Datenbank-Programmierung mit Visual C# 2012 (Buch + E-Book)

Grundlagen, Rezepte, Anwendungsbeispiele

von
Walter Doberenz, Thomas Gewinnus

1. Auflage

Thematische Gliederung:

Microsoft Programmierung

Microsoft 2013

Verlag C.H. Beck im Internet:
www.beck.de
ISBN 978 3 86645 466 8

Kapitel 10

SQLite – ein Mini ganz groß

In diesem Kapitel:

In diesem Kapitel wollen wir Ihnen eine sinnvolle Alternative sowohl für den meist überdimensionierten Einsatz des Microsoft SQL Servers, egal ob Express oder LocalDB, als auch für die Verwendung von Microsoft Access-Datenbanken als lokale Datenspeicher vorstellen.

Die Hauptforderungen nach

- einfacher Installation/Distribution
- Unterstützung bekannter Technologien (ADO.NET, LINQ to SQL, Entity Framework)
- Aufhebung der Restriktionen bezüglich der maximalen Datenbankgröße[1]
- Unterstützung für Datenbindung
- gute Performance
- Plattformunabhängigkeit des Datenformats
- und, last but not least, die Datensicherheit

werden von dem im Folgenden vorgestellten SQLite in jedem Fall erfüllt. Das hat mittlerweile auch Microsoft erkannt – SQLite fungiert neuerdings als Mini-Datenbank für die Windows Store Apps[2]. Grund für diesen Rückgriff auf eine externe Lösung ist der gänzliche Mangel an hauseigener Datenbankunterstützung für diese Apps, die vorhandene IndexedDB ist für C#-Apps derzeit nicht nutzbar.

HINWEIS Wir beschränken uns an dieser Stelle ganz bewusst auf lokale Datenspeicher, viele Anwendungen erfordern nach wie vor keine Server-Infrastruktur und werden mit viel zu viel Ballast (zusätzliche Dienste, Probleme mit UAC, Datensicherung etc.) beim Kunden »abgeworfen«. Administratoren und Anwender sind Ihnen sicher dankbar dafür, wenn Sie eine einfach installierbare Anwendung anbieten, die nicht gleich das gesamte System »umgräbt«, um ein paar Datensätze zu speichern. Vielfach reicht auch schon eine XML-Datei, aber das ist eine andere Geschichte.

Was ist eigentlich SQLite?

Bei SQLite handelt es sich um eine Desktop-Datenbankengine, die im Gegensatz zum SQL Server ohne eine extra Server-Anwendung auskommt. Die komplette Funktionalität wird von **einer** DLL bereitgestellt, die Anwendung greift direkt auf den eigentlichen Datenspeicher zu. Der Clou an dieser Lösung: Sie können trotz allem mit SQL als Abfragesprache arbeiten, müssen sich also nicht erst an eine neue Schnittstelle gewöhnen[3].

Einen grundsätzlichen Überblick zum Datenformat, zur verwendeten SQL-Syntax und zur DLL-Schnittstelle bietet Ihnen die folgende Website

WWW http://www.sqlite.org/

[1] Insbesondere dieser Punkt dürfte für viele Programmierer von Interesse sein, ist doch das Datenlimit von 2 GByte bei Access-Datenbanken nicht mehr zeitgemäß.

[2] Zumindest so lange, bis Microsoft endlich eine eigen Lösung auf die Beine gestellt hat.

[3] Am besten können Sie SQLite noch mit dem SQL Server Compact vergleichen, beide haben einen konzeptionell ähnlichen Ansatz.

Im Folgenden wollen wir Ihnen mit einer unverbindlichen Gegenüberstellung der Vor- und Nachteile die Entscheidung für oder gegen SQLite erleichtern.

Vorteile

Davon bietet SQLite jede Menge:

- Die Datenbankengine ist winzig im Vergleich zu den etablierten Produkten (die DLL hat lediglich eine Größe von ungefähr 1 MB).
- Es ist keinerlei administrativer Aufwand notwendig, wenn Sie mal vom Speichern der eigentlichen Datendatei absehen.
- Das Format ist ideal für die Verwendung im Zusammenhang mit dem Compact Framework, da geringer Ressourcenbedarf.
- Alle Daten sind in einer Datei zusammengefasst, endlose Dateilisten, wie bei dBase oder Paradox, sind nicht zu befürchten.
- Die komplette Engine befindet sich in einer bzw. zwei Dateien (Compact Framework).
- SQLite implementiert eine Großteil der SQL92-Spezifikation, Sie können also Ihre SQL-Know-How weiter nutzen und müssen nicht umlernen.
- SQLite-Datenbanken sind plattformkompatibel, d.h., Sie können die Datei problemlos mit anderen Systemen auslesen und bearbeiten. Für fast jede Plattform und Programmiersprache werden entsprechende Schnittstellen angeboten. Dies ist im Zusammenhang mit dem Datenaustausch zu Android- und iOS-Anwendungen interessant.
- SQLite ist in einigen Punkten schneller[1] als eine entsprechende SQL Server Compact-Datenbank und die Dateien sind kleiner. Im Gegensatz zum SQL Server Compact kann man bei einer maximalen Datenbankgröße von 2 Terabyte kaum noch von einer Größenbegrenzung sprechen.
- Datenbanken können verschlüsselt werden.
- Unterstützung für Trigger, Views und Constraints.
- SQLite unterstützt verschiedene Formen der Volltextsuche, ein Feature, auf das wir z.B. bei Access-Datenbanken schon lange warten.
- Es sind ADO.NET 2.0 Provider verfügbar, auch die Verwendung des Entity Frameworks ist möglich.
- Optional ist auch ein Zugriff per ODBC-Treiber möglich.
- SQLite ist komplett kostenlos, der Quellcode ist ebenfalls verfügbar.
- **SQLite ist eine der wenigen Datenbankengines, die Sie derzeit in einer WinRT-App überhaupt zum Laufen bekommen.**

HINWEIS Insbesondere der letzte Punkt ist ein echtes »Killerfeature«, wir gehen ab Seite 695 auf die spezifische Lösung im Rahmen von WinRT ein.

[1] Hier kommt es jedoch auf eine sinnvolle Indizierung der Tabellen an, andernfalls bricht die Performance recht schnell ein.

Nachteile

Jede Medaille hat zwei Seiten und so müssen Sie auch bei SQLite mit einigen Einschränkungen und Nachteilen leben.

- Grundsätzlich sollten Sie immer das Konzept als Desktop-Datenbank im Auge behalten. Sie können zwar mit mehreren Anwendungen auf die Datendatei zugreifen, allerdings ist der Schreibmechanismus der Engine etwas eigenwillig, nur ein Prozess kann exklusiv auf die Datenbank zugreifen, Lesezugriffe werden in dieser Zeit geblockt.
- Keine Unterstützung für Stored Procedures und UDFs, Sie können jedoch eigene Scalar- und Aggregat-Funktionen schreiben, die als Callback in Ihrer Anwendung abgelegt sind.
- Es sind keine geschachtelten Transaktionen möglich.
- Keine direkte Replikationsunterstützung, Sie können jedoch eine zweite Datenbank mit ATTACH einbinden und nachfolgend die Daten mit einer Abfrage über die betreffenden Tabellen synchronisieren.
- Keine Unterstützung für Nutzer- und Rechteverwaltung, es handelt sich um eine Desktop-Datenbank, die Sie jedoch verschlüsseln können.

Vorbereitungen

Haben Sie sich für SQLite als Datenformat entschieden, ist der nächste Schritt die Auswahl eines geeigneten Datenproviders, der uns auch unter .NET wie gewohnt zur Verfügung steht.

Download/Installation

Die Autoren haben sich in diesem Fall für *System.Data.SQLite*, einen kostenlosen Wrapper und ADO-.NET 2.0/3.5-Provider, entschieden, da dieser sehr gut dokumentiert und auch aktuell ist. Ganz nebenbei ist auch eine entsprechende Integration in Visual Studio vorhanden. Sie können also die Datenbanken, wie vom Microsoft SQL Server gewohnt, in der Visual Studio-IDE bearbeiten und abfragen (Server-Explorer).

Herunterladen können Sie die Installation unter der Adresse:

WWW http://system.data.sqlite.org/

Laden Sie das *Setup for 32-bit Windows (.NET Framework 4.5)* herunter, wenn Sie über Visual Studio 2012 verfügen, für Visual Studio 2010 nutzen Sie das *Setup for 32-bit Windows (.NET Framework 4.0)*.

HINWEIS Achten Sie darauf, für welche Framework-Version Sie die Installation herunterladen!

Nach dem Download führen Sie das Setup-Programm aus, um die Designtime-Unterstützung in Visual Studio zu integrieren.

Die Besonderheit dieses Projekts ist eine Unterstützung sowohl für das komplette, als auch für das Compact-Framework. Arbeiten Sie mit dem normalen Framework, wird für Ihr Projekt bzw. das Zielsystem lediglich die Datei *System.Data.SQLite.DLL* benötigt, in dieser befindet sich die SQLite-Engine und der für

uns wichtige .NET-Wrapper. Für den Einsatz mit dem Compact Framework müssen Sie die Dateien *System.Data.-SQLite.DLL* (Unterordner *\\CompactFramework*) und *SQLite.Interop.066.DLL* auf dem Zielsystem bereitstellen.

HINWEIS *System.Data.SQLite* ist »lediglich« ein .NET-Wrapper für die originale SQLite-Engine (aktuell 3.7.15.2), Sie profitieren also auch automatisch von Verbesserungen und Neuerungen an der SQLite-Engine.

Mehr über die SQLite-Engine erfahren Sie unter folgender Adresse:

WWW http://www.sqlite.org/docs.html

Integration in Ihr C#-Projekt

Möchten Sie den Provider in Ihr Projekt integrieren, fügen Sie zunächst einen Verweis auf die Assembly *System.Data.SQLite* hinzu und legen die Eigenschaft *Lokale Kopie* auf *True* fest. Benötigen Sie zusätzlich auch Unterstützung für LINQ, fügen Sie noch die Assembly *System.Data.SQLite.Linq* hinzu.

Der einfachste Weg zur Integration in Ihr C#-Projekt führt jedoch über den NuGet-Manager. Wählen Sie eines der vier möglichen Pakete und klicken Sie auf *Installieren*:

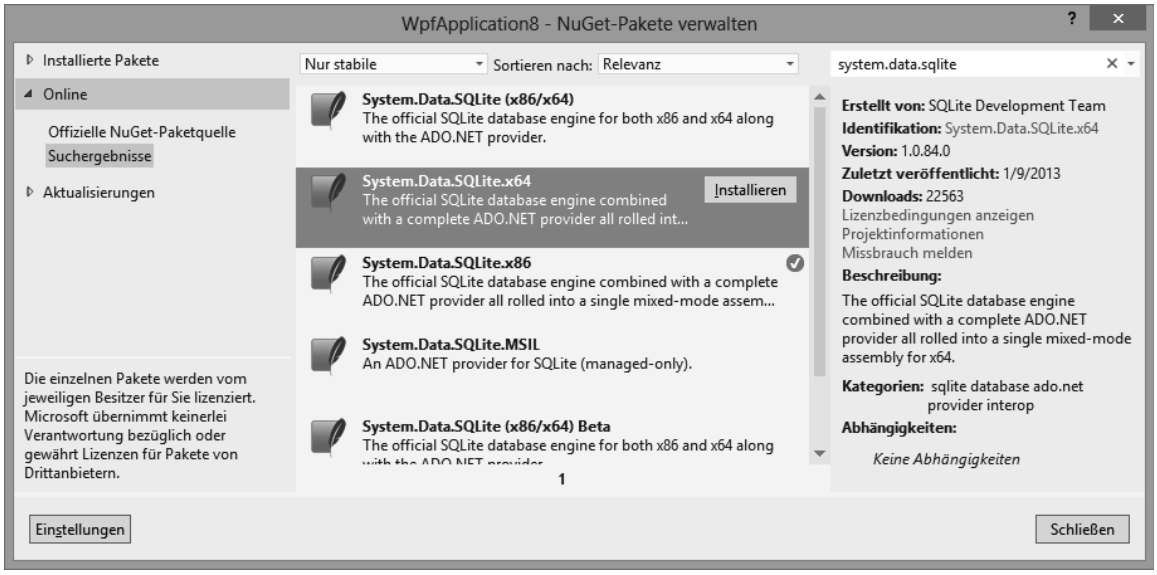


Abbildung 10.1 Installation per NuGet-Paket

Sollten Sie das Paket *System.Data.SQLite (x86/x64)* gewählt haben, werden Ihrem Projekt zwei Unterverzeichnisse *x86* und *x64* hinzugefügt, in denen jeweils die plattformspezifische *SQLite.Interop.dll* abgelegt ist. Im Hauptverzeichnis finden Sie die Assembly *System.Data.SQLite.dll*.

Datenbank-Tools

Eine Datenbank-Engine ist ja gut und schön, aber wer hat schon Lust, Datenbanken ausschließlich per Code zu erstellen bzw. zu administrieren? Aus diesem Grund möchten wir zunächst einen Blick auf einige der verfügbaren Werkzeuge werfen, bevor wir auf die Details der Programmierung mit C# eingehen.

Verwalten von SQLite-Datenbanken mit Visual Studio

Dank vollständiger Integration in die Visual Studio-IDE stellt es kein Problem dar, über den Server-Explorer eine neue Datenbank zu erzeugen. Wählen Sie einfach im Server-Explorer die Schaltfläche *Mit Datenbank verbinden* und ändern Sie die Datenquelle in *SQLite Database File*. Der dazugehörige Datenanbieter *.NET Framework DataProvider for SQLite* ist bereits automatisch ausgewählt (siehe folgende Abbildung).

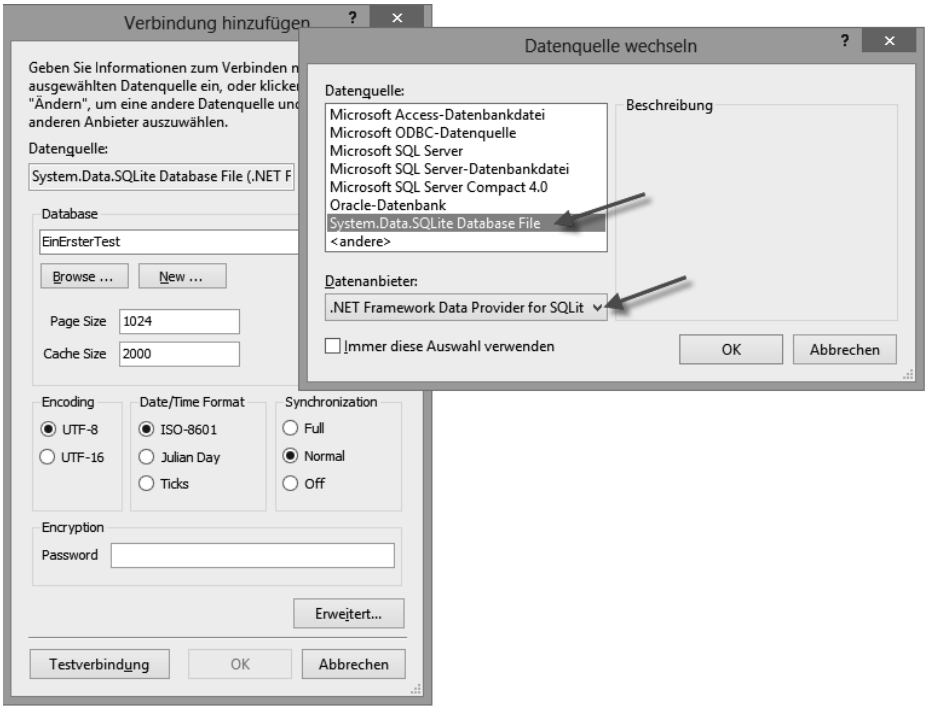


Abbildung 10.2 Neue SQLite-Datenbank erzeugen

Im eigentlichen Verbindungsdialog können Sie jetzt die neue Datenbank erzeugen. Legen Sie hier auch Page-Size und Cache-Size fest. Das Date-/Time-Format belassen Sie am besten bei *ISO-8601* (formatierte Zeichenkette). Mit *Synchronisation* ist das Verhalten beim Speichern von Änderungen gemeint:

- die Einstellung *Normal* führt dazu, dass Änderungen immer dann geschrieben werden, wenn kritische Codeabschnitte durchlaufen werden
- *Full* führt zu Schreibzugriffen bei jeder Änderung und
- *Off* bedeutet, dass die Schreibpuffer nicht explizit geschrieben werden

HINWEIS Welche Datei-Extension Sie für die Datenbank verwenden ist egal, empfehlenswert ist *.db* oder *.db3*.

Optional haben Sie die Möglichkeit, die Datenbank mit einem Passwort zu schützen, versprechen Sie sich davon aber bitte nicht eine extreme Sicherheit, es handelt sich lediglich um einen RC4-Algorithmus mit 128 Bit-Schlüssel.

Schließen Sie den Dialog ab, können Sie sich mittels Server-Explorer um die inneren Werte der Datenbank kümmern, d.h. Tabellen und Sichten erstellen, Trigger und Check-Einschränkungen festlegen und, last but not least, auch Indizes erzeugen.

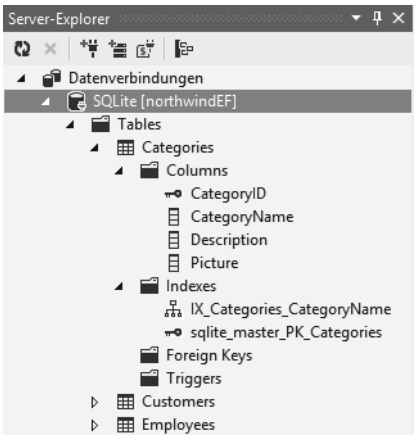


Abbildung 10.3 SQLite-Beispieldatenbank *NorthwindEF* im Server-Explorer

HINWEIS Der Editor für das Bearbeiten von Tabellenlayouts ist leider noch nicht ausgereift, da fehlen noch einige Funktionen. Wir verweisen Sie deshalb besser an das im Folgenden vorgestellte Tool.

Database .NET

Mit diesem Programm, quasi der »eierlegenden Wollmilchsau« des Datenbankentwicklers, haben Sie unter anderem auch Ihre SQLite-Datenbanken voll im Griff.

Laden Sie sich die frei verfügbare Anwendung unter folgender Adresse herunter:

WWW http://fishcodelib.com/Database.htm

Nach dem Download und dem Entpacken der einzigen EXE-Datei können Sie bereits loslegen, eine Installation oder Registrierung ist nicht nötig. Damit eignet sich das Programm auch phantastisch für den USB-Stick zum Mitnehmen, vor allem deshalb, weil neben SQLite auch noch folgende Datenbanken unterstützt werden:

- Microsoft Access, Microsoft Excel
- Firebird
- dBase, FoxPro
- OData, Generic OLE DB, Generic ODBC

- SQL Server, LocalDB, SQL Server Compact, SQL Azure
- MySQL
- Oracle
- IBM DB2
- IBM Informix
- PostgreSQL
- Sybase ASE

HINWEIS Wer mag, kann auch eine Pro-Version erwerben, diese bietet einen erweiterten SQL-Editor, Datenbankdiagramme, einen Profiler etc.

Doch zurück zum Programm. Öffnen Sie eine bestehende SQLite-Datenbank über den Menüpunkt *Datei/VerbindenSQLite*, bzw. erstellen Sie auf diesem Weg auch eine neue Datenbank. Nachfolgend können Sie sich entweder im SQL-Editor oder per Assistent an der Datenbank austoben.

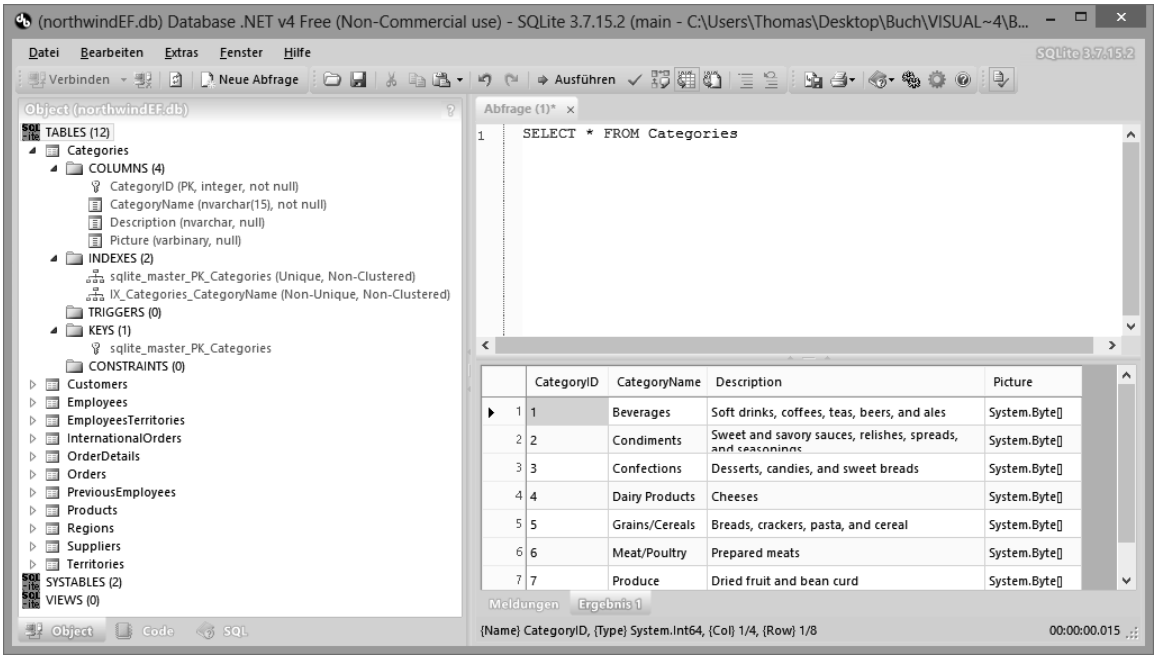


Abbildung 10.4 *Database .NET* in Aktion

Für die Bearbeitung des Tabellenlayouts steht Ihnen ein leistungsfähiger Editor zur Verfügung, Sie müssen also nicht umständlich mit endlosen CREATE TABLE-Statements herumhantieren:

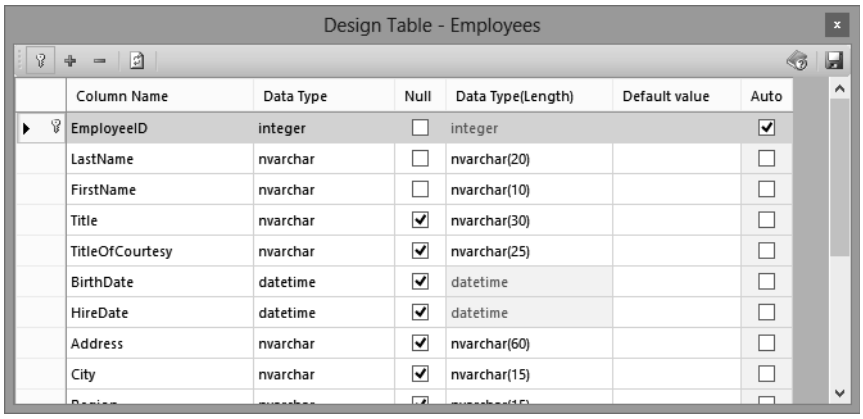


Abbildung 10.5 Entwurf des Tabellenlayouts in *Database .NET*

Ein recht praktisches Feature, Sie können sich aus der fertigen Datenbank auch ein SQL-Skript erstellen lassen, um die Datenbank beispielsweise erst beim Kunden per Skript zu generieren:

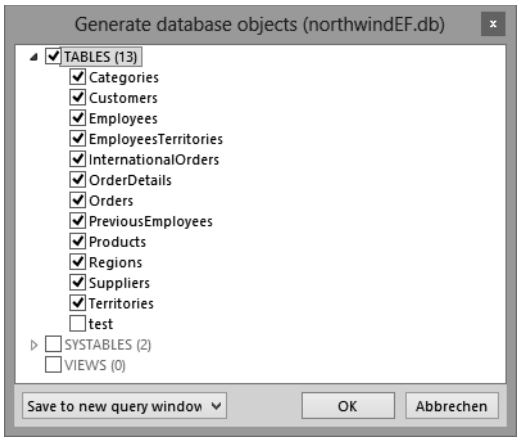


Abbildung 10.6 SQL-Skript generieren
(Sie können einzelne Tabellen auswählen)

HINWEIS *Database .NET* unterstützt beide Versionen der Volltextsuche von SQLite (FTS3/FTS4).

SQLite Administrator

Ein weiteres empfehlenswertes Tool ist *SQLite Administrator,* das Sie kostenlos unter folgender Adresse herunterladen können:

WWW http://sqliteadmin.orbmu2k.de/

Entpacken Sie einfach den Inhalt der heruntergeladenen ZIP-Datei in ein Verzeichnis Ihrer Wahl und starten Sie die Datei *sqliteadmin.exe*. Nach Beantwortung der Frage nach der gewünschten Anzeigesprache können Sie auch schon loslegen:

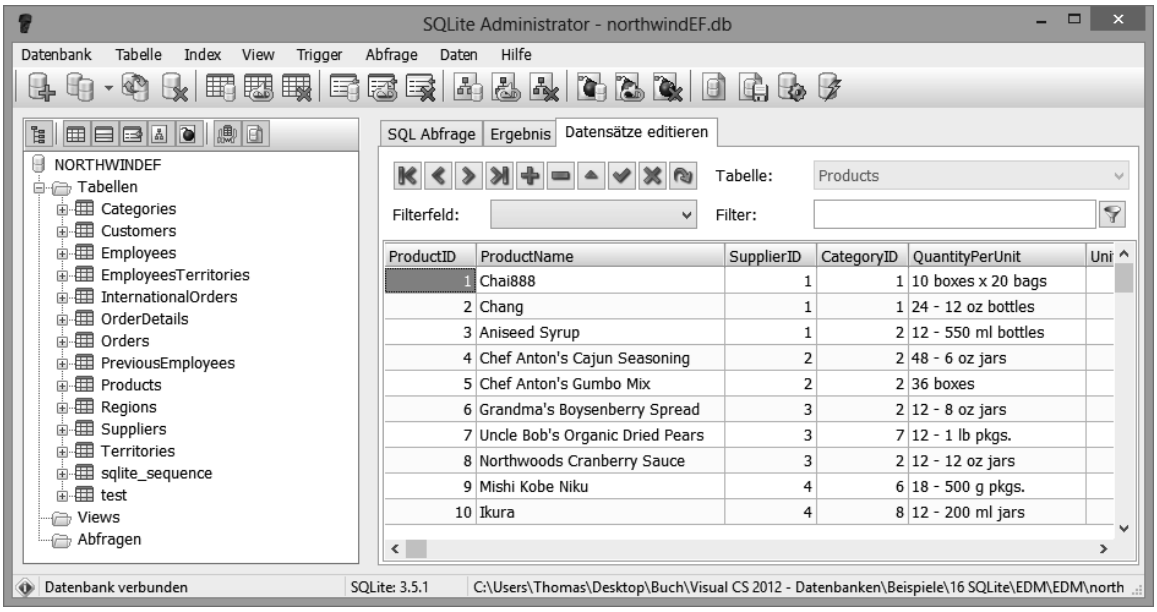


Abbildung 10.7 Der *SQLite Administrator* in Aktion

Wer kein großer Freund von SQL ist, kann hier auch auf recht einfache Art und Weise eigene Trigger und Sichten definieren:

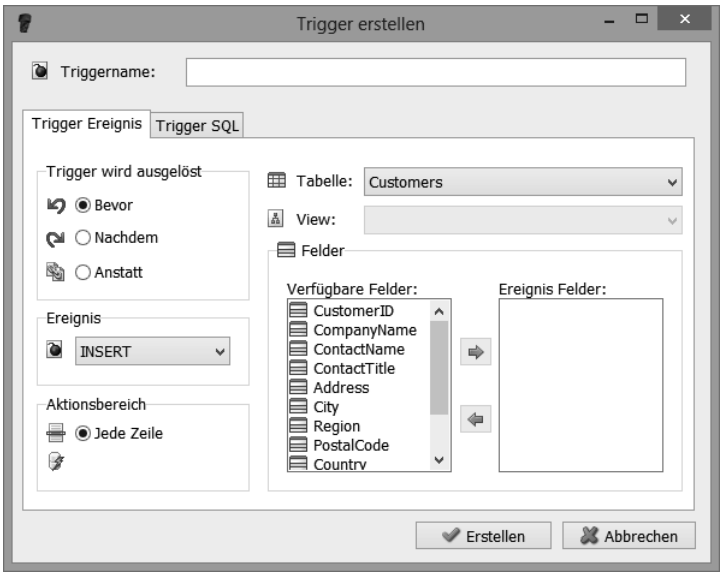


Abbildung 10.8 Neuen Trigger erstellen

Die allgemeine Programmbedienung sollte Sie vor keine allzu großen Herausforderungen stellen, wir gehen deshalb nicht weiter darauf ein.

Praktische Aufgabenstellungen

Da wir ja mit einem ADO.NET-Provider arbeiten, treffen die Ausführungen der ADO.NET-Kapitel 3 und 4 im Grunde auch auf SQLite-Datenbanken zu, wir wollen deshalb hier auf Wiederholungen verzichten.

Auch SQL steht nicht im Mittelpunkt dieses Abschnitts, sondern ist lediglich Mittel zum Zweck. Wir beschränken uns im Weiteren auf einige SQLite-spezifische Aufgabestellungen und Besonderheiten.

Datenbank/Datenbankobjekte per Code erstellen

Hier sind wir endlich an einem der interessantesten Punkte für den Programmierer angelangt. Wie erstelle ich zur Laufzeit meine Datenbank-Objekte? Anhand einiger kleinerer Beispiele zeigen wir Ihnen die grundsätzliche Vorgehensweise.

HINWEIS Voraussetzung für die folgenden Beispiele ist die Einbindung des Wrappers *System.Data.Sqlite* (siehe dazu Seite 670).

BEISPIEL

Erzeugen einer neuen, leeren Datenbank

```
...
using System.Data.SQLite;
...
        private void button1_Click(object sender, EventArgs e)
        {
```

Mit der statischen *CreateFile*-Methode erzeugen Sie im Anwendungsverzeichnis eine neue Datenbank *test3.db3*:

```
            SQLiteConnection.CreateFile(AppDomain.CurrentDomain.BaseDirectory + "test.db3");
        }
```

HINWEIS Der Wrapper unterstützt nicht die Angabe *|DataDirectory|*, verwenden Sie deshalb in diesem Fall die Eigenschaft *BaseDirectory*, wie im obigen Beispiel gezeigt.

BEISPIEL

Ein Passwort für den Datenbankzugriff festlegen

```
...
using System.Data.SQLite;
...
        private void button2_Click(object sender, EventArgs e)
        {
```

Öffnen der Verbindung (mehr zum Connectionstring siehe Seite 678):

```
            conn = new SQLiteConnection("Data Source=" + AppDomain.CurrentDomain.BaseDirectory +
                                        "test.db3");
            conn.Open();
```

Passwort festlegen:

```
conn.ChangePassword("geheim");
```

Verbindung schließen:

```
conn.Close();
```

Vor dem erneuten Öffnen der Datenbank ist jetzt die Angabe des Passwortes erforderlich, andernfalls tritt ein Laufzeitfehler auf

```
conn.SetPassword("geheim");
conn.Open();
conn.Close();
}
```

HINWEIS Die Fehlermeldung *File opened that is not a database file, file is encrypted or is not a database* ist vielleicht etwas missverständlich, fangen Sie mögliche Fehler also ab, und geben Sie eigene Meldungen an den Anwender aus.

BEISPIEL

Wer es gern übersichtlicher hat, kann auch einen *SQLiteConnectionStringBuilder* verwenden:

```
...
SQLiteConnectionStringBuilder csb = new SQLiteConnectionStringBuilder();
csb.DataSource = AppDomain.CurrentDomain.BaseDirectory + "test.db3";
csb.Password = "geheim";
SQLiteConnection.CreateFile(csb.ConnectionString);
...
```

Mögliche Connectionstring-Parameter

Als Ergänzung zum vorhergehenden Abschnitt wollen wir Ihnen im Folgenden noch die wichtigsten Parameter für den Connectionstring vorstellen (siehe Tabelle 10.1).

Parameter	Bedeutung	Standard
Data Source	Kompletter Datenbankpfad inklusive Dateiname	
UseUTF16Encoding	Welche Kodierung soll verwendet werden (True/False)	False
DateTimeFormat	Das verwendete Datumsformat (Ticks/ISO8601)	ISO8601
BinaryGUID	Speicherformat für GUID-Spalten (True = Binär, False = Text)	True
Cache Size	Cachegröße in Bytes	2000
Synchronous	Wann wird der Puffer geschrieben (Normal/Full/Off)	Normal
Page Size	Seitengröße in Bytes	1024
Password	Optional eine Passwortangabe	
Pooling	Verwendet Connection Pooling (True/False)	False

Tabelle 10.1 Die wichtigsten Connectionstring-Parameter

Parameter	Bedeutung	Standard
FaillfMissing	True – ist die Datenbank nicht vorhanden, wird ein Fehler ausgelöst False – ist die Datenbank nicht vorhanden, wird sie automatisch erzeugt	False
Max Page Count	Beschränkung der Seitenzahl (und damit der Datenbankgröße)	0 = keine
Legacy Format	True – verwendet das kompatiblere 3.x Datenbankformat False – verwendet das neuere 3.3x Datenbankformat mit besserer Kompression	False
Default Timeout	Timeout in Sekunden	30
Journal Mode	Delete – Löschen des Journals nach einem Commit Persist – Das Journal wir geleert und verbleibt auf der Festplatte Off – Das Journal wird nicht erzeugt Memory – Das Journal wir d im Speicher gehalten WAL – Es wird ein Write-Ahead-Log verwendet (erst ab Version 3.7.0)	Delete
Read Only	True – Datenbank schreibgeschützt öffnen False – Datenbank mit Schreib-Lesezugriff öffnen	False
Max Pool Size	Connectionpool-Größe	100

Tabelle 10.1 Die wichtigsten Connectionstring-Parameter *(Fortsetzung)*

Tabellen erzeugen

Nach dem Erzeugen der Datenbank herrscht noch gähnende Leere, dem wollen wir jetzt abhelfen. Erster Schritt ist meist das Erstellen neuer Tabellen, alle anderen Objekte bauen ja mehr oder weniger darauf auf.

BEISPIEL

Erzeugen einer neuen Tabelle (beachten Sie, dass wir jetzt ein Passwort übergeben müssen, da die Datenbank verschlüsselt ist).

```
private void button3_Click(object sender, EventArgs e)
{
    conn = new SQLiteConnection("Data Source=" + AppDomain.CurrentDomain.BaseDirectory +
                                "test.db3");
    conn.SetPassword("geheim");
    conn.Open();
```

Die folgende Zeilen dürften Ihnen bereits bekannt vorkommen (siehe ADO.NET-Kapitel 3).

Mittels *SQLiteCommand*-Objekt wird ein SQL-Statement an die Datenbankengine abgesetzt, um die gewünschte Tabelle zu erstellen:

```
    SQLiteCommand cmd = conn.CreateCommand();
    cmd.CommandText = "CREATE TABLE IF NOT EXISTS kunden (" +
                      "     Id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT," +
                      "     Vorname VARCHAR(50) NOT NULL," +
                      "     Nachname VARCHAR(50) NOT NULL," +
                      "     Telefon VARCHAR(50)" +
                      " );";
    cmd.ExecuteNonQuery();
    cmd.Dispose();
    conn.Close();
}
```

HINWEIS Auf dem gleichen Weg können Sie auch alle anderen Datenbankobjekte (Views, Trigger etc.) erzeugen, die Vorgehensweise unterscheidet sich nicht von der bei einem *OleDBCommand*- oder eine *SQLCommand*-Objekt. Über die zulässigen SQL-Befehle und deren Syntax klärt Sie die Hilfedatei zum *System.Data.SQLite-Wrapper* auf.

Bitte umlernen – Datentypen einmal anders

Kommen wir noch einmal zu den obigen Datentypen zurück. Grundsätzlich kennt SQLite nur die folgenden Datentypen:

- NULL
- INTEGER (1, 2, 3, 4, 6 oder 8 Bytes, je nach Subtyp)
- REAL (8 Byte IEEE-Fließkommazahl)
- TEXT (UTF-8, UTF-16BE oder UTF-16LE)
- BLOB

Alle anderen Datentypen, die Sie angeben, werden intern einem der obigen Datentypen zugeordnet. Geben Sie also beispielsweise NVARCHAR(25) an, ist das zwar recht vorbildlich, intern können Sie aber in der erzeugten TEXT-Spalte speichern so viel Sie wollen. Hier hilft dann nur eine Constraint weiter.

Bevor wir Sie jetzt mit endlosen Zuordnungslogiken verwirren, werfen Sie lieber einen Blick auf die folgende Tabelle, die für Sie als C#-Programmierer sicher wesentlich aufschlussreicher ist.

SQL-Datentyp	System.Data.SQLite-Datentyp
TINYINT	*DbType.Byte*
SMALLINT	*DbType.Int16*
INT	*DbType.Int32*
COUNTER, AUTOINCREMENT, IDENTITY, LONG, INTEGER, BIGINT	*DbType.Int64*
VARCHAR, NVARCHAR, CHAR, NCHAR, TEXT, NTEXT, STRING, MEMO, NOTE, LONGTEXT, LONGCHAR, LONGVARCHAR	*DbType.String*
DOUBLE, FLOAT	*DbType.Double*
REAL	*DbType.Single*
BIT, YESNO, LOGICAL, BOOL	*DbType.Boolean*
NUMERIC, DECIMAL, MONEY, CURRENCY	*DbType.Decimal*
TIME, DATE, TIMESTAMP, DATETIME, SMALLDATE, SMALLDATETIME	*DbType.DateTime*
BLOB, BINARY, VARBINARY, IMAGE, GENERAL, OLEOBJECT	*DbType.Binary*
GUID, UNIQUEIDENTIFIER	*DbType.Guid*

Tabelle 10.2 Datentypen SQL versus System.Data.SQLite-Provider

HINWEIS Die Datentypen TEXT und BLOB haben eine maximale Kapazität von 1.000.000.000 Bytes.

Einschränkungen definieren

Wie schon erwähnt, können Sie sich nicht darauf verlassen, dass sich die Datenbankengine auch darum kümmert, dass in eine NVARCHAR(25)-Spalte auch nur 25 Zeichen eingegeben werden können. Wer hier für Konsistenz und Ordnung sorgen will, kommt um eine ganze Reihe von Constraint bzw. CHECK-Klauseln nicht herum.

BEISPIEL

Verwendung von CHECK-Klauseln

```
CREATE TABLE test (
```

Speichert Zeichenkette mit maximal 10 Zeichen:

```
   Spalte1 VARCHAR(10) CHECK (LENGTH(Spalte1) < 11),
```

Speichert Zeichenkette mit den zulässigen Werten *Mann* und *Frau*:

```
   Spalte2 VARCHAR(4)  NOT NULL CHECK (Spalte2 IN ('Mann', 'Frau')),
```

Speichert Gleitkommazahl mit den zulässigen Werten *NULL* oder *größer 5*:

```
   Spalte3 REAL NULL CHECK (Spalte3 > 5)
);
```

Wir könnten sicherlich noch viele weiteren Möglichkeiten aufzeigen, aber da verweisen wir Sie besser an die Dokumentation auf der SQLite-Website.

Datenbankzugriff per DataSet realisieren

Haben Sie die Datenbank mit den entsprechenden Datenbankobjekten erzeugt, wollen Sie sicher auch auf die Tabellen und Abfragen zugreifen. Nichts leichter als das, denn haben Sie Kapitel 4 (*DataSet*-Objekt) eingehend studiert, werden Sie keine Probleme haben, auch auf SQLite-Datenbanken zuzugreifen.

Zwei kleine Beispiele zeigen den Zugriff auf die Tabelle *Kunden*, die wir im vorhergehenden Beispiel erstellt hatten.

BEISPIEL

Laden der Daten in ein *DataSet*

```
using System.Data.SQLite;
...
    public partial class Form1 : Form
    {
        SQLiteConnection conn;
        DataSet ds;
        SQLiteDataAdapter da;
```

```
        private void button4_Click(object sender, EventArgs e)
        {
```

Verbindung öffnen:

```
            conn = new SQLiteConnection("Data Source=" + AppDomain.CurrentDomain.BaseDirectory +
                                        "test.db3");
            conn.SetPassword("geheim");
            conn.Open();
```

Datenauswahl:

```
            da = new SQLiteDataAdapter("SELECT * FROM kunden", conn);
```

Wir wollen automatisch im *DataSet* neue Ids vergeben:

```
            da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
```

DataSet füllen:

```
            ds = new DataSet();
            da.Fill(ds, "Kunden");
```

DataGridView anbinden:

```
            dataGridView1.DataSource = ds;
            dataGridView1.DataMember = "Kunden";
            conn.Close();
        }
```

Das dürfte Ihnen recht bekannt vorkommen, lediglich die Klassenbezeichner unterscheiden sich etwas (aus *OleDbDataAdapter* wird ein *SQLiteDataAdapter* etc.).

Abschließen sollen die Daten auch aus dem *DataSet* zurück in die Datenbank geschrieben werden.

BEISPIEL

Zurückschreiben der Änderungen in die SQLite-Datenbank

```
        private void button5_Click(object sender, EventArgs e)
        {
```

Wir erzeugen die nötigen UPDATE-, INSERT und DELETE- Statements per *CommandBuilder*:

```
            SQLiteCommandBuilder cb = new SQLiteCommandBuilder(da);
            conn.SetPassword("geheim");
            conn.Open();
```

Änderungen schreiben:

```
            da.Update(ds, "Kunden");
            conn.Close();
        }
```

HINWEIS Alternativ können/sollten Sie ein typisiertes Dataset verwenden, auch dieses wird unterstützt.

Besonderheit: InMemory-Datenbank

Auf eine Besonderheit der SQLite-Engine wollen wir an dieser Stelle noch einmal besonders eingehen. Die Rede ist von der Möglichkeit, SQLite-Datenbanken komplett im Arbeitsspeicher abzulegen. Alle Abfragen funktionieren wie bekannt, es wird jedoch nichts auf die Platte ausgelagert.

HINWEIS Dass Sie an dieser Stelle nicht von der maximalen Datenbankgröße von 2 TB Gebrauch machen können, dürfte Ihnen sicher klar sein.

Ein Beispiel zeigt die Vorgehensweise.

BEISPIEL

Erzeugen einer InMemory-Datenbank und Verwendung eines *DataSet*-Objekts

```
using System.Data.SQLite;
...
    public partial class Form1 : Form
    {
```

Die zentralen Zugriffsobjekte:

```
        SQLiteConnection conn;
        DataSet ds;
        SQLiteDataAdapter da;

        private void button8_Click(object sender, EventArgs e)
        {
```

Datenbank im Speicher erzeugen:

```
            conn = new SQLiteConnection("Data Source=:memory:");
            conn.Open();
```

Objekte auf die bekannte Weise erstellen:

```
            SQLiteCommand cmd = conn.CreateCommand();
            cmd.CommandText = "CREATE TABLE IF NOT EXISTS kunden (" +
                              "     Id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT," +
                              "     Vorname VARCHAR(50) NOT NULL," +
                              "     Nachname VARCHAR(50) NOT NULL," +
                              "     Telefon VARCHAR(50)" +
                              " );";
            cmd.ExecuteNonQuery();
```

Daten abfragen und anzeigen:

```
            da = new SQLiteDataAdapter("SELECT * FROM kunden", conn);
            da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
            ds = new DataSet();
            da.Fill(ds, "Kunden");
            dataGridView1.DataSource = ds;
            dataGridView1.DataMember = "Kunden";
        }
```

HINWEIS Beachten Sie, dass die Connection in diesem Fall **nicht geschlossen** wird! Die Datenbank wäre sonst »futsch«.

Daten vom *DataSet* zurück in die InMemory-Datenbank schreiben:

```
        private void button9_Click(object sender, EventArgs e)
        {
            SQLiteCommandBuilder cb = new SQLiteCommandBuilder(da);
            da.Update(ds, "Kunden");
        }
```

Hier fragen wir probeweise noch weitere Daten aus der InMemory-Datenbank ab:

```
        private void button1_Click(object sender, EventArgs e)
        {
            SQLiteCommand cmd = conn.CreateCommand();
            da = new SQLiteDataAdapter("SELECT nachname FROM kunden", conn);
            ds = new DataSet();
            da.Fill(ds, "Kunden");
            dataGridView1.DataSource = ds;
            dataGridView1.DataMember = "Kunden";
        }
    }
}
```

HINWEIS Leider wird durch SQLite keine Cursor-Programmierung und damit möglicherweise ein Recordset-artiges Konstrukt angeboten. So werden die Daten teilweise doppelt im Speicher gehalten (InMemory-Datenbank und DataSet).

Datenzugriff mit dem Entity Framework

Auch hier werden Sie nicht von großartigen Neuigkeiten überrascht, Sie können problemlos für eine SQLite-Datenbank ein passendes Entity Data Modell erstellen und mit diesem wie gewohnt arbeiten.

HINWEIS Voraussetzung dafür ist allerdings die korrekte Installation des SQLite-DataProviders (siehe dazu Seite 670).

Der grundsätzliche Ablauf ist auch hier:

1. Wählen Sie in einem vorhandenen Projekt den Menüpunkt *Projekt/Neues Element hinzufügen/Daten/ADO.NET Entity Data Model.*
2. Mit *Generate from DataBase* rufen Sie den *Enity Data Model Wizard* auf. Hier klicken Sie auf *New Connection*.
3. Wechseln Sie im nächsten Dialog die Datenquelle und wählen Sie *System.Data.SqLite Database File*.
4. Nach einem Klick auf *OK* können Sie bereits die Verbindung zur gewünschten SQLite-Datenbank herstellen.
5. Wählen Sie nun die Tabellen aus, die in das Datenmodell eingeschlossen werden sollen.

Die folgende Abbildung 10.9 zeigt das Ganze noch einmal in einer Übersicht.

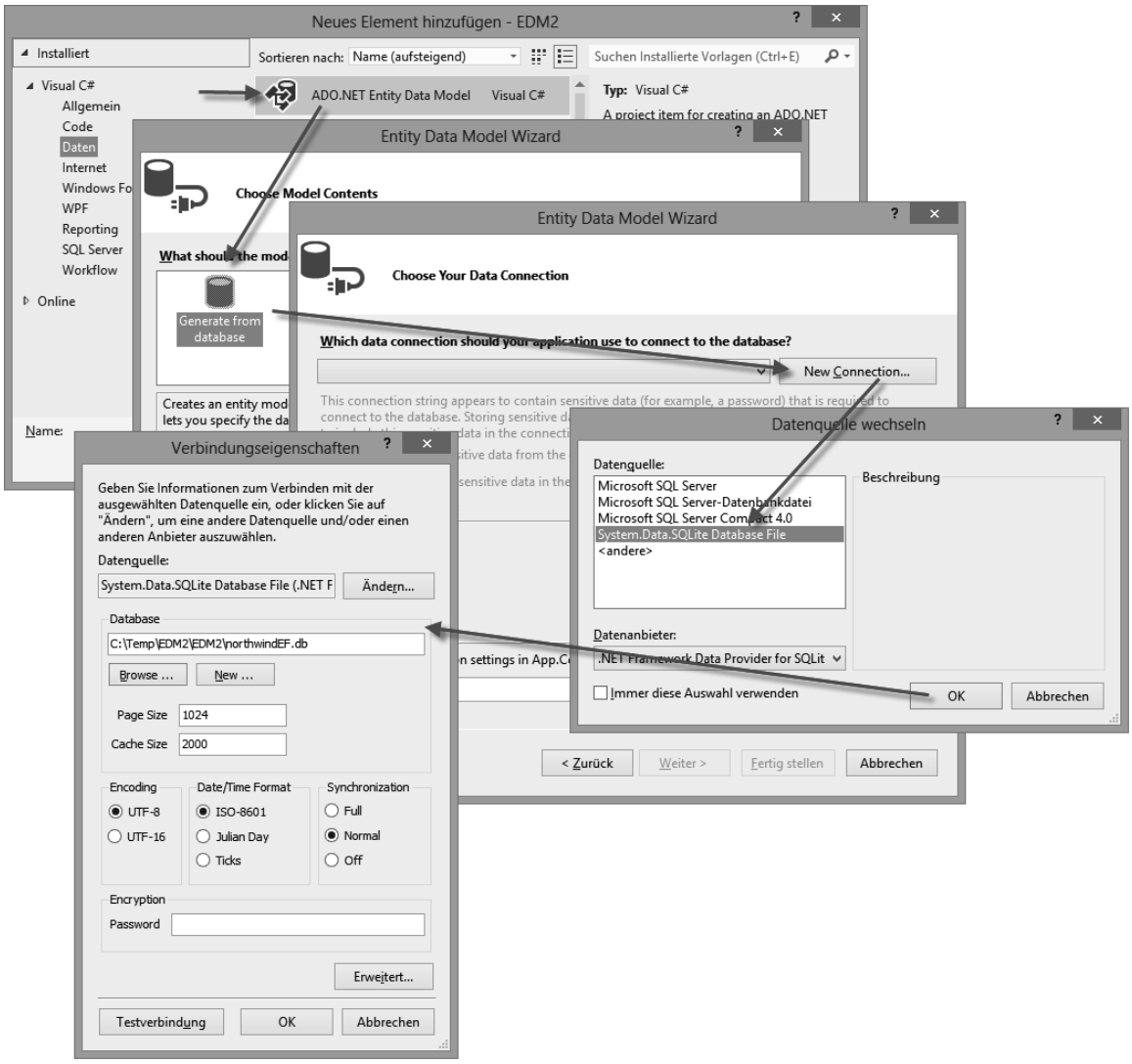


Abbildung 10.9 Übersicht zum Aufbau einer Verbindung

Leider wird das |*DataDirectory*|-Konstrukt im Connectionstring in diesem Fall nicht unterstützt, der Pfad zur Datenbank ist hart kodiert. Machen Sie es sich leicht und editieren Sie nach dem Erstellen des Modells die Datei *App.Config*. Ersetzen Sie die Pfadangabe zum Beispiel durch »XXXX«. Zur Laufzeit ersetzen wir diese Angabe mit dem Pfad zur eigentlichen Datenbankdatei.

BEISPIEL

Dynamisches Anpassen des Connectionstrings

```
using System.Configuration;
...
```

```
private void button1_Click(object sender, EventArgs e)
{
   string connstr = ConfigurationManager.ConnectionStrings["northwindEFEntities"].ConnectionString;
   connstr = connstr.Replace("XXXX", AppDomain.CurrentDomain.BaseDirectory + "northwindef.db");
   db = new northwindEFEntities(connstr);
```

Datenbindung herstellen (Windows Forms):

```
   dataGridView1.DataSource = db.Products;
}
```

Der zugehörige Connectionstring in der Datei *App.config*:

```
  <connectionStrings>
    <add name="northwindEFEntities" connectionString="metadata=res://*/NWModel.csdl|
res://*/NWModel.ssdl|res://*/NWModel.msl;provider=System.Data.SQLite;provider connection string='data
source=&quot;XXXX&quot;'" providerName="System.Data.EntityClient" />
  </connectionStrings>
```

Arbeiten Sie mit einem *DbContext* (statt mit einem *ObjectContext*, wie im obigen Beispiel), müssen Sie zur Anpassung des Connectionstrings wie folgt vorgehen, da Sie dem Konstruktor keinen Connectionstring übergeben können.

BEISPIEL

Connectionstring für *DataContext* anpassen

```
...
            db = new northwindEFEntities();
            string connstr = db.Database.Connection.ConnectionString;
            connstr = connstr.Replace("XXXX", AppDomain.CurrentDomain.BaseDirectory +
                                      "northwindef.db");
            db.Database.Connection.ConnectionString = connstr;
```

Daten lokal laden:

```
            db.Products.Load();
```

Datenanbindung an die lokalen Daten (WPF):

```
            this.DataContext = db.Products.Local;
...
```

HINWEIS Weitere Informationen zur Arbeit mit dem Entity Framework siehe Kapitel 12.

Die Bedeutung von Transaktionen bei SQLite

Sicherlich haben Sie schon etwas mit den SQLite-Klassen herumgespielt. Solange Sie aber nur einige wenige Datensätze zwischen Programm und Datenbank austauschen, werden Sie kaum über einen der größten Fallstricke von SQLite stolpern. Doch wehe, Sie möchten in einem Schwung zum Beispiel 10.000 Datensätze aus einer XML-Datei oder einer anderen Quelle importieren. In diesem Fall werden Sie zunächst maßlos

enttäuscht sein. So benötigt der Import obiger 10.000 Datensätze auf einem schnellen Rechner sage und schreibe 90 Sekunden, ein Wert der wohl nicht mehr ganz zeitgemäß ist.

An dieser Stelle lohnt es sich, einen Blick auf die Abläufe beim Speichern von Datensätze in SQLite-Datenbanken zu werfen. Fügen Sie Daten in eine SQLite-Datenbank ein, und ist dieses INSERT-Statement nicht in einer Transaktion gekapselt, wird intern automatisch eine Transaktion gestartet (verbunden mit dem Erstellen einer Journaldatei), die INSERT-Anweisung ausgeführt und die Transaktion abgeschlossen (COMMIT). Das Ganze erfolgt 10.000 Mal, der immense Zeitbedarf dürfte damit klar erkennbar sein.

Die Lösung: Eine explizite Transaktion für den kompletten Einfügevorgang.

BEISPIEL

Einfügen von Datensätzen per Transaktion

```
...
       Stopwatch watch = new Stopwatch();
...
           watch.Reset();
           watch.Start();
           using (SQLiteConnection conn = new SQLiteConnection("Data Source=" +
                       AppDomain.CurrentDomain.BaseDirectory + "test.db3"))
           {
               try
               {
                   conn.Open();
                   using (SQLiteTransaction trans = conn.BeginTransaction())
                   {
                       using (SQLiteCommand cmd = conn.CreateCommand())
                       {
                           cmd.CommandText =
                              "INSERT INTO Kunden (Vorname, Nachname) VALUES (@vorname, @nachname)";
                           cmd.Parameters.Add("@vorname", System.Data.DbType.String, 50);
                           cmd.Parameters.Add("@nachname", System.Data.DbType.String, 50);
                           cmd.Prepare();
                           for (int i = 0; i < 10000; i++)
                           {
                               cmd.Parameters["@vorname"].Value = "xxxxxxxxxxx" +
                                                                  i.ToString();
                               cmd.Parameters["@nachname"].Value = "yyyyyyyyyy" +
                                                                  i.ToString();
                               cmd.ExecuteNonQuery();
                           }
                       }
                       trans.Commit();
                   }
               }
               catch (SQLiteException ex)
               {
                   MessageBox.Show(ex.Message, "Fehler");
               }
           }
           watch.Stop();
           MessageBox.Show("Zeit: " + watch.ElapsedMilliseconds.ToString() + " (ms)");
```

Auf dem gleichen PC benötigt obiges Beispiel nur noch 99 Millisekunden für das Einfügen der Daten.

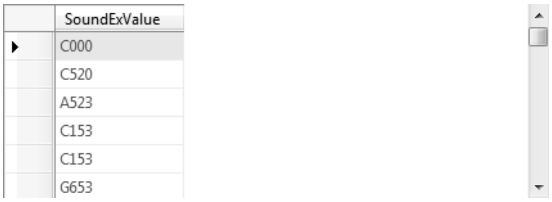
SOUNDEX verwenden

Im Unterschied zum SQL Server Compact bietet die SQLite-Engine auch eine *SoundEx*-Funktion, die Sie in Ihren Abfragen nutzen können:

BEISPIEL

Verwendung von SOUNDEX

```
SELECT
    SOUNDEX(ProductName) AS SoundExValue
FROM
    Products
```



SoundExValue
C000
C520
A523
C153
C153
G653

Abbildung 10.10 Ergebnis obiger Abfrage

Volltextabfragen realisieren

Auch hier bietet SQLite mehr als der SQL Server Compact[1]. Sie können für entsprechend erzeugte Tabellen problemlos Volltextabfragen mit MATCH realisieren.

Die Verwendung ist für den gestandenen SQL Server-Programmierer allerdings etwas »merkwürdig«. So wird nicht etwa eine bestehende Tabelle für die Volltextsuche genutzt, sondern Sie erstellen eine neue »virtuelle« Tabelle, bei der Sie auch noch eine recht eigenartige Syntax verwenden.

Doch der Reihe nach, ein Beispiel sagt uns nicht nur in diesem Fall mehr als tausend Worte.

BEISPIEL

Volltextsuche für E-Mails realisieren (das komplette Beispiel finden Sie in den Begleitdateien)

Zunächst die Namespaces einbinden:

```
using System.Data.SQLite;
using System.IO;
...
    public partial class Form1 : Form
    {
```

Die erforderlichen Objekte für die Arbeit mit einem *DataSet* bereitstellen:

```
        SQLiteConnection conn;
        DataSet ds;
        SQLiteDataAdapter da;
```

[1] Der SQL Server Express kann mit Volltextsuche installiert werden, aber da sind wir schon fast wieder bei einem kompletten SQL Server mit Diensten etc.

Hier erstellen wir zunächst die Datenbank:

```
private void button1_Click(object sender, EventArgs e)
{
    if (File.Exists(AppDomain.CurrentDomain.BaseDirectory + "test.db3"))
         File.Delete(AppDomain.CurrentDomain.BaseDirectory + "test.db3");
    SQLiteConnection.CreateFile(AppDomain.CurrentDomain.BaseDirectory + "test.db3");
```

Jetzt können wir die Verbindung öffnen und die neue Tabelle erzeugen:

```
    conn = new SQLiteConnection("Data Source=" + AppDomain.CurrentDomain.BaseDirectory +
                                "test.db3");
    conn.Open();
    SQLiteCommand cmd = conn.CreateCommand();
    cmd.CommandText = "CREATE VIRTUAL TABLE EMails USING FTS3 (Betreff, Body);";
    cmd.ExecuteNonQuery();
```

Was passiert hier im Detail? Eine Tabelle *Emails* wird unter Verwendung der Volltextsuche (FTS3) mit den beiden Spalten *Betreff* und *Body* erzeugt. Vermutlich vermissen Sie die Typangaben bei dieser Art von Tabellendefinition, aber das wäre unnötige »Folklore«, diese Werte würden in jedem Fall ignoriert. Spalten werden hier immer als Text interpretiert, eine ID benötigen Sie nicht, die wird automatisch über eine interne RowId bereitgestellt[1]. Eine Abfrage à la *SELECT rowid, betreff, body FROM emails* ist also problemlos realisierbar.

Jetzt fügen wir einfach ein paar Datensätze in die Tabelle ein:

```
    cmd.CommandText = "INSERT INTO EMails (Betreff, Body) values (@betreff, @body);";
    cmd.Parameters.Add("@betreff", DbType.AnsiString);
    cmd.Parameters.Add("@body", DbType.AnsiString);

    cmd.Parameters["@betreff"].Value = "Buch fertig";
    cmd.Parameters["@body"].Value = "Nach ja, es sind noch zwei Wochen!";
    cmd.ExecuteNonQuery();

    cmd.Parameters["@betreff"].Value = "Rechnung schreiben";
    cmd.Parameters["@body"].Value = "Sehr geehrte Damen und Herrn, wie Sie vielleicht" +
                                    "befürchtet haben ...";
    cmd.ExecuteNonQuery();

    cmd.Parameters["@betreff"].Value = "Zwei Tage Zeit";
    cmd.Parameters["@body"].Value = "Bis Mittwoch sollten Sie fertig sein!";
    cmd.ExecuteNonQuery();
```

Wir fragen die Tabelle ab und zeigen diese per *DataSet* in einem *DataGridView* an:

```
    da = new SQLiteDataAdapter("SELECT * FROM emails", conn);
    ds = new DataSet();
    da.Fill(ds, "emails");
    dataGridView1.DataSource = ds;
    dataGridView1.DataMember = "emails";
    conn.Close();
}
```

[1] Im Falle einer VIRTUAL TABLE ist dieser Wert auch wirklich eindeutig, Sie können die RowId also problemlos als Schlüssel in einer anderen Tabelle verwenden.

Nun schreiten wir zur Abfrage. Ausgehend von den Eingaben einer *TextBox* (der zukünftige SQL-String) erstellen wir ein zweites *DataSet*, das wir in einem weiteren *DataGridView* anzeigen:

```
        private void button2_Click(object sender, EventArgs e)
        {
            conn = new SQLiteConnection("Data Source=" + AppDomain.CurrentDomain.BaseDirectory +
                                        "test.db3");
            conn.Open();
```

Sicherheitshalber eine kleine Fehlerbehandlung, falls wir wider Erwarten einen falschen SQL-Befehl eingeben:

```
            try
            {
                da = new SQLiteDataAdapter(textBox1.Text, conn);
                ds = new DataSet();
                da.Fill(ds, "emails");
                dataGridView2.DataSource = ds;
                dataGridView2.DataMember = "emails";
            }
            catch (Exception ex)
            {  MessageBox.Show(ex.Message); }
            conn.Close();
        }
    }
}
```

Eine erste Beispielabfrage zeigt die folgende Abbildung 10.11:

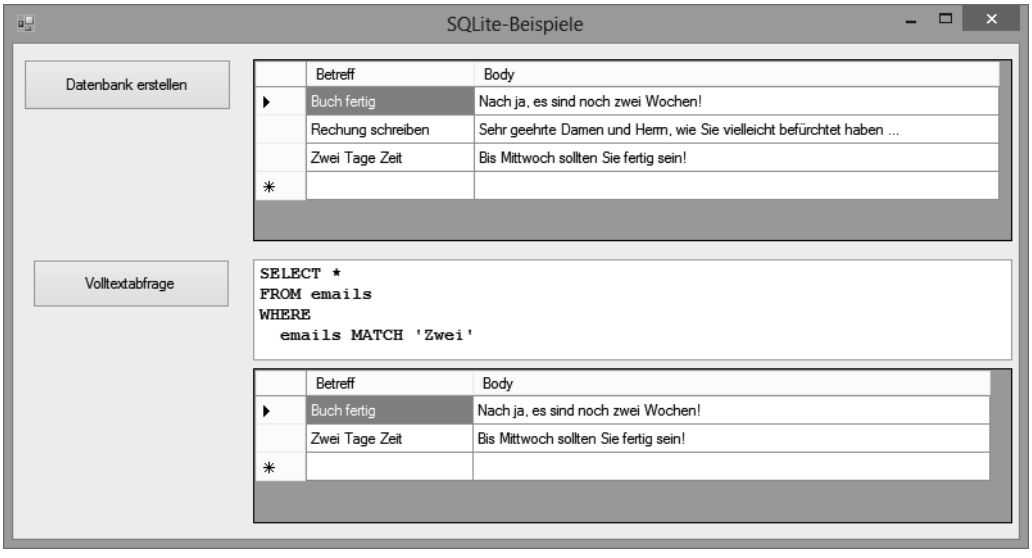


Abbildung 10.11 Das Beispielprogramm in Aktion

Vermutlich wird sich mancher die Augen reiben, fragen wir doch mit

```
... WHERE emails MATCH 'Zwei'
```

eine eigentlich nicht vorhandene Tabellenspalte ab. Sie vermuten richtig, wenn Sie davon ausgehen, dass in diesem Fall alle Spalten abgefragt werden. Die virtuelle Tabelle verfügt zu diesem Zweck über eine interne Spalte mit dem gleichen Namen wie die Tabelle.

Alternativ können Sie auch einen einzelnen Spaltennamen angeben:

```
... WHERE body MATCH 'Zwei'
```

Geht es darum, einen Fund davon abhängig zu machen, dass zwei Wörter nahe beieinander stehen, können Sie NEAR verwenden. Standardmäßig dürfen die Wörter nicht weiter als zehn Terme voneinander entfernt sein, Sie können jedoch auch einen anderen Wert festlegen.

BEISPIEL

Verwendung von NEAR

Maximal 10 Terme Abstand:

```
SELECT * FROM emails WHERE emails MATCH 'ja NEAR Zwei'
```

Maximal 25 Terme Abstand:

```
SELECT * FROM emails WHERE emails MATCH 'ja NEAR/25 Zwei'
```

Neben dem zusätzlichen Operator NEAR können Sie auch AND, OR oder NOT einsetzen, mehrere Token einzeln angeben, Fundstellen lokalisieren etc. Weitere grundlegende Informationen zur Volltext-Engine und deren Syntax finden Sie unter folgender Adresse:

WWW http://www.sqlite.org/fts3.html

Recht interessant ist auch die Möglichkeit, Teile der Fundstelle als Suchergebnis abzurufen. Nutzen Sie dazu die Funktion *Snippet*:

BEISPIEL

Ausschnitt der Fundstelle anzeigen:

```
SELECT
   Snippet(emails)
FROM
   emails
WHERE
   emails MATCH 'Damen'
```

HINWEIS Geben Sie für die *Snippet*-Funktion die virtuelle Spalte mit dem Namen der Tabelle an.

Zurückgegeben wird ein HTML-Fragment, das die Umgebung der Fundstelle zeigt. Der eigentliche Suchtext wird mit fett hervorgehoben (HTML-Ansicht vorausgesetzt), lange Texte werden mit »...« abgeschnitten.

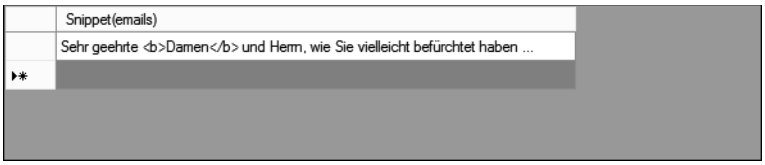


Abbildung 10.12 Der Rückgabewert

HINWEIS Wer weitere Informationen über die Funde abrufen will (Offset, Tabellenspalte etc.), kann dies mit den Funktionen *Offsets* und *Matchinfo* tun.

Noch eine kleine Ergänzung zum Schluss: Neben dem oben schon beschriebenen FTS3-Suchmodul wird mittlerweile auch ein FTS4-Modul unterstützt, das einige zusätzliche Informationen für die *Matchinfo*-Funktion bereitstellen kann. Diese zusätzlichen Informationen erfordern intern zwei weitere Tabellen, was sich auch in einem etwas erhöhten Platzbedarf niederschlägt. FTS4 unterstützt zusätzlich Hooks für das Komprimieren bzw. Dekomprimieren der Daten.

Für neue Projekte wird FTS4 empfohlen, das Verfahren ist in einigen Fällen signifikant schneller, Altprojekte belassen Sie aus Kompatibilitätsgründen am besten bei FTS3.

Und wie nutzen wir nun das FTS4-Modul? Ganz einfach, verwenden Sie im einfachsten Fall statt »FTS3« die Angabe »FTS4«:

```
CREATE VIRTUAL TABLE EMails USING FTS4 (Betreff, Body);
```

Eigene skalare Funktionen in C# realisieren

Nichts leichter als das, auch hier sind SQLite sowie der Wrapper recht komfortabel. Mit Hilfe der Klasse *SQLiteFunction* haben Sie im »Handumdrehen« eine eigene Funktion für Ihre SQL-Abfragen realisiert. Sie müssen lediglich die entsprechende Methode überschreiben.

BEISPIEL

Eine Levenshtein-Funktion[1] realisieren, welche die »Distanz«, d.h. die Ähnlichkeit zwischen zwei Strings berechnet. Dazu werden Einfüge-, Lösch- und Ersetzungsvorgänge bewertet.

Mit den Attributen steuern wir die wichtigsten Eigenschaften der Funktion (Name, Argumentzahl (-1 = beliebig), Funktionsart (in diesem Fall eine skalare Funktion). Der Name der Klasse ist irrelevant.

```
[SQLiteFunction(Name = "Levenshtein", Arguments = 2, FuncType = FunctionType.Scalar)]
class myLevenshtein : SQLiteFunction
{
```

[1] Gefunden unter *http://dotnetperls.com/levenshtein*

Durch Überschreiben der *Invoke*-Methode realisieren wir bereits die gewünschte Funktion:

```
        public override object Invoke(object[] args)
        {
```

Argumente auswerten:

```
            string s = args[0].ToString();
            string t = args[1].ToString();
```

Hier folgt der Algorithmus:

```
            int n = s.Length;
            int m = t.Length;
            int[,] d = new int[n + 1, m + 1];
            if (n == 0) return m;
            if (m == 0) return n;
            for (int i = 0; i <= n; d[i, 0] = i++)
            {}
            for (int j = 0; j <= m; d[0, j] = j++)
            {}
            for (int i = 1; i <= n; i++)
            {
                for (int j = 1; j <= m; j++)
                {
                    int cost = (t[j - 1] == s[i - 1]) ? 0 : 1;
                    d[i, j] = Math.Min(
                                    Math.Min(d[i - 1, j] + 1, d[i, j - 1] + 1),
                                    d[i - 1, j - 1] + cost);
                }
            }
```

Funktionsergebnis liefern:

```
            return d[n, m];
        }
    }
```

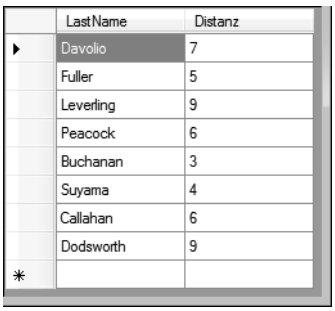
Das folgende Beispiel zeigt, wie Sie die Funktion *Levenshtein* in einer Abfrage verwenden können:

BEISPIEL

Verwendung der obigen Funktion

```
DataSet ds;
SQLiteDataAdapter da;
SQLiteConnection  conn = new SQLiteConnection("Data Source=" +
                         AppDomain.CurrentDomain.BaseDirectory  + "northwindef.db");
conn.Open();
da = new SQLiteDataAdapter("SELECT lastname, Levenshtein(lastname, 'Bucha') As " +
                           "Distanz FROM employees", conn);
ds = new DataSet();
da.Fill(ds, "Mitarbeiter");
dataGridView1.DataSource = ds;
dataGridView1.DataMember = "Mitarbeiter";
conn.Close();
```

Das Ergebnis der Abfrage:



LastName	Distanz
Davolio	7
Fuller	5
Leverling	9
Peacock	6
Buchanan	3
Suyama	4
Callahan	6
Dodsworth	9

Abbildung 10.13 Der Rückgabewert unserer Abfrage

Eigene Aggregat-Funktionen in C# realisieren

Auch in diesem Fall hilft uns die Klasse *SQLiteFunction* weiter, überschreiben Sie einfach die Methoden *Step* und *Final,* um eine eigene Aggregatfunktion zu realisieren.

BEISPIEL

Eine *MaxLenght*-Funktion implementieren (die maximale Stringlänge bestimmen)

```
using System.Data.SQLite;
...
```

Per Attribut bestimmen wir die Funktionsparameter:

```
[SQLiteFunction(Name = "MaxLength", Arguments = -1, FuncType = FunctionType.Aggregate)]
class MyMaxLength : SQLiteFunction
{
```

Diese Methode wird für jeden übergebenen Datensatz aufgerufen (Argumente sind die Feldwerte, der wievielte Aufruf und eine interne Verwaltungsvariable, in der wir das Zwischenergebnis speichern):

```
    public override void Step(object[] args, int nStep, ref object contextData)
    {
```

Den Feldwert auslesen:

```
        string s =  args[0].ToString();
```

Die interne Verwaltungsvariable initialisieren:

```
        if (contextData == null)
            contextData = 0;
        else
```

Zwischenschritt berechnen:

```
            contextData = Math.Max((int)contextData, s.Length);
    }
```

Das Endergebnis zurückgeben:

```
        public override object Final(object contextData)
        {
            return contextData;
        }
    }
```

Zwecks Abruf des Ergebnisses verwenden wir ein *Command*-Objekt:

BEISPIEL

Die Verwendung im Detail

```
conn = new SQLiteConnection("Data Source=" + AppDomain.CurrentDomain.BaseDirectory +
                            "northwindef.db");
conn.Open();
SQLiteCommand cmd = conn.CreateCommand();
cmd.CommandText = "SELECT MaxLength(lastname) FROM employees";

int i =  Convert.ToInt32(cmd.ExecuteScalar());
MessageBox.Show("Ergebnis = " + i.ToString());
```

Der Rückgabewert wird in unserem Beispiel »9« sein, »Dodsworth« ist der längste Eintrag.

SQLite – die Datenbank für Windows Store Apps

Im Eifer des Gefechts haben die Microsoft-Entwickler wohl eine »Kleinigkeit« vergessen. Prinzipiell steht Ihnen derzeit »ab Werk« keine Datenbankengine für Ihre App zur Verfügung! Das dürfte zunächst ein kompletter Show-Stopper für viele Anwendungen sein, die derzeit noch um eine lokale Desktop-Datenbank herum aufgebaut sind. Schnelle und vor allem einfache Abhilfe ist hier nicht in Sicht. Mittlerweile scheint aber Microsoft zur Einsicht gelangt zu sein, dass hier eine gewaltige Lücke klafft.

Auf der Website

WWW http://timheuer.com/blog/archive/2012/06/05/howto-video-using-sqlite-in-metro-style-app.aspx

wird eine Lösung vorgestellt, die allerdings nicht von Microsoft stammt.

Für alle, die sich jetzt voller Euphorie auf die SQLite-Entwicklung stürzen wollen, gleich ein beachtlicher Dämpfer:

HINWEIS In WinRT steht Ihnen ADO.NET nicht zur Verfügung. Also keine DataSets etc., wie im bisherigen Kapitel beschrieben, alle Zugriffe erfolgen über SQLite-spezifische Methoden, der Portierungsaufwand einer bisherigen ADO.NET-Anwendung sollte also nicht unterschätzt werden!

Installation in einem WinRT-Projekt

Mit der SQLite-DLL allein ist es nicht getan, wir benötigen zusätzlich noch einen Wrapper, der uns die Library-Funktionen in ein halbwegs nutzbares Format überträgt. Hilfreich dabei ist die von Frank Krueger erstellte *sqlite-net*-Library.

Sehen wir uns zunächst die Installationsschritte an, bevor wir zur Verwendung der Library kommen:

1. Laden Sie die eigentliche SQLite-DLL herunter. Gegen Sie dazu auf folgende Adresse:

   WWW http://www.sqlite.org/download.html

   In der Rubrik *Precompiled Binaries* ***for Windows Runtime*** wählen Sie entweder die x86- oder die x64-Version (wir verwenden die x86-Version).
2. Entpacken Sie die heruntergeladene ZIP-Datei.
3. Erstellen oder öffnen Sie jetzt ein App-Projekt und kopieren Sie die *sqlite3.dll* aus der ZIP-Datei in Ihr Projekt. Setzen Sie für die DLL die Eigenschaft *In Ausgabeverzeichnis kopieren* auf *Immer kopieren*.
4. Erstellen Sie einen Verweis auf das Microsoft *Visual c++ Runtime Package*:

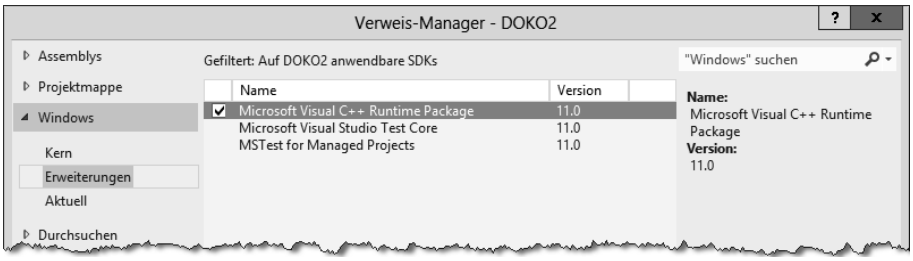


Abbildung 10.14 Verweis hinzufügen

Nach Einfügen des Verweises werden Sie feststellen, dass dieser mit einem kleinen Warnhinweis versehen ist. Ursache ist, dass wir jetzt nicht mehr plattformunabhängig sind, d.h., wir müssen das Projektmappenausgabeziel anpassen (von *Any CPU* auf *x86*):

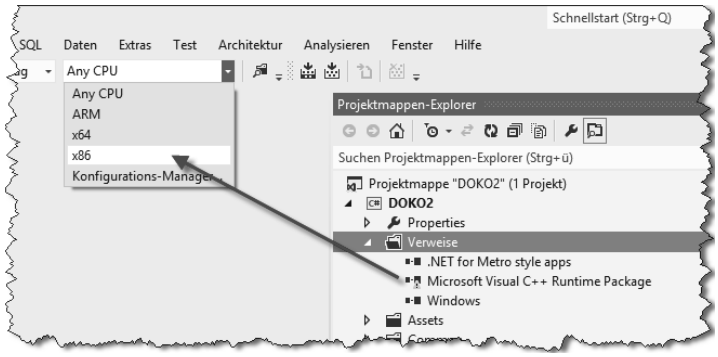


Abbildung 10.15 Anpassen der Zielplattform

5. Klicken Sie nun im Menü *Projekte* auf *NuGet-Packete* verwalten. Geben Sie in der Suchmaske des folgenden Dialogs *sqlite-net* ein, um das Paket zu suchen. Wählen Sie dann die Schaltfläche *Installieren.* Nachfolgend wird Ihr App-Projekt um zwei Wrapperdateien (*SQLite.cs* und *SQLiteAsync.cs*) erweitert.

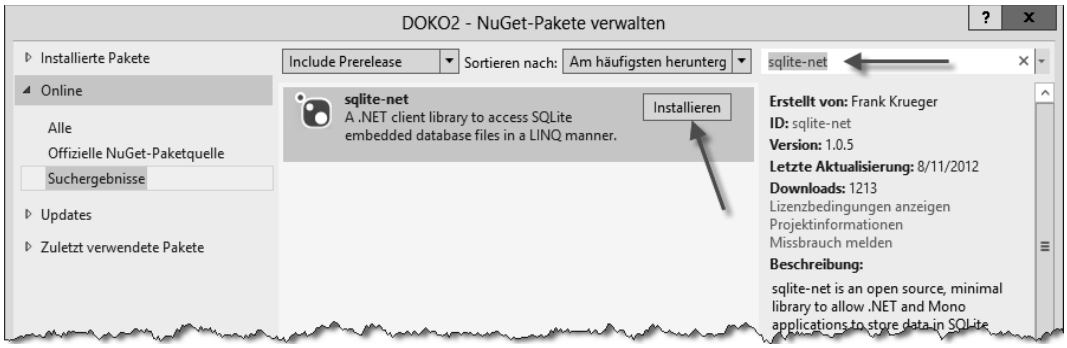


Abbildung 10.16 NuGet-Paket installieren

Damit sind alle Vorbereitungen abgeschlossen und wir können uns der eigentlichen Programmierung zuwenden.

Erstellen der Datenbank

Drei grundsätzliche Varianten bieten sich bei der vorliegenden SQLite-Lösung an:

- Sie erstellen die Datenbank per Code aus vorhandenen Klassen.
- Sie nutzen SQL-Befehle zum Aufbau der Datenbank (oder führen ein Skript aus).
- Sie liefern eine leere Datenbank mit, die Sie beim ersten Start in das App-Datenverzeichnis kopieren (Sie denken bitte daran, dass Sie auf das Installationsverzeichnis nur lesend zugreifen dürfen).

Die folgenden Beispiele zeigen die Grundansätze.

BEISPIEL

Erstellen der Datenbank aus Klassen zur Laufzeit

```
...
using SQLite;
using Windows.Storage;
using System.Threading.Tasks;
...
    public sealed partial class BasicPage1 : SQL_Einführung.Common.LayoutAwarePage
    {
```

Unsere Verbindung zur Datenbank:

```
        private SQLiteConnection db;
        private TableQuery<Artikel> artikeltabelle;

        public BasicPage1()
        {
```

```
            this.InitializeComponent();
            Loaded += BasicPage1_Loaded;
        }
```

Mit dem Laden der Seite wird auch die Datenbank im App-Datenverzeichnis erstellt (ist diese bereits vorhanden, wird sie automatisch geöffnet):

```
        async void BasicPage1_Loaded(object sender, RoutedEventArgs e)
        {
            db = new SQLite.SQLiteConnection(
                      Path.Combine(ApplicationData.Current.LocalFolder.Path, "firma.db"));
```

Erstellen einer Tabelle:

```
            db.CreateTable<Artikel>();
```

Im Hintergrund wird der Befehl CREATE TABLE IF NOT EXISTS ausgeführt, die Tabelle wird also nicht überschrieben. Die Spalteninformationen und Attribute werden intern anhand des übergebenen Typs bestimmt, an dieser Stelle haben Sie also recht wenig Arbeit.

```
        }
...
```

Ganz anders ist das in der Klasse, die später die Datenbank-Entitäten repräsentieren soll. Hier definieren Sie zunächst Eigenschaften wie Sie es gewohnt sind. Nachfolgend steuern Sie über zusätzliche Attribute, welche Eigenschaften die aus den Properties der Klasse generierten Spalten haben sollen:

```
using SQLite;
...
namespace SQL_Einführung
{
```

Unsere Beispiel-Klasse *Artikel*:

```
    public class Artikel : INotifyPropertyChanged
    {
        public Artikel()
        { }
```

Ein Konstruktor zum einfachen Erstellen neuer Instanzen:

```
        public Artikel(string bezeichnung, double preis )
        {
            this._bezeichnung = bezeichnung;
            this._preis = (float) preis;
            this._anzahl = 0;
            this._ausverkauft = true;
        }
```

Unser Primärschlüssel (ein Zählerwert):

```
        private int _id;

        [AutoIncrement, PrimaryKey]
        public int Id
        {
            get { return _id; }
```

```
            set {
                _id = value;
                this.NotifyPropertyChanged("Id");
            }
        }
```

Ein indiziertes Textfeld in der Datenbank mit der Länge 100:

```
        private string _bezeichnung;

        [Indexed, MaxLength(100)]
        public string Bezeichnung
        {
            get { return _bezeichnung; }
            set {
                _bezeichnung = value;
                this.NotifyPropertyChanged("Bezeichnung");
            }
        }
```

Ein Float-Feld:

```
        private float _preis;
        public float Preis
        {
            get { return _preis; }
            set {
                _preis = value;
                this.NotifyPropertyChanged("Preis");
            }
        }
```

Ein Integer-Feld:

```
        private int _anzahl;
        public int Anzahl
        {
            get { return _anzahl; }
            set {
                _anzahl = value;
                this.NotifyPropertyChanged("Anzahl");
            }
        }
```

Ein Boolean-Feld:

```
        private bool _ausverkauft;
        public bool Ausverkauft
        {
            get { return _ausverkauft; }
            set {
                _ausverkauft = value;
                this.NotifyPropertyChanged("Ausverkauft");
            }
        }
```

Dieses Feld wird nur in der App benötigt, es gibt keine Entsprechung in der Datenbank:

```
        [Ignore]
        public string ID_Bezeichnung
        {
            get
            {
                return _id.ToString() + "_" + Bezeichnung;
            }
        }
```

Für die Anzeige im Listenfeld eine überschriebene *ToString*-Methode:

```
        public override string ToString()
        {
            return string.Format("Id:{0} Bez:{1} Preis:{2}", _id, _bezeichnung, _preis);
        }
...
```

Wie Sie sehen, besteht Ihre Hauptarbeit im Definieren der Mapperklassen. Folgende Attribute sind verfügbar:

- *PrimaryKey*
  Kennzeichnet das Feld als Primärschlüssel.
- *AutoIncrement*
  Erstellt ein Zählerfeld (meist gleichzeitig der Primärschlüssel).
- *Indexed*
  Das Feld soll indiziert werden.
- *MaxLength(<anzahl>)*
  Das Textfeld soll eine maximale Länge von <anzahl> haben. Der Standardwert sind 140 Zeichen.
- *Ignore*
  Dieses Feld wird nicht in der Datenbank gespeichert

Folgende Datentypen werden durch den Mapper unterstützt:

- Integer
- Boolean (intern Integer mit 1=*true*)
- Enumerations (intern *Integer*)
- Gleitkommawerte (intern als *float*)
- String (intern *varchars* mit durch *MaxLength* festgelegter Länge)
- *DateTime*

Ja, ja, einen Enterprise-SQL-Server können Sie so nicht ersetzen, aber das dürfte in den meisten Fällen auch gar nicht Ihr Ziel sein.

Neben obigem Weg gibt es auch einen anderen: Sie erstellen die Datenbank bereits fix und fertig und liefern eine leere Version davon mit der App aus. Diese Datenbank befindet sich zunächst im Installationsverzeichnis der App, das bekanntermaßen schreibgeschützt ist. Sie müssen also nur Sorge dafür tragen, diese Datei in das App-Datenverzeichnis zu kopieren.

BEISPIEL

Datenbank-Vorlage aus dem Installations- in das Datenverzeichnis kopieren

```
using SQLite;
using System.Diagnostics;
using Windows.Storage;
using Windows.ApplicationModel;
...
        async void BasicPage1_Loaded(object sender, RoutedEventArgs e)
        {
            if (!await ApplicationData.Current.LocalFolder.FileExistsAsync("firma.db"))
            {
                StorageFile file = await
                   Package.Current.InstalledLocation.GetFileAsync("Vorlage_firma.db");
                await file.CopyAsync(ApplicationData.Current.LocalFolder, "firma.db");
            }
            db = new   SQLite.SQLiteConnection(Path.Combine(
                       ApplicationData.Current.LocalFolder.Path, "firma.db"));
...
```

Vermutlich werden Sie vergebens nach der Methode *FileExistsAsync* Ausschau halten, denn diese schreiben wir uns selbst:

```
  static class Erweiterungsmethoden
    {
        public async static Task<bool> FileExistsAsync(this StorageFolder folder,
                                                       string filename)
        {
            try
            {
                await folder.GetFileAsync(filename);
                return true;
            }
            catch (Exception)
            { return false; }
        }
    }
```

HINWEIS Mit dem SQLite-Administrator (siehe *http://sqliteadmin.orbmu2k.de/*) können Sie die Datenbanken komfortabel entwerfen.

Daten einfügen, lesen und abfragen

An dieser Stelle halten wir uns mit weitschweifigen Ausführungen zurück, ein kleines Beispiel soll genügen.

BEISPIEL

Manipulieren der SQLite-Datenbank

```
...
        async void BasicPage1_Loaded(object sender, RoutedEventArgs e)
        {
...
            db = new SQLite.SQLiteConnection(
```

```
                Path.Combine(ApplicationData.Current.LocalFolder.Path, "firma.db"));
            db.CreateTable<Artikel>();  // ein CREATE TABLE IF NOT EXISTS
```

Wir fragen die Anzahl der Datensätze in der Tabelle *Artikel* ab:

```
            int anzahl = db.ExecuteScalar<int>("SELECT COUNT(*) FROM artikel");
```

Sind keine Datensätze vorhanden, erstellen wir vier neue Einträge:

```
            if (anzahl == 0)
            {
                db.RunInTransaction(() =>
                    {
                        db.Insert(new Artikel("Kuchenhörnchen", 1.45));
                        db.Insert(new Artikel("Brötchen", 0.46));
                        db.Insert(new Artikel("Brot", 1.99));
                        db.Insert(new Artikel("Eis rot", 1.35));
                    });
            }
```

Wir fragen die komplette Tabelle ab und zeigen diese an:

```
            artikeltabelle = db.Table<Artikel>();
            ListBox1.ItemsSource = artikeltabelle;
        }
...
```

Einen einzelnen Artikel einfügen und die Tabelle erneut abfragen:

```
        private void Button_Click_2(object sender, RoutedEventArgs e)
        {
```

Artikel erzeugen:

```
            Artikel art = new Artikel("Weissbrot", 1.66);
```

In der Datenbank speichern:

```
            db.Insert(art);
```

Den Autowert abfragen:

```
            Debug.WriteLine("Der neue Autowert ist: " + art.Id.ToString());
            artikeltabelle = db.Table<Artikel>();
            ListBox1.ItemsSource = artikeltabelle;
        }
```

Wir löschen den ersten Artikel in der Liste:

```
        private void Button_Click_3(object sender, RoutedEventArgs e)
        {
            db.Delete(artikeltabelle.First());
```

Tabelle erneut abfragen und anzeigen:

```
            artikeltabelle = db.Table<Artikel>();
            ListBox1.ItemsSource = artikeltabelle;
        }
```

Wir fragen alle Artikel ab, die einen Preis größer 1,4 Euro haben:

```
        private void Button_Click_4(object sender, RoutedEventArgs e)
        {
           ListBox1.ItemsSource = db.Query<Artikel>(
                                              "SELECT * FROM artikel WHERE preis > 1.4");
        }
...
```

Damit belassen wir es an dieser Stelle, es steht zu vermuten, dass Microsoft sich noch aufrafft und im Laufe der nächsten Monate etwas sinnvolles auf die Beine stellt.

Tipps & Tricks

Im Folgenden zeigen wir Ihnen einige interessante Lösungen und Möglichkeiten mit und für SQLite.

Für Liebhaber der Kommandozeile – Sqlite3.exe

Unter der Adresse

WWW http://www.sqlite.org/download.html

finden Sie in der Rubrik *Precompiled Binaries For Windows* das gewünschte Tool *sqlite3.exe* (siehe *sqlite-shell-win32-x86-3071502.zip*). Laden Sie die ZIP-Datei herunter und entpacken Sie den Inhalt (eine EXE) in ein Verzeichnis Ihrer Wahl.

HINWEIS Dies ist quasi das Pendant zu *sqlcmd.exe* vom Microsoft SQL Server.

Möchten Sie eine Datenbank öffnen genügt es, wenn Sie die betreffende Datenbank per Drag & Drop auf die EXE ziehen.

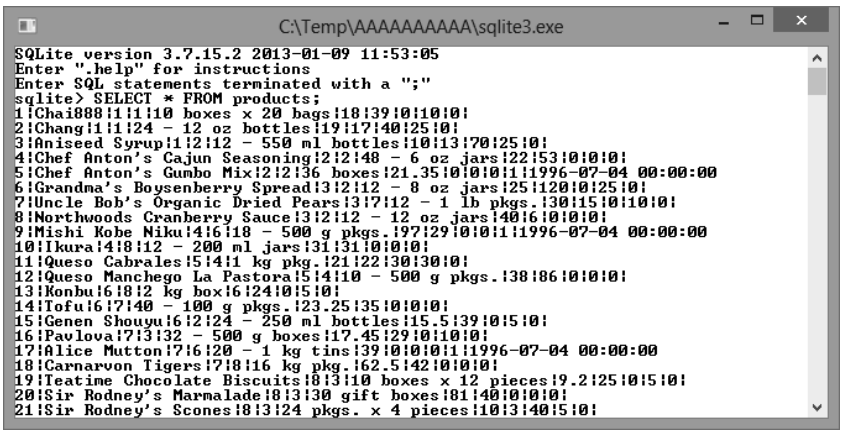


Abbildung 10.17 Abfragen per Kommandozeilentool

Nachfolgend können Sie schon Ihre SQL-Kenntnisse prüfen, alternativ stehen Ihnen auch einige zusätzliche Kommandos zur Verfügung. Über deren Verwendung informieren Sie sich bitte in der Onlinehilfe.

Im Folgenden zeigen wir Ihnen an einem kleinen Beispielskript, wie Sie einen Datenbankexport im SQL-Format realisieren können.

BEISPIEL

Ein Mini-Skript *ExportSQLite.cmd*

```
sqlite3.exe %1 .dump >> output.sql
```

Ziehen Sie jetzt eine Datenbank per Drag & Drop auf dieses Skript, so wird die komplette Datenbank im SQL-Format ausgegeben (Abbildung 10.18).

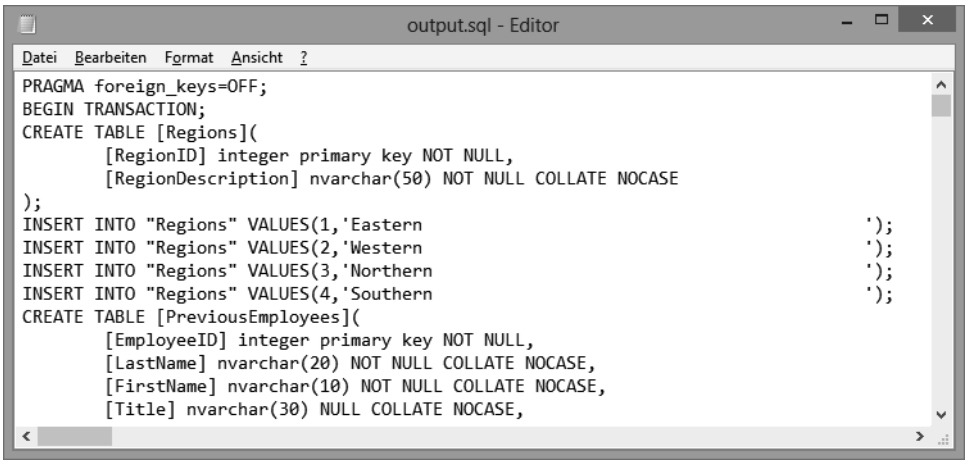


Abbildung 10.18 Ausschnitt aus der Datei *output.sql*

Diese Funktionalität dürfte vor allem beim Exportieren der Daten in Fremdformate recht nützlich sein, einige kleine Anpassungen genügen meist und Sie können das Skript beispielsweise auf einen Microsoft SQL Server einspielen.

Eine SQLite-Datenbank reparieren

Mit dem blöden Spruch, dass eine Sicherungskopie immer der bessere Weg zu einer intakten Datenbank sei, ist Ihnen nicht geholfen, wenn keine Sicherungskopie existiert bzw. wenn diese zu alt ist. Wenn also »das Kind in den Brunnen gefallen ist« und Ihre App mysteriöse Fehler[1] produziert, sollten Sie sich mit dem Gedanken anfreunden, dass die Datenbank beschädigt ist.

Mit dem im vorhergehenden Abschnitt vorgestellten Programm *sqlite3.exe* können Sie zunächst eine Fehlerprüfung realisieren:

[1] z.B. *The database disk image is malformed.*

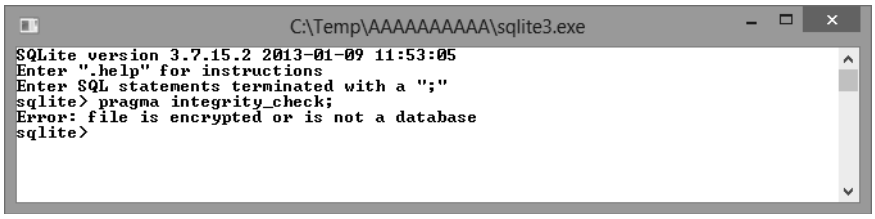


Abbildung 10.19 Integrität der Datenbank prüfen

Mit

```
pragma integrity_check;
```

starten Sie eine Integritätsprüfung, die in obigem Beispiel leider negativ ausfällt. Jetzt gilt es zu retten, was zu retten ist.

Zwei Varianten bieten sich an:

- Sie verwenden den *.restore*-Befehl von *sqlite3.exe*
- Sie exportieren die defekte Datenbank im SQL-Format

Die folgende Abbildung zeigt die erste Variante in Aktion:

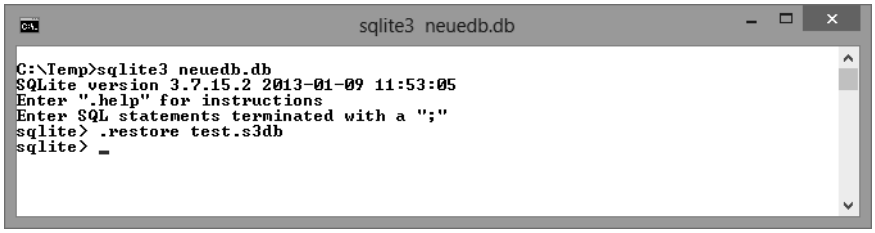


Abbildung 10.20 Wiederherstellen einer Datenbank

Nun zu Variante 2: Mit dem Skript aus dem vorhergehenden Abschnitt können Sie ein SQL-Skript der kompletten Datenbank erstellen. Dieses kann nachbearbeitet und später ausgeführt werden. Abschließend sollten Sie wieder eine funktionstüchtige Datenbank besitzen.

HINWEIS Eine Garantie, dass mit den beiden obigen Verfahren die Datenbank in jedem Fall wiederhergestellt werden kann, lässt sich nicht geben. Hier bleibt als Rettungsanker letztendlich nur die Sicherungskopie.

Eine Beispieldatenbank herunterladen

Für erste Tests eignet sich nach wie vor die *Northwind*-Datenbank recht gut. Eine entsprechende Portierung in das SQLite-Format finden Sie unter der Adresse:

WWW http://sqlite.phxsoftware.com/forums/t/1377.aspx

Eine Datenbank ver- und entschlüsseln

Wie Sie eine Datenbank verschlüsseln, haben wir Ihnen ja bereits ab Seite 677 gezeigt, an dieser Stelle nur noch mal ein kurzer Auszug.

BEISPIEL

Datenbank verschlüsseln

```
using System.Data.SQLite;
...
            conn = new SQLiteConnection("Data Source=test.db3");
            conn.Open();
            conn.ChangePassword("geheim");
```

Wie Sie sehen verwenden wir *ChangePassword*, um eine unverschlüsselte Datenbank zu verschlüsseln. Auf gleichem Weg können wir diese Datenbank auch wieder entschlüsseln.

BEISPIEL

Datenbank entschlüsseln

```
using System.Data.SQLite;
...
            conn = new SQLiteConnection("Data Source=test.db3");
```

Wir setzen das Passwort:

```
            conn.SetPassword("geheim");
            conn.Open();
```

Und hier wird es gelöscht:

```
            conn.ChangePassword("");
            conn.Close();
```

Eine verschlüsselte Datenbank öffnen

Hier haben Sie drei Möglichkeiten:

- Sie übergeben das Passwort im Connectionstring:

```
conn = new SQLiteConnection("Data Source=test.db3; Password=geheim");
```

- Sie verwenden die *SetPassword*-Methode mit einem String:

```
conn.SetPassword("geheim");
```

- Sie übergeben der *SetPassword*-Methode ein Byte-Array:

```
cnn.SetPassword(new byte[] { 0xFF, 0xEE, 0xDD, 0x10, 0x20, 0x30 });
```

Die Datenbank defragmentieren

Wie auch bei jedem anderen Datenbanksystem wird die Datendatei durch Löschvorgänge und Änderungen mit der Zeit fragmentiert, d.h., es verbleiben »Leerstellen« in der Datei und diese wächst immer mehr an. Um es kurz zu machen: SQLite stellt die SQL-Anweisung VACUUM zur Verfügung, um die Datenbank zu reorganisieren.

BEISPIEL

Datenbank defragmentieren

```
conn = new SQLiteConnection("Data Source=" + AppDomain.CurrentDomain.BaseDirectory +
                            "test.db3");
conn.Open();
SQLiteCommand cmd = conn.CreateCommand();
cmd.CommandText = "VACUUM;";
cmd.ExecuteNonQuery();
```

Mehrere Datenbanken verknüpfen

Möchten Sie Abfragen über mehrere Datenbankdateien realisieren, hilft Ihnen die ATTACH-Anweisung weiter. Führen Sie diese in einer geöffneten Verbindung aus, wird eine weitere Datenbankdatei eingebunden. Sie haben nachfolgend die Möglichkeit, Abfragen über Tabellen aus zwei verschiedenen Datenbanken zu realisieren.

BEISPIEL

Öffnen der Testdatenbank *test.db* und die Datenbank *Northwindef.db* anhängen

```
conn = new SQLiteConnection("Data Source=" + AppDomain.CurrentDomain.BaseDirectory +
                            "test.db");
conn.Open();
```

Jetzt wird die Datenbank angehängt, wir vergeben den Aliasnamen »XYZ«:

```
SQLiteCommand cmd = conn.CreateCommand();
cmd.CommandText = @"ATTACH DATABASE 'northwindef.db' AS XYZ;";
cmd.ExecuteNonQuery();
```

Ab sofort können Sie über die Connection auch *Northwind*-Tabellen abfragen:

```
da = new SQLiteDataAdapter("SELECT * FROM XYZ.products", conn);
ds = new DataSet();
da.Fill(ds, "Produkte");
dataGridView1.DataSource = ds;
dataGridView1.DataMember = "Produkte";
conn.Close();
```

HINWEIS Zusammen mit der Anweisung INSERT INTO können Sie per SQL auf diese Weise Tabellendaten zwischen zwei Datenbanken austauschen.

Testen, ob eine Tabelle vorhanden ist

Möchten Sie eine Tabelle löschen, wenn diese bereits existiert, nutzen Sie folgende SQL-Anweisung:

```
DROP TABLE IF EXISTS [TabellenName];
```

Nachfolgend können Sie die Tabelle dann neu erstellen:

```
CREATE TABLE [TabellenName] (
   ...
);
```

Möchten Sie prüfen, ob eine spezifische Tabelle in der Datenbank bereits vorhanden ist, nutzen Sie folgende Hilfsfunktion:

```
bool TableExists(string tblName)
{
  return (conn.GetSchema("Tables").Select("Table_Name = '" + tblName + "'").Length > 0);
}
```

Die Verwendung zeigt das folgende Beispiel:

BEISPIEL

Test auf vorhandene Tabelle

```
if (TableExists("Kunden")) MessageBox.Show("Tabelle ist vorhanden!");
```

Eine Abfrage/Tabelle kopieren

SQLite stellt »ab Werk« keine SELECT INTO-Abfrage zur Verfügung (Einfügen von Abfragewerten in neue Tabelle) und so bleibt nichts anderes übrig, als dies mit zwei Abfragen zu »simulieren«.

BEISPIEL

Das Abfrageergebnis soll in eine neue Tabelle *Abfragedaten* kopiert werden. Ist die Tabelle nicht vorhanden, soll Sie erstellt werden.

```
CREATE TABLE IF NOT EXISTS Abfragedaten AS SELECT * FROM Products;
INSERT INTO Abfragedaten SELECT * FROM Products;
```

Die erste Anweisung erstellt die Tabellenstruktur (ohne Index, Contraints etc.), die zweite Anweisung kopiert die Daten.

HINWEIS Sie können beide Anweisungen mit einem *SQLiteCommand*-Aufruf ausführen, vergessen Sie jedoch das Semikolon nicht!

Backup/Restore implementieren

Prinzipiell genügt es, wenn Sie im unbenutzten Zustand die komplette Datenbank-Datei einfach kopieren, es sind alle Informationen enthalten. Doch wie wollen Sie zum Beispiel einen InMemory-Datenbank kopieren? Hier hilft Ihnen die Backup-API von SQLite weiter.

BEISPIEL

Backup und Restore für eine InMemory-Datenbank implementieren

```
...
using System.Data.SQLite;
...
```

Reguläres Erstellen der Speicherdatenbank:

```
SQLiteConnection conn;
DataSet ds;
SQLiteDataAdapter da;

private void button8_Click(object sender, EventArgs e)
{
    conn = new SQLiteConnection("Data Source=:memory:");
    conn.Open();
    SQLiteCommand cmd = conn.CreateCommand();
    cmd.CommandText = "CREATE TABLE IF NOT EXISTS kunden (" +
                      "     Id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT," +
                      "     Vorname VARCHAR(50) NOT NULL," +
                      "     Nachname VARCHAR(50) NOT NULL," +
                      "     Telefon VARCHAR(50)" +
                      " );";
    cmd.ExecuteNonQuery();
    da = new SQLiteDataAdapter("SELECT * FROM kunden", conn);
    da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
    ds = new DataSet();
    da.Fill(ds, "Kunden");
    dataGridView1.DataSource = ds;
    dataGridView1.DataMember = "Kunden";
}
```

Sichern der Speicherdatenbank:

```
private void button2_Click(object sender, EventArgs e)
{
    SQLiteConnection ziel = new SQLiteConnection("Data Source=" +
                                                  AppDomain.CurrentDomain.BaseDirectory +
                                                 "Backup.db");
    ziel.Open();
    conn.BackupDatabase(ziel, "main", "main", -1, null, 0);
    ziel.Close();
}
```

Nach diesem Aufruf finden Sie im Anwendungsverzeichnis die Datei *Backup.db*. Diese enthält die komplette Struktur und alle Daten der Quell-Datenbank, in diesem Fall die Speicherdatenbank.

Wiederherstellen der Speicherdatenbank:

```
        private void button3_Click(object sender, EventArgs e)
        {
```

Im Zweifel wird eine bestehende Speicherdatenbank gelöscht und neu erzeugt:

```
            if (conn != null) conn.Clone();
            conn = new SQLiteConnection("Data Source=:memory:");
            conn.Open();
```

Daten von der Platte laden:

```
            SQLiteConnection quelle = new SQLiteConnection("Data Source=" +
                                                           AppDomain.CurrentDomain.BaseDirectory +
                                                           "Backup.db");
            quelle.Open();
            quelle.BackupDatabase(conn, "main", "main", -1, null, 0);
            quelle.Close();
        }
```

Haben Sie größere Datenmengen zu bewältigen, können Sie auch eine Callback-Routine ansteuern, die den Fortschritt visualisiert:

BEISPIEL

Verwendung eines Callbacks

```
...
        private bool SQLiteBackupCallback(SQLiteConnection source, string sourceName,
                                          SQLiteConnection destination, string destinationName,
                                          int pages, int remainingPages, int totalPages, bool retry)
        {
            listBox1.Items.Add("Pages " + pages.ToString() + "  remainingPages " +
                               remainingPages.ToString() + "  totalPages " + totalPages.ToString());
            return true;
        }
```

Beim Backup aktualisieren wir die Anzeige bei jeder geschriebenen Seite:

```
...
    conn.BackupDatabase(ziel, "main", "main", 1, this.SQLiteBackupCallback, 0);
...
```

Beim Restore aktualisieren wir nur bei jeder fünften Seite:

```
...
    quelle.BackupDatabase(conn, "main", "main", 5, this.SQLiteBackupCallback, 0);
...
```

Abbildung 10.21 Die Anzeige in der *ListBox*

Tabellen zwischen Datenbanken kopieren

Wie im Beispiel »Mehrere Datenbanken verknüpfen« (Seite 707) gezeigt, lassen sich weitere Datenbanken per ATTACH in eine bestehende Connection einbinden. Über den Aliasnamen können Sie die Tabellen der beiden Datenbanken auseinanderhalten. Verbinden Sie dieses Möglichkeit mit der im Beispiel »Tabellen zwischen Datenbanken kopieren« (Seite 711) gezeigten Vorgehensweise, steht dem Kopieren einzelner Tabellen bzw. Abfragen in eine weitere Datenbank nichts mehr im Weg.

BEISPIEL

Tabelle zwischen Datenbanken kopieren

Ausgehend von einer geöffneten Verbindung zur *Ziel*-Datenbank binden wir die Datenbank *Northwind-ef.db* ein:

```
ATTACH DATABASE 'northwindef.db' AS NW;
```

Falls noch nicht vorhanden, erstellen wir die Zieltabelle:

```
CREATE TABLE IF NOT EXISTS Ziel.Abfragedaten AS SELECT * FROM NW.Products;
```

Abschließend kopieren wir die Daten:

```
INSERT INTO Ziel.Abfragedaten SELECT * FROM NW.Products;
```

Ersatz für TOP

Als T-SQL-Programmierer ist Ihnen sicher die TOP-Klausel ein Begriff, damit haben Sie die Möglichkeit, die Anzahl der Datensätze, die als Abfrageergebnis zurückgegeben werden, zu beschränken.

SQLite kennt diese Klausel nicht, Sie müssen stattdessen die LIMIT-Klausel verwenden.

BEISPIEL

Aus

```
SELECT TOP 10 * FROM Products
```

wird

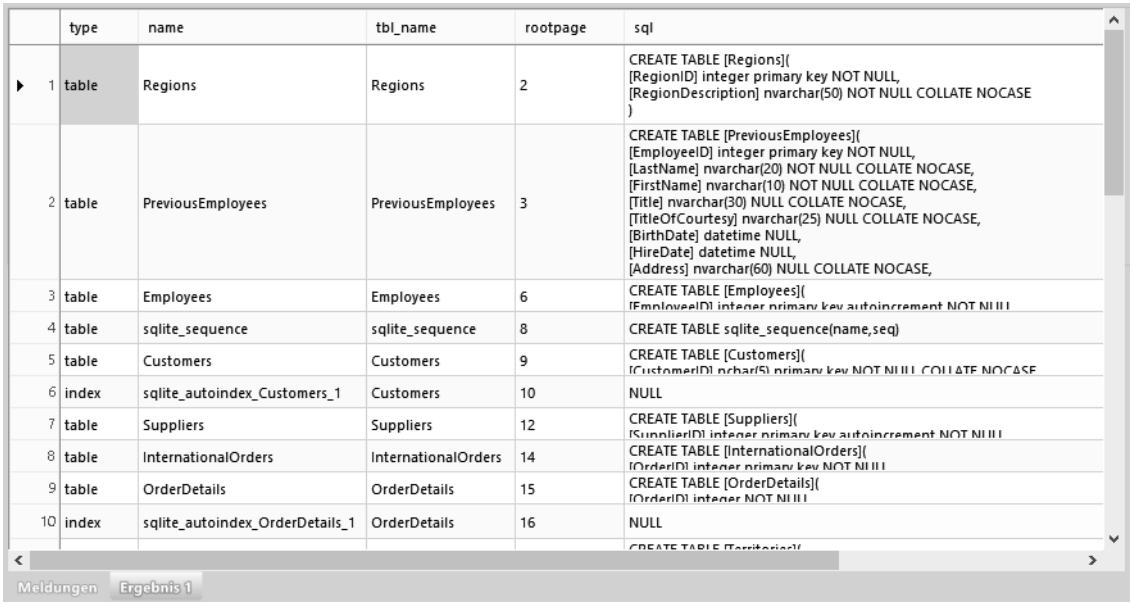
```
SELECT * FROM Products LIMIT 10
```

Metadaten auswerten

Wer mehr über eine SQLite-Datenbank erfahren möchte, kann einen Blick in die System-Tabelle *sqlite_master* werfen:

```
SELECT * FROM sqlite_master
```

Die Anweisung zaubert folgende Daten auf den Bildschirm[1]:



	type	name	tbl_name	rootpage	sql
1	table	Regions	Regions	2	CREATE TABLE [Regions]([RegionID] integer primary key NOT NULL, [RegionDescription] nvarchar(50) NOT NULL COLLATE NOCASE)
2	table	PreviousEmployees	PreviousEmployees	3	CREATE TABLE [PreviousEmployees]([EmployeeID] integer primary key NOT NULL, [LastName] nvarchar(20) NOT NULL COLLATE NOCASE, [FirstName] nvarchar(10) NOT NULL COLLATE NOCASE, [Title] nvarchar(30) NULL COLLATE NOCASE, [TitleOfCourtesy] nvarchar(25) NULL COLLATE NOCASE, [BirthDate] datetime NULL, [HireDate] datetime NULL, [Address] nvarchar(60) NULL COLLATE NOCASE,
3	table	Employees	Employees	6	CREATE TABLE [Employees]([EmployeeID] integer primary key autoincrement NOT NULL
4	table	sqlite_sequence	sqlite_sequence	8	CREATE TABLE sqlite_sequence(name,seq)
5	table	Customers	Customers	9	CREATE TABLE [Customers]([CustomerID] nchar(5) primary key NOT NULL COLLATE NOCASE
6	index	sqlite_autoindex_Customers_1	Customers	10	NULL
7	table	Suppliers	Suppliers	12	CREATE TABLE [Suppliers]([SupplierID] integer primary key autoincrement NOT NULL
8	table	InternationalOrders	InternationalOrders	14	CREATE TABLE [InternationalOrders]([OrderID] integer primary key NOT NULL
9	table	OrderDetails	OrderDetails	15	CREATE TABLE [OrderDetails]([OrderID] integer NOT NULL
10	index	sqlite_autoindex_OrderDetails_1	OrderDetails	16	NULL

Abbildung 10.22 Ansicht der Daten

Wie Sie sehen, ist für jedes Objekt die komplette DDL-Anweisung gespeichert, Sie können diese Informationen entsprechend parsen.

Eine etwas übersichtlichere Variante für die Darstellung der Tabelleneigenschaften bietet sich an mit der Anweisung

```
pragma table_info([<Tabellenname>])
```

Die zurückgegebene Tabelle entspricht im Wesentlichen der Darstellung in einem Tabellen-Layout-Editor:

	cid	name	type	notnull	dflt_value	pk
1	0	ProductID	integer	1	NULL	1
2	1	ProductName	nvarchar(40)	1	NULL	0
3	2	SupplierID	integer	0	NULL	0
4	3	CategoryID	integer	0	NULL	0
5	4	QuantityPerUnit	nvarchar(20)	0	NULL	0
6	5	UnitPrice	money	0	0	0
7	6	UnitsInStock	smallint	0	0	0
8	7	UnitsOnOrder	smallint	0	0	0
9	8	ReorderLevel	smallint	0	0	0
10	9	Discontinued	bit	1	0	0
11	10	DiscontinuedDate	datetime	0	NULL	0

Abbildung 10.23 Rückgabewert für die Tabelle *Products*

[1] bzw. in einen *DataReader* (hier verwenden wir *Database .NET* zur Abfrage)

Möchten Sie die letzten Werte der Autoinkrement-Felder für einzelne Tabellen abrufen, nutzen Sie die Tabelle *sqlite_sequenz*:

	name	seq
▶ 1	Employees	9
2	Orders	11077
3	Products	77
4	Suppliers	29
5	Categories	15
6	test	3
✳ 7	NULL	NULL

Abbildung 10.24 Abfrage der Zählerfelder

Mit diesen drei Anweisungen können Sie schon eine ganze Menge über unbekannte Datenbanken herausfinden, für weitergehende Informationen verweisen wir Sie wieder an die SQLite-Dokumentation im Internet.

Timestamp als Defaultwert verwenden

Da die Syntax etwas gewöhnungsbedürftig ist, wollen wir noch auf einen Spezialfall eingehen. Möchten Sie das aktuelle Datum als Default-Wert für eine Tabellenspalte verwenden, definieren Sie die Spalte bitte wie folgt:

```
CREATE TABLE Test (
   Id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
   Eingabedatum DATETIME DEFAULT (datetime('now')),
...
);
```

Übergeben Sie keinen Wert an die Spalte, trägt die SQLite-Engine automatisch das aktuelle Datum und die Zeit in die Spalte ein.

Export in XML-Format

Direkte Unterstützung für einen XML-Export bietet SQLite nicht, mit Hilfe einer *DataTable* haben Sie aber auch dieses Problem schnell gemeistert:

BEISPIEL

Exportieren von Abfragedaten im XML-Format

```
        private void button5_Click(object sender, EventArgs e)
        {
            da = new SQLiteDataAdapter("SELECT * FROM kunden", conn);
            da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
            ds = new DataSet();
            da.Fill(ds, "Kunden");
```

DataTable abrufen und Daten exportieren:

```
            DataTable dt = ds.Tables["Kunden"];
            dt.WriteXml(AppDomain.CurrentDomain.BaseDirectory + "Kunden.xml");
        }
```

Das Ergebnis ist folgende ineffiziente Struktur, die recht viel Speicher belegt:

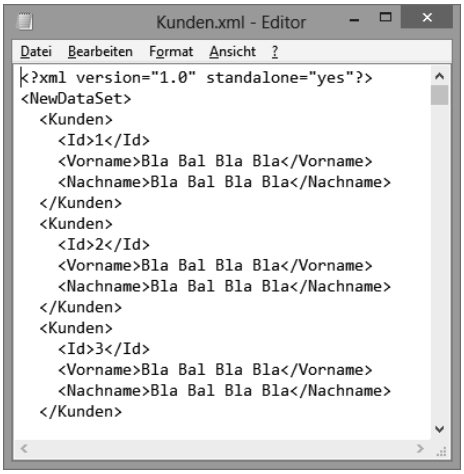


Abbildung 10.25 Die erzeugten XML-Daten

Besser und vor allem platzsparender ist die Verwendung von Attributen für das Mapping der einzelnen Tabellenspalten.

```
...
            DataTable dt = ds.Tables["Kunden"];
            foreach (DataColumn dc in dt.Columns)
                dc.ColumnMapping = MappingType.Attribute;
            dt.WriteXml(AppDomain.CurrentDomain.BaseDirectory + "Kunden.xml");
...
```

Das sieht doch schon besser aus:

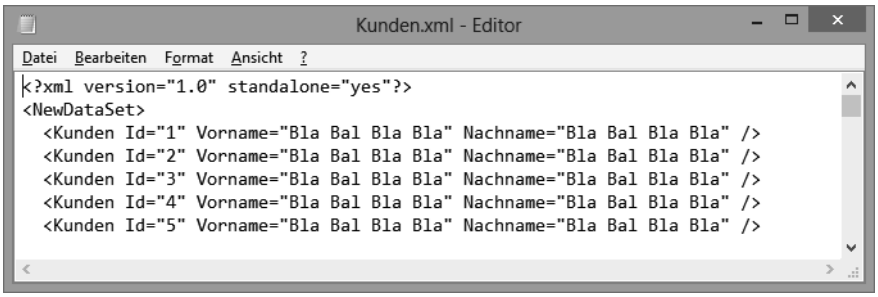


Abbildung 10.26 XML-Daten mit Attributen

Sicher könnten wir hier noch reichlich Informationen zu SQLite-Datenbanken und deren Verwendung hinzufügen, aber leider fehlt uns dafür wie immer der Platz[1]. Was bleibt ist ein finales Fazit.

Fazit

Sie haben es sicher bemerkt, SQLite ist – sowohl was die Distribution als auch die Features anbelangt – die ideale **Desktop-Datenbank**. Gerade solche Funktionen, wie Volltextsuche und InMemory-Datenbank, sucht man bei Microsoft in dieser Rubrik vergeblich. Die gegenüber Access-Datenbanken nicht vorhandene Einschränkung bei der maximalen Dateigröße ist ein weiteres gewichtiges Argument.

Auch wer an der Verbindung zu mobilen Lösungen (iOS, Android) arbeitet, wird mit diesem Datenformat seine Freude haben. Sie können eine Datenbank problemlos zwischen den Plattformen austauschen, Sie müssen nicht mal die Indizes neu aufbauen.

Die Kompatibilität zu den Microsoft-Technologien ist dank ADO.NET-Datenprovider hervorragend gelöst, was aber auch gleich zu einer Einschränkung führt: Im Zusammenhang mit dem in diesem Kapitel vorgestellten Provider *System.Data.SQLite* wird mancher die Unterstützung für eine Cursor-Programmierung vermissen. Gerade im Desktop-Bereich bzw. bei Verwendung einer InMenory-Datenbank ist es recht sinnfrei, mit einem *DataSet* zu arbeiten, denn so halten Sie die Daten zweimal im Speicher. Ein Pendant zum bekannten *SqlCeResultSet* werden Sie also nicht finden.

Natürlich hat die Beschränkung auf *DataReader*, *DataSet* und *DataCommand* auch ihre Vorteile: Sie kommen erst gar nicht in die Versuchung, wieder mit Cursorn zu programmieren. Damit ist Ihr Programm später wesentlich einfacher in Richtung SQL Server zu migrieren. Allerdings haben die Microsoft Produkte hier die Nase vorn, stimmen doch die Datentypen bei SQL Server und SQL Server Express überein.

Einen Einsatz im Netzwerk würden wir aus prinzipiellen Gründen (Sicherheit, Locking etc.) nicht empfehlen. Da sind Sie mit einem echten SQL Server viel besser beraten.

[1] Es mussten ohnehin schon viel zu viele Kapitel in das E-Book ausgelagert werden.