TABELLEN MIT .NET ERSTELLEN

Von Excel zu FlexCel

Im Büroalltag gilt Excel als Schweizer Taschenmesser. Die Bibliothek TMS FlexCel befähigt eigene Anwendungen, mit Excel-Tabellen zu arbeiten.

Excel und Tabellen, das gehört zusammen. Dabei gilt die Tabellenkalkulation aus dem Hause Microsoft nicht nur unter den Mitarbeitern aus der Controlling-Abteilung zu den Standardwerkzeugen. Fast jeder Anwender hat auf seinem Rechner eine Tabelle im Excel-Format, und wenn es nur die Übersicht mit den aktuellen Kontaktdaten ist. Excel ist neben Word, PowerPoint und Outlook Bestandteil der Office-Suite von Microsoft. Bedienung und Funktionsweise gelten als allgemein bekannt und für allerlei Aufgaben als nützlich. Lizenziert wird Excel heute über die Microsoft-365-Suite, die neben den klassischen Office-Anwendungen auch den Zugriff auf die Videokonferenzsoftware Microsoft Teams und den Cloud Speicherdienst OneDrive enthält. Es gibt unterschiedliche Lizenzierungsmodelle für kleinere und mittlere Unternehmen, für Großunternehmen, für die private Nutzung und für den Bildungssektor.

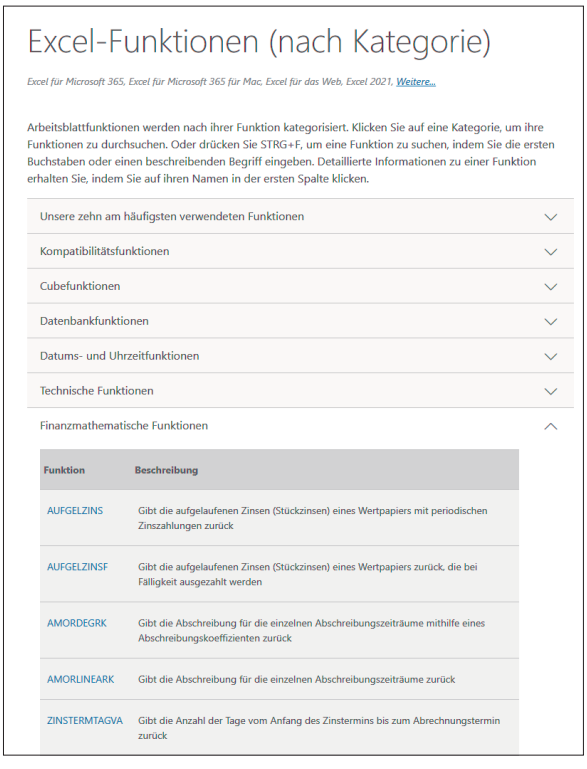
Die Funktionsvielfalt in Excel ist inzwischen sehr groß, sodass es sich für sehr unterschiedliche Aufgaben verwenden lässt. Es unterstützt mittlerweile leistungsfähige Funktionen aus den unterschiedlichsten Bereichen [1], so zum Beispiel aus der Finanzmathematik oder der Statistik (Bild 1). Dabei stellt es nicht nur Funktionen aus dem jeweiligen Grundlagenbereich bereit, sondern auch fortschrittliche und komplexe Berechnungen, Datenauswertungen, Visualisierungen und so weiter sind damit möglich.

Diese Funktionsvielfalt und die weite Verbreitung von Excel führen dazu, dass die Anwenderinnen und Anwender mit dem Tool gut vertraut sind und es sich als Quasistandard für Tabellenkalkulationssoftware etabliert hat. Das gilt inzwischen nicht nur für das Betriebssystem Windows. Excel steht als vollwertige Desktop-Anwendung auch auf macOS zur Verfügung. Für die Nutzung im Web gibt es eine Online-Version und für Android und iOS ebenfalls entsprechende Apps.

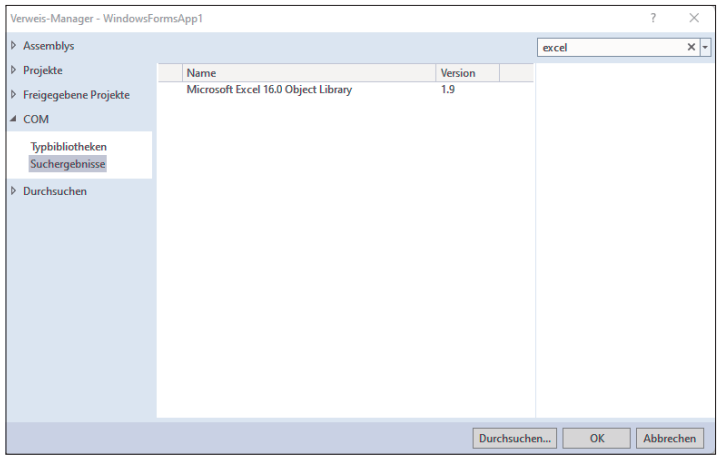
Kommen alternative Softwareprodukte zum Einsatz, zum Beispiel die Tabellenkalkulation Calc aus der Suite Libre Office, dann gibt es inzwischen sehr gut funktionierende Import- und Export-Datenfilter, um auch komplexe Dateien inklusive Berechnungen, Grafiken et cetera auszutauschen.

Warum erzählen wir Ihnen das alles? Arbeitet man an einer eigenen datenorientierten Applikation, dann müssen häufig Daten mit anderen Anwendungen ausgetauscht werden. Typische Anwendungsfälle und Anforderungen sind:

- **Import von Daten:** Daten, die in der eigenen Anwendung verarbeitet werden sollen, liegen in Form einer Excel-Datei vor. Auch aus Drittprogrammen gibt es sehr häufig die Möglichkeit, die betreffenden Daten in das Excel-Format zu exportieren und in Excel aufzubereiten.



Die Excel-Funktionen, nach Kategorien zusammengefasst (Bild 1)



Die Interop-Assembly für die Office-Automation, die in .NET-Anwendungen eingebunden werden kann (Bild 2)

- **Export von Daten:** Daten aus der Geschäftsanwendung sollen im Excel-Format exportiert werden. Dann können sie gemäß den Anforderungen weiterverarbeitet, grafisch aufbereitet oder anderweitig verwendet werden.
- **Erstellen von Tabellen und Berichten:** Die eigene Applikation soll in der Lage sein, Excel-Dateien zu erzeugen, die unterschiedliche Funktionen enthalten. Dabei sollen nicht nur Datenwerte in die Zellen geschrieben werden, sondern diese werden durch Formeln und Bezüge untereinander verknüpft. Interaktive Grafiken dienen der Visualisierung oder es werden druckfähige Berichte erzeugt.

Der letzte Punkt beschreibt eine typische Anforderung aus der Praxis für Geschäftsanwendungen. Excel soll dabei nicht nur als Datenaustauschformat verwendet werden, sondern man möchte aus der Applikation mithilfe der Daten komplexe Excel-Dateien programmgesteuert generieren. Es muss also möglich sein, Funktionen, Zellbezüge, Berechnungen, Grafiken und Zellformatierungen über den Quellcode in der Anwendung im Excel-Dateiformat zu erstellen.

Tabelle 1: Technische Eigenschaften und Grundfunktionen von FlexCel

Technische Eigenschaften, Systemvoraussetzungen	Wichtige Funktionen
Native .NET-Komponenten, keine Microsoft-Office-Installation und zusätzliche Bibliotheken erforderlich.	Lesen und Schreiben von XLS- und XLSX-Dateien, einschließlich verschlüsselter Dateien.
Unterstützt alle Excel-Dateiformate, das heißt Excel 2 bis 2021.	Programmatisches Erstellen von Excel-Dateien mit Werten, Zellbezügen, Formatierungen, Bildern, Kommentaren et cetera.
In .NET- und .NET-Core-Anwendungen einzusetzen.	Neuberechnung von Formeln mit Unterstützung von über 300 Excel-Funktionen.
Anwendungen für Windows (Win Forms, WPF, UWP, WinUI) und in Xamarin für iOS und Android.	Export von nativen PDF-Dateien, standardkonformen HTML-Dateien und SVG-Dateien.
Quellcode in C# geschrieben, 100 Prozent Managed Code.	Berichts-Engine, mit der sich komplexe Berichte mit Excel als Berichtsdesigner erstellen lassen; der Anwender kann diese anpassen.
Verwendung auf Client und Server.	Effizientes und zügiges Erstellen von Excel-Dateien auch bei großen Datentabellen und Batch-Verarbeitung.
Umfassende Online-Dokumentation mit API-Beschreibung und Beispielen.	Tool API-Mate, um bestehende Excel-Dokumente in C#/VB.NET-Quelltext umzuwandeln.

Office-Automatisierung

Das Lesen und Schreiben des Excel-Dateiformats ist keine neue Anforderung und lässt sich auf unterschiedliche Art und Weise angehen. Zum einem ist danach zu unterscheiden, ob für das Lesen und Schreiben eine lizenzierte Office-Installation auf dem Rechner vorausgesetzt werden kann oder ob es auch ohne lokale Excel-Installation funktioniert.

Ist Excel auf dem betreffenden Rechner eingerichtet, dann besteht die Möglichkeit, das API von Microsoft Office im eigenen Programm zu verwenden. Dieses Vorgehen wird unter dem Stichwort .NET-Interoperabilität beschrieben. Konkret handelt es sich dabei um die Verwendung von Office-Interop-Objekten [2].

Um in einer .NET-Anwendung auf diese Weise mit Office-Applikationen zu arbeiten, ist es notwendig, die zugehörige Assembly in das Projekt als Verweis aufzunehmen (Bild 2). Über diese Schnittstelle können Sie dann Excel „fernsteuern“, also zum Beispiel eine Datei erstellen und Daten in diese schreiben oder aus ihr lesen. Dieses Vorgehen hat jedoch einige Nachteile:

- **Office-Software:** Es ist eine Installation der Office-Programme auf dem betreffenden Rechner notwendig. Sie müssen also prüfen und letztendlich voraussetzen, dass auf dem Computer des Kunden Excel eingerichtet ist, ansonsten funktioniert dieses Vorgehen nicht.
- **.NET Framework:** Die Kommunikation erfolgt über die .NET-Interoperabilität, das heißt, es geht nur mit .NET-Applikationen. Der Einsatz von .NET Core und anderen Betriebssystemen ist über diese Schnittstelle nicht möglich.
- **Beschränkung auf den Client:** Das Szenario ist für eine Nutzung auf dem Client vorgesehen, das heißt, Excel und die Anwendung zum Fernsteuern müssen auf dem Desktop-Rechner lokal vorhanden sein. Die Automatisierung von Office auf einem Server ist stark eingeschränkt.
- **Laufzeitverhalten:** Die Interaktion über den Weg der Office-Automatisierung ist gerade bei größeren Datenmengen und Dateien nicht sehr gut.

Die beiden größten Einschränkungen sind die Notwendigkeit einer Office-Installation (Lizenz) und dass die Office-Automatisierung nicht für einen Einsatz auf einem Server gedacht ist. Läuft das Softwaresystem aber auf dem Server, dann lassen sich über Office-Interop nur eingeschränkt Excel-Dokumente erzeugen. Zu diesen Einschränkungen gibt Microsoft auch Hinweise. Dort heißt es: „Microsoft empfiehlt Entwicklern dringend, Alternativen zur Automatisierung von Office zu suchen, wenn sie serverseitige Lösungen entwickeln müssen.“ [3]

Im Übrigen stehen diesem Vorhaben auch lizenzrechtliche Probleme entgegen, denn Office darf auf dem Server nur eingesetzt werden, wenn die Clients selbst über lizenzierte Versionen der jeweiligen Office-Produkte verfügen.

Zwischenfazit: Es ist eine andere Lösung nötig. Diese besteht üblicherweise darin, eine alternative Bibliothek für die Arbeit mit Office-Dateien und in diesem Fall mit Excel zu verwenden. ▶

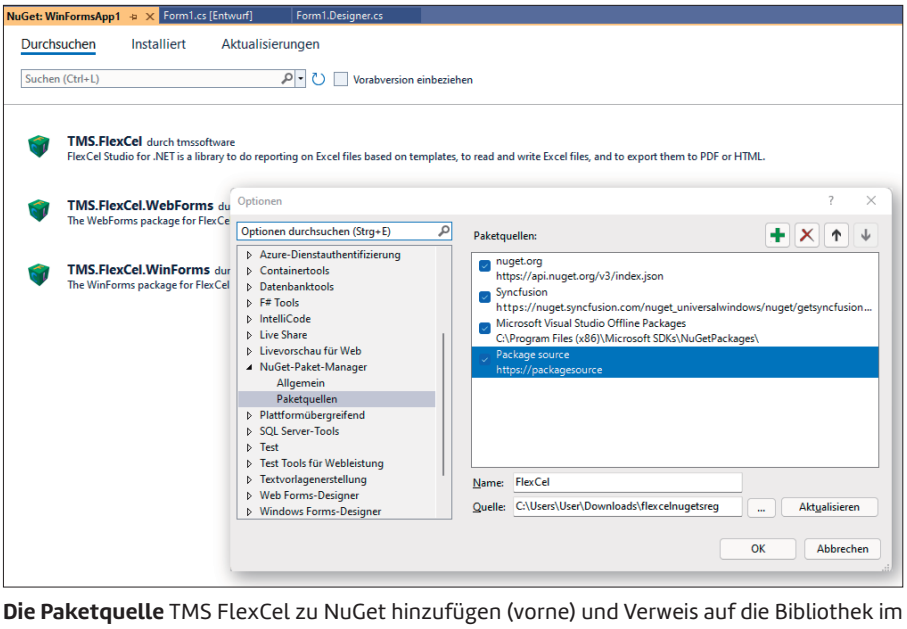
TMS FlexCel Studio für Delphi

Die Bibliothek TMS FlexCel Studio gibt es nicht nur für .NET, sondern auch für Delphi [16], konkret für die User-Interface-Frameworks Visual Component Library (VCL) für Windows-Betriebssysteme und FireMonkey (FMX) für eine plattformübergreifende Entwicklung. In diesem Fall wird mit der Programmiersprache Delphi (Object Pascal) oder C++ programmiert, als integrierte Entwicklungsumgebung kommt Delphi zum Einsatz. Funktionen und Vorgehensweise dieser Version von TMS FlexCel Studio sind mit der .NET-Bibliothek identisch.

Nutzung von Bibliotheken

Hier gibt es einige Optionen auf dem Markt, die einen unterschiedlichen Funktionsumfang aufweisen, für verschiedene Programmiersprachen geeignet sind und anderen Lizenzbedingungen unterliegen. Für den Einsatz zusammen mit .NET gibt es beispielsweise Aspose.Cells [4], Spire.XLS for .NET [5], XlsIORenderer.Net.Core [6], TMS FlexCel Studio for .NET [7], SpreadsheetGear for .NET Standard [8], ExcelMapper [9] und GemBox.Spreadsheet [10]. Dabei handelt es sich sowohl um einzelne Bibliotheken, die als NuGet-Package vorliegen und oft unter der MIT-Lizenz bereitgestellt werden, als auch um kommerzielle und funktionsreiche Softwaresuiten. Wer beispielsweise nur eine Excel-Datei mit einigen Daten lesen und schreiben will, kann prüfen, ob er mit einer einfachen und kostenfreien Bibliothek ans Ziel kommt.

Anders sieht es aus, wenn umfassende Excel-Dateien mit einer großen Funktionsvielfalt aus der App heraus zu erstellen und zu bearbeiten sind, das heißt, wenn die Dateien Datenwerte, Formeln, Grafiken, Berichte und so weiter enthalten sollen. Hier bietet sich der Einsatz einer professionellen Bibliothek mit entsprechendem Funktionsumfang, Support und Unterstützung für unterschiedliche Systeme an. Davon gibt es eine ganze Reihe. Dazu gehört die Softwaresuite TMS FlexCel Studio for .NET, die hier vorgestellt wird.



Die Paketquelle TMS FlexCel zu NuGet hinzufügen (vorne) und Verweis auf die Bibliothek im Projekt (hinten) aufnehmen (Bild 3)

TMS FlexCel Studio for .NET

Die Software TMS FlexCel Studio for .NET dient dem Lesen, Erstellen und Editieren von einfachen bis komplexen Excel-Dateien mit einer großen Anzahl von Funktionen. Die wesentlichen technischen Eigenschaften und Funktionen fasst Tabelle 1 zusammen.

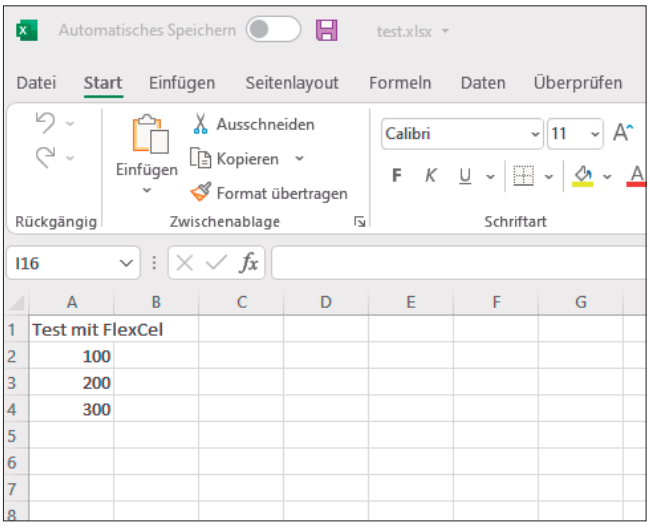
Die Bibliothek ist zwar kostenpflichtig (Einzellizenz 195 Euro, Unternehmenslizenz 775 Euro), es gibt jedoch eine Trial-Edition für Experimente und um die Eignung im konkreten Projekt zu beurteilen. Sie können TMS FlexCel direkt beim Hersteller entweder als NuGet-Package herunterladen oder als komplettes Installationspaket für die unterschiedlichen Visual-Studio-Versionen, auch für VS 2022. Aktuell ist die Version 7.14. Auch das NuGet-Testpaket ist nur über die TMS-Website erhältlich. Sie laden es von dort herunter, entpacken es und fügen es in Visual Studio als lokale Quelle hinzu (Bild 3 vorne) Die Software steht auch für Delphi zur Verfügung, siehe den Kasten TMS FlexCel Studio für Delphi.

Das erste Experiment ist denkbar einfach. Es besteht aus einer Windows-Forms-Anwendung (.NET 6), in die das NuGet-Paket eingebunden wird (Bild 3 hinten). Dazu muss dieses als lokale Quelle der Paketverwaltung hinzugefügt werden; dann sind Verweise aus dem Projekt auf die Bibliothek möglich. Danach lässt sich schon eine „Hello World"-Anwendung erstellen. In diesem Fall geht es um das Generieren einer einfachen Excel-Datei. Legen Sie einen *Button* und eine *Click()*-Methode an. Letztere führt den folgenden Quellcode aus:

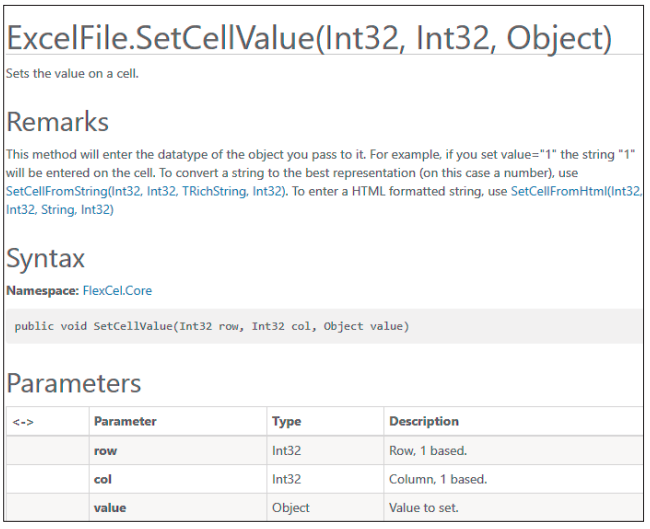
```
XlsFile xls = new XlsFile(1,
  TExcelFileFormat.v2021, true);
                    xls.SetCellValue(1, 1,
                      "Test mit FlexCel");
xls.SetCellValue(2, 1, 100.0);
xls.SetCellValue(3, 1, 200,0);
xls.SetCellValue(4, 1, new
  TFormula("=Sum(A2:A3)"));
xls.Save(Path.Combine(Environment.
  GetFolderPath(Environment.
  SpecialFolder.Personal),
  "test.xlsx"));
```

Dabei gilt:

- *new XlsFile()* erzeugt ein neues Excel-Dokument mit einem Tabellenblatt im angegebenen Excel-Format und der Option, dass eine vorhandene Datei überschrieben wird.
- *SetCellValue()* schreibt einen Wert (etwa Zeichenkette, Zahl, Formel) in eine bestimmte Zelle der Tabelle.
- *Save()* speichert die Excel-Datei am angegebenen Ort.



Die durch Quellcode erzeugte Excel-Datei (Bild 4)



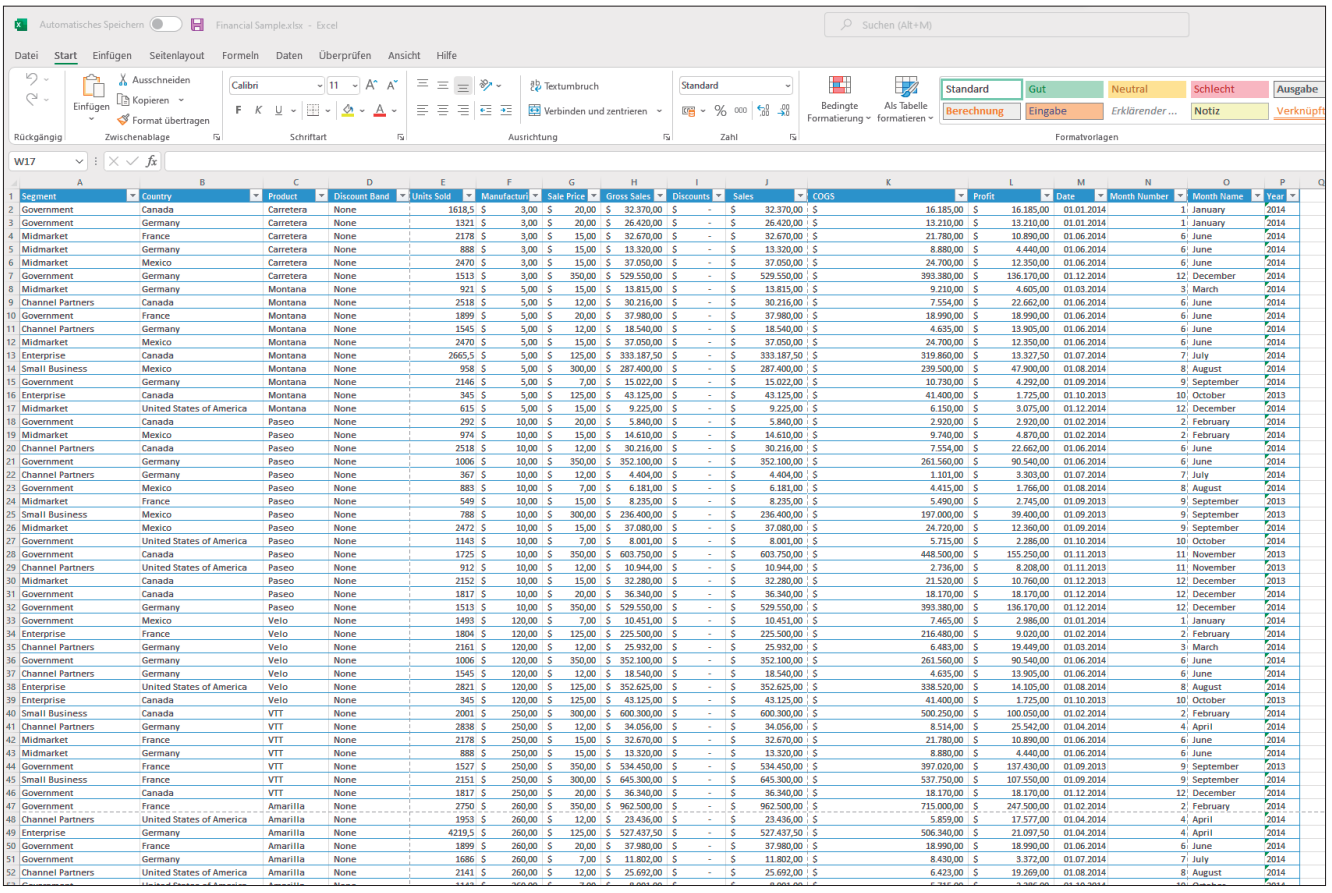
Die API-Dokumentation der Methode SetCellValue() (Bild 5)

Die so erzeugte Excel-Datei kann danach in Excel geöffnet werden (Bild 4). Damit haben Sie eine erste Excel-Datei aus C# erstellt. Nun ist es Zeit, sich einige ausgewählte Funktionen der Bibliothek näher anzusehen.

Funktionen von TMS FlexCel

TMS FlexCel bietet dem Programmierer ein sehr umfassendes API, das einen Großteil der Funktionen von Microsofts Tabellenkalkulation kapselt. Diese gliedern sich in die folgenden Bereiche:

- *Core*: Basisklassen zur Nutzung der Bibliothek
- *Draw*: Zeichenfunktionen
- *Pdf*: Klassen für das Erstellen von PDF-Dateien
- *Render*: Klassen zum Darstellen einer Excel-Tabelle auf dem Bildschirm, zum Drucken oder um sie in andere Dateiformate zu exportieren ▶



Ein Auszug aus Microsofts Datentabelle mit Finanzbeispielen (Bild 6)

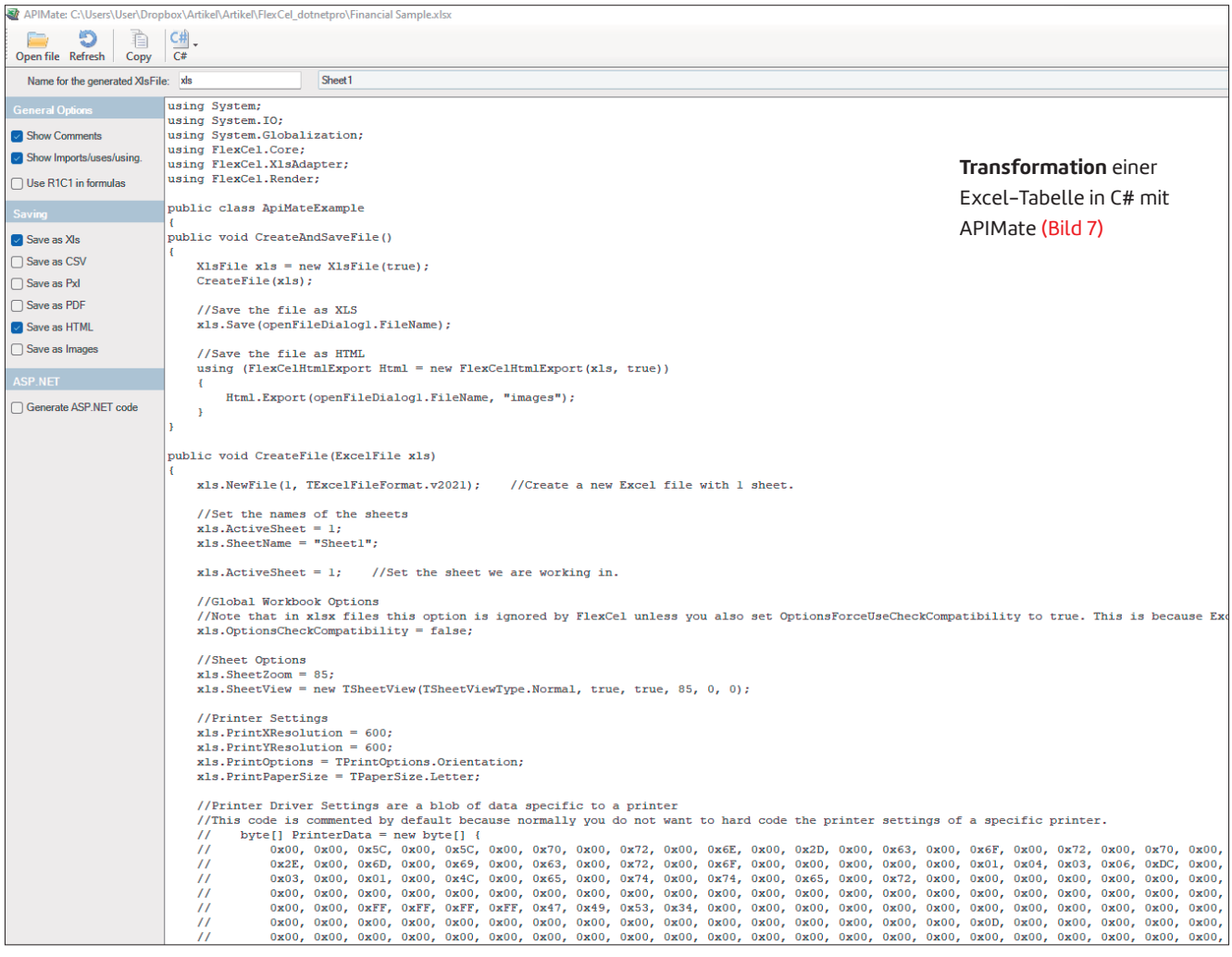
- *Report*: Komponenten zum Erstellen von Excel-Berichten basierend auf einer Vorlage
- *AddinFunctions*: Excel-Add-in-Funktionen, die von FlexCel bei der Neuberechnung verwendet werden
- *XlsAdapter*: Natives API zum Lesen und Schreiben einer Excel-Datei
- *Winforms/AspNet*: Spezifische Klassen für Windows-Forms- und ASP.NET-Anwendungen

Die API-Dokumentation liefert einen Überblick und enthält genaue Beschreibungen der Klassen und Methoden. Als Beispiel sei die Methode *SetCellValue()* der Klasse *ExcelFile* genannt. Diese erwartet die Angabe der Zelle in Form eines Zeilen- und Spaltenindex (1-basiert) und einen Wert des Typs *Object* für den Inhalt der Zelle. Dieser Objektwert kann beispielsweise ein Zahlenwert, eine Zeichenkette oder eine Formel sein (Bild 5).

Aus Excel werde Quellcode

Bisher war das Ziel, eine Excel-Datei mithilfe von Quellcode zu erzeugen. Das kann ein Szenario sein, wird jedoch nur bis zu einer gewissen Komplexität realistisch sein. Umfangreiche Excel-Tabellen und Excel-basierte Berichte entstehen in der Praxis, indem die Anwenderinnen und Anwender Daten erfassen oder importieren, Berechnungen durch Zellbezüge herstellen, Filter einbauen, die Zellen individuell formatieren, Grafiken ergänzen und so weiter. Eine derartig komplexe Tabelle kann dann zum Beispiel als Vorlage dienen, um diese wiederum programmatisch zu generieren und dann die statischen Werte in der Tabelle durch Variablen zu ersetzen. In diesem Fall stammen die Daten der Anwendung aus Berechnungen, Datenbankabfragen et cetera. Daraus wird eine Tabelle oder ein Bericht im Excel-Format erstellt, die oder der dann wiederum durch die Anwender weiterverarbeitet werden kann. Um eine vorhandene Excel-Tabelle in C#-(VB-) Quellcode zu transformieren, gibt es das Tool APIMate, das kostenfrei für Windows und macOS aus dem Internet heruntergeladen werden kann [11].

Jetzt bedarf es noch einer Datentabelle mit mehr als einer Handvoll Einträge. Als solche soll die Excel-Tabelle *Financial Sample.xlsx* von Microsoft dienen [12]. Es handelt sich um eine Tabelle mit 700 Datensätzen, sortiert in 16 Spalten und mit einer Filtermöglichkeit (Bild 6). Diese Excel-Tabelle soll mit C#-Quellcode transformiert werden. Der entsprechende Quellcode wird dann die Ausgangsbasis für die weiteren Arbeiten darstellen, zum Beispiel für das Ersetzen von statischen Werten durch Variablen, den Export der Daten in andere Formate und anderes mehr.



Transformation einer Excel-Tabelle in C# mit APIMate (Bild 7)

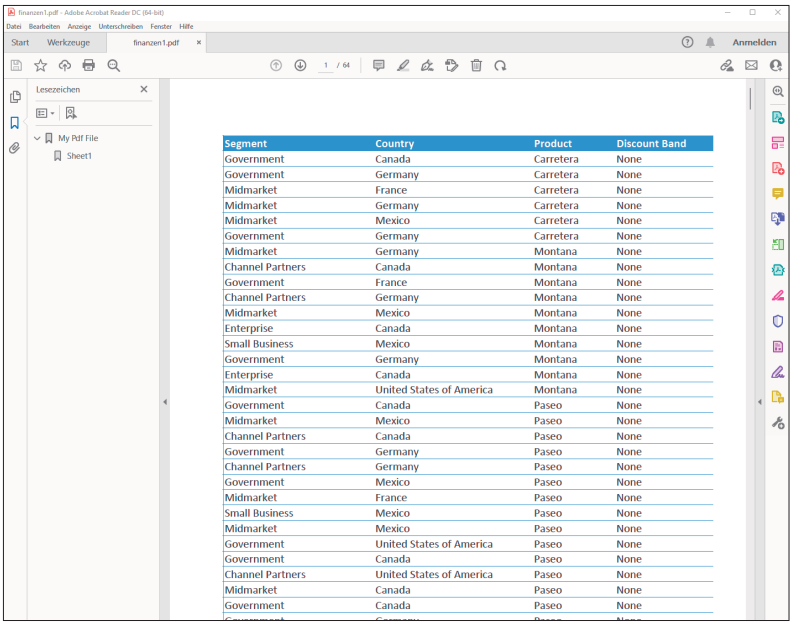
Listing 1: Methode zum Speichern als XLS, PDF und HTML

```
public void CreateAndSaveFile()
{
  XlsFile xls = new XlsFile(true);
  CreateFile(xls);

  //Save the file as XLS
  xls.Save(Path.Combine(Environment.GetFolderPath(
    Environment.SpecialFolder.Personal),
    "finanzen1.xlsx"));

  //Save the file as Pdf
  using (FlexCelPdfExport Pdf =
      new FlexCelPdfExport(xls, true))
  {
    using (FileStream PdfStream = new FileStream(
      Path.Combine(Environment.GetFolderPath(
      Environment.SpecialFolder.Personal),
      "finanzen1.pdf"), FileMode.OpenOrCreate))
    {
      Pdf.BeginExport(PdfStream);
      Pdf.PageLayout = TPageLayout.Outlines;
      Pdf.ExportAllVisibleSheets(false,
        "My Pdf File");
      Pdf.EndExport();
    }
  }

  //Save the file as HTML
  using (FlexCelHtmlExport Html =
      new FlexCelHtmlExport(xls, true))
  {
    Html.Export(Path.Combine(Environment.
      GetFolderPath(Environment.SpecialFolder.
      Personal), "finanzen1.html"), "images");
  }
}
```

Für eine Transformation starten Sie APIMate und öffnen die Excel-Tabelle. Im Tool können Sie einige Parameter für die Konvertierung einstellen, etwa die Anzeige von Kommentaren, die Aufnahme der *using*-Anweisungen und das Erzeugen von Methoden für verschiedene Speicherformate. Aktivieren Sie das Generieren von Methoden zum Speichern in den Formaten XLS, PDF und HTML. Der erstellte Quellcode wird direkt in der Programmiersprache C#, alternativ in Visual Basic angezeigt (Bild 7).

Der Quellcode wird in das Projekt in Visual Studio übernommen. Alle Daten, das heißt die Werte in den einzelnen Zellen, alle Formatierungen, die Filter und auch die Einstellungen zum Drucklayout, die in Excel in der Ausgangstabelle vorgenommen wurden, werden nun über C#-Code erstellt. Scrollen Sie dazu durch die betreffende Klasse in Visual Studio.

Wenn Sie jetzt – zum Beispiel durch einen Klick auf einen Button – die Methode *CreateAndSaveFile()* aus Listing 1 aufrufen, dann wird die Tabelle im Speicher erstellt und als Excel-, PDF- und HTML-Datei gespeichert, in diesem Fall im Dokumentenordner des Nutzers. Dafür stellt das API von FlexCel entsprechende Methoden und Klassen bereit. Beim Export in HTML lässt sich ergänzend ein Name für einen Unterordner für mögliche einzubettende Bilder übergeben. Die Tabelle enthält solche Elemente zunächst nicht.



Segment	Country	Product	Discount Band
Government	Canada	Carretera	None
Government	Germany	Carretera	None
Midmarket	France	Carretera	None
Midmarket	Germany	Carretera	None
Midmarket	Mexico	Carretera	None
Government	Germany	Carretera	None
Midmarket	Germany	Montana	None
Channel Partners	Canada	Montana	None
Government	France	Montana	None
Channel Partners	Germany	Montana	None
Midmarket	Mexico	Montana	None
Enterprise	Canada	Montana	None
Small Business	Mexico	Montana	None
Government	Germany	Montana	None
Enterprise	Canada	Montana	None
Midmarket	United States of America	Montana	None
Government	Canada	Paseo	None
Midmarket	Mexico	Paseo	None
Channel Partners	Canada	Paseo	None
Government	Germany	Paseo	None
Channel Partners	Germany	Paseo	None
Government	Mexico	Paseo	None
Midmarket	France	Paseo	None
Small Business	Mexico	Paseo	None
Midmarket	Mexico	Paseo	None
Government	United States of America	Paseo	None
Government	Canada	Paseo	None
Channel Partners	United States of America	Paseo	None
Midmarket	Canada	Paseo	None
Government	Canada	Paseo	None

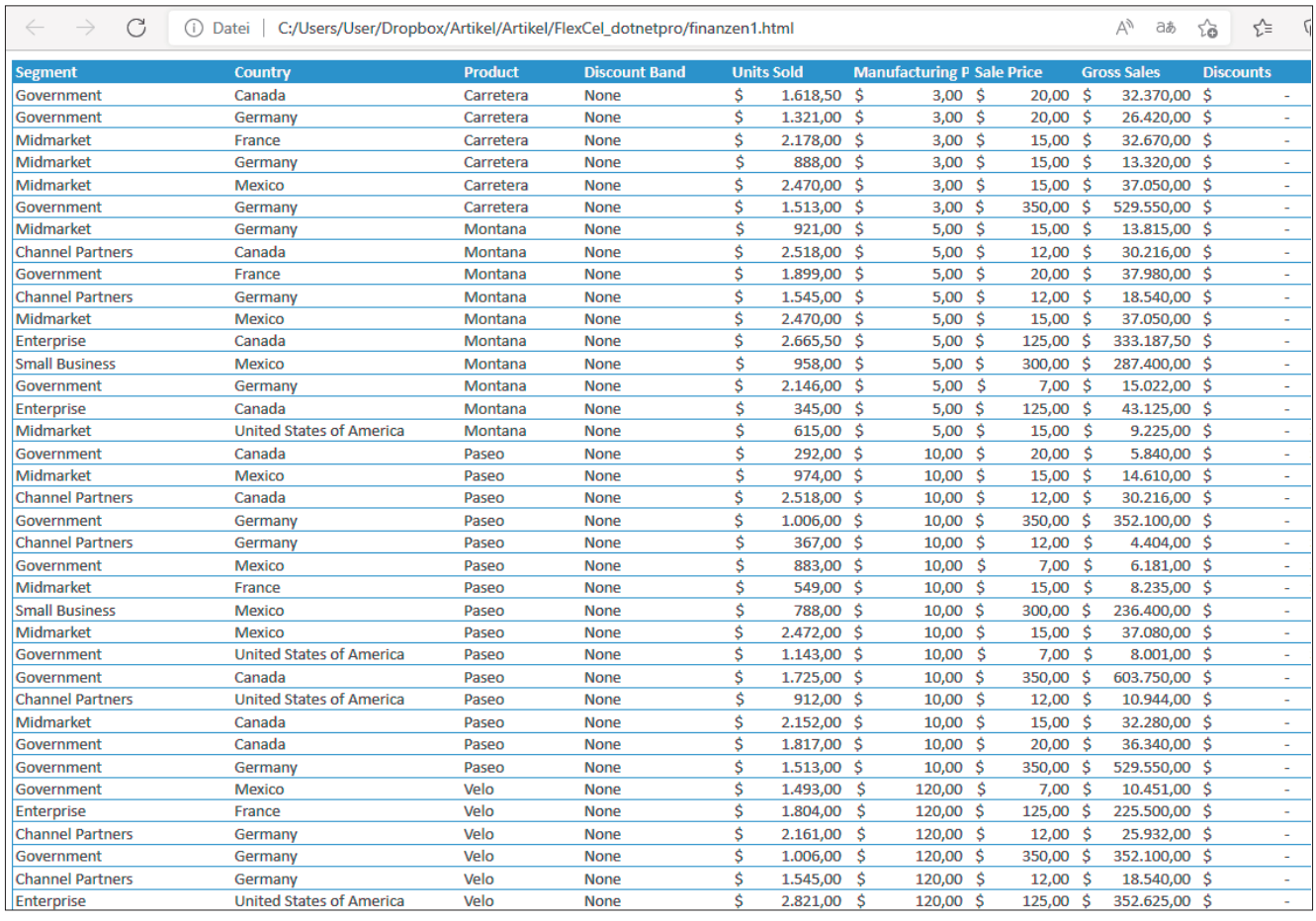
Der Export der Tabelle in eine PDF-Datei (Bild 8)

Das Ergebnis ist eine Excel-Tabelle, die sich in nichts von der Ausgangstabelle unterscheidet, eine PDF-Datei (Bild 8) und eine HTML-Datei (Bild 9). Beachten Sie: Der Export in PDF erfolgte durch eine native Funktion der Bibliothek ohne den Einsatz von Excel.

Werfen wir nun noch einen Blick auf den durch APIMate automatisch erzeugten Quellcode und die korrespondierende Excel-Tabelle. Bild 10 stellt die die beiden Elemente nebeneinander, das heißt eine jeweilige Excel-Zeile und den dazu erzeugten Quellcode.

Berichte

FlexCel enthält auch einen Berichtsgenerator, der Excel als Berichtsdesigner verwendet. Das funktioniert so: Sie erstellen eine Vorlage für einen Bericht in Excel, schreiben einige Tags und führen den Bericht aus. FlexCel ersetzt diese ▶



Datei | C:/Users/User/Dropbox/Artikel/Artikel/FlexCel_dotnetpro/finanzen1.html

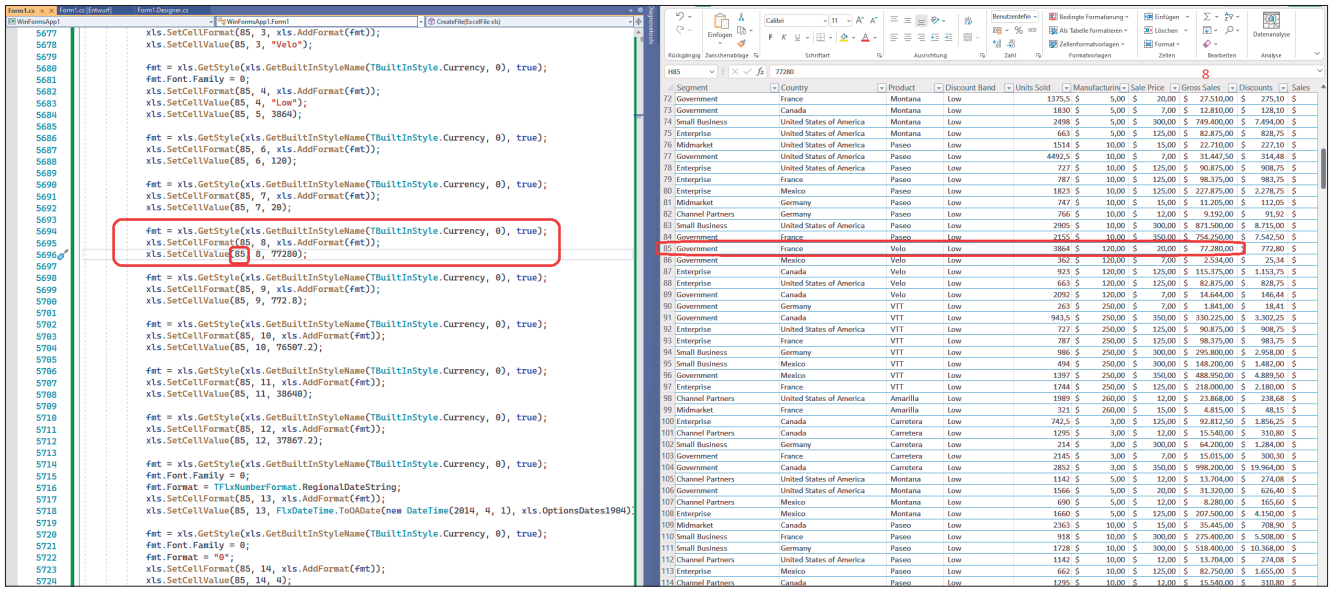
Segment	Country	Product	Discount Band	Units Sold	Manufacturing P	Sale Price	Gross Sales	Discounts
Government	Canada	Carretera	None	$ 1.618,50	$ 3,00	$ 20,00	$ 32.370,00	$ -
Government	Germany	Carretera	None	$ 1.321,00	$ 3,00	$ 20,00	$ 26.420,00	$ -
Midmarket	France	Carretera	None	$ 2.178,00	$ 3,00	$ 15,00	$ 32.670,00	$ -
Midmarket	Germany	Carretera	None	$ 888,00	$ 3,00	$ 15,00	$ 13.320,00	$ -
Midmarket	Mexico	Carretera	None	$ 2.470,00	$ 3,00	$ 15,00	$ 37.050,00	$ -
Government	Germany	Carretera	None	$ 1.513,00	$ 3,00	$ 350,00	$ 529.550,00	$ -
Midmarket	Germany	Montana	None	$ 921,00	$ 5,00	$ 15,00	$ 13.815,00	$ -
Channel Partners	Canada	Montana	None	$ 2.518,00	$ 5,00	$ 12,00	$ 30.216,00	$ -
Government	France	Montana	None	$ 1.899,00	$ 5,00	$ 20,00	$ 37.980,00	$ -
Channel Partners	Germany	Montana	None	$ 1.545,00	$ 5,00	$ 12,00	$ 18.540,00	$ -
Midmarket	Mexico	Montana	None	$ 2.470,00	$ 5,00	$ 15,00	$ 37.050,00	$ -
Enterprise	Canada	Montana	None	$ 2.665,50	$ 5,00	$ 125,00	$ 333.187,50	$ -
Small Business	Mexico	Montana	None	$ 958,00	$ 5,00	$ 300,00	$ 287.400,00	$ -
Government	Germany	Montana	None	$ 2.146,00	$ 5,00	$ 7,00	$ 15.022,00	$ -
Enterprise	Canada	Montana	None	$ 345,00	$ 5,00	$ 125,00	$ 43.125,00	$ -
Midmarket	United States of America	Montana	None	$ 615,00	$ 5,00	$ 15,00	$ 9.225,00	$ -
Government	Canada	Paseo	None	$ 292,00	$ 10,00	$ 20,00	$ 5.840,00	$ -
Midmarket	Mexico	Paseo	None	$ 974,00	$ 10,00	$ 15,00	$ 14.610,00	$ -
Channel Partners	Canada	Paseo	None	$ 2.518,00	$ 10,00	$ 12,00	$ 30.216,00	$ -
Government	Germany	Paseo	None	$ 1.006,00	$ 10,00	$ 350,00	$ 352.100,00	$ -
Channel Partners	Germany	Paseo	None	$ 367,00	$ 10,00	$ 12,00	$ 4.404,00	$ -
Government	Mexico	Paseo	None	$ 883,00	$ 10,00	$ 7,00	$ 6.181,00	$ -
Midmarket	France	Paseo	None	$ 549,00	$ 10,00	$ 15,00	$ 8.235,00	$ -
Small Business	Mexico	Paseo	None	$ 788,00	$ 10,00	$ 300,00	$ 236.400,00	$ -
Midmarket	Mexico	Paseo	None	$ 2.472,00	$ 10,00	$ 15,00	$ 37.080,00	$ -
Government	United States of America	Paseo	None	$ 1.143,00	$ 10,00	$ 7,00	$ 8.001,00	$ -
Government	Canada	Paseo	None	$ 1.725,00	$ 10,00	$ 350,00	$ 603.750,00	$ -
Channel Partners	United States of America	Paseo	None	$ 912,00	$ 10,00	$ 12,00	$ 10.944,00	$ -
Midmarket	Canada	Paseo	None	$ 2.152,00	$ 10,00	$ 15,00	$ 32.280,00	$ -
Government	Canada	Paseo	None	$ 1.817,00	$ 10,00	$ 20,00	$ 36.340,00	$ -
Government	Germany	Paseo	None	$ 1.513,00	$ 10,00	$ 350,00	$ 529.550,00	$ -
Government	Mexico	Velo	None	$ 1.493,00	$ 120,00	$ 7,00	$ 10.451,00	$ -
Enterprise	France	Velo	None	$ 1.804,00	$ 120,00	$ 125,00	$ 225.500,00	$ -
Channel Partners	Germany	Velo	None	$ 2.161,00	$ 120,00	$ 12,00	$ 25.932,00	$ -
Government	Germany	Velo	None	$ 1.006,00	$ 120,00	$ 350,00	$ 352.100,00	$ -
Channel Partners	Germany	Velo	None	$ 1.545,00	$ 120,00	$ 12,00	$ 18.540,00	$ -
Enterprise	United States of America	Velo	None	$ 2.821,00	$ 120,00	$ 125,00	$ 352.625,00	$ -

Hier wurde die Tabelle in HTML exportiert (Bild 9)

Tags durch die Werte aus der Datenbank oder sonstige Berechnungen.

Der große Vorteil eines Berichtsdesigners besteht darin, dass die Anwender den Bericht, zum Beispiel das Layout, selbst anpassen können und dazu kein Quellcode geschrieben werden muss. Dabei werden die Ebenen wie in Bild 11 unterschieden [13].

- *Data Model*: Das Datenmodell. Die Angaben können in einer Datenbank oder in Listen von Objekten gespeichert werden.



```
xls.SetCellFormat(85, 3, xls.AddFormat(fmt));
xls.SetCellValue(85, 3, "Velo");

fmt = xls.GetStyle(xls.GetBuiltInStyleName(TBuiltInStyle.Currency, 0), true);
fmt.Font.Family = 0;
xls.SetCellFormat(85, 4, xls.AddFormat(fmt));
xls.SetCellValue(85, 4, "Low");
xls.SetCellValue(85, 5, 3864);

fmt = xls.GetStyle(xls.GetBuiltInStyleName(TBuiltInStyle.Currency, 0), true);
xls.SetCellFormat(85, 6, xls.AddFormat(fmt));
xls.SetCellValue(85, 6, 120);

fmt = xls.GetStyle(xls.GetBuiltInStyleName(TBuiltInStyle.Currency, 0), true);
xls.SetCellFormat(85, 7, xls.AddFormat(fmt));
xls.SetCellValue(85, 7, 20);

fmt = xls.GetStyle(xls.GetBuiltInStyleName(TBuiltInStyle.Currency, 0), true);
xls.SetCellFormat(85, 8, xls.AddFormat(fmt));
xls.SetCellValue(85, 8, 77280);

fmt = xls.GetStyle(xls.GetBuiltInStyleName(TBuiltInStyle.Currency, 0), true);
xls.SetCellFormat(85, 9, xls.AddFormat(fmt));
xls.SetCellValue(85, 9, 772.8);

fmt = xls.GetStyle(xls.GetBuiltInStyleName(TBuiltInStyle.Currency, 0), true);
xls.SetCellFormat(85, 10, xls.AddFormat(fmt));
xls.SetCellValue(85, 10, 76507.2);

fmt = xls.GetStyle(xls.GetBuiltInStyleName(TBuiltInStyle.Currency, 0), true);
xls.SetCellFormat(85, 11, xls.AddFormat(fmt));
xls.SetCellValue(85, 11, 38640);

fmt = xls.GetStyle(xls.GetBuiltInStyleName(TBuiltInStyle.Currency, 0), true);
xls.SetCellFormat(85, 12, xls.AddFormat(fmt));
xls.SetCellValue(85, 12, 37867.2);

fmt = xls.GetStyle(xls.GetBuiltInStyleName(TBuiltInStyle.Currency, 0), true);
fmt.Font.Family = 0;
fmt.Format = TFlxNumberFormat.RegionalDateString;
xls.SetCellFormat(85, 13, xls.AddFormat(fmt));
xls.SetCellValue(85, 13, FlxDateTime.ToOADate(new DateTime(2014, 4, 1), xls.OptionsDates1904));

fmt = xls.GetStyle(xls.GetBuiltInStyleName(TBuiltInStyle.Currency, 0), true);
fmt.Font.Family = 0;
fmt.Format = "0";
xls.SetCellFormat(85, 14, xls.AddFormat(fmt));
xls.SetCellValue(85, 14, 4);
```

Eine Zeile in Excel und der von APIMate dazu erzeugte C#-Code (Bild 10)

- *Interface*: Diese Schicht verbindet die Daten- und die Präsentationsschicht.
- *Presentation*: Hier erfolgt die Definition aller visuellen Aspekte des Berichts, wie Datenposition, Schriftarten, Farben und so weiter.

Der Vorteil: Das Design ist sauber auf drei verschiedene Ebenen verteilt. Die meiste Arbeit wird auf der Präsentationsebene mit Excel erledigt. Der Anwender gestaltet den Bericht visuell in Excel und arbeitet mit Platzhaltern und Verweisen (Tags), die später durch Echtzeitdaten ersetzt werden.

Ein simples Beispiel demonstriert es: Erstellen Sie zuerst eine Excel-Tabelle (Präsentationsebene), zum Beispiel mit den folgenden Einträgen in den Zellen:

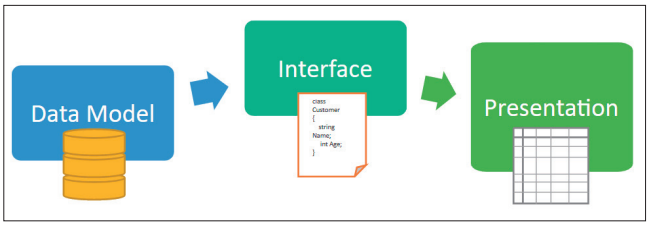
- A1: <#Kunde.ID>
- B1: <#Kunde.Name>
- C1: <#Kunde.Ort>

Die Syntax soll darauf hinweisen, dass hier später „echte" Daten eingefügt werden. Die Tabelle wird nun minimal formatiert und ein Filter gesetzt. Ebenso ist noch der Bereich zu definieren, in dem später die Daten programmgesteuert eingespielt werden sollen. Gehen Sie dazu in Excel in das Dialogfeld *Formeln | Namen definieren* und legen Sie einen Namen mit der Bezeichnung *__Kunde__* mit einer Referenz auf die Zelle *A1* des aktuellen Arbeitsblatts fest (Bild 12). Achten Sie auf die doppelten Unterstriche am Anfang und am Ende des Namens und speichern Sie die Datei unter dem Namen *report_kunden_template.xlsx* im Dokumentenordner und schließen Sie Excel. Das ist die Vorlage für den Bericht.

Im zweiten Schritt erstellen Sie ein Windows-Forms-Projekt in Visual Studio mit Zugriff auf die FlexCel-Bibliothek und fügen beispielsweise in einer Click-Methode eines Buttons den Quellcode aus Listing 2 ein. Sie definieren eine Klasse *Kunde* mit drei Eigenschaften für ID, Name und Ort. Dazu kommt dann noch die Methode *CreateReport()*, die eine Liste von Kunden aufbaut. Anschließend erfolgt das Erstellen eines Berichts, der auf das eben definierte Template (*report_kunden_template.xlsx*) zugreift und das Ergebnis als neue Excel-Datei *kunden.xls* ausgibt. Starten Sie die Anwendung und führen Sie die Methode aus. Template und Daten aus der Anwendung werden miteinander „verbunden" (Bild 13).

Drittens ändern Sie nun testweise die Vorlage, zum Beispiel die Farben, und starten die Anwendung erneut. Der Bericht wird nun auf der Basis der neuen Vorlage erstellt.

Wie von Wunderhand fügen sich Vorlage und Daten zu einem neuen Bericht zusammen. Die Nutzer können damit das Layout des Berichts eigenständig ohne Änderungen am Quellcode anpassen. Wie der Aufbau der Excel-Vorlage für den Bericht erfolgt, das heißt die Definition der Tags, Referenzen et cetera, und wie man diese aus dem Quellcode der Anwendung adressiert, ist in der Dokumentation in den Abschnitten *Reports Developer Guide* und *Reports Designer Guide* des User Guide für FlexCel beschrieben [11].

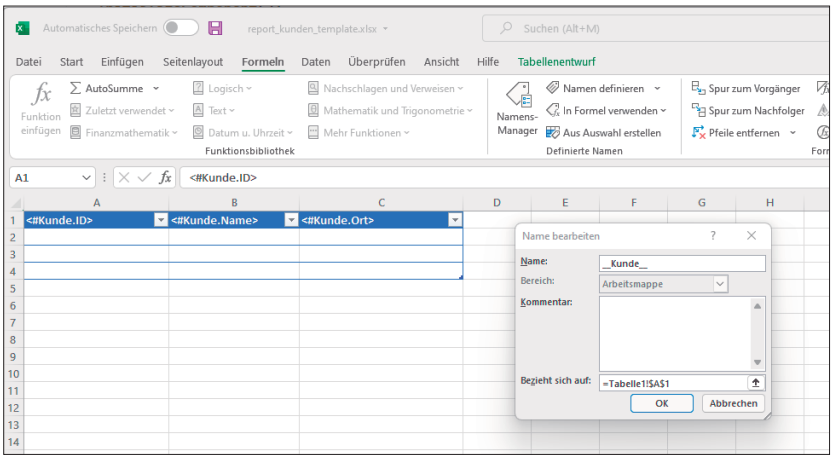


Die Trennung von Daten, Layout und Präsentation bei einem Bericht (Bild 11)

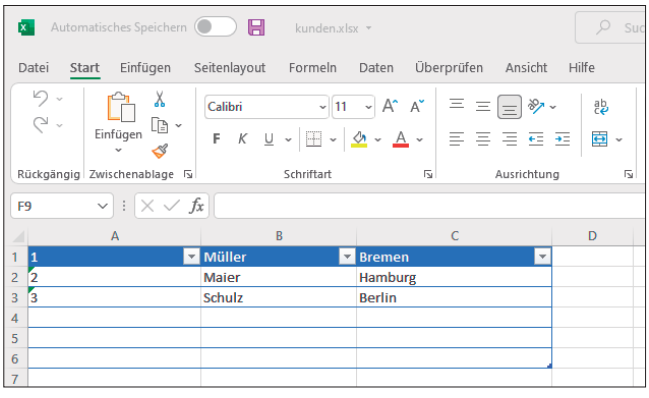
Berichte sind die flexiblere und anwenderfreundlichere Variante, um Excel-Dateien aus Quellcode zu erstellen. Hinsichtlich der Verarbeitungsgeschwindigkeit empfiehlt sich jedoch die direkte Anwendung des API der Bibliothek.

Auf dem Server

Das obige Beispiel einer .NET-Applikation hat die Funktionen der Bibliothek gezeigt, die direkt im Client des Nutzers ausgeführt werden. In der Praxis wird es häufig so sein, dass die .NET-Anwendung auf dem Server läuft und die Clients der Anwender dann über das Netzwerk auf die Server- ▶



Die Excel-Tabelle dient als Präsentationsschicht für den Bericht (Bild 12)



Ein Bericht, aus Template und Daten erzeugt (Bild 13)

anwendung zugreifen. Während diese auf .NET basiert, besteht bei der Technologie der Client-Anwendungen freie Wahl. Hier kann es sich sowohl um eine Desktop-Anwendung als auch um eine mobile App oder eine Webanwendung handeln. Weder lokal noch auf dem Server wird eine Excel-Installation benötigt, um beispielsweise Excel-Tabellen zu erstellen oder zu bearbeiten.

Große Tabellen und Laufzeitverhalten

In kleineren und mittelgroßen Excel-Tabellen ist die Verarbeitungsgeschwindigkeit kein besonderes Thema. In der Praxis kommen jedoch auch Tabellen mit mehr als 100 000 Zellen und gegebenenfalls mehreren Tausend Formeln zum Einsatz. Dann kann das Einfügen einer Zeile in der Mitte der Tabelle Excel schon einige Zeit beschäftigen. In einigen unglücklichen Fällen – wenn die Zellbezüge beispielsweise sehr umfangreich und komplex sind – kann das auch einen Absturz der Tabellenkalkulation mit sich bringen. Leistungsfähigere Hardware und die Nutzung der 64-Bit-Version sind zwei bekannte Lösungsansätze. In der Excel-Dokumentation findet sich ein eigener Abschnitt, um das Laufzeitverhalten einer Excel-Datei zu verbessern [14].

Das Problem der Verarbeitungsgeschwindigkeit besteht auch, wenn man Excel-Dateien über Programmcode erstellt oder eine vorhandene Datei öffnen möchte. TMS FlexCel.NET verwendet verwalteten Code, das heißt, es gelten die Leistungsmerkmale der .NET-Plattform, zum Beispiel in Bezug auf die Nutzung der Speicherbereinigung. Wird eine Excel-Datei geöffnet, dann muss diese im Speicher gehalten werden. Der Speicherverbrauch steigt dann schnell an, wenn es sich um große und komplexe Dateien, zum Beispiel mit vielen Zellbezügen oder Grafiken, handelt. Ein Beispiel: Gibt es in einer Zelle einen Verweis auf eine Zelle in einem anderen Tabellenblatt, dann müssen bei Anpassungen beide Tabellenblätter komplett im Speicher gehalten werden.

Die Dokumentation von TMS FlexCel nennt einige Ansätze, das Laufzeitverhalten zu verbessern [15]:

- **Virtueller Modus:** Manchmal benötigen Sie nicht alle Funktionen einer Tabellenkalkulation, beispielsweise wenn Sie nur Werte aus Zellen lesen wollen. In diesen Fällen können Sie den virtuellen Modus einsetzen. Die Datei wird geöffnet und jede Zelle wird gelesen, jedoch nicht in das Speichermodell geladen. Am Ende erhalten Sie ein *ExcelFile*-Objekt, das keine Zellen enthält, aber Diagramme, Zeichnungen, Kommentare et cetera. Die Zellen können dann verwendet werden, während sie gelesen werden.
- **64-Bit:** Es kann sinnvoll sein, FlexCel im 64-Bit-Modus zu verwenden. Das ist gerade bei einer Ausführung auf dem Server von Bedeutung. Die konkreten Auswirkungen bei einem Wechsel von 32- auf 64-Bit hängen auch von der konkreten .NET-Version und vom Aufbau der Excel-Datei ab.
- **Hardware:** Läuft die Anwendung auf dem Server und die Anwender greifen über Clients auf die Applikation und damit auch auf FlexCel zu, dann sollte beispielsweise die Server-CPU genügend Kerne bieten, damit die Anforderungen unabhängig im Prozessor bearbeitet werden können. Optimierungen an der Speicherbereinigung für die serverseitige Verwendung (siehe Dokumentation) können Verbesserungen bringen.

Listing 2: Excel-Datei aus Bericht heraus erstellen

```
public class Kunde
{
  public string ID { get; set; }
  public string Name { get; set; }
  public string Ort { get; set; }

}

public void CreateReport()
{
  var Kunden = new List<Kunde>();
  Kunden.Add(new Kunde() { ID = "1",
      Name = "Müller", Ort = "Bremen" });
  Kunden.Add(new Kunde() { ID = "2",
      Name = "Maier", Ort = "Hamburg" });
  Kunden.Add(new Kunde() { ID = "3",
      Name = "Schulz", Ort = "Berlin" });
  using (FlexCelReport report =
      new FlexCelReport(true))
  {
    report.AddTable("Kunde", Kunden);
    report.Run(Path.Combine(Environment.
      GetFolderPath(Environment.SpecialFolder.
      Personal), "report_kunden_template.xlsx"),
      Path.Combine(Environment.GetFolderPath(
      Environment.SpecialFolder.Personal),
      "kunden.xlsx"));
  }
}
```

Grundsätzlich gelten in Bezug auf das Laufzeitverhalten bei einer nativen Excel-Bibliothek andere Aspekte als bei der Verwendung einer OLE-Automation, bei der die Schnittstelle selbst ein Flaschenhals ist. Bei einer nativen Bibliothek sollten daher viele Perfomance-Probleme in der Praxis keine Rolle spielen.

Fazit

Excel-Dateien zu erstellen ist eine häufige Anforderung für betriebswirtschaftliche Anwendungen, die intensiv mit Daten arbeiten. Dabei geht es oft nicht nur um einen bloßen Import oder Export der Daten, sondern es sind umfassende Tabellen mit Formeln, Zellbezüge oder komplette interaktive Berichte zu erstellen. Um das umzusetzen, empfiehlt sich am einfachsten eine fertige Bibliothek.

Die hier vorgestellte Bibliothek TMS FlexCel Studio for .NET bietet sehr viele Funktionen und gestattet es, über Quellcode einen sehr großen Teil an Excel-Funktionen umzusetzen. Dennoch ist das API einfach und kommt schnell zum

gewünschten Ergebnis. Mit standardisierten Excel-Berichten können Anwender auf jeden Fall etwas anfangen. Statt komplizierter eigener Implementierungen oder unbekannter Berichtsgeneratoren genügt eine Schaltfläche *Export nach Excel …*, und der Anwender kommt damit garantiert zurecht. ■

[1] Excel-Funktionen (nach Kategorie), www.dotnetpro.de/SL2206FlexCel1
[2] Vorgehensweise: Zugreifen auf Office-Interop-Objekte (C#-Programmierleitfaden), www.dotnetpro.de/SL2206FlexCel2
[3] Was bei der serverseitigen Automatisierung von Office zu beachten ist, www.dotnetpro.de/SL2206FlexCel3
[4] Manipulate Excel Files via .NET APIs, www.dotnetpro.de/SL2206FlexCel4
[5] Spire.XLS, www.dotnetpro.de/SL2206FlexCel5
[6] Overview of Syncfusion Excel (XlsIO) library, www.dotnetpro.de/SL2206FlexCel6
[7] TMS FlexCel Studio for .NET, www.dotnetpro.de/SL2206FlexCel7
[8] Excel Compatibility for iOS, Android, Linux, macOS, UWP and More, www.dotnetpro.de/SL2206FlexCel8
[9] ExcelMapper, www.dotnetpro.de/SL2206FlexCel9
[10] GemBox.Spreadsheet, www.dotnetpro.de/SL2206FlexCel10
[11] Getting Started with FlexCel Studio for the .NET Framework, www.dotnetpro.de/SL2206FlexCel11
[12] Microsoft, Herunterladen der Excel-Finanzbeispielarbeitsmappe für Power BI, www.dotnetpro.de/SL2206FlexCel12
[13] FlexCel Reports Developer Guide, www.dotnetpro.de/SL2206FlexCel13
[14] Excel-Leistung: Verbesserung der Berechnungsleistung, www.dotnetpro.de/SL2206FlexCel14
[15] FlexCel Performance Guide, www.dotnetpro.de/SL2206FlexCel15
[16] TMS FlexCel for VCL & FMX, www.dotnetpro.de/SL2206FlexCel16

Elena Bochkor
arbeitet am Entwurf und Design mobiler Anwendungen und Webseiten. Weitere Informationen zu diesen und anderen Themen der IT finden Sie unter http://larinet.com. Folgen Sie ihr auf Instagram unter **www.instagram.com/larinetcommunication**.

Dr. Veikko Krypczyk
ist begeisterter Entwickler und Fachautor. Weitere Informationen zu diesen und anderen Themen der IT finden Sie unter http://larinet.com. Folgen Sie ihm auf Instagram unter **www.instagram.com/larinetcommunication**.

dnpCode	A2206FlexCel