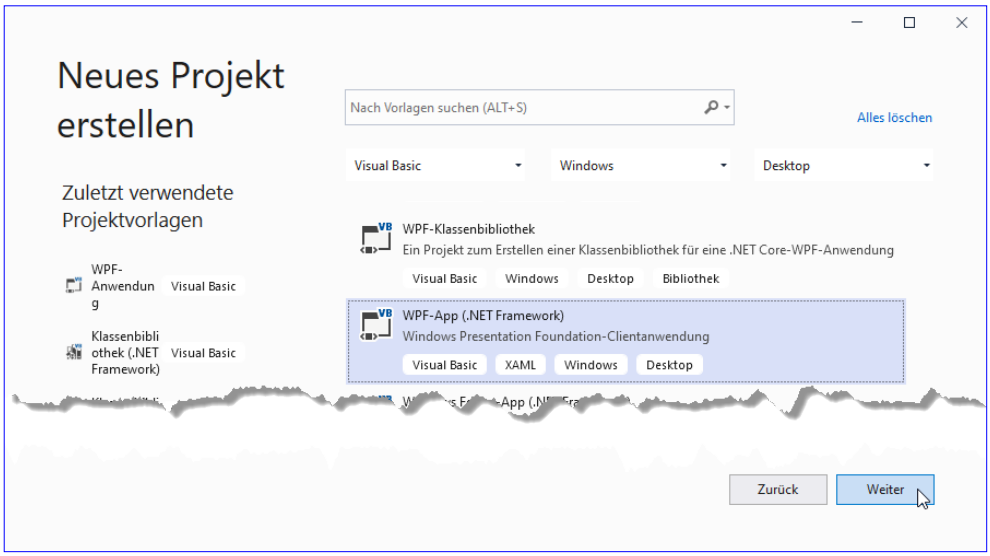


Bild 1: Erstellen eines WPF-Projekts

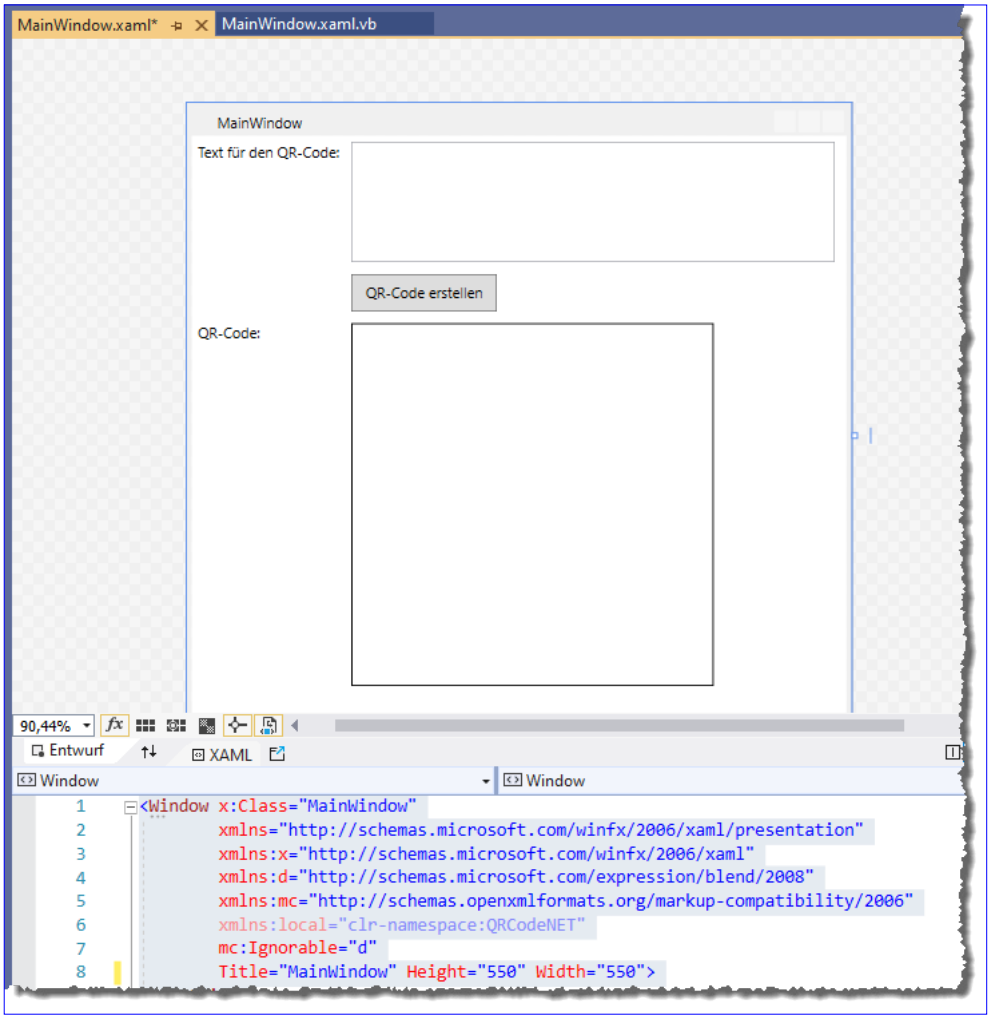


Bild 2: Unsere angehende Lösung im WPF-Entwurf

```
<Window x:Class="MainWindow" ... Title="MainWindow" Height="550" Width="550">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition Width="*"></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="*"></RowDefinition>
        </Grid.RowDefinitions>
        <Label>Text für den QR-Code:</Label>
        <TextBox x:Name="txtQRCode" Grid.Column="1" Width="400" Height="100" Margin="5" TextWrapping="Wrap" ></TextBox>
        <Button x:Name="btnQRCodeErstellen" Grid.Column="1" Grid.Row="1" Width="120" Height="30" Margin="5"
            HorizontalAlignment="Left">QR-Code erstellen</Button>
        <Label Grid.Row="2">QR-Code:</Label>
        <Border BorderThickness="1" BorderBrush="DarkGray" Grid.Column="1" Grid.Row="2" Margin="5" Width="300"
                Height="300" HorizontalAlignment="Left">
            <Image x:Name="imgQRCode" Margin="5" Stretch="Fill"></Image>
        </Border>
    </Grid>
</Window>
```

Listing 1: Code für die Gestaltung des Fensters und der enthaltenen Steuerelemente

die letzte die restliche Breite im Fenster einnehmen (**Width="*"**).

Für die Zeilen verfahren wir analog und legen im **Grid.RowDefinitions** vier **RowDefinition**-Elemente fest.

Diese Zeilen und Spalten geben wir mit den Eigenschaften **Grid.Column** und **Grid.Row** der nachfolgend hinzugefügten Steuerelemente so an, dass festgelegt ist, welches Steuerelement in welcher Zelle landet.

Für das erste **Label**-Element fehlen diese beiden Eigenschaften, da wir hier den Standardwert **0** verwenden, damit das Bezeichnungsfeld in der linken, oberen Spalte landet. Die nachfolgend definierte **TextBox** soll in der zweiten Spalte der ersten Zeile landen, daher stellen wir hier **Grid.Column** auf den Wert **1** ein (der Index für die Zeilen und Spalten ist 0-basiert). Für die Textbox stellen wir eine entsprechende Höhe und Breite und einen Abstand zum umgebenden Grid ein. Außerdem legen wir mit **TextWrapping="Wrap"** fest, dass wir mehrzeilige Texte eingeben können. Der Name des Textfeldes ist **txtQRCodeErstellen**.

Die nächste Zeile enthält in der zweiten Spalte eine Schaltfläche namens **btnQRCodeErstellen**. Die dadurch aufzurufende Prozedur definieren wir gleich.

Darunter zeigen wir ein **Image**-Steuerelement an, welches den erzeugten QR-Code schließlich anzeigen soll. Dieses Element nennen wir **imgQRCode**.

Damit man es überhaupt sieht, haben wir um dieses Element herum noch ein **Border**-Element mit der Dicke **1** und der Farbe **DarkGray** gelegt.

Ereignisprozedur hinzufügen

Damit wir beim Anklicken der Schaltfläche eine Prozedur ausführen können, stellen wir für diese das Attribut **Click** ein.

Bild 3: Anlegen des Ereignishandlers

Um die Prozedur schnellstmöglich anzulegen und bearbeiten zu können, geben wir nach dem Attribut **Click** ein Gleichheitszeichen ein, woraufhin der IntelliSense-Eintrag **<Neuer Ereignishandler>** auftaucht (siehe Bild 3). Betätigen wir nun die Tabulatur-Taste, wird automatisch der Name der Ereignisprozedur eingetragen:

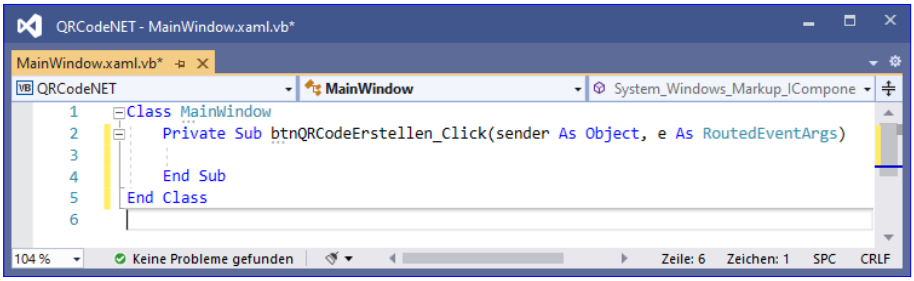


Bild 4: Der neue Ereignishandler

```
Click="btnQRCodeErstellen_Click"
```

Außerdem finden wir durch das anschließende Betätigen der Taste **F12** (während sich die Einfügemarke auf dem Attribut befindet) die neu angelegte Prozedur im Codefenster des Klassenmoduls des WPF-Fensters vor (siehe Bild 4).

Um ein solches NuGet-Paket zu einem Projekt hinzuzufügen, brauchen wir nur den passenden Manager über den Menübefehl **Projekt|NuGet-Pakete verwalten...** aufzurufen.

Daraufhin erschient der Bereich zur Verwaltung der NuGet-Pakete. Dass wir dem aktuellen Projekt noch keine solchen hinzugefügt haben, sehen wir im leeren Bereich unter dem Registerreiter **Installiert**.

PDFs aufteilen und zusammenfügen mit .NET

Gelegentlich möchte man PDF-Dateien bearbeiten – und das am besten ohne eigenes Zutun, sondern komplett programmgesteuert. Ein Kunde ist beispielsweise an mich herangetreten und wollte ein riesiges PDF-Dokument, das eine Menge Rechnungen in einer einzigen Datei enthielt, aufteilen. Als Ergebnis wünscht er sich einzelne Dateien, die jeweils eine Rechnung enthielten. Grund genug, einmal zu schauen, wie man mit .NET an dieses Problem herangehen kann. Genügend NuGet-Pakete mit Funktionen zum Bearbeiten von PDF-Dokumenten sind verfügbar, sodass wir die Qual der Wahl haben. Für diese Aufgabe haben wir das NuGet-Paket PDFsharp herangezogen, das verspricht, mit PDF-Seiten umgehen zu können – egal, ob es um das Entfernen, Hinzufügen, Extrahieren oder Zusammenstellen neuer Dokumente geht.

Anforderungen an ein NuGet-Paket zum Analysieren und Bearbeiten von PDF-Dokumenten

Speziell auf die oben angegebene Aufgabenstellung bezogen können wir zwei Anforderungen definieren:

- Wir benötigen eine Funktion, mit der wir ein PDF-Dokument seitenweise durchlaufen und im Text der Seiten nach bestimmten Ausdrücken suchen können. Diese Aufgabe erledigt PDFsharp nicht zufriedenstellend, daher schauen wir uns dazu in einem weiteren Artikel ein besser geeignetes Paket an.
- Außerdem benötigen wir eine Funktion, mit der wir einzelne Seiten eines PDFs-Dokuments extrahieren und als eigenständige Dokumente speichern können.

Beides würde theoretisch auch Adobe Acrobat Pro bieten, aber dies ist kostenpflichtig und wir sollen sehen, ob wir auch mit kostenloser Software weiterkommen.

NuGet-Paket suchen

Also starten wir Visual Studio und legen ein neues Projekt namens **PDFTools** an. Das Projekt soll den Typ **WPF-App (.NET Framework)** erhalten (siehe Bild 1), damit wir die Funktionen über eine Benutzeroberfläche steuern können – beispielsweise, um den Namen der zu bearbeitenden PDF-Datei anzugeben und die gewünschten Funktionen aufzurufen.

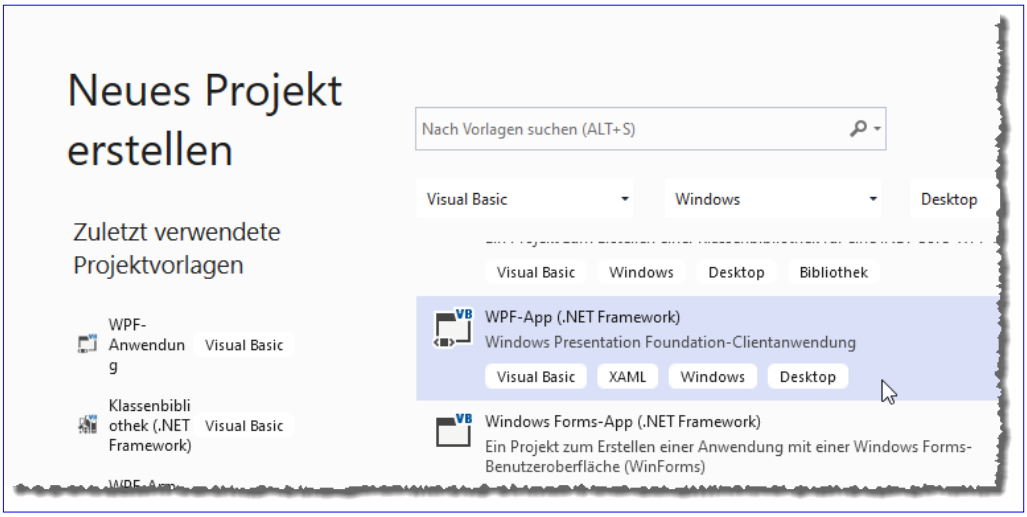


Bild 1: Anlegen einer neuen WPF-Anwendung

Nachdem das Projekt angelegt und geöffnet wurde, rufen wir den Menübefehl **Projekt|NuGet-Pakete verwalten...** auf. Dies öffnet das Fenster **NuGet: PDFTools**, wo wir zum Bereich **Durchsuchen** wechseln.

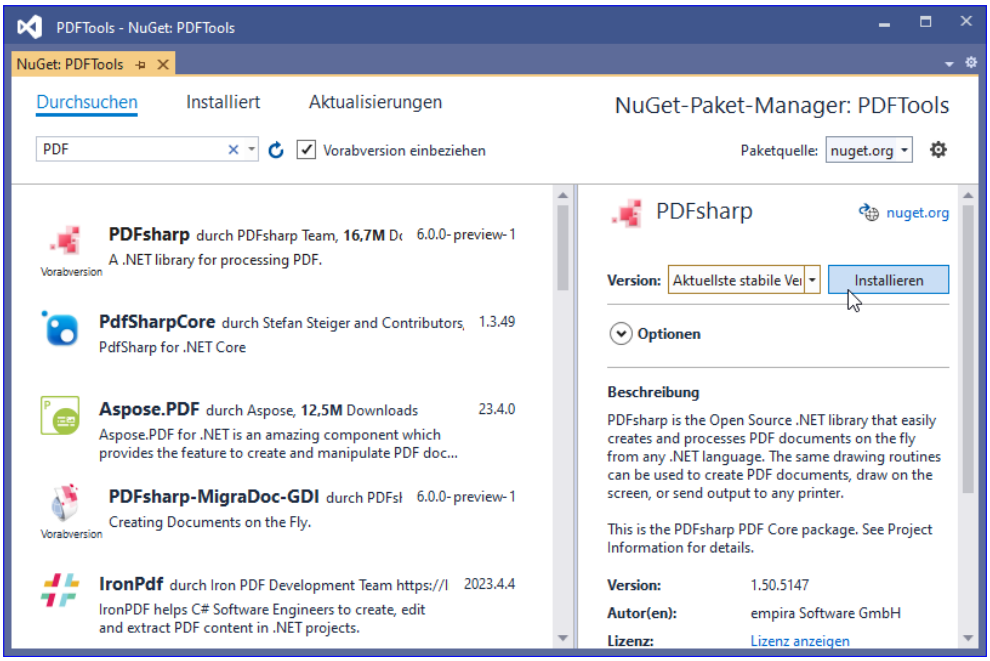


Bild 2: Hinzufügen einer PDF-Bibliothek

Hier geben wir einfach einmal den Suchbegriff **PDF** ein und erhalten direkt einige Treffer. Direkt der erste ist recht bekannt und heißt **PDFsharp**. Interessant ist, dass es unter der MIT-Lizenz läuft. Die Bedingungen kannst Du mit einem Klick auf **Lizenz anzeigen** einsehen.

Wir wählen **PDFsharp** in der Version **Aktuellste stabile Version** aus und klicken auf **Installieren** (siehe Bild 2). Damit landen die benötigten Dateien im Projektverzeichnis und werden als Verweise eingebunden.

Funktion zum Auswählen des PDF-Dokuments hinzufügen

Bevor wir uns mit **PDFsharp** beschäftigen, fügen wir der Startseite **MainWindow.xaml** schon einmal ein Textfeld zum Anzeigen und eine Schaltfläche zum Auswählen der zu untersuchenden und gegebenenfalls zu bearbeitenden PDF-Datei hinzu. Dazu definieren wir die Steuerelemente wie folgt (für den kompletten Code siehe Beispielprojekt im Download):

```
<Label>PDF-Dokument:</Label>
<StackPanel Orientation="Horizontal" Grid.Column="1">
    <TextBox x:Name="txtPDFDokument"
        Width="300"></TextBox>
    <Button x:Name="btnDateiauswahl"
        Click="btnDateiauswahl_Click">Auswählen</Button>
</StackPanel>
```

Für die Schaltfläche **btnDateiauswahl** hinterlegen wir die Prozedur aus Listing 1 in der Code behind-Datei **MainWindow.xaml.vb**. Diese Prozedur ruft einen Dialog zur Auswahl der zu bearbeitenden PDF-Datei auf.

Zum Ausprobieren klicken wir auf **F5** und sehen, dass alles wie gewünscht funktioniert.

Seitenzahl eines PDF-Dokuments ermitteln

Zum Warmwerden wollen wir die Seitenzahl des ausgewählten PDF-Dokuments ermitteln. Dazu fügen wir eine neue Schaltfläche zum Code hinzu:

```
<Button x:Name="btnSeitenzahl" Grid.Row="1"
    Click="btnSeitenzahl_Click">Seitenzahl</Button>
```

Die dadurch ausgelöste Prozedur beantwortet gleich zwei Fragen: Wie kann ich überhaupt ein PDF-Doku-

```
Imports Microsoft.Win32

Class MainWindow
    Private Sub btnDateiauswahl_Click(sender As Object, e As RoutedEventArgs)
        Dim objFileDialog As New OpenFileDialog()
        objFileDialog.Filter = "PDF-Dateien (*.pdf)|*.pdf|Alle Dateien (*.*)|*.*"
        objFileDialog.FilterIndex = 1
        objFileDialog.RestoreDirectory = True
        If objFileDialog.ShowDialog() = True Then
            txtPDFDokument.Text = objFileDialog.FileName
        End If
    End Sub
End Class
```

Listing 1: Öffnen eines Dateiauswahl-Dialogs

ment für die Analyse und Bearbeitung mit **PDFsharp** öffnen und wie finde ich die Seitenzahl des geöffneten Dokuments heraus?

Bevor wir damit starten, fügen wir der Klasse **MainWindow.xaml.vb** die folgenden weiteren beiden **Namespace**-Verweise hinzu:

```
Imports PdfSharp.Pdf
Imports PdfSharp.Pdf.IO
```

Danach schauen wir uns die Prozedur aus Listing 2 an. Die Prozedur deklariert drei Variablen:

- **strPfad**: Nimmt dem Pfad aus dem Textfeld entgegen.
- **objDocument**: Speichert einen Verweis auf das PDF-Dokument und hat den Typ **PdfDocument**.
- **intSeitenzahl**: Nimmt die Anzahl der Seiten des PDF-Dokuments auf.

Das Öffnen des Dokuments erfolgt zu diesem Zweck mit der Methode **Open** und dem Parameter **ReadOnly** – wir wollen schließlich erst einmal nur lesend auf die Datei zugreifen. Danach fragen wir mit der **PageCount**-Methode die Seitenzahl des Dokuments ab und geben diese in einer Meldung aus.

Alle Seiten in eine eigene Datei speichern

Bevor wir die PDF-Datei nach bestimmten Inhalten durchsuchen und damit bestimmen, an welchen

```
Private Sub btnSeitenzahl_Click(sender As Object, e As RoutedEventArgs)
    Dim strPfad As String
    Dim objDocument As PdfDocument
    Dim intSeitenzahl As Integer
    strPfad = txtPDFDokument.Text
    objDocument = PdfReader.Open(strPfad, PdfDocumentOpenMode.ReadOnly)
    intSeitenzahl = objDocument.PageCount
    MsgBox("Anzahl der Seiten: " & intSeitenzahl.ToString)
    objDocument.Close()
End Sub
```

Listing 2: Öffnen eines PDF-Dokuments und Auslesen der Seitenzahl

```
Private Sub btnSeitenExtrahieren_Click(sender As Object, e As RoutedEventArgs)
    Dim strPfad As String
    Dim objDocument As PdfDocument
    Dim intSeitenzahl As Integer
    Dim objDocumentSeite As PdfDocument
    Dim intSeite As Integer
    Dim strVerzeichnis As String
    strPfad = txtPDFDokument.Text
    strVerzeichnis = strPfad.Substring(0, InStrRev(strPfad, "\"))
    objDocument = PdfReader.Open(strPfad, PdfDocumentOpenMode.Import)
    intSeitenzahl = objDocument.PageCount
    For intSeite = 0 To intSeitenzahl - 1
        objDocumentSeite = New PdfDocument()
        objDocumentSeite.AddPage(objDocument.Pages(intSeite))
        objDocumentSeite.Save(strVerzeichnis & "Seite_" & (intSeite + 1).ToString() & ".pdf")
        objDocumentSeite.Close()
    Next
    objDocument.Close()
End Sub
```

Listing 3: Extrahieren jeder einzelnen Seite eines Dokuments in ein neues Dokument

Stellen jeweils ein neues Dokument beginnt, wollen wir erst einmal die Technik zum Speichern einzelner Seiten dokumentieren. Dazu durchlaufen wir einfach alle Seiten und speichern diese jeweils in einer eigenen Datei.

Die Prozedur hinterlegen wir für eine weitere neue Schaltfläche namens **btnSeitenExtrahieren** (siehe Listing 3). Die Prozedur deklariert neben den bereits aus dem vorherigen Beispiel bekannten Variablen noch zwei neue Variablen:

- **objDocumentSeite**: Speichert die Referenz auf die jeweils neu zu erstellenden Dokumente.
- **intSeite**: Laufvariable zum Durchlaufen der Seiten.
- **strVerzeichnis**: Speichert das Basisverzeichnis, in dem die neuen Seiten angelegt werden sollen.

Die Prozedur ermittelt zuerst das Verzeichnis aus **strPfad**, also ohne den Dateinamen des PDF-Dokuments, welches wir in seine einzelnen Seiten zerlegen wollen. Dazu nutzen wir die **Substring**-Methode der Variablen **strPfad** und holen uns damit den Teil bis zum ersten Backslash (\) von hinten aus gesehen.

Dann öffnen wir das Quelldokument, was wir diesmal mit dem zweiten Parameter **PdfDocumentOpenMode.Import** erledigen. Wieder lesen wir die Seitenzahl in die Variable **intSeitenzahl** ein. Diese nutzen wir, um in einer **For Next**-Schleife alle Seiten zu durchlaufen. Der Index der **Pages**-Auflistung, über die wir auf die Seiten zugreifen wollen, ist 0-basiert, daher verwenden wir **0** als Startwert und **intSeitenzahl - 1** als Endwert der Schleife.

Innerhalb der Schleife legen wir mit der **New**-Anweisung jeweils ein neues Objekt des Typs **PdfDocument** an und referenzieren es mit der Variablen **objDocumentSeite**.

Diesem fügen wir mit der **AddPage**-Methode eine neue Seite hinzu. Der Parameter dieser Methode ent-

hält den Verweis auf die jeweilige Seite des Quelldokuments. Anschließend speichern wir das Dokument mit der **Save**-Methode. Den Speicherpfad setzen wir aus dem Verzeichnis aus **strVerzeichnis** und einem Namen zusammen, der aus **Seite_**, dem Wert der Variablen **intSeite + 1** und der Dateiendung **.pdf** besteht, also beispielsweise **Seite_1.pdf**.

Als Beispiel haben wir eine Ausgabe von **Visual Basic Entwickler** verwendet, das Ergebnis sieht ausschnittsweise wie in Bild 3 aus.

Kriterien zum Aufteilen finden

Nun wollen wir allerdings nicht einfach alle Seiten extrahieren (auch wenn es sein kann, dass es dafür Anwendungsfälle gibt), sondern wir wollen bestimmte Stellen ermitteln. In unserem Beispiel wollen wir die einzelnen Artikel einer Ausgabe in jeweils einer eigenen PDF-Datei speichern.

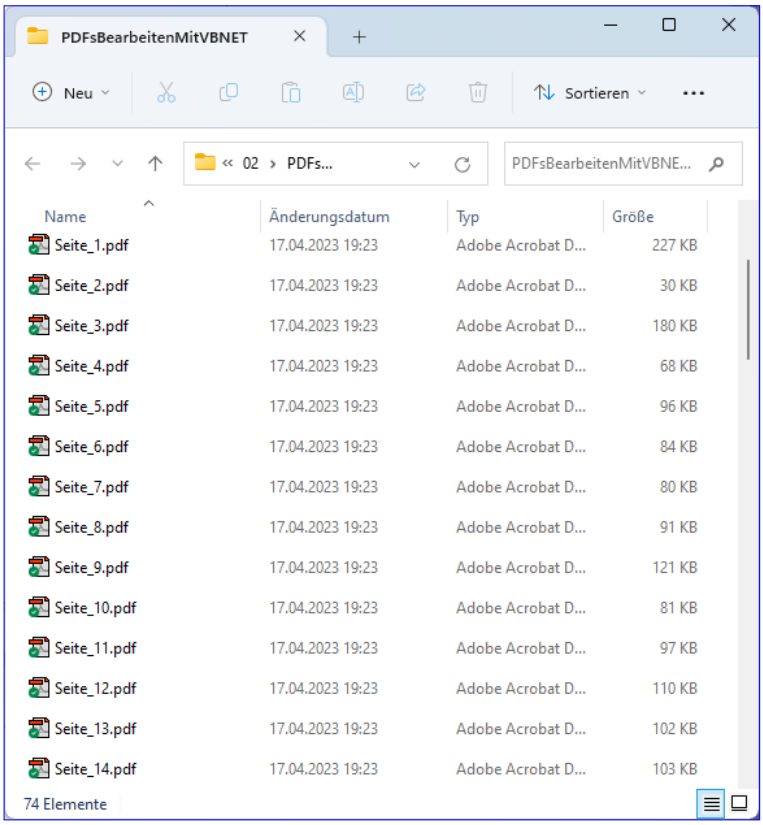


Bild 3: Exportierte Seiten