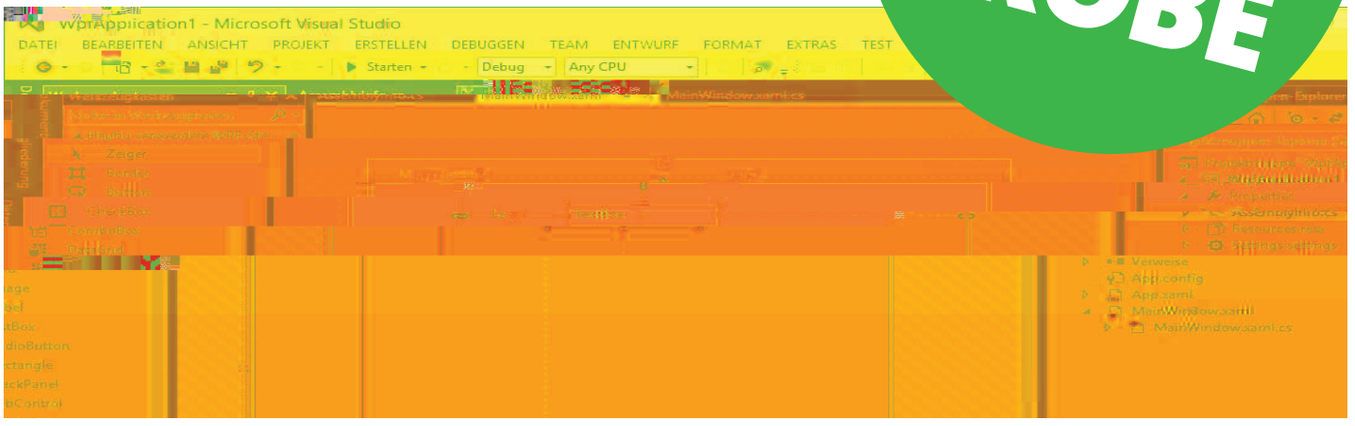
AUSGABE 04/2016

DATENBANK ENTWICKLER

Gratis LESE-PROBE

MAGAZIN FÜR DIE DATENBANKENTWIC
VISUAL STUDIO FÜR DESKTOP, WEB U



TOP-THEMEN:

André Minhorst Verlag

Impressum

DATENBANKENTWICKLER

Borkhofer Str. 17
47137 Duisburg

Redaktion: Dipl.-Ing. André Minhorst

LocalDB-Datenbanken nutzen

LocalDB ist ein spezieller Ausführungsmodus von SQL Server Express, den Sie gut für die Entwicklung und die Weitergabe von Datenbankanwendungen auf Basis von .NET-Technologien nutzen können. Außerdem ist diese Variante des SQL Servers wesentlich leichtgewichtiger und in weniger als einer Minute zu installieren. Der Download umfasst gerade mal 45 MB.

LocalDB herunterladen

Den direkten Download, der nur die Installationsdatei für **LocalDB** enthält, finden Sie beispielsweise unter folgendem Link:

```
https://download.microsoft.com/download/E/1/2/E12B3655-D817-49BA-B934-CEB9DAC0BAF3/SqlLocalDB.msi
```

LocalDB installieren

Nachdem Sie die nur rund 45 MB große Datei **SqlLocalDB.msi** heruntergeladen haben, können Sie diese gleich durch einen Doppelklick installieren. Dazu brauchen Sie nur wenige Schritte im Installationsassistenten zu absolvieren (siehe Bild 1).

Alternativ führen Sie die Installation ohne weitere Interaktion über die Eingabeaufforderung oder eine entsprechende Batch-Datei aus.

Die Eingabeaufforderung muss dabei mit Administratorrechten geöffnet werden. Dazu geben Sie zunächst **cmd** in

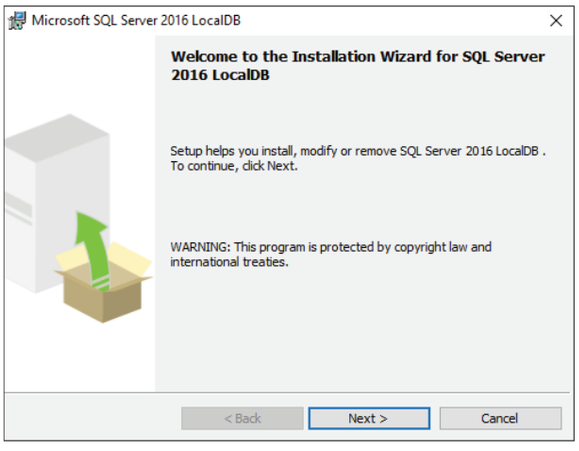


Bild 1: Start der Installation

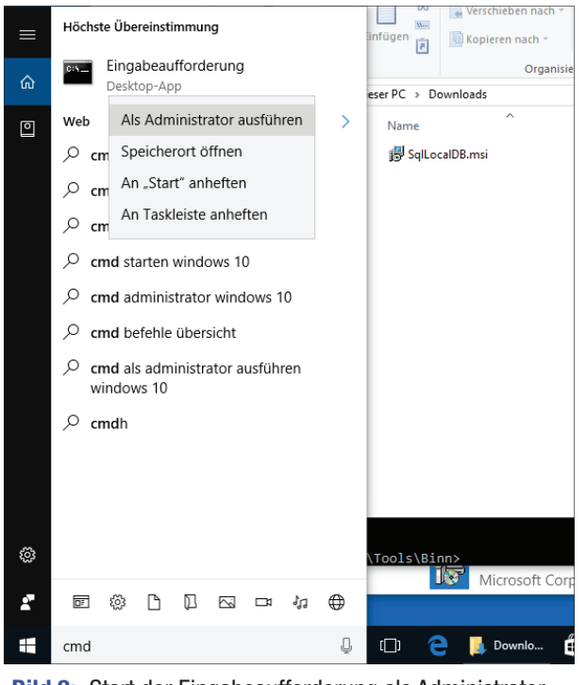


Bild 2: Start der Eingabeaufforderung als Administrator

das Feld **Web und Windows durchsuchen** von Windows ein. Dort erscheint dann ein Eintrag namens Eingabeaufforderung, auf den Sie mit der rechten Maustaste klicken. Im Kontextmenü wählen Sie dann den Eintrag **Als Administrator ausführen** aus (siehe Bild 2).

Hier geben Sie nun die folgende Anweisung ein, um LocalDB ohne weitere Rückfragen zu installieren:

```
msiexec /i SqlLocalDB.msi /qn IACCEPTSQLLOCALDBLICENSETERMS=YES
```

Ob **LocalDB** installiert wurde, können Sie nun beispielsweise in der Systemsteuerung unter **Programme und Features** prüfen (siehe Bild 3).

LocalDB testen

Was machen wir nun mit LocalDB? Als Erstes könnten wir herausfinden, wie die Instanz benannt ist, damit wir Verbindungszeichenketten zusammensetzen oder per **SQL**

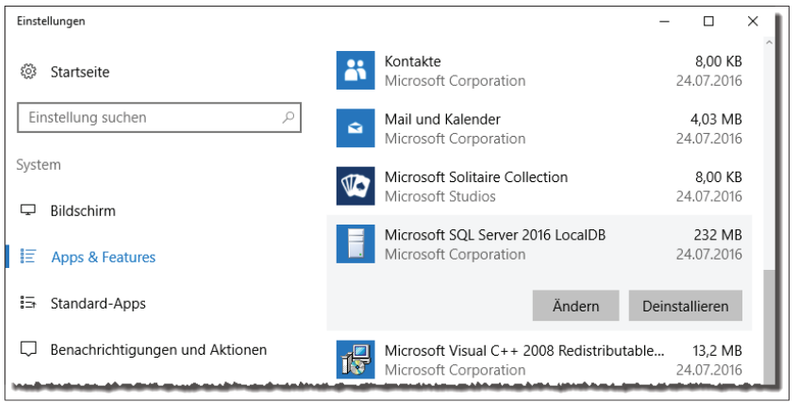


Bild 3: **LocalDB** wurde erfolgreich installiert.

Server Management Studio darauf zugreifen können. Dazu bleiben Sie gleich in der Eingabeaufforderung und setzen beispielsweise den folgenden Befehl ab:

```
sqllocaldb info
```

Dies sollte den Namen der Standardinstanz zurückgeben:

```
MSSQLLocalDB
```

Wenn Sie LocalDB wie oben beschrieben allein installiert haben und Sie diese nicht etwa mit Visual Studio oder einer SQL Server Installation auf den Rechner gebracht haben, erscheint vielleicht auch diese Meldung:

```
Der Befehl "sqllocaldb" ist entweder falsch geschrieben
oder konnte nicht gefunden werden.
```

In diesem Fall wechseln Sie zum Verzeichnis, in dem sich die Datei **sqllocaldb.exe** befindet, normalerweise in diesem hier:

```
C:\Program Files\Microsoft SQL Server\130\Tools\Binn
```

Wenn Sie nicht jedesmal in dieses Verzeichnis wechseln wollen, können Sie das Verzeichnis auch in die Umgebungsvariable **Path** aufnehmen. Dadurch fügt Windows das Verzeichnis zum Suchpfad hinzu, der immer bei Eingabe von Befehlen in die Kommandozeile durchsucht wird.

Geben Sie im Feld **Web und Windows durchsuchen** den Ausdruck **Path** ein und klicken Sie dann oben auf **Systemumgebungsvariablen bearbeiten**. Klicken Sie im nun erscheinenden Dialog **Systemeigenschaften** auf die Schaltfläche **Umgebungsvariablen**.

Es erscheint ein weiterer Dialog namens **Umgebungsvariablen**. Je nachdem, ob Sie den Pfad zur Datei **sqllocaldb.exe** nur für den aktuell angemeldeten Benutzer oder systemweit eintragen möchten, markieren Sie im oberen oder im unteren Bereich den Eintrag **Path**.

Klicken Sie dann auf **Hinzufügen** und kopieren Sie den Pfad zur Datei **sqllocaldb.exe** dort hinein. Nach dieser Aktion müssen Sie sich gegebenenfalls abmelden und erneut anmelden oder das System neu starten. Wenn Sie nun **sqllocaldb info** in die Eingabeaufforderung eingeben, erhalten Sie das Ergebnis aus Bild 4. Die Ausgabe entspricht dem Namen der Standardinstanz von **localdb**.

Damit wissen Sie auch gleich, wie Sie dieses Programm als Datenbankserver etwa in Verbindungszeichenfolgen oder auch im SQL Server Management Studio ansprechen können – nämlich mit **(localdb)\MSSQLLocalDB**.

Wenn Sie eine Übersicht aller Befehle von **sqllocaldb** wünschen, geben Sie einfach **sqllocaldb** in die Eingabeaufforderung ein.

Sollten Sie eine neue Instanz von **localdb** erzeugen wollen, erledigen Sie dies mit dem **create**-Parameter und geben den Namen der benannten Instanz an:

```
C:\Users\andre>sqllocaldb create MeineInstanz
Die LocalDB-Instanz "MeineInstanz" wurde mit Version
13.0.1601.5 erstellt.
```

Die neue Instanz wird dann mit dem Parameter **info** ausgegeben:

```
C:\Users\andre>sqllocaldb info
MeineInstanz
MSSQLLocalDB
```

Um die Version von **localdb** zu erhalten, verwenden Sie den Parameter **versions**:

Bild 4: Test der Anwendung **sqllocaldb.exe**

```
C:\Users\andre>sqllocaldb versions
Microsoft SQL Server 2016 (13.0.1601.5)
```

Um eine Instanz zu starten, verwenden Sie den Parameter **start** und geben den Namen der benannten Instanz an:

```
C:\Users\andre>sqllocaldb start "MeineInstanz"
Die LocalDB-Instanz "MeineInstanz" wurde gestartet.
```

Um die Standardinstanz zu starten, lassen Sie hier einfach den Namen der Instanz weg:

```
C:\Users\andre>sqllocaldb start
Die LocalDB-Instanz "MSSQLLocalDB" wurde gestartet.
```

Sollten Sie die Instanz nicht mehr benötigen, können Sie diese mit dem **delete**-Parameter löschen (mit Angabe des Namens für eine benannte Instanz, ohne Name für die Standardinstanz). Wenn die Instanz noch läuft, liefert dies einen Fehler wie den folgenden:

```
C:\Users\andre>sqllocaldb delete "MeineInstanz"
Fehler beim Löschen der LocalDB-Instanz "MeineInstanz".
Fehler:
Der angeforderte Vorgang für die LocalDB-Instanz kann
nicht ausgeführt werden, weil die angegebene Instanz gera-
de verwendet wird. Beenden Sie die Instanz, und wiederho-
len Sie den Vorgang.
```

Wenn die Instanz noch läuft, müssen Sie diese zuerst beenden. Dies gelingt mit dem **stop**-Parameter:

```
C:\Users\andre>sqllocaldb stop "MeineInstanz"
Die LocalDB-Instanz "MeineIns-
tanz" wurde beendet.
```

Danach gelingt auch das Löschen der Instanz:

```
C:\Users\andre>sqllocaldb delete
"MeineInstanz"
Die LocalDB-Instanz "MeineInstanz" wurde gelöscht.
```

Die Standardinstanz **SQLLocalDB** können Sie zwar auch löschen, aber diese bleibt dennoch erhalten.

SQL Server Management Studio 2016

Um komfortabel auf die Datenbanken einer LocalDB-Installation zugreifen zu können, steht Ihnen neben Visual Studio auch noch das SQL Server Management Studio zur Verfügung.

Wenn Sie LocalDB in einer aktuellen Variante nutzen, kann es bei Verwendung einer älteren Version von SQL Server Management Studio passieren, dass verschiedene Funktionen nicht zur Verfügung stehen – in unserem Fall konnten etwa keine Tabellen zu einer Datenbank hinzugefügt oder geändert werden.

Beim Einsatz von LocalDB in der Version 13 benötigen Sie beispielsweise das SQL Server Management Studio 2016, das Sie hier herunterladen können:

```
https://msdn.microsoft.com/de-de/library/mt238290.aspx
```

Sollte dieser Link nicht mehr zur Verfügung stehen, suchen Sie einfach mit den Stichworten **SQL Server Management Studio 2016 Download** bei Google nach dem aktuellen Download-Link.

Dies lädt die Datei **SSMS-Setup-DEU.exe** auf Ihren Rechner, die Sie nun installieren können. Die Installation verläuft weitgehend ohne Rückfragen, braucht aber ungleich länger als die zügige Installation von LocalDB. Danach können Sie aber

Datenbanken erstellen mit dem SSMA

Wer von Access kommt und nun Anwendungen auf Basis von WPF, C# und dem Entity Framework erstellen möchte, wird feststellen, dass dies mit Access-Datenbanken als Backend nicht gelingt – es lässt sich schlicht kein Entity Framework für Access-Datenbanken erstellen. Da kann es dann hilfreich sein, wenn Sie, nachdem wir uns bereits mit der Migration von Access zum SQL Server beschäftigt haben, auch neue Datenbanken im SQL Server erstellen können. Dieser Artikel zeigt die wichtigsten Techniken für die Erstellung von Datenbanken über die Benutzeroberfläche des Microsoft SQL Server Management Studios (SSMA).

Mit der für kleine Unternehmen kostenfreien Community Edition des SQL Servers oder auch mit den kostenpflichtigen Versionen kommt auch das SQL Server Management Studio auf Ihren Rechner. Das ist ein Tool, mit dem Sie Datenbanken verwalten können – und dazu gehört natürlich auch das Erstellen neuer Datenbanken.

Datenbanken für das Entity Framework

Wenn Sie einmal eine Migration einer reinen Access-Datenbank zu einer Kombination aus Access-Frontend und SQL Server-Backend durchgeführt haben, waren Sie sicher nicht mit der Migration der Tabellen zufrieden. Stattdessen haben Sie auch noch Stored Procedures, Trigger und weitere SQL Server-spezifische Objekte erstellt, um Geschäftslogik auf den SQL Server zu transferieren.

Bei Verwendung des Entity Frameworks ist das nicht notwendig. Hier reicht es aus, wenn die Tabellen mit den Daten auf dem SQL Server landen, die Geschäftslogik soll in anderen Schichten der Anwendung untergebracht werden. Wie und wo dies geschieht, werden wir uns in späteren Artikeln ansehen.

Fürs Erste reicht es uns, zu wissen, dass wir vom Entity Framework aus direkt auf die Tabellen der Datenbank zugreifen und dass diese die Informationen liefern, um das Entity Framework auf Basis der Tabellen und ihrer Beziehungen untereinander aufbauen zu können.

Dies ist erfordlich, wenn Sie das Entity Framework auf Basis der Tabellenstruktur erstellen wollen. Der Assistent kann dann Informationen über Primärschlüssel- und Fremdschlüsselfelder direkt in entsprechende Beziehungen umwandeln.

Benennung von Tabellen und Feldern

Der Entity Framework-Assistent übernimmt die Bezeichnungen von Tabellen und Feldern für die zu erstellenden Klassen. Das heißt, dass Sie sich wohl oder übel von einigen unter Access liebgewonnenen Gewohnheiten verabschieden sollten.

Dort haben Sie Tabellen vermutlich mit dem Präfix **tbl** ausgestattet und den Rest der Tabelle entsprechend dem Plural der enthaltenen Elemente benannt, also beispielsweise **tblKunden**.

Dies würde dazu führen, dass die Klasse im Entity Framework, welche die Datensätze der Tabelle **tblKunden** abbildet, automatisch auch **tblKunden** heißen würde. Das entspricht nicht den gängigen Benennungen für Klassen, die erstens im Singular und zweitens ohne Präfix erscheinen sollten – in diesem Fall also einfach **Kunde**.

Nun gibt es im Entity Framework-Assistenten sogar eine Option, mit der man abfangen kann, dass die Tabellen entsprechend dem Plural der gespeicherten Elemente benannt sind.

Dabei ist die Option allerdings nur auf den Plural für englische Benennungen optimiert (und somit auf das angehängte **s** für den Plural), sodass **Customers** in **Customer** geändert werden kann. Bei **Kunden** und **Kunde** sieht es anders aus, sodass wir zwei Möglichkeiten für unsere deutschsprachig

benannten Tabellen haben: entweder wir benennen diese direkt im Singular oder wir vergeben Plural-Namen. In beiden Fällen müssen später etwa bei Verwendung im Entity Framework noch die eine oder andere individuelle Anpassung vornehmen.

Wenn Sie eine Anwendung auf einer bereits vorhandenen Datenbank aufbauen müssen, bleibt Ihnen aber ohnehin keine Wahl – dann sind spätere Optimierungen sowieso unumgänglich, wenn Sie mit vernünftigen Klassennamen arbeiten wollen.

In diesem Artikel wollen wir uns auf eine Datenbank konzentrieren, die auf der neuen **LocalDB**-Instanz läuft – ein Feature, das seit der Version 2012 mit Visual Studio geliefert wird.

Um mit dem SQL Server Management Studio zu arbeiten, benötigen Sie eine entsprechende Installation. Die Software können Sie auf verschiedene Arten auf den Rechner bringen, beispielsweise durch Installation eines entsprechenden SQL Server-Pakets (siehe Artikel **SQL Server 2014 Express installieren**).

Wenn Sie, wie im vorliegenden Artikel beschrieben, eine neue SQL Server-Datenbank mit der Datenbank-Engine **LocalDB** erstellen wollen, brauchen Sie nicht das komplette SQL Server-Paket, das Ihnen nicht nur eine Menge Speicherplatz auf der Festplatte belegt, sondern auch eine Menge Zeit bei der Installation raubt. Also installieren Sie einfach nur das SQL Server Management Studio, das Sie unter dem folgenden Link finden:

```
https://msdn.microsoft.com/de-de/library/
mt238290.aspx
```

Dies liefert die Datei **SSMS-Setup-DEU.exe**, welche Sie einfach ausführen.

Mit LocalDB verbinden

Um sich mit dem als Datenbankserver verwendeten LocalDB zu verbinden (oder auch mit einem anderen Datenbankserver wie etwa der Vollversion des SQL Servers), starten Sie das

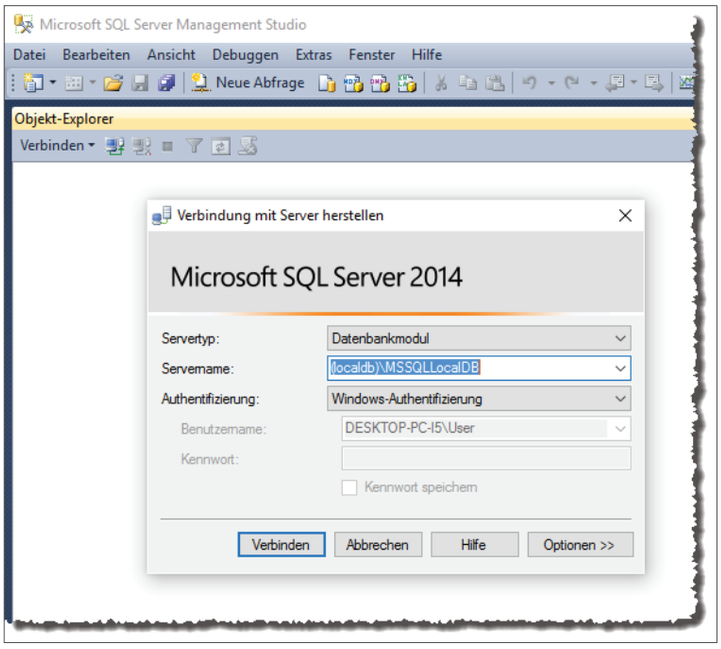


Bild 1: Mit dem SQL Server verbinden – hier mit **LocalDB**

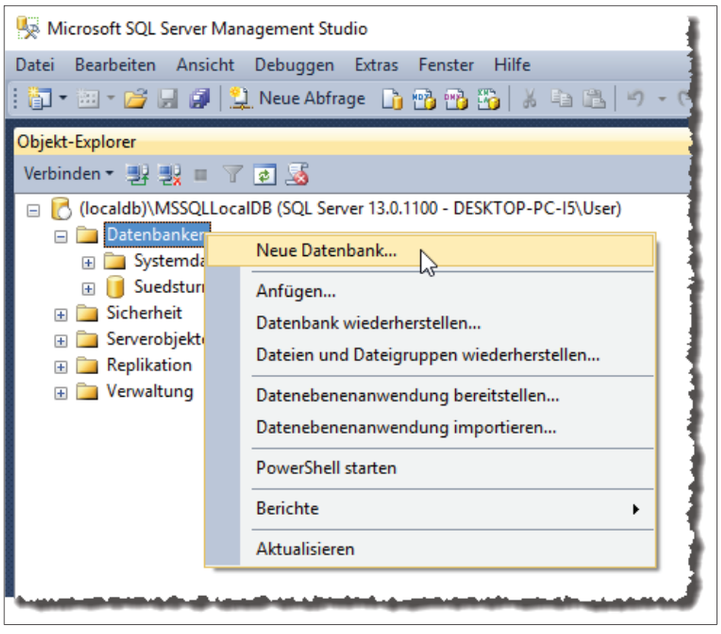


Bild 2: Anlegen einer neuen Datenbank

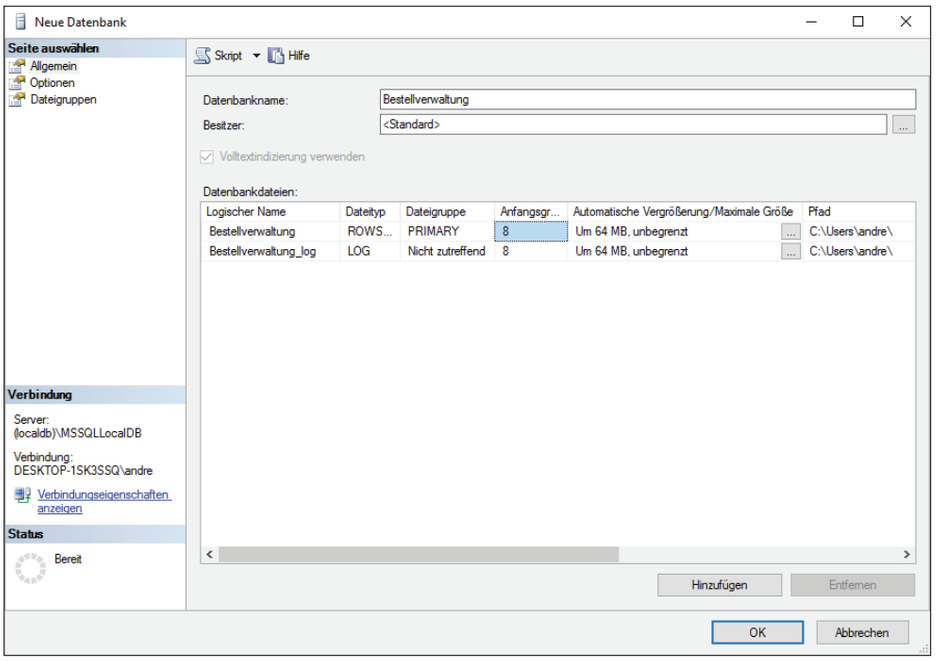


Bild 3: Angabe des Datenbanknamens

SQL Server Management Studio. Es erscheint der Dialog aus Bild 1, der Sie zur Eingabe des Servernamens auffordert.

Verwenden Sie die Standardinstanz von **LocalDB**, geben Sie einfach folgenden Ausdruck ein:

```
(localdb)\MSSQLLocalDB
```

Sie können auch das Kombinationsfeld aufklappen, um die Suche nach den verfügbaren SQL Servern zu starten, aber dies nimmt eine Menge Zeit in Anspruch – schneller geht es mit der direkten Eingabe.

Für den Zugriff auf eine benannte Instanz von LocalDB geben Sie statt **MSSQLLocalDB** den entsprechenden Namen der Instanz an.

Datenbank anlegen

Nach erfolgreichem Verbindungsaufbau können Sie gleich eine neue Datenbank anlegen, und zwar über den Eintrag **Neue Datenbank** des Kontextmenüs des Elements **Datenbanken** im Objekt-Explorer (siehe Bild 2).

Der folgende Dialog namens **Neue Datenbank** erwartet die Eingabe der Bezeichnung für die Datenbank (siehe Bild 3). Geben Sie hier den Text **Bestellverwaltung** an und klicken Sie auf **OK**, um die Änderung zu übernehmen und den Dialog zu schließen. Unter **Pfad** können Sie außerdem angegeben, in welchem Verzeichnis die beiden Datenbankdateien gespeichert werden sollen.

Tabellen anlegen

Die erste Tabelle legen Sie nun an, indem Sie unter **Bestellverwaltung** mit der rechten Maustaste auf das Element **Tabellen** klicken und den Eintrag **Tabelle...** des Kontextmenüs auswählen (siehe Bild 4).

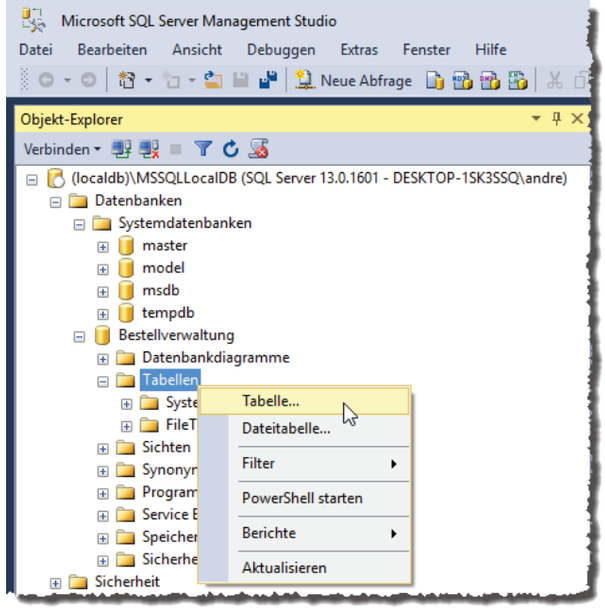


Bild 4: Neue Tabelle anlegen

Es erscheint nun die Entwurfsansicht der neuen Tabelle.

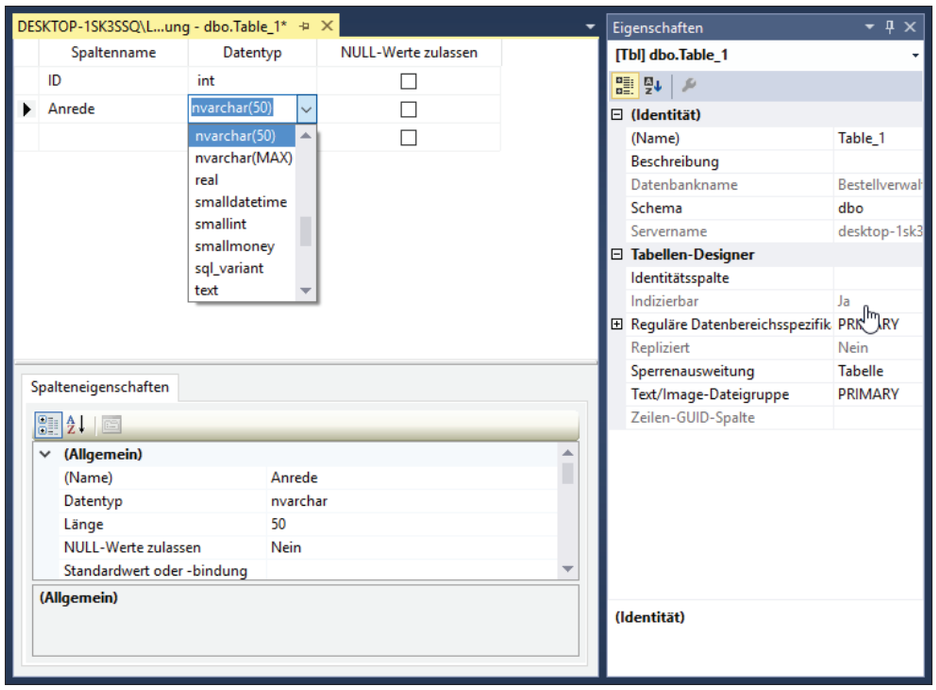


Bild 5: Entwurfsansicht einer neuen Tabelle

Diese liefert drei wichtige Bereiche (siehe Bild 5):

- Links oben finden Sie die Liste der Felder mit den Feldnamen, Felddatentypen und der Angabe, ob für das Feld Nullwerte zulässig sind.
- Links unten zeigt der SSMA die Eigenschaften des aktuell markierten Feldes beziehungsweise der Spalte an. Hier finden sich noch weitere Eigenschaften zu den einzelnen Feldern.
- Rechts sehen Sie die Eigenschaften der aktuell bearbeiteten Tabelle. Hier tragen Sie beispielsweise den Tabellennamen, eine Beschreibung et cetera ein.

Erste Tabelle: Anreden

Die erste Tabelle soll gemäß der weiter oben besprochenen Konvention nach dem enthaltenen Element im Plural benannt werden, im Falle einer Tabelle zum Speichern von Anreden also schlicht **Anreden**. Tragen Sie diesen Namen für die Eigenschaft **Name** der Tabelle im rechten Bereich ein. Der Artikel **Datenzugriff per Entity Framework** zeigt, wie wir die Tabellen der hier erstellten Datenbank per Entity Framework nutzen und welche Änderungsschritte notwendig sind, wenn Sie die Tabelle anders als **Anreden** nennen, aber im Entity Framework ein Objekt namens **Anrede** verwenden wollen.

Primärschlüsselfeld anlegen

Als erstes Feld legen wir ein Primärschlüsselfeld an. Auch hier sollten Sie, genau wie beim Tabellennamen, eine Konvention befolgen. Die beiden gängigsten Varianten lauten, für das Primärschlüsselfeld einer Tabelle immer die Bezeichnung **ID** zu verwenden oder noch den Namen des enthaltenen Elements voranzustellen, also **AnredeID**. Wie auch immer Sie sich entscheiden, sollten Sie sicherstellen, dass Sie bei Ihrer selbst gewählten Benennungskonvention bleiben.

Wenn man sich jedoch anschaut, wie die verschiedenen Varianten später beim objektorientierten Zugriff auf die jeweiligen Objekte und Eigenschaften aussehen, liegt die Wahl der kürzeren Variante, nämlich ID für das Primärschlüsselfeld, nahe. Dann würde der Zugriff etwa über **Anrede.ID** erfolgen und nicht über **Anrede.AnredeID**, was ja doch irgendwie doppelt gemoppelt ist.

Das Primärschlüsselfeld soll keine Null-Werte zulassen, also deaktivieren Sie das Kontrollkästchen für die Zeile **AnredeID** und die Spalte **NULL-Werte zulassen**.

Den Datentyp des Primärschlüsselfeldes stellen Sie auf **int** ein. Danach wollen wir noch die Autowert-Eigenschaft

umsetzen. Dazu wechseln Sie zum Bereich links unten, wo sich die Spalteneigenschaften befinden. Dort finden Sie die Eigenschaft **Identitätsspezifikation**, für die Sie weitere Untereigenschaften aufklappen können. Dort legen Sie **(Ist Identity)** auf **Ja** fest und behalten die Werte für **ID-Ausgangswert** und **ID-Inkrement** bei (siehe Bild 6).

Damit ist das Feld **AnredeID** theoretisch aber immer noch kein Primärschlüsselfeld. Dies stellen Sie schließlich über den Kontextmenüeintrag **Primärschlüssel festlegen** ein (siehe Bild 7).

Die als Primärschlüsselfeld definierte Spalte sieht schließlich wie in Bild 8 aus.

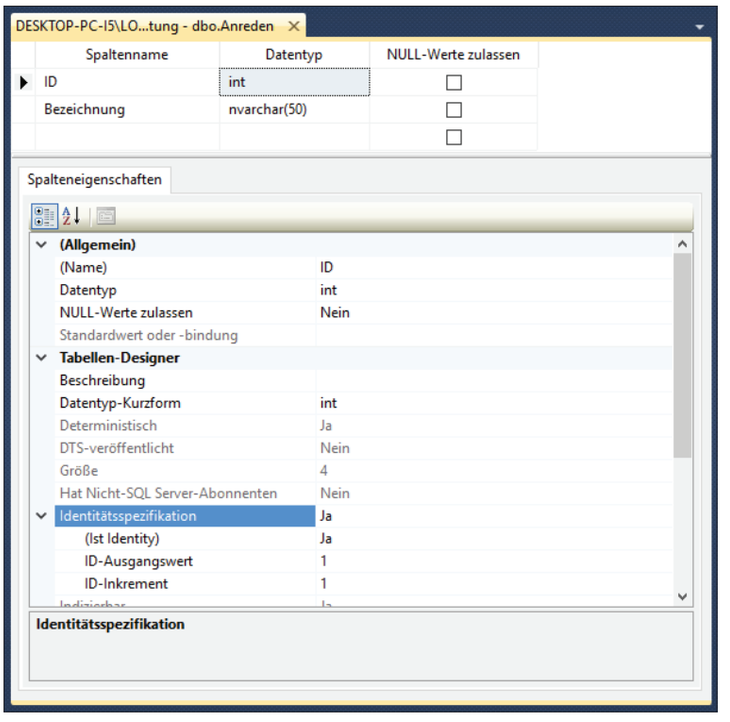


Bild 6: Festlegen der Identitätsspezifikation

Weiteres Feld definieren

Das zweite Feld dieser Tabelle soll die Anrede aufnehmen. Hierfür legen wir den Datentyp **nvarchar(50)** fest. Auch dieses Feld soll keine Null-Werte enthalten. Außerdem definieren wir für diese Spalte die Bezeichnung **Anrede** – welche auch sonst? Nun, es gäbe noch die Möglichkeit, etwa auf einen Wert wie **Name** oder **Bezeichnung** auszuweichen. Unter Access gab es aber immer Probleme mit Feldnamen, die wie Schlüsselwörter verschiedener Funktionen hießen – und **Name** ist nun einmal die Bezeichnung einiger Eigenschaften im Access-Bereich.

Sind bei Verwendung etwa als Datenquelle für das Entity Framework ähnliche Probleme zu erwarten? Ja – es gibt eine Einschränkung bei der Vergabe von Feldnamen, zumindest wenn diese später als Eigenschaften im Entity Framework genutzt werden sollen: Ein Feldname darf dort nicht mit dem Namen der Klasse beziehungsweise der Entität übereinstimmen, sonst wird dieser automatisch umbenannt. Wenn wir die Tabelle **Anreden** mit den beiden Feldern **AnredeID** und **Anrede** als Datenquelle einer Entity Framework-Klasse nutzen wollen, würde das Feld **Anrede** automatisch in die Eigenschaft **Anrede1** umbenannt, da die Klasse dann logischerweise **Anrede** heißen soll. Auch der Versuch, diese Eigenschaft im Entity Framework umzubenennen, scheitert mit der Meldung aus Bild 9.

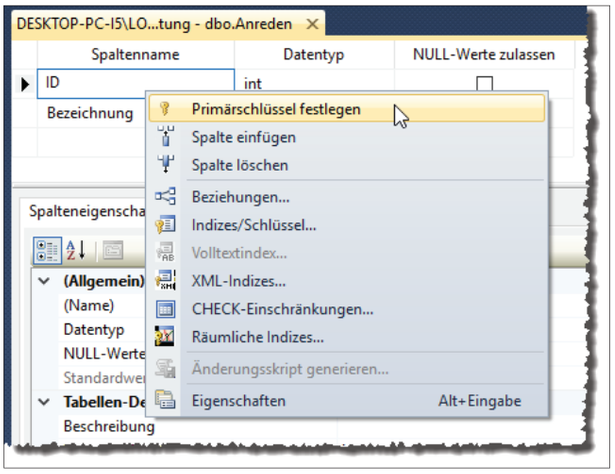


Bild 7: Primärschlüssel der Tabelle festlegen

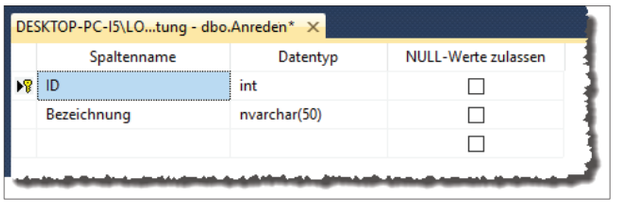


Bild 8: Definierter Primärschlüssel

Welche Schlussfolgerung ziehen wir daraus? Wir verwenden statt der Bezeichnung **Anrede**, die mit dem Entitätsnamen übereinstimmt, eine andere Bezeichnung. In diesem Fall weichen wir auf den Feldnamen **Bezeichnung** aus.

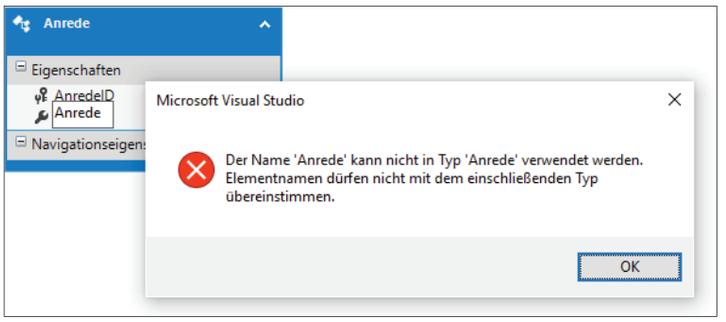


Bild 9: Im Entity Framework dürfen Eigenschaften und Klassen nicht den gleichen Namen verwenden.

Wir haben zwar bereits geeignete Datentypen für die beiden Felder ausgewählt, aber wenn sie von Access kommen, dürfte es Sie interessieren, mit welchen Datentypen unter SQL Server man die Datentypen unter Access ersetzen kann. Daher finden Sie nachfolgende eine Auflistung der passenden Datentypen für Access und SQL Server.

Vor dem Doppelpunkt einer jeden Zeile finden Sie den Datentyp unter Microsoft Access, hinter dem Doppelpunkt haben wir den Datentyp im SQL Server aufgeführt:

- **Autowert** (**Long Integer**): **integer** mit Spaltenoption **IDENTITY**
- **Autowert** (**Replikations-ID**): **uniqueidentifier** mit der Funktion **NEWID()** als Standardwert
- **Zahl** (**Byte**): **tinyint**
- **Zahl** (**Integer**): **smallint**
- **Zahl** (**Long Integer**): **integer**
- **Zahl** (**Single**): **real**
- **Zahl** (**Double**): **float**
- **Zahl** (**Dezimal**): **decimal** oder **numeric**
- **Zahl** (**Replikations-ID**): **uniqueidentifier**
- **Währung**: **money**
- **Ja/Nein**: **bit** mit Option **NOT NULL**
- **Text**: **nvarchar(n)** für Unicode und **varchar(n)** für ANSI; **(n)** steht für die maximale Länge
- **Memo**: **ntext** oder **nvarchar(max)** für Unicode und **text** oder **varchar(max)** für ANSI
- **Hyperlink**: **ntext** oder **nvarchar(max)**, jedoch ohne Hyperlink-Funktionalität
- **Datum/Uhrzeit**: **datetime**
- **OLE-Objekt**: **image** oder **varbinary(max)**
- **Anlage**: **ntext** oder **nvarchar(max)**
- **Text** als **RichText**: **ntext** oder **nvarchar(max)** inklusive HTML-Tags

Zusammengefasst verwenden wir für die Tabelle **Anreden** für das Feld **ID** ein **integer**-Feld mit der oben bereits beschriebenen **IDENTITY**-Eigenschaft und der Primärschlüssel-Eigenschaft. Das Feld Bezeichnung statten wir mit dem Datentyp **nvarchar(50)** aus. Außerdem belassen wir für beide Felder die Eigenschaft **NULL-Werte zulassen** bei dem Wert **Nein**, denn beide Felder müssen für jeden Datensatz unbedingt gefüllt werden.

Datensätze anlegen

Bevor wir Datensätze über eine entsprechende Benutzeroberfläche hinzufügen können, wird es noch ein paar

Artikel dauern, daher fügen wir der Tabelle **Anreden** die benötigten Datensätze über das SQL Server Management Studio hinzu.

Dazu klicken Sie mit der rechten Maustaste auf den Eintrag **dbo.Anreden** im Objekt-Explorer und wählen aus dem Kontextmenü den Befehl **Oberste 200 Zeilen bearbeiten** aus. In der nun erscheinenden Ansicht geben Sie die beiden Werte **Herr** und **Frau** und gegebenenfalls **Firma** in jeweils einem neuen Datensatz ein (siehe Bild 10).

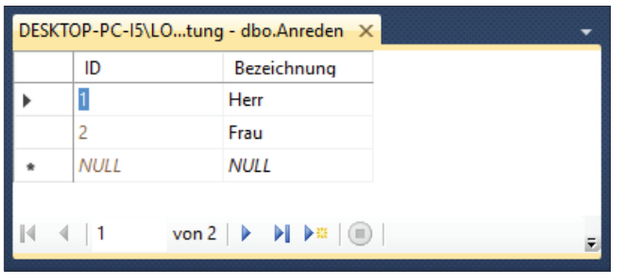


Bild 10: Hinzufügen einiger Datensätze zur Tabelle **Anrede**

Kunden-Tabelle

Als Nächstes erstellen wir die Tabelle zum Speichern der Kunden. Diese Tabelle soll entsprechend unserer Konvention **Kunden** heißen. Das Primärschlüsselfeld nennen wir wieder **ID**, legen den Datentyp auf **int** fest, stellen **Identitätsspezifikation** auf **Ja** ein und aktivieren die Primärschlüssel-Eigenschaft.

Die ersten Felder heißen **Firma**, **Vorname** und **Nachname** und erhalten jeweils den Datentyp **nvarchar(255)**. Außerdem aktivieren wir für alle drei die Eigenschaft **NULL-Werte zulassen**. Immerhin kann es sein, dass für einen Kunden nur der Firmenname oder nur eine Privatperson angegeben werden soll.

Fehlt noch ein Feld, mit dem wir die Anrede für den jeweiligen Kunden auswählen. Dieses Feld soll einen der Primärschlüsselfeldwerte der Tabelle **Anreden** aufnehmen und somit als Fremdschlüsselfeld einer Beziehung zwischen diesen beiden Tabellen dienen.

Im Gegensatz zum Primärschlüsselfeld, das immer **ID** heißen soll, legen wir für Fremdschlüsselfelder eine andere Konvention fest: Die Bezeichnung besteht aus dem Namen der Tabelle, welche den Primärschlüsselwert zu der Beziehung beiträgt, und der Zeichenfolge **ID**.

Im Falle des Feldes zum Festlegen einer Anrede heißt dieses also etwa **AnredeID**. Damit wir über dieses Feld eine Beziehung zwischen den Tabellen herstellen können, muss es den

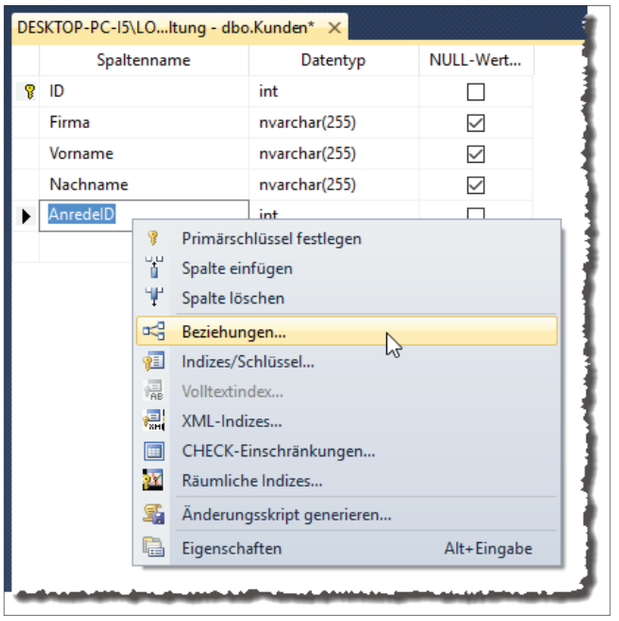


Bild 11: Aufrufen des Dialogs zum Verwalten von Beziehungen

gleichen Datentyp wie das Primärschlüsselfeld aufweisen, in diesem Fall also **int**.

Wie aber stellen wir nun die Beziehung zwischen den beiden Tabellen her? Dies gelingt auch vom Entwurf der Tabelle aus, die das Fremdschlüsselfeld der Beziehung enthalten soll, in diesem Fall also **Kunden**. Klicken Sie hier in der Entwurfsansicht mit der rechten Maustaste auf die Liste der Felder und betätigen Sie den Befehl **Beziehungen** (siehe Bild 11).

Der nun erscheinende Dialog mit der Überschrift **Fremdschlüsselbeziehungen** sieht wie in Bild 12 aus – zumindest, nachdem Sie das Feld **Tabellen- und Spaltenspezifikationen** aktiviert haben. Vorher findet man recht wenig Mög-

SQL Server-Interaktion mit dem Profiler verfolgen

Wenn Sie per ADO.NET, LINQ to Entities oder anderen Zugriffstechniken auf die Datenbank eines SQL Servers zugreifen, werden die von Ihnen programmierten Codezeilen meist im Hintergrund in SQL-Befehle umgewandelt. In vielen Fällen kann es interessant sein, was dort tatsächlich geschieht. Ein geeignetes Werkzeug ist der SQL Server Profiler, der zwar demnächst ausläuft, aber eine schnelle Möglichkeit der Nachverfolgung liefert.

Wir wollen in diesem Artikel die schnelle und unkomplizierte Verfolgung der Zugriffe auf den SQL Server beschreiben. Dazu ist eine Installation des SQL Server Profilers in einer aktuellen und dem SQL Server entsprechenden Version Voraussetzung. Nachdem Sie den SQL Server Profiler gestartet haben, erwartet Sie zunächst das leere Anwendungsfenster (siehe Bild 1). Hier klicken Sie direkt auf die Schaltfläche **Neue Ablaufverfolgung**.

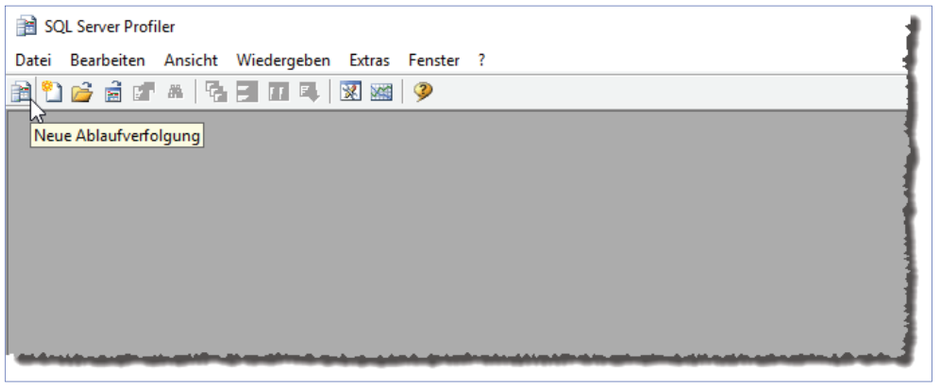


Bild 1: Der Profiler nach dem Start der Anwendung

Daraufhin erscheint der Dialog aus Bild 2, mit dem Sie sich an der SQL Server-Instanz anmelden, die Sie untersuchen möchten. In diesem Fall geht es um eine Instanz von LocalDB in der Version 11.0. Nach der Auswahl/Eingabe des Servernamens klicken Sie auf die Schaltfläche **Verbinden**.

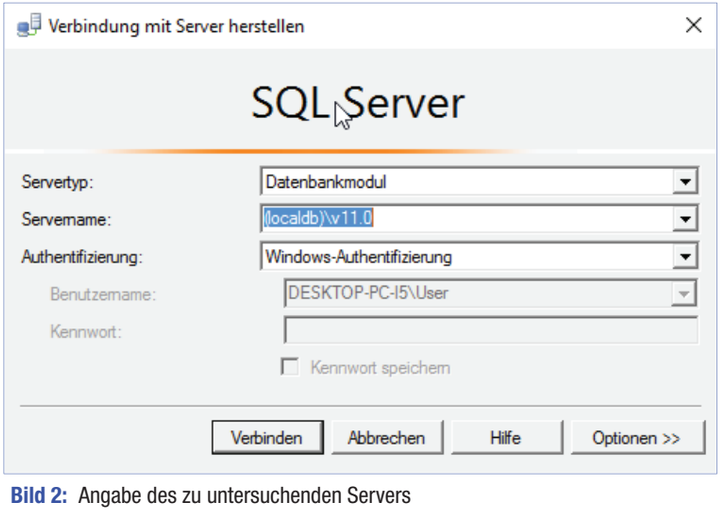


Bild 2: Angabe des zu untersuchenden Servers

Damit erscheint der Dialog **Ablaufverfolgungseigenschaften**, der einige Einstellungen ermöglicht (siehe Bild 3). Die erste ist die Vergabe eines Namens für die Ablaufverfolgung. Außerdem stellen Sie ein, ob die protokollierten Daten in einer Datei oder in einer SQL Server-Tabelle gespeichert werden sollen. Sollten Sie letztere Variante wählen und wollen Sie damit die Performance untersuchen, sollten Sie eine Tabelle einer Datenbank verwenden, die von einer anderen SQL Server-Instanz verwaltet wird.

Dazu öffnet der Profiler, sobald Sie die Option **In Tabelle speichern** gewählt haben, direkt einen weiteren Dialog zum Herstellen einer Verbindung mit einem SQL Server, sodass Sie die Gelegenheit haben, auch eine alternative Instanz auszuwählen.

Die SQL-Anweisungen zum Speichern der Profiler-Daten werden übrigens nicht mitgespeichert.

Das Speichern in einer SQL Server-Tabelle ist dann notwendig, wenn Sie viele Daten oder einen längeren Zeitraum beobachten wollen – die Darstellung im Profiler selbst ist nicht dazu geeignet, beispielsweise nach bestimmten Tabellen oder Ereignissen zu filtern. Für die im Rahmen des Zugriffes durch die in diesem Magazin beschriebenen Beispiele reicht die Darstellung direkt im Profiler allerdings aus.

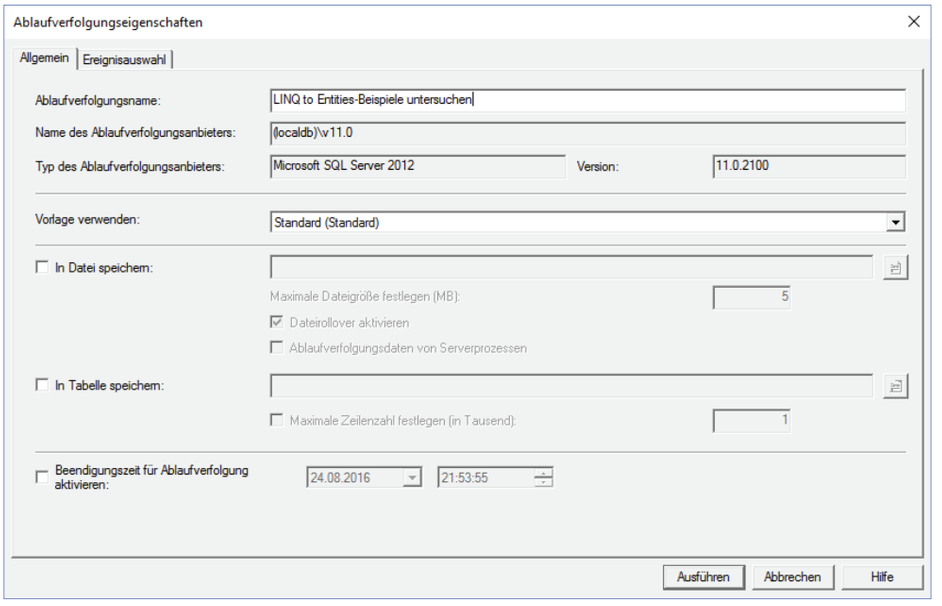


Bild 3: Eingabe der Ablaufverfolgungseigenschaften

Nach Datenbank filtern

Interessant wird es, wenn Sie die zu untersuchenden Zugriffe auf die Datenbank auf einer Server-Instanz ausführen, die noch andere, gegebenenfalls rege frequentierte Datenbanken verwaltet. In diesem Fall müssen Sie noch festlegen, dass der Profiler nur die Zugriffe auf die von Ihnen verwendete Datenbank protokolliert. Dies erledigen Sie auf der zweiten Registerseite des Dialogs **Ablaufverfolgungseigenschaften** (siehe Bild 4).

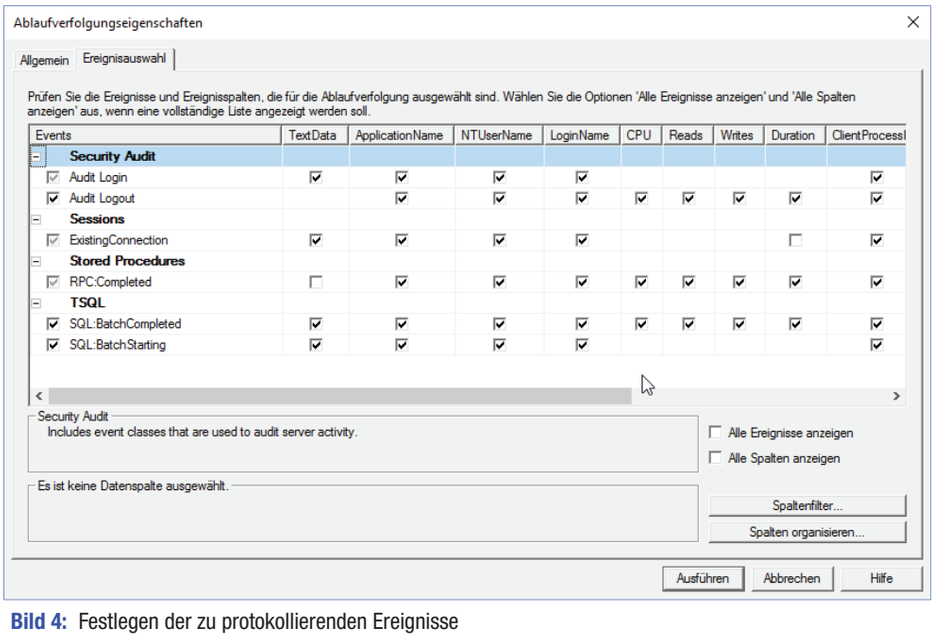


Bild 4: Festlegen der zu protokollierenden Ereignisse

Diese Registerseite enthält eine Matrix aus Ereignissen und den für die jeweiligen Ereignisse aufgezeichneten Daten. Wenn Sie festlegen möchten, dass nur die Zugriffe auf eine bestimmte Datenbank protokolliert werden sollen, müssen Sie zunächst einmal die ID dieser Datenbank herausfinden und einen entsprechenden Filter definieren.

Dazu öffnen Sie das SQL Server Management Studio und verbinden sich mit dem

Entity Data Model für eine Datenbank erstellen

In den vorangegangenen Ausgaben von DATENBANKENTWICKLER haben Sie bereits erfahren, wie Sie per ADO.NET auf die Daten der Tabellen einer Datenbank zugreifen. Nun nutzen wir nicht mehr direkt ADO.NET, sondern das Entity Framework als Datenlieferant, welches eine ganz andere Art des Zugriffs ermöglicht. Dieser Artikel zeigt zunächst, wie Sie die Verbindung zu einer Beispieldatenbank einrichten und dann die für den Zugriff per C# oder die Anzeige in WPF-Fenstern und -Steuerelementen nötigen Objekte erstellen.

Im Artikel **Datenbanken erstellen mit dem SSMA** haben wir eine kleine Beispieldatenbank mit dem SQL Server Management Studio auf Basis von **LocalDB** erstellt. Auf diese wollen wir nun mit dem Entity Framework zugreifen, um die Vorbereitung für den Zugriff per C# oder die Anzeige in mit WPF definierten Benutzeroberflächen zu erledigen.

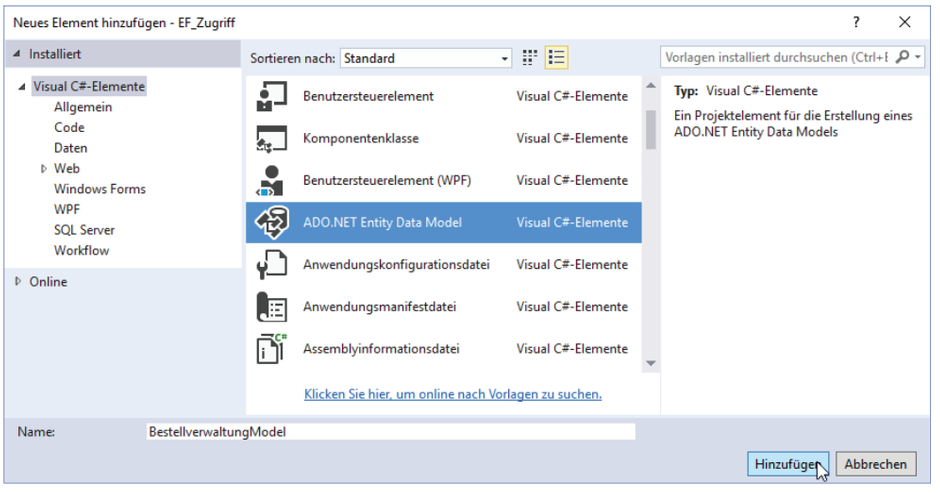


Bild 1: Hinzufügen eines neuen Entity Data Models

Entity Data Model anlegen

Als Erstes erstellen wir dazu das Entity Data Model auf Basis der Tabellen der Beispieldatenbank. Dazu legen Sie ein neues Projekt des Typs **C#|Konsolenanwendung** an und nennen es **EF_Zugriff**.

Wählen Sie dazu im Objektmappen-Explorer auf dem Kontextmenü des Projekts den Eintrag **Hinzufügen|Neues Element** aus. Der Typ des gewünschten neuen Objekts heißt **ADO.NET Entity Data Model**, der Name soll **BestellverwaltungModel** lauten (siehe Bild 1).

Im nun erscheinenden Dialog **Assistent für Entity Data Model** wählen Sie unter dem Schritt **Modellinhalte auswählen** den Eintrag **EF Designer aus Datenbank** aus, da wir ja bereits eine Datenbank vorbereitet haben (siehe Bild 2).

Daraufhin erscheint der Dialog-Schritt **Wählen Sie Ihre Datenverbindung aus**. Hier klicken Sie auf **Neue Verbindung...** und wählen im folgenden Dialog **Datenquelle auswählen** den Eintrag **Microsoft SQL Server** aus (siehe Bild 3).

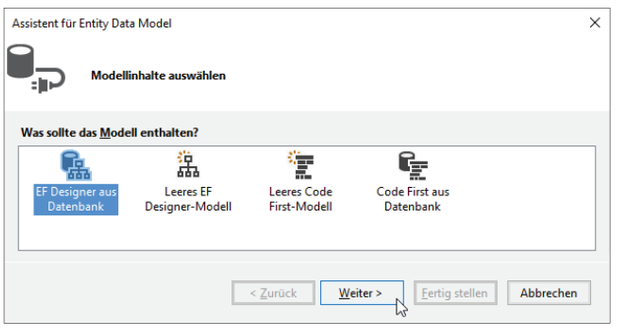


Bild 2: Art des Modellinhalts

Tragen Sie dort den Namen des Servers ein oder wählen Sie ihn aus der Liste aus, soweit vorhanden. Wenn Sie, wie oben vorgeschlagen, LocalDB verwenden und eine einzige Instanz von LocalDB auf dem Rechner installiert haben, können Sie hier **(localdb)\SQLLocalDB** eintragen, sonst geben Sie die Version mit an – also etwa **(localdb)\v11.0** wie in Bild 4. Wählen Sie dann unter **Mit Datenbank verbinden** die gewünschte Datenbank aus, sofern diese schon von LocalDB verwaltet wird. Anderenfalls können Sie eine **.mdf**-Datei unter **Datenbankdatei anhängen** angeben. Klicken Sie auf **OK**.

Damit kehren Sie zum Schritt **Wählen Sie Ihre Datenverbindung aus** und können dort mit einem Klick auf die Schaltfläche **Weiter >** zum nächsten Schritt wechseln.

Der Assistent fragt Sie nun, ob Sie, falls Sie eine Datenbankdatei angegeben haben, diese in den Projektordner kopieren möchten. Dies macht beispielsweise Sinn, wenn Sie diese Datei mit dem Projekt weitergeben möchten.

Im nächsten Schritt fragt der Assistent Sie unter dem Titel **Wählen Sie Ihre Version**, welche Version des Frameworks

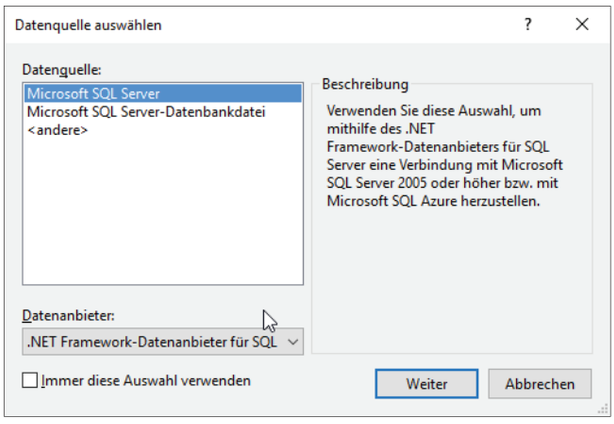


Bild 3: Auswahl der Datenquelle

Sie verwenden möchten. Wir wählen den Wert **Entity Framework 6.x** aus.

Im nächsten Schritt wird es nun interessant: Hier finden wir die Objekte der Datenbank vor, in diesem Fall die wie im Artikel **Datenbanken erstellen mit dem SSMA** angelegten Tabellen (siehe Bild 5). Um alle hier angegebenen Tabellen gleichzeitig zu markieren, klicken Sie einfach auf den Eintrag **Tabellen**. Unten finden Sie noch zwei Optionen, mit denen Sie die Erstellung der Klassen des Entity Frameworks steuern

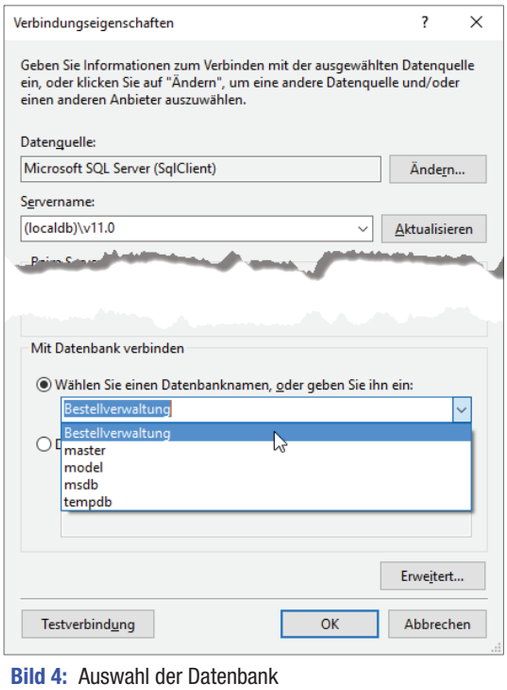


Bild 4: Auswahl der Datenbank

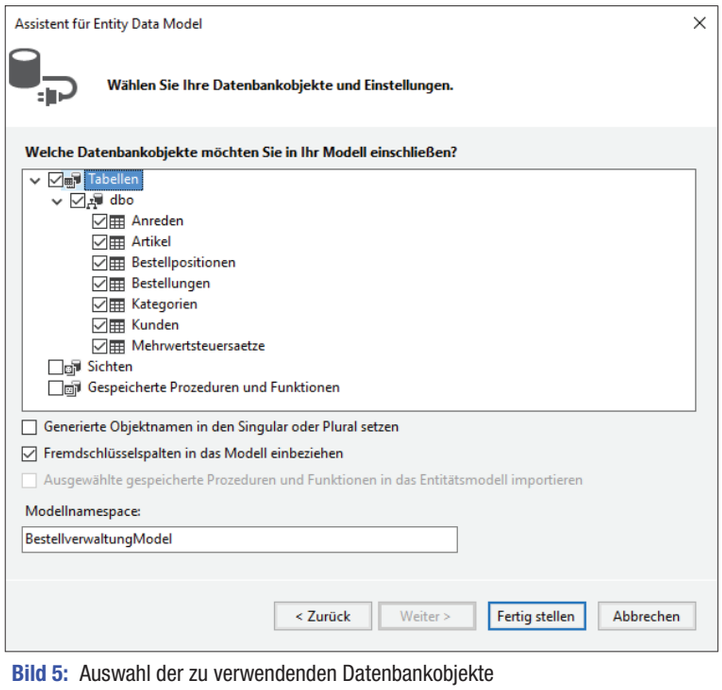


Bild 5: Auswahl der zu verwendenden Datenbankobjekte

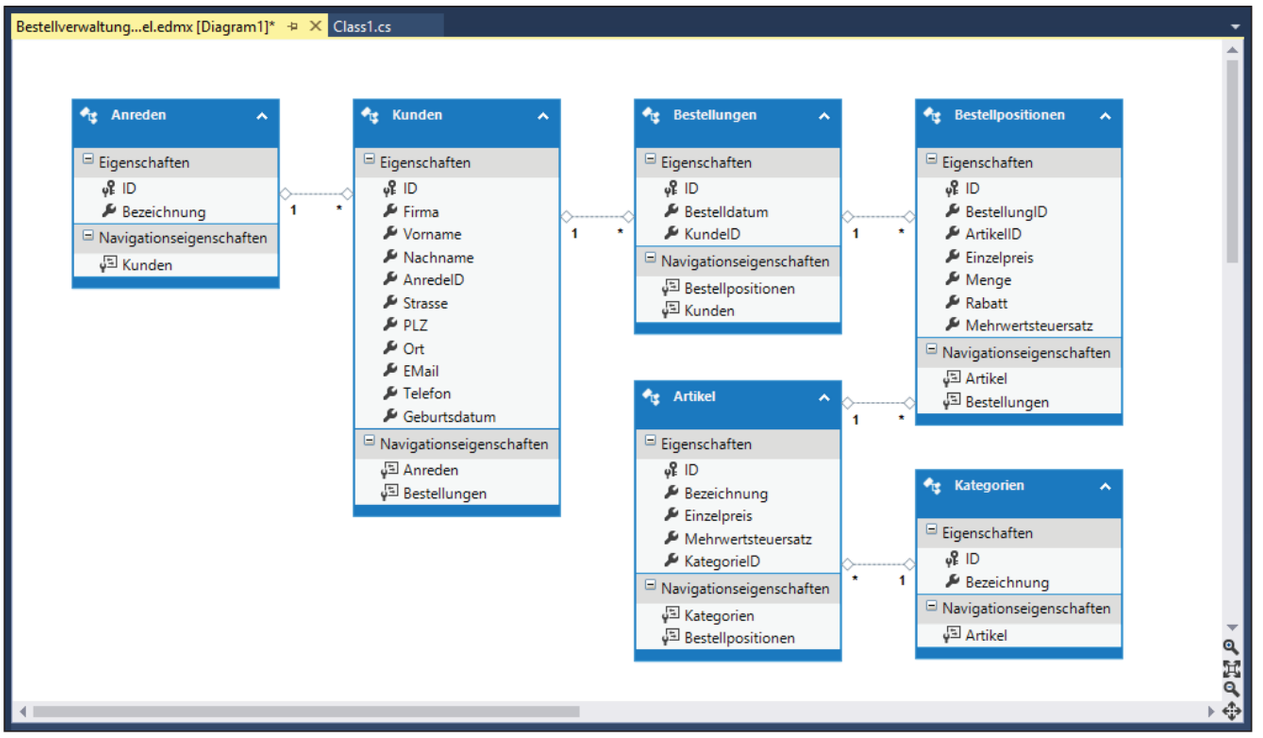


Bild 6: Entity Data Model der Bestellverwaltung

können. Die erste ist für uns nicht relevant, da wir den Tabellen des Datenmodells deutsche Bezeichnungen gegeben haben – bei englischen Bezeichnungen würde einem diese Option etwas Arbeit durch das Anhängen/Entfernen von s bei Singular/Plural abnehmen (**Customer**/**Customers**). Die Option **Fremdschlüsselspalten in das Modell einbeziehen** behalten wir hingegen bei. Außerdem legen wir als Name des Namespaces für das Model den vorgegebenen Wert **BestellverwaltungModel** fest.

Damit können Sie nun auf **Fertig stellen** klicken. Das Ergebnis finden Sie nach einer kurzen Zeit in Bild 6. Die Abbildung zeigt das Entity Data Model auf Basis der angegebenen Tabellen samt ihrer Verknüpfungen beziehungsweise Fremdschlüsselfelder. Die Ansicht sieht etwas anders aus als das Datenmodell, das Sie etwa vom Beziehungsfenster von Access kennen. Hier werden die Beziehungspfeile nicht direkt mit den betroffenen Feldern der verknüpften Tabellen verbunden, sondern einfach an irgendeiner Stelle. Dafür zeigen die einzelnen Entitäten jeweils die verknüpften Elemente unten im Bereich **Navigationseigenschaften** an. Die Felder selbst werden nun **Eigenschaften** genannt.

Entitäten: Aus Plural mach Singular

An dieser Stelle sehen Sie, dass die Entitäten des Entity Data Models genauso heißen wie die Tabellen, von denen sie abgeleitet wurden. Schön und gut – aber wenn wir später im Code auf diese Entitäten zugreifen wollen, was letztlich Klassen sind, auf deren Basis sie Objekte erstellen, dann fällt eines auf: Ein Objekt etwa zur Abbildung eines Kunden soll wohl kaum beispielsweise **Kunden** heißen, sondern **Kunde**, und Gleiches gilt dann auch für die Klassen (nur dass wir Objekte, also Instanzen von Klassen, nicht genau wie die Klassen benennen, sondern diese beispielsweise mit einem kleinen Anfangsbuchstaben versehen – also **kunde** oder **bestellung**).

Nun sollten wir also dafür sorgen, dass die Klassen nicht mehr **Kunden** oder **Bestellungen** heißen, sondern **Kunde** oder **Bestellung**. Dazu benennen Sie gleich hier und jetzt

LINQ to Entities: Daten abfragen

In den vorangegangenen Ausgaben von DATENBANKENTWICKLER haben Sie bereits erfahren, wie Sie per ADO.NET auf die Daten der Tabellen einer Datenbank zugreifen. Nun nutzen wir nicht mehr direkt ADO.NET, sondern das Entity Framework als Datenlieferant, welches eine ganz andere Art des Zugriffs ermöglicht. Dieser Artikel zeigt, wie Sie per C#-Konsolenanwendung auf die per Entity Data Model bereitgestellten Daten zugreifen.

Beispielprojekt

Das Beispielprojekt soll über ein geeignetes Entity Data Model auf die Tabellen einer LocalDB-Datenbank zugreifen. Wie Sie das Entity Data Model erstellen, beschreiben wir im Artikel **Entity Data Model für eine Datenbank erstellen**.

Alle Beispielmethoden finden Sie in der Klasse **EDM_ZugriffPerCSharp_Beispiele.cs**. Um die Beispiele auszuprobieren, starten Sie einfach das Projekt.

Aufruf der Beispielmethoden

Damit wir die nachfolgend programmierten Methoden einfach aufrufen können, verwenden wir wieder die bereits in früheren Artikeln verwendete Klasse **Program** mit einer **Main**-Methode, die alle öffentlichen, statischen Methoden aus öffentlichen Klassen auflistet und zur Ausführung anbietet. In einer neuen Klasse namens **EDM_ZugriffPerCSharp_Beispiele.cs** legen wir dann die später vorgestellten Methoden an.

LINQ to Entities

Die Abfragesprache für den Zugriff auf die Daten des Entity Frameworks und damit auf das Entity Data Model heißt **LINQ to Entities**.

Zugriff auf die Daten der Tabelle Anrede

Unter Access würden Sie nun, wenn Sie alle Datensätze einer Tabelle wie etwa Anrede durchlaufen wollten, ein **Database**-Objekt und ein **Recordset**-Objekt erzeugen, Letzteres mit einer entsprechenden SQL-Anweisung füllen und dann per VBA in einer **Do While**-Schleife durch die einzelnen Datensätze navigieren. Unter C# und dem Entity Framework sieht das etwas anders aus – im Überblick wie in Listing 1. Hier nutzen Sie ein automatisch generiertes Objekt, das die Klasse **DbContext** implementiert, die in unserem Fall beispielsweise **BestellverwaltungEntities** heißt.

Wir erstellen ein neues Objekt auf Basis dieses Typs und speichern es in der Variablen **context**:

```
BestellverwaltungEntities context =
                        new BestellverwaltungEntities();
```

Dann definieren wir eine Variable des Typs **var** und nennen diese **anreden**. Sie soll mit der Liste der Anreden des Objekts **context** gefüllt werden:

```
var anreden = context.Anreden;
```

Schließlich durchlaufen wir die Elemente der Auflistung **anreden** in einer **foreach**-Schleife und weisen das jeweils aktuelle Objekt der Variablen **anrede** zu. Innerhalb der

```
public static void AlleAnredenAusgeben_ForEach() {
    BestellverwaltungEntities context = new BestellverwaltungEntities();
    var anreden = context.Anreden;
    foreach (var anrede in anreden) {
        Console.WriteLine("{0} {1}", anrede.ID, anrede.Bezeichnung);
    }
}
```

Listing 1: Methode zur Ausgabe aller Anreden der Beispieldatenbank

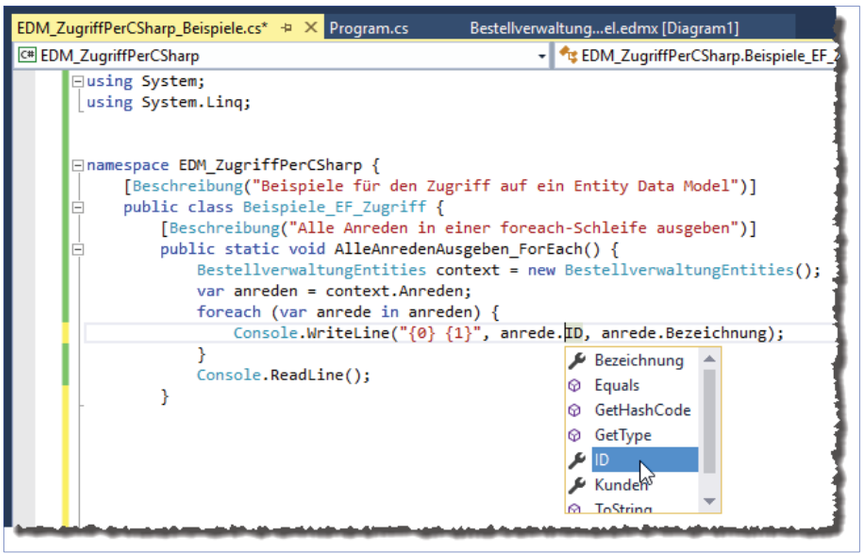


Bild 1: IntelliSense für den Datenzugriff

Schleife geben wir dann die Werte der Eigenschaften **ID** und **Bezeichnung** aller Elemente aus:

```
foreach (var anrede in anreden) {
    Console.WriteLine("{0} {1}",
                      anrede.ID, anrede.Bezeichnung);
}
```

Das Ergebnis sieht etwa so aus:

```
1 Herr
2 Frau
```

Praktischerweise blendet Visual Studio beim Eingeben der Eigenschaften für das Objekt **anrede** per IntelliSense die verfügbaren Einträge ein (siehe Bild 1).

Fertig – damit wäre der erste Zugriff auf die Daten der Tabelle **Anreden** unserer Beispieldatenbank schon erledigt! Im Folgenden sehen wir uns nun weitere Möglichkeiten an.

Verbesserung: using verwenden

Damit die Verbindung zur Datenbank nach dem Abrufen der Daten wieder geschlossen wird, fassen Sie die für den Datenzugriff verwendeten Methoden in ein **using**-Konstrukt ein. Der Grund, ohne weitere Erläuterungen: Die **DbContext**-Klasse, von der unsere Klasse erbt, implementiert das Interface **IDisposable**. Die Nutzung solcher Objekte sollte man innerhalb der **using**-Direktive realisieren. Auf diese Weise wird beim Verlassen des von der **using**-Direktiven eingefassten Abschnitts auf jeden Fall die **Dispose**-Methode aufgerufen, was zum Beispiel die genutzten Ressourcen wieder freigibt. Die verbesserte Version des vorherigen Beispiels sieht nun wie in Listing 2 aus.

Achtung: Die **using**-Variante funktioniert nicht wie gewünscht, wenn Sie die der **using**-Direktiven übergebene Variable bereits vorher deklarieren, wie es hier der Fall ist. Die Variable existiert dann auch noch nach Verlassen von **using**:

```
BestellverwaltungEntities context =
                              new BestellverwaltungEntities()
using (context) {
    ... }
```

```
public static void AlleAnredenAusgeben_ForEach_Besser() {
    using (BestellverwaltungEntities context = new BestellverwaltungEntities()) {
        var anreden = context.Anreden;
        foreach (var anrede in anreden) {
            Console.WriteLine("{0} {1}", anrede.ID, anrede.Bezeichnung);
        }
    }
    Console.ReadLine();
}
```

Listing 2: Methode zur Ausgabe aller Anreden der Beispieldatenbank mit **using**

Kunden sortiert ausgeben

Wenn Sie die Daten einer Tabelle sortiert erhalten möchten, können Sie das durch eine Erweiterung einer Abfrage erreichen. Alle Kunden in unsortierter Reihenfolge würde folgendes Statement liefern:

```
var kunden = context.Kunden;
```

Wenn Sie eine Sortierung hinzufügen wollen, etwa nach dem Feld **Nachname**, haben Sie zwei Möglichkeiten – eine mit der Methodensyntax und eine mit der Abfragesyntax.

Abfragen mit der Methodensyntax

Für beide fügen Sie zunächst eine Referenz auf den Namespace **System.Linq** zur Klassendatei hinzu:

```
using System.Linq;
```

Nun erstellen Sie eine Methode wie in Listing 3. Der Unterschied zur Ermittlung der unsortierten Kunden liegt in der folgenden Zeile:

```
var kunden = context.Kunden.OrderBy(d => d.Nachname);
```

Hier haben wir noch die Methode **OrderBy** an die Eigenschaft **Kunden** angehängt und dieser als Parameter die Vorgabe für die Sortierung mitgegeben, nämlich **d => d.Nachname**.

Dabei handelt es sich um einen sogenannten Lambda-Ausdruck. Lambda-Ausdrücke haben wir in diesem Magazin noch nicht besprochen, daher gehen wir hier nur darauf ein, wie diese für den vorliegenden Anwendungszweck formuliert werden.

Der Lambda-Ausdruck, den wir hier als Parameter der **OrderBy** (und später auch der **OrderByDescending**-, der **Where**- und anderer Methoden) nutzen, enthält zwei Teile: Der Teil vor dem Lambda-Zeichen (**=>**, nicht zu verwechseln mit **=>**) enthält den Namen für das, was wir im Teil hinter dem Lambda-Zeichen untersuchen wollen.

Wir untersuchen immer das Objekt, für das wir die Methode **OrderBy** (**OrderByDescending**, **Where**, ...) aufrufen, in diesem Fall also **context.Kunde**, also die Auflistung der Kunden-Entitäten, und geben dieser im vorliegenden Fall den Namen **d** (Sie können auch einen anderen Buchstaben oder Ausdruck statt **d** verwenden).

Dies können wir dann im zweiten Teil des Lambda-Ausdrucks aufgreifen. Im Falle der **OrderBy**-Methode geben wir also beispielsweise an, wonach wir sortieren wollen – hier die Eigenschaft **Nachname** der **Kunde**-Entitäten.

In der Variablen **kunden** landet dann die sortierte Liste der **Kunde**-Entitäten, die wir wieder in einer **foreach**-Schleife durchlaufen. Diesmal geben wir den Primärschlüsselwert sowie den Nachnamen der Kunden aus:

```
8 Ackermann
9 Alberts
10 Becker
11 Fröhlich
4 Meier
1 Minhorst
2 Müller
7 Schmidt
```

```
public static void KundenNachNachname_MethodenSyntax() {
    using (BestellverwaltungEntities context = new BestellverwaltungEntities()) {
        var kunden = context.Kunden.OrderBy(d => d.Nachname);
        foreach (var kunde in kunden) {
            Console.WriteLine("{0} {1}", kunde.ID, kunde.Nachname);
        }
    }
    Console.ReadLine();
}
```

Listing 3: Methode zur Ausgabe der Kunden, sortiert nach dem Nachnamen (Methodensyntax)

Abfragen mit der Abfragesyntax

Die zweite Methode zum Abfragen von Daten unter LINQ to

Entities verwendet die sogenannte Abfragesyntax. Hier stellen Sie einen Abfrageausdruck zusammen, der eher wie ein SQL-Ausdruck aussieht. Um das gleiche Ergebnis wie im obigen Beispiel mit der Methodensyntax zu erzielen, ersetzen Sie die Zeile mit der **OrderBy**-Methode und dem Lambda-Ausdruck durch die folgende Zeile:

```
var kunden =
from d in context.Kunden
orderby d.Nachname
select d;
```

Dies sieht für erfahrene SQL-Nutzer etwas intuitiver aus, da hier einige Elemente wie **from**, **orderby** und **select** auftauchen. Die Reihenfolge ist hier etwas anders als beim Lambda-Ausdruck. Die Zuweisung des Ergebnisses erfolgt wieder an die Variable **kunden**. Danach geben wir an, woher die Daten kommen und welchen Alias-Namen wir nachfolgend dafür verwenden wollen (**from d in context.Kunden**).

Dann legen wir die Operation fest, in diesem Fall wieder die Sortierung (**orderby d.Nachname**). Schließlich gibt **select d** an, was zurückgegeben werden soll. Beim vollständigen Beispiel ändert sich also nur die Zeile mit der Definition der Abfrage (siehe Listing 4).

Sowohl bei der Methodensyntax als auch bei der Abfragensyntax unterstützt Visual Studio Sie mit IntelliSense.

Absteigende Sortierung

Unter SQL haben Sie für eine absteigende Sortierung das Schlüsselwort **DESC** hinten angefügt (**ORDER BY Nachname DESC**). Unter LINQ to Entities verwenden Sie bei der Methodensyntax einfach eine andere Methode (**OrderByDescending**):

```
public static void KundenNachNachname_Abfragesyntax() {
    using (BestellverwaltungEntities context = new BestellverwaltungEntities()) {
        var kunden = from d in context.Kunden orderby d.Nachname select d;
        foreach (var kunde in kunden) {
            Console.WriteLine("{0} {1}", kunde.ID, kunde.Nachname);
        }
    }
    Console.ReadLine();
}
```

Listing 4: Methode zur Ausgabe der Kunden, sortiert nach dem Nachnamen (Abfragesyntax)

```
var kunden =
context.Kunden.OrderByDescending(d => d.Nachname);
```

Bei der Abfragensyntax fügen Sie ähnlich wie bei SQL ein Schlüsselwort hinzu, hier **descending**:

```
var kunden =
from d in context.Kunden
orderby d.Nachname descending
select d;
```

Hinweis: Die Abfrage wird zwar in den Beispielen bereits vor der Schleife definiert, aber die Daten werden erst beim ersten Zugriff in der Schleife geladen.

Kunden nach bestimmten Kriterien ausgeben

Nun steigern wir den Schwierigkeitsgrad und wollen nur bestimmten Datensätze der Tabelle **Kunden** als Objekt ermitteln und durchlaufen. Unter Access/VBA brauchen wir dazu nur die **SELECT**-Anweisung in der **OpenRecordset**-Methode zu ändern. Unter C#/LINQ to Entities ist es auch nicht viel komplizierter – genau genommen haben Sie die Grundlagen dazu schon oben mit der Sortierung kennen gelernt.

Wenn Sie beispielsweise den Kunden ermitteln wollen, dessen Eigenschaft **Nachname** den Wert **Minhorst** enthält, formulieren Sie die Abfrage in der Methodensyntax wie folgt (siehe Methode **KundeMitNachname_Methodensyntax**).

Sie verwenden also die **Where**-Methode statt der **OrderBy**-Methode:

```
var kunden =
context.Kunden.Where(d => d.Nachname == "Minhorst");
```

In der Abfragensyntax erledigen Sie die Aufgabe mit dieser Zeile (siehe Methode **KundeMitNachname_Abfragesyntax**). Hier kommt also das **where**-Schlüsselwort mit dem Vergleichskriterium hinzu:

```
var kunden =
from d in context.Kunden
where d.Nachname == "Minhorst"
select d;
```

Platzhalter?

Von Access/SQL kennen Sie als Platzhalter das Sternchen (*) für kein, ein oder mehrere Zeichen oder das Fragezeichen für ein Zeichen, vom SQL Server das Prozentzeichen (%). Unter LINQ to Entities gibt es Platzhalter in diesem Sinne nicht.

Auf einem Umweg können Sie auch damit arbeiten, aber dazu kommen wir gegebenenfalls in einem späteren Artikel.

Für die meisten Fälle sollten jedoch einige spezielle Funktionen ausreichen, etwa solche, mit denen Sie nach Inhalten suchen können, die mit einem bestimmten Ausdruck beginnen, einen Ausdruck enthalten oder damit enden.

Kunden, deren Nachname mit einem bestimmten Ausdruck beginnt

Suchen Sie nach Objekten, von denen ein Eigenschaftswert mit einem bestimmten Ausdruck beginnt – etwa der Nachname mit dem Buchstaben **A** –, dann nutzen Sie eine erweiterte Variante der **Where**-Methode beziehungsweise des **Where**-Schlüsselworts.

Bei der Methodensyntax nutzen Sie dann nicht etwa einen Ausdruck wie **d => d.Nachname == "A%"**, denn Platzhalter gibt es in dieser Form ja nicht. Stattdessen nutzen Sie die Funktion **StartsWith** der entsprechenden Eigenschaft und übergeben dieser den Wert, mit dem der Nachname der zu liefernden Objekte beginnen soll (**d => d.Nachname.StartsWith("A")**), siehe Methode **KundenMitNachnameBeginntMitA_Methodensyntax**:

```
var kunden =
context.Kunden.Where(d => d.Nachname.StartsWith("A"));
```

Für die Abfragensyntax sieht das wie folgt aus – hier nutzen wir die Funktion **StartsWith** analog zur Methodensyntax (siehe Methode **KundenMitNachnameBeginntMitA_Abfragesyntax** in der Beispieldatenbank):

```
var kunden =
from d in context.Kunden
where d.Nachname.StartsWith("A") select d;
```

Die verfügbaren Funktionen wie **StartsWith**, **EndsWith**, **Contains** und viele andere werden per IntelliSense angezeigt, nachdem Sie den Punkt hinter dem Objekt angegeben haben (siehe Bild 2).

Andere Zeichenketten-Funktionen

Es gibt noch eine Reihe weiterer Funktionen, die Sie für eine Eigenschaft angeben können, um diese nach einer enthaltenen Zeichenkette zu durchsuchen – zum Beispiel die folgenden beiden:

- **contains**: Liefert alle Elemente, welche die angegebene Zeichenkette enthalten.

```
public static void KundenMitNachnameEnthaeltA_Abfragensyntax() {
    using (BestellverwaltungEntities context = new BestellverwaltungEntities()) {
        var kunden = from d in context.Kunden where d.Nachname.Contains("I") select d;
        foreach (var kunde in kunden) {
            Console.WriteLine("{0} {1}", kunde.ID, kun
        }
    }
    Console.ReadLine();
}
```

GetHashCode
GetType
ID
Nachname
Ort
PLZ
Strasse
Telefon
ToString

Bild 2: Auswahl der möglichen Funktionen per IntelliSense

- **endswith**: Sucht die Elemente, die mit der angegebenen Zeichenkette enden.

Sortieren und Filtern gleichzeitig

Gelegentlich wollen Sie gleichzeitig filtern und sortieren oder auch mehrere Filter- oder Sortierkriterien übergeben. Alle Kunden, deren Name ein **i** enthält, sortiert der folgende Ausdruck in der Abfragensyntax auch direkt nach dem Nachnamen:

```
var kunden =
from d in context.Kunden
where d.Nachname.Contains("I")
orderby d.Nachname
select d;
```

In der Methodensyntax hängen Sie einfach die beiden Methoden hintereinander an die zu untersuchende/zu sortierende Eigenschaft an:

```
var kunden = context.Kunden
.Where(d => d.Nachname.Contains("i"))
.OrderBy(d => d.Nachname);
```

Nach mehreren Felder filtern

Auf die gleiche Weise können Sie auch nach mehreren Feldern gleichzeitig sortieren oder filtern. Hängen Sie einfach die gewünschten Kriterien hintereinander.

Alle Kunden, deren **Nachname** mit **A** beginnt und deren **AnredeID** den Wert **1** enthält, liefert die folgende Zeile in der Abfragensyntax:

```
var kunden =
from d in context.Kunden
where d.Nachname.StartsWith("A")
where d.AnredeID == 1
orderby d.Nachname
select d;
```

Und hier der Ausdruck in der Methodensyntax:

```
var kunden = context.Kunden
.Where(d => d.Nachname.StartsWith("A"))
.Where(d => d.AnredeID == 1)
.OrderBy(d => d.Nachname);
```

Im Gegensatz zu SQL können Sie die Reihenfolge von **where**- und **orderby**-Elementen vertauschen.

Nach mehreren Feldern sortieren

Allerdings gelingt es nicht, einfach mehrere **orderby**-Ausdrücke anzugeben – es wird dann einfach immer der letzte Ausdruck ausgewertet, wie in folgendem Beispiel:

```
//sortiert nur nach dem Nachnamen:
var kunden =
from d in context.Kunden
orderby d.Vorname,
orderby d.Nachname
select d;
```

Für das Sortieren nach mehreren Eigenschaften in der Abfragensyntax geben Sie die zu sortierenden Eigenschaften einfach in der gewünschten Reihenfolge an – genau wie unter SQL:

```
var kunden =
from d in context.Kunden
orderby d.Vorname, d.Nachname
select d;
```

Für eine absteigende Sortierung hängen Sie noch das Schlüsselwort **descending** an – oder, um explizit die aufsteigende Sortierung festzulegen, das Schlüsselwort **ascending**. Hier werden beide verwendet:

```
var kunden =
from d in context.Kunden
orderby d.Vorname ascending,
d.Nachname descending select d;
```

Bei der Methodensyntax gibt es das **thenby**-Schlüsselwort:

LINQ to Entities: Daten bearbeiten

In den vorangegangenen Ausgaben von DATENBANKENTWICKLER haben Sie bereits erfahren, wie Sie per ADO.NET auf die Daten der Tabellen einer Datenbank zugreifen. Nun nutzen wir nicht mehr direkt ADO.NET, sondern das Entity Framework als Datenlieferant, welches eine ganz andere Art des Zugriffs ermöglicht. Dieser Artikel zeigt, wie Sie die per Entity Data Model bereitgestellten Daten mit LINQ to Entities ändern, löschen und neu anlegen.

Beispielprojekt

Das Beispielprojekt soll über ein geeignetes Entity Data Model auf die Tabellen in einer LocalDB-Datenbank zugreifen. Wie Sie das Entity Data Model erstellen, beschreiben wir im Artikel **Entity Data Model für eine Datenbank erstellen**.

In Bild 1 sehen Sie die Klassen des Beispiels, die prinzipiell wie ein Datenmodell aussehen. Diese Klassen bilden die Grundlage der Beispiele für einige Artikel, aber später werden wir diese gegebenenfalls noch etwas erweitern. Alle Beispielmethoden finden Sie in der Klasse **Beispiele_EF_Bearbeiten.cs**. Um die Beispiele auszuprobieren, starten Sie einfach das Projekt.

Aufruf der Beispielmethoden

Damit wir die nachfolgend programmierten Methoden einfach aufrufen können, verwenden wir wieder die bereits in früheren Artikeln verwendete Klasse **Program** mit einer **Main**-Methode, die alle öffentlichen, statischen Methoden aus öffentlichen Klassen auflistet und zur Ausführung anbietet. In einer neuen Klasse namens **Beispiele_EF_Bearbeiten** legen wir dann die später vorgestellten Methoden an.

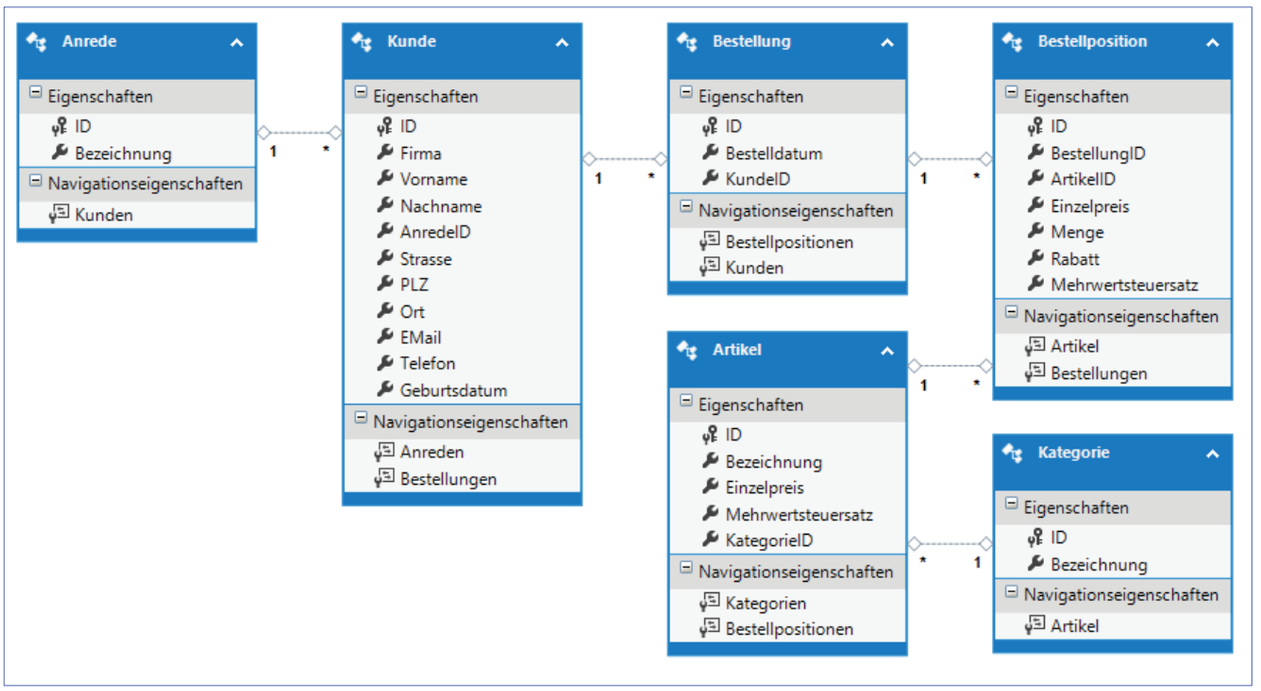


Bild 1: Klassendiagramm für die Beispiele dieses Artikels

```
public static void KundeAendern() {
    using (BestellverwaltungEntities context = new BestellverwaltungEntities()) {
        Kunde kunde = context.Kunden.Find(1);
        Console.WriteLine("Kunde vorher: {0} {1} {2}", kunde.ID, kunde.Vorname, kunde.Nachname);
        kunde.Vorname = "Andreas";
        Console.WriteLine("Kunde nachher: {0} {1} {2}", kunde.ID, kunde.Vorname, kunde.Nachname);
        context.SaveChanges();
    }
    using (BestellverwaltungEntities context = new BestellverwaltungEntities()) {
        Kunde kunde = context.Kunden.Find(1);
        Console.WriteLine("Kunde neu eingelesen: {0} {1} {2}", kunde.ID, kunde.Vorname, kunde.Nachname);
    }
    Console.ReadLine();
}
```

Listing 1: Einlesen, ändern und erneutes Einlesen eines Kunden

Einen Kunden bearbeiten

Im Artikel **LINQ to Entities: Daten abfragen** haben Sie bereits erfahren, wie Sie die Daten aus einer Tabelle in ein oder mehrere Objekte laden und diese in der Konsole ausgeben können. Genau das erledigen wir jetzt auch – mit dem Unterschied, dass wir auch noch eine Änderung an diesem Kunden vornehmen und diese Änderung speichern.

Den vollständigen Code des Beispiels sehen Sie in Listing 1. Die Methode **KundeAendern** erstellt eine Variable des Typs **BestellverwaltungEntities**, welche die verschiedenen Klassen wie **Kunde**, **Anrede** et cetera bereithält, und speichert den Verweis darauf in der Variablen **context**, welche innerhalb des **using**-Konstrukts ihre Gültigkeit behält. Über die **Find**-Methode des **Kunden**-DBSets von **context** ermittelt die Prozedur dann den Kunden mit dem Wert **1** in der Identitätsspalte, in diesem Fall im Feld **ID**. Die Methode gibt nun die Inhalte der drei Felder **ID**, **Vorname** und **Nachname** des **Kunde**-Objekts in der Konsole aus. Dann ändert sie durch eine einfache Zuweisung des Wertes **Andreas** an die Eigenschaft **Vorname** den Inhalt des Objekts. Die erneute Ausgabe zeigt den geänderten Inhalt. Nun folgt der entscheidende Schritt: Die Methode ruft die **SaveChanges**-Methode des **context**-Objekts auf. Damit sollen die Änderungen in die Datenbank übertragen werden. Um dies zu prüfen, erstellen wir einen neuen Kontext auf die Datenbank, lesen den Kundendatensatz erneut in das Objekt **kunde** ein und geben die aktuellen Werte von **ID**, **Vorname** und **Nachname** aus. Das Ergebnis: Die Änderung wurde in die Datenbank übernommen.

Wenn Sie den SQL Server-Profiler mitlaufen lassen, können Sie sich die durch die **SaveChanges** aufgerufene SQL-Anweisung ansehen (mehr dazu im Artikel **SQL Server-Interaktion mit dem Profiler verfolgen**):

```
exec sp_executesql N'UPDATE [dbo].[Kunde]
SET [Vorname] = @0
WHERE ([ID] = @1)
',N'@0 nvarchar(255),@1 int',@0=N'Andreas',@1=1
```

Hier wurde in der Tat nur das Feld **Vorname** für den Datensatz mit dem Wert **1** im Feld **ID** geändert.

Sie können die Änderungen natürlich auch direkt im SQL Server mitverfolgen, indem Sie dort die Herkunftstabelle (hier **Kunde**) mit der rechten Maustaste anklicken und den Kontextmenüeintrag **Oberste 200 Zeilen bearbeiten** auswählen. Sie können die dortige Ansicht aktualisieren, indem Sie die Tastenkombination **Strg + R** betätigen.

Einen neuen Kunden hinzufügen

Kommen wir nun zum Hinzufügen eines neuen Kunden. Hier betrachten wir zunächst, was wir bisher wissen: Wenn wir

WPF/EDM: Kundenübersicht

WPF-Fenster können zwar auch Daten ADO.NET-Datenquellen anzeigen, aber optimal sind sie für den Zugriff auf Daten aus einem Entity Data Model vorbereitet. Dieser Artikel zeigt, wie Sie die per Entity Data Model abgebildeten Daten aus einer Haupt- und einer Lookup-Tabelle in einem WPF-Fenster anzeigen. Dabei soll das Fenster die Navigation durch die Datensätze und auch das Anlegen neuer Datensätze ermöglichen. Im Vergleich zu einem früheren Artikel zeigen wir nun, wie Sie die Aufgabe manuell erledigen, also ohne die Unterstützung von Assistenten.

Um eine datenbankbasierte WPF-Anwendung zu erstellen, legen Sie zunächst ein neues Projekt auf Basis der Vorlage **Visual C#|WPF-Anwendung** an.

Dann fügen Sie ein Entity Data Model auf Basis unserer Beispieldatenbank **Bestellverwaltung** hinzu, wie es im Artikel **Entity Data Model für eine Datenbank erstellen** im Detail beschrieben wird. Hier die Kurzform ohne Screenshots und Umwege:

- Betätigen Sie die Tastenkombination **Strg + Umschalt + A** und wählen Sie im Dialog **Neues Element hinzufügen** den Eintrag **ADO.NET Entity Data Model** aus. Geben Sie als Name **BestellverwaltungModel** an und klicken Sie auf **Hinzufügen**.
- Behalten Sie im Dialog **Assistent für Entity Data Model** den Wert **EF Designer aus Datenbank** bei und klicken Sie auf **Weiter**.
- Klicken Sie unter **Wählen Sie Ihre Datenbankverbindung aus** auf **Neue Verbindung ...** und wählen Sie dort **Microsoft SQL Server** aus.
- Geben Sie im Dialog **Verbindungseigenschaften** den Servernamen ein (zum Beispiel **(localdb)\v11.0** oder **(localdb)\mssqllocaldb**, wenn Sie LocalDB verwenden).
- Wählen Sie die Datenbank aus oder geben Sie die anzuhängende Datenbankdatei an. Testen Sie die Verbindung und klicken Sie auf **OK**.
- Zurück im **Assistent für Entity Data Model** bestätigen Sie den Namen für die Verbindungseinstellungen (hier **BestellverwaltungEntities**).
- Wenn Sie möchten, bestätigen Sie die Frage, ob Sie die Datenbankdatei zum Projekt hinzufügen möchten.
- Behalten Sie **Entity Framework 6.x** als Version bei.
- Wählen Sie alle gewünschten Tabellen aus und geben Sie den Namen für den Modellnamespace an (hier **BestellverwaltungModel**).
- Klicken Sie auf **Fertigstellen**. Das Entity Data Model wird nun erstellt.

Nun nehmen Sie noch eventuell nötige Änderungen an den Namen der Entitäten oder den Namen der Entitätsmengen vor. Die Beispieldatenbank verwendet Tabellen mit Namen im Singular. Sie sollten daher die Eigenschaft **Name der Entitätenmenge** für alle Entitäten über das Modelldiagramm (hier **Bestellverwaltungmodel.edmx**) auf die Pluralform einstellen (also **Kategorien** statt **Kategorie** und so weiter). Anschließendes Speichern aktualisiert den Code. Für dieses Beispiel arbeiten wir mit den Tabellen **Kunden** und **Anreden**, wobei wir als Namen für die Entitätenmengen die Namen **Kunden** und **Anreden** übernehmen und die Entitätsklassen selbst mit **Kunde** und **Anrede** benennen.

Um die Namen der Entitätsklassen anzupassen, öffnen Sie die Datei **BestellverwaltungModel.edmx**, markieren

nacheinander die Entitäten und ändern die Bezeichnungen der Entitäten von Singular auf Plural (siehe Bild 1). Speichern nicht vergessen, um die Änderungen auf den Code zu übertragen – dies kann ein paar Sekunden dauern.

Kunden im DataGrid

Als Erstes wollen wir die Kunden der Tabelle **Kunden** im DataGrid anbieten. Dieses Datagrid legen wir in einem neuen Fenster namens **KundenuebersichtEinfach** an, das Sie per **Strg + Umschalt + A** und Auswahl des Eintrags Fenster sowie Angabe der Bezeichnung **KundenuebersichtEinfach** anlegen.

Um dieses Fenster vom Fenster **MainWindow.xaml** aus zu öffnen, fügen wir diesem eine neue Schaltfläche namens **btnKundenuebersichtEinfach** hinzu:

```
<Button x:Name="btnKundenuebersichtEinf
ach" Content="Kundenübersicht einfach"
HorizontalAlignment="Left" Margin="31,20,0,0"
VerticalAlignment="Top" Width="143" Click="btnKundenuebers
ichtEinfach_Click"/>
```

Für die Eigenschaft **Click** hinterlegen wir eine Methode, die wie folgt aussieht:

```
private void btnKundenuebersichtEinfach_Click(
                    object sender, RoutedEventArgs e) {
    KundenuebersichtEinfach wnd =
                        new KundenuebersichtEinfach();
    wnd.Show();
}
```

Kunden anzeigen – einfache Variante

Dem Fenster **Kundenuebersicht-Einfach** fügen Sie nun das **Datagrid**-Steuerelement hinzu, welches die Felder der Tabelle **Kunden** anzeigen soll. Der XAML-Code sieht dann etwa wie folgt aus:

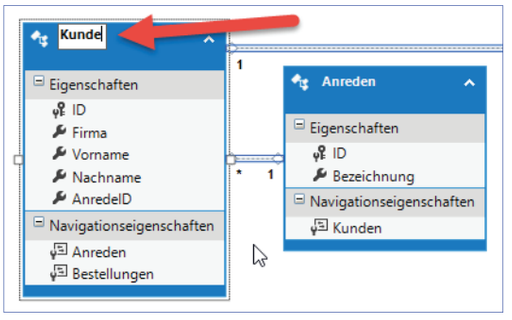


Bild 1: Ändern der Entitätsnamen

```
<Window x:Class="Bestellverwaltung_EDM.Kundenuebersicht"
...
        Title="Kundenuebersicht" Height="300" Width="600">
    <Grid>
        <DataGrid x:Name="dgKunden"
            HorizontalAlignment="Left" Margin="10,10,10,10"
            VerticalAlignment="Top" />
    </Grid>
</Window>
```

Sie sehen hier keinerlei Datenbindungseigenschaften. Dies erledigen wir direkt vom Code aus. Der komplette Code der Klasse sieht so aus:

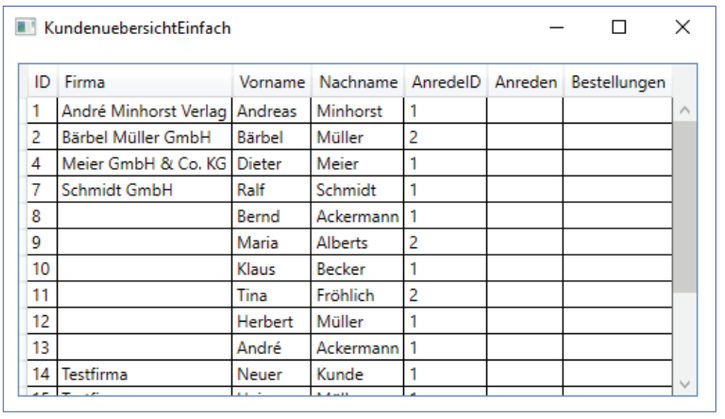
```
using System.Linq;
using System.Windows;

namespace Bestellverwaltung_EDM {
    public partial class KundenuebersichtEinfach : Window {
        public KundenuebersichtEinfach() {
            InitializeComponent();
            using (BestellverwaltungEntities DBContext =
                        new BestellverwaltungEntities()) {
                var kunden = DBContext.Kunden.ToList();
                dgKunden.ItemsSource = kunden;
            };
        }
    }
}
```

Hier verwenden wir die im Artikel **LINQ to Entities: Daten abfragen** ausführlich erläuterten Techniken, um die Daten aus einer Tabelle über das Entity Data Model in eine Variable zu füllen. Dazu erstellen wir einen Datenbank-Kontext des Typs **BestellverwaltungEntities**. Dieses bietet die Entitätsauflistung **Kunden**, die wir mit der **ToList**-Methode in eine Liste füllen und an die Variable

kunden übergeben. Diese landet schließlich über die Eigenschaft **ItemsSource** im DataGrid:

```
using (BestellverwaltungEntities DBContext =
            new BestellverwaltungEntities()) {
    var kunden = DBContext.Kunden.ToList();
    dgKunden.ItemsSource = kunden;
};
```



ID	Firma	Vorname	Nachname	AnredeID	Anreden	Bestellungen
1	André Minhorst Verlag	Andreas	Minhorst	1		
2	Bärbel Müller GmbH	Bärbel	Müller	2		
4	Meier GmbH & Co. KG	Dieter	Meier	1		
7	Schmidt GmbH	Ralf	Schmidt	1		
8		Bernd	Ackermann	1		
9		Maria	Alberts	2		
10		Klaus	Becker	1		
11		Tina	Fröhlich	2		
12		Herbert	Müller	1		
13		André	Ackermann	1		
14	Testfirma	Neuer	Kunde	1		

Bild 2: Erster Entwurf des **DataGrid**-Steuerelements mit Kundendaten

Das Ergebnis sehen Sie in Bild 2. Weil das Attribut **AutoGenerateColumns** in der XAML-Definition des **DataGrid**-Steuerelements nicht explizit gesetzt wurde, hat dieses den Standardwert **True**, was dazu führt, dass alle Felder der Datenquelle im **DataGrid**-Steuerelement abgebildet werden. Da wir die Daten mit der Methode **ToList()** in die Variable **kunden** geschrieben haben, werden die verknüpften Objekte **Anreden** und **Bestellungen** nicht gefüllt. Damit können Sie nun bereits die üblichen Aktionen wie etwa das Sortieren durchführen, aber es werden noch nicht einmal die Anreden angezeigt, sondern nur die Werte des Feldes **AnredeID** – und die beiden verknüpften Eigenschaften **Anreden** und **Bestellungen** wollen wir auch loswerden.

Manuelle Definition der anzuzeigenden Felder

Wie wir oben beschrieben haben, werden die Felder aktuell angezeigt, weil die Option **AutoGenerateColumns** für das **DataGrid**-Steuerelement aktiviert ist. Wir wollen aber individuelle Einstellungen für die angezeigten Daten vornehmen. Daher deaktivieren wir zunächst mit dem Attribut **AutoGenerateColumns="False"** die automatische Generierung. Außerdem definieren wir selbst die anzuzeigenden Spalten und deren Spaltenüberschriften und Inhalte. Dies sieht dann wie in der Definition aus Listing 1 aus, wo wir für Spalten jeweils ein Element des Typs **DataGridTextColumn**

```
<Window x:Class="Bestellverwaltung_EDM.Kundenuebersicht_Columns" ...
                                                    Title="Kundenuebersicht_Columns" Height="300" Width="600">
    <Grid>
        <DataGrid x:Name="dgKunden" HorizontalAlignment="Left" Margin="10,10,10,10" VerticalAlignment="Top"
                                                                              AutoGenerateColumns="False">
            <DataGrid.Columns>
                <DataGridTextColumn Header="ID" Binding="{Binding ID}" Width="30"></DataGridTextColumn>
                <DataGridTextColumn Header="Firma" Binding="{Binding Firma}" Width="*"></DataGridTextColumn>
                <DataGridTextColumn Header="Anrede" Binding="{Binding Anrede}" Width="50"></DataGridTextColumn>
                <DataGridTextColumn Header="Vorname" Binding="{Binding Vorname}" Width="*"></DataGridTextColumn>
                <DataGridTextColumn Header="Nachname" Binding="{Binding Nachname}" Width="*"></DataGridTextColumn>
                <DataGridTextColumn Header="Straße" Binding="{Binding Strasse}" Width="*"></DataGridTextColumn>
                <DataGridTextColumn Header="PLZ" Binding="{Binding PLZ}" Width="*"></DataGridTextColumn>
                <DataGridTextColumn Header="Ort" Binding="{Binding Ort}" Width="*"></DataGridTextColumn>
            </DataGrid.Columns>
        </DataGrid>
    </Grid>
</Window>
```

Listing 1: Benutzerdefinierte Festlegung der Spalten

verwenden. Als Datenbindung geben wir für die Eigenschaft **Binding** jeweils Werte wie **{Binding ID}** an.

Den Code dieses Beispiels finden Sie im Fenster **Kundenuebersicht_Columns.xaml**. Die Ansicht entspricht weitgehend der aus dem ersten Beispiel.

Kunden per Eigenschaft füllen

Nun gehen wir einen Schritt weiter: Bisher haben wir die Datenquelle im C#-Code im Code behind-Modul für die Eigenschaft ItemsSource angegeben. Nun wollen wir die Zuweisung in den XAML-Code verschieben. All dies erledigen wir in dem Beispielfenster **KundenuebersichtBinding.xaml**. Warum wollen wir die folgende Zeile gern aus dem Code behind-Modul heraushaben?

```
dgKunden.ItemsSource = kunden;
```

Weil wir so die Benutzeroberfläche und die Anwendungslogik besser voneinander trennen können. Später, also in folgenden Ausgaben dieses Magazins, werden wir uns mit MVVM beschäftigen, einem Entwurfsmuster, mit dem genau diese Trennung erreicht werden soll. Ein erster Schritt dorthin ist, dass wir die Eigenschaften und Inhalte der Benutzeroberfläche nicht direkt per Code ändern, sondern dass wir per Code Eigenschaften bereitstellen, welche die Elemente der Benutzeroberfläche dann referenzieren. In diesem Fall wollen wir also nicht vom Code aus die Datenquelle für das **DataGrid**-Steuerelement festlegen, sondern nur eine Eigenschaft bereitstellen, welche die Daten liefert (in diesem Fall eine Liste von **Kunde**-Objekten). Auf diese Eigenschaft greifen wir dann mit einem entsprechenden Attribut vom XAML-Code des Fensters zu.

Dies sieht dann wie in Listing 2 aus. Hier sind zwei verschiedene Eigenschaften wichtig:

- Die Eigenschaft **ItemsSource** des **DataGrid**-Steuerelements legt fest, von welcher Eigenschaft die Daten bezogen werden.

- Die Eigenschaft **Binding** legt fest, welche Eigenschaft/welches Feld der mit **ItemsSource** definierten Datenquelle als Inhalt der Textspalten des **DataGrid**-Steuerelements genutzt werden.

Woher aber weiß die XAML-Definition, woher **Kunden** stammt? Dies legen wir in der Code behind-Datei fest. Diese verwendet folgende Namespaces, wobei **System.Collections.Generic** neu ist:

```
using System.Collections.Generic;
using System.Linq;
using System.Windows;
```

```
<Window x:Class="Bestellverwaltung_EDM.KundenuebersichtBinding" ...
        Title="KundenuebersichtBinding" Height="300" Width="300">
    <Grid>
        <DataGrid x:Name="dgKunden" HorizontalAlignment="Left" Margin="10,10,10,10" VerticalAlignment="Top"
                                                    ItemsSource="{Binding Kunden}" AutoGenerateColumns="False">
            <DataGrid.Columns>
                <DataGridTextColumn Header="ID" Binding="{Binding ID}" Width="30"></DataGridTextColumn>
                <DataGridTextColumn Header="Firma" Binding="{Binding Firma}" Width="*"></DataGridTextColumn>
            ...
            </DataGrid.Columns>
        </DataGrid>
    </Grid>
</Window>
```

Listing 2: Die Datenbindung erfolgt nun komplett über Eigenschaften.

WPF/EDM: Kundendetails

Wenn Sie aus einer Kundenübersicht die Details eines Kunden anzeigen oder einen neuen Kunden anlegen möchten, benötigen Sie ein geeignetes weiteres Fenster. Dieses soll die Daten des zu bearbeitenden Kundendatensatzes oder auch einen neuen, leeren Datensatz anzeigen – je nach Anforderung. Wie Sie dies auf Basis von Daten aus einem Entity Data Model erledigen, zeigt dieser Artikel.

Voraussetzungen

Für das Nachvollziehen dieses Beispiels erstellen Sie ein Entity Data Model wie im Artikel **WPF/EDM: Kundenübersicht** beschrieben. Das dort beschriebene Fenster mit einer Übersicht der gespeicherten Kunden verwenden wir, um das im vorliegenden Artikel beschriebene Detailfenster zu öffnen.

Fenster zum Hinzufügen und Bearbeiten von Kunden

Bevor wir Kunden bearbeiten oder hinzufügen können, benötigen wir ein entsprechendes Fenster. Dieses definieren wir mit der **.xaml**-Datei aus Listing 1 (gekürzte Fassung).

Hier haben wir ein Raster von vier Spalten und sieben Zeilen, wobei die oberen Zeilen für die Beschriftungen und die Felder des Datensatzes und die unteren für Schaltflächen zum Speichern und zum Abbrechen der aktuellen Änderungen vorgesehen sind. Der Entwurf sieht wie in Bild 1 aus. Die Bindung der Felder verläuft wie schon beim Übersichtsfenster, nur dass hier kein DataGrid mit einzelnen Spalten, sondern jeweils unabhängige Textfelder und ein Kombinationsfeld zum Einsatz kommen. Die Bindung erfolgt über die entsprechenden Eigenschaften des **Kunde**-Objekts.

Aus Gründen, die wir später erläutern, nutzen wir hier jedoch eine ungebundene Kopie des zu bearbeitenden **Kunde**-Objekts beziehungsweise ein neues **Kunde**-Objekt, wenn ein neuer Kunde angelegt werden soll. Dieses Objekt nennen wir **KundeTemp**. Da wir in der Definition kein übergeordnetes Element haben, dem wir das **KundeTemp**-Objekt zuweisen

```
<Window x:Class="Bestellverwaltung_EDM.Kundendetails" ...Title="Kundendetails" Height="300" Width="300" Icon="users3.ico">
    <Grid>
        //... Definition von vier Spalten und sieben Zeilen
        <TextBlock Grid.Column="0" Grid.Row="0" Margin="3" Text="ID:" />
        <TextBlock Grid.Column="0" Grid.Row="1" Margin="3" Text="Firma:" />
        <TextBlock Grid.Column="0" Grid.Row="2" Margin="3" Text="Anrede:" />
        <TextBlock Grid.Column="0" Grid.Row="3" Margin="3" Text="Vorname:" />
        <TextBlock Grid.Column="0" Grid.Row="4" Margin="3" Text="Nachname:" />
        <TextBox ... Text="{Binding KundeTemp.ID}" IsEnabled="False"/>
        <TextBox ... Text="{Binding KundeTemp.Firma}" />
        <ComboBox ... ItemsSource="{Binding Anreden}" SelectedItem="{Binding KundeTemp.Anreden}"
                                                    DisplayMemberPath="Bezeichnung" SelectedValuePath="Id" />
        <TextBox ... Text="{Binding KundeTemp.Vorname}" />
        <TextBox ... Text="{Binding KundeTemp.Nachname}" />
        <Button x:Name="btnAbbrechen" ... Content="Abbrechen" Click="btnAbbrechen_Click" />
        <Button x:Name="btnSpeichern" ... Content="Speichern" Click="btnSpeichern_Click" />
    </Grid>
</Window>
```

Listing 1: Benutzerdefinierte Festlegung der Spalten

können (etwa der **ItemsSource** eines **DataGrid**-Elements), tragen wir die Werte für die Eigenschaft **Text** der **Textbox**-Steuerelemente in der Form **{Binding KundeTemp.<Eigenschaftsname>}** ein.

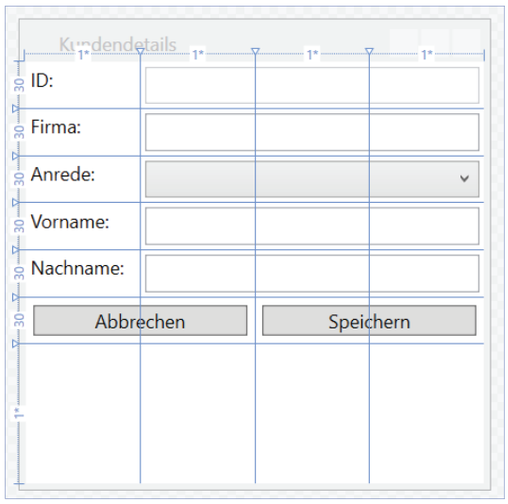


Bild 1: Entwurf des Detailfensters

Kombinationsfeld für die Anreden

Das Kombinationsfeld zur Anzeige der Anreden füllen wir mit mehr als einer Eigenschaft. Das Attribut **ItemsSource** erhält den Wert **{Binding Anreden}**, wodurch die Eigenschaft **Anreden** der Code behind-Klasse als Datenherkunft dieses Steuerelements dient.

Damit es den für diesen Kunden festgelegten Eintrag anzeigt, legen wir für das Attribut **SelectedItem** den Wert **{Binding KundeTemp.Anreden}** fest. Dies liefert genau das benötigte **Anrede**-Objekt.

Das Attribut **DisplayMemberPath** gibt an, welche Eigenschaft des enthaltenen Objekts angezeigt werden soll, hier also die Eigenschaft **Bezeichnung**. **SelectedValuePath** erhält die Eigenschaft **ID** des gewählten **Anrede**-Objekts.

Falls Sie von Access kommen: **ItemsSource** entspricht der **Datensatzherkunft**, **SelectedItem** ist mit der Access-Eigenschaft **ItemSelected** vergleichbar, **DisplayMemberPath** bilden Sie unter Access mit **Spaltenanzahl** und **Spaltenbreiten** ab und **SelectedValuePath** entspricht der Eigenschaft **Gebundene Spalte** bezogen auf die Felder der Datensatzherkunft.

Beim Öffnen des Fensters

Wenn Sie das Fenster öffnen, müssen Sie entscheiden, ob Sie ein neues **Kunde**-Objekt anlegen oder ein vorhandenes bearbeiten möchten. Wenn Sie mit Access/VBA arbeiten, wissen Sie, dass Sie solche Eigenschaften mit dem Befehl zum Öffnen des jeweiligen Formulars übergeben konnten, beispielsweise mit

```
DoCmd.OpenForm "frmKunde",
DataMode:=acFormAdd
```

für einen neuen Datensatz oder

```
DoCmd.OpenForm "frmKun-
de", DataMode:=acFormEdit,
WhereCondition:="ID = " & KundeID
```

zum Bearbeiten eines vorhandenen Datensatzes. Unter C# können wir solche Parameter frei definieren, und zwar durch Anpassung der Konstruktor-Methode. Bevor wir dazu kommen, ein kurzer Blick auf die Variablen dieser Klasse:

```
public Kunde KundeTemp { get; set; }
private Kunde KundeAktuell;
BestellverwaltungEntities DBContext;
private List<Anreden> anreden;
```

KundeTemp nimmt ein temporäres **Kunde**-Objekt auf, wobei es sich wahlweise um den zu editierenden oder den neu zu erstellenden Kunden handelt.

KundeAktuell ist ein Verweis auf den zu bearbeitenden Kunden. **DBContext** speichert den Datenbankkontext. Und **anreden** speichert die Anreden und dient später als Datenherkunft des Kombinationsfeldes zur Auswahl der Anreden.

Außerdem deklarieren wir in der Klasse zwei Ereignishandler. Der erste heißt **ItemAdded** und soll ausgelöst werden, wenn ein neuer Datensatz hinzugefügt wurde und gespeichert werden soll.

Der zweite heißt **ItemChanged** und soll feuern, wenn ein zum Bearbeiten übergebener Kunde bearbeitet und gespeichert wurde. Die Deklarationen sehen so aus: