

# Datasets: lokal bearbeitet, zentral gespeichert

Das Dataset ist die wichtigste Komponente in ADO.NET für die Arbeit mit Datenbanken. Im Gegensatz zum alten ADO-Recordset-Objekt hat es keinerlei Verbindung zur Datenbank. Die Client-Instanzen führen Änderungen an ihren lokalen Datasets unabhängig voneinander durch. Es kann vorkommen, dass zwei Anwender den gleichen Datensatz ändern. Werden die Änderungen zur Datenbank übertragen, treten Konflikte auf.

\_\_\_ Mit einer Datenbankanwendung arbeiten mehrere Anwender gleichzeitig. Wie soll mit der Situation umgegangen werden, dass die Anwender den gleichen Datensatz ändern? Im Englischen wird das als *Concurrency Control* bezeichnet. Es gibt dafür drei Möglichkeiten:

- optimistisches Sperren,
- pessimistisches Sperren und
- die Strategie „Der Letzte gewinnt“.

Das pessimistische Sperren ist vom Standpunkt des Programmierers die einfachste Variante. Wird ein Datensatz von einem Client gelesen, bleibt er so lange gesperrt, bis die Änderungen zurückgeschrieben werden oder bis der Datensatz wieder freigegeben wird. In der Zwischenzeit können andere Anwender diesen Datensatz nicht bearbeiten. Je mehr Anwender mit dem System arbeiten, desto mehr werden sie sich gegenseitig blockieren. Für die Benutzer ist so ein System unbefriedigend.

Beim optimistischen Sperren wird der Datensatz gelesen, aber nicht gesperrt, sodass andere Anwender ihn ebenfalls lesen und bearbeiten können. Nur während des eigentlichen Updates auf der Datenbank wird der Datensatz für kurze Zeit gesperrt, bis das Update abgeschlossen ist. Andere Anwender, die ebenfalls diesen Datensatz in ihr *DataSet* geladen haben, bekommen davon nichts mit. Versucht ein weiterer Anwender einen Datensatz zu aktualisieren, der in der Zwischenzeit geändert wurde, so sollte ein Feh-

ler ausgelöst werden. In dieser Situation muss entschieden werden, wie damit umgegangen werden soll. Sollen die Änderungen des ersten Anwenders überschrieben werden? Das wäre der Der-Letzte-gewinnt-Ansatz, der sicherlich nur in wenigen Situationen ausreichend ist. Soll der zweite Anwender seine Änderungen noch einmal überarbeiten und dabei die Änderungen des ersten berücksichtigen? Dieses wird sicherlich die häufiger eingesetzte Variante sein. Der Vorteil liegt darin, dass das System besser skaliert, da nur für kurze Zeit Datensatzsperrungen aktiv sind. Zudem wird sichergestellt, dass ein Anwender nicht die Änderungen eines anderen überschreibt. Der Nachteil: Der Entwickler hat mehr zu programmieren und die einzelne Update-Operation dauert etwas länger.

Das .NET Framework stellt das *DataSet* und den *DataAdapter* zur Verfügung. Der *DataAdapter* dient dabei als Brücke zwischen *DataSet* und Datenbank. Die *Fill*-Methode des *DataAdapters* füllt ein *DataSet* mit Daten aus der Datenbank, die *Update*-Methode schreibt alle Änderungen zurück in die Datenbank [1].

Für die Behandlung von Konflikten stellen die beiden Klassen *DataSet* und *DataAdapter* Hilfsmittel zur Verfügung. Im Folgenden wird ein Beispielprojekt beschrieben, das diese Hilfsmittel im Einsatz zeigt. Das Projekt liegt auf der Heft-CD vor.

Das Projekt besteht aus zwei Formularen: *frmDS.cs* und *frmConcuError.cs*. Das erste Formular hat die Aufgabe, die Daten anzuzeigen und Änderungen zu ermöglichen, das zweite zeigt aufgetretene Konflikte beim Aktualisieren der Daten an.

## Der DataAdapter

Im Formular *frmDS.cs* befinden sich neben einer Reihe von Steuerelementen auch sechs ADO.NET-Komponenten: eine *Connection*, ein *DataAdapter* und vier *Command*-Objekte, die die Kommunikation mit der Datenbank übernehmen. Diese Objekte wurden mit einem Assistenten von Visual Studio .NET erzeugt.

Wenn Sie das mit einem eigenen Formular nachvollziehen wollen, gehen Sie wie folgt vor: Fügen Sie dem Projekt ein neues Formular hinzu und ziehen Sie mit Drag-and-Drop eine *SqlConnection* von der Toolbox auf Ihr Formular. Konfigurieren Sie die Verbindung mithilfe des Eigenschaften-Fensters, sodass sie auf die Datenbank *Kunde* Ihres Servers verweist. Anschließend fügen Sie Ihrem Formular einen *SqlDataAdapter* hinzu. Jetzt startet automatisch ein Assistent von Visual Studio .NET, der Ihnen bei der Konfiguration des *DataAdapters* hilft. Zunächst wählen Sie

## SUMMARY

### Auf einen Blick

Was passiert, wenn zwei Anwender den gleichen Datensatz in der Datenbank ändern wollen? Wie werden Konflikte bei den nichtverbundenen DataSets entdeckt und gelöst? Anhand eines Beispielprojekts wird dieses Problem untersucht.

### Eingesetzte Anwendungen

Visual Studio .NET, SQL Server oder anderes DBMS

### CD-Code

Basics06

### Autor

Marcel Gnoth ist Senior Consultant bei der Berliner NTeam GmbH. Seine Arbeitsschwerpunkte liegen in den Bereichen COM, .NET und verteilte Informationssysteme, zu denen er auch Trainings durchführt. Sie erreichen ihn unter [marcel@gnoth.net](mailto:marcel@gnoth.net) oder über [www.gnoth.net](http://www.gnoth.net).



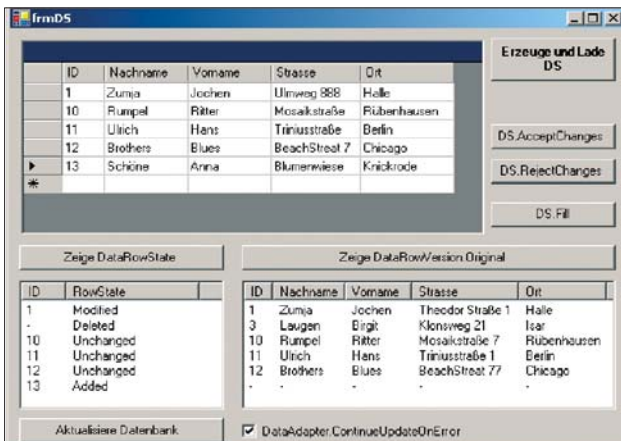


Abbildung 1 | Der Client speichert Änderungen zunächst nur lokal.

im Assistenten die eben konfigurierte *Connection* aus. Der *DataAdapter* benutzt vier SQL-Kommandos, um mit der Datenbank zu kommunizieren. Sie haben die Wahl, ob die Kommunikation über Stored Procedures geschehen soll oder über SQL-Anweisungen, die in Ihrer Anwendung gespeichert werden. Wählen Sie für dieses Beispiel: *Use SQL Statements*. Die SQL-Befehle sind dann im Quelltext zu finden.

Im dritten Schritt bestimmen Sie, welche Daten der *DataAdapter* laden soll. Mit dem Query Builder erhalten Sie folgenden Befehl:

```
SELECT ID, Nachname, Vorname, Strasse, Ort
FROM Kunde
```

Schauen Sie sich die Optionen unter *Advanced Options* an. Diese sollten alle drei angehakt sein. Was bedeuten sie?

**Listing 1 Aktualisieren der Datenbank und Prüfen auf Konflikte.**

```
private void cmdUpdateDB_Click(object sender, System.EventArgs e) {
    if(chkContinueUpdateOnError.Checked==true)
        //Fehler werden erst einmal verschluckt und später ausgewertet
        daKunde.ContinueUpdateOnError=true;
    else
        daKunde.ContinueUpdateOnError=false;

    daKunde.Update(tblKunde);

    if(chkContinueUpdateOnError.Checked==true && tblKunde.HasErrors){
        //Und jetzt geht's an die Fehler ...
        foreach(DataRow dr in tblKunde.GetErrors()){
            string s;
            if (dr.RowState==DataRowState.Deleted){
                s = "ID = " + dr["ID",DataRowVersion.Original]+ "\n";
                s = s + " Error = " + dr.RowError ;
            } else {
                s = "ID = " + dr["ID"]+ "\n";
                s = s + " Error = " + dr.RowError ;
            }
            MessageBox.Show("Fehler beim Aktualisieren der Datenbank in der
            Zeile: " + s);
            frmConcuError f = new
            frmConcuError(dr,cnKunde.ConnectionString);
            f.ShowDialog(this);
        }
    }
}
```

Ein *DataAdapter* füllt ein *DataSet* mit Daten. Dafür wird das SQL-Statement benutzt, welches mit dem Query Builder erzeugt wird. Soll der Adapter aber auch Änderungen in die Datenbank zurückschreiben, benötigt er drei weitere SQL-Anweisungen. Mit der Option *Generate Insert, Update and Delete Statements* werden diese drei Kommandos basierend auf der *Select*-Anweisung generiert. Diese Kommandos könnten auch später zur Laufzeit zum *DataAdapter* hinzugefügt werden.

Die zweite Option modifiziert das *Update*- und das *Delete*-Kommando, sodass Konflikte mit Änderungen anderer Benutzer entdeckt werden. Dafür gibt es zwei Möglichkeiten. Das *DataSet* speichert jeden Wert in zwei Versionen: einmal den Originalwert, den es aus der Datenbank erhalten hat, und zum zweiten die aktuellen Werte, die die Änderungen des Anwenders enthalten. Die Option ergänzt bei dem *Update*- und dem *Delete*-Kommando die *Where*-Klausel. Jetzt wird nicht mehr nur der Primärschlüssel in der *Where*-Klausel angegeben, sondern es werden auch alle Felder auf ihre Originalwerte geprüft.

```
UPDATE Kunde
SET Nachname = @Nachname,
    Vorname = @Vorname,
    Strasse = @Strasse,
    Ort = @Ort
WHERE
    (ID = @Original_ID) AND
    (Nachname = @Original_Nachname) AND
    (Ort = @Original_Ort) AND
    (Strasse = @Original_Strasse) AND
    (Vorname = @Original_Vorname);
SELECT ID, Nachname, Vorname, Strasse, Ort
FROM Kunde WHERE (ID = @ID)
```

Hat ein anderer Anwender in der Zwischenzeit ein Feld dieses Datensatzes geändert, kann diese *Where*-Klausel den Datensatz nicht mehr finden, und die Änderung wird auf keinem Datensatz ausgeführt. Der *DataAdapter* prüft, ob die Anweisung einen Datensatz geändert hat (*RecordsAffected*). Ist das der Fall, wird eine *System.Data.DBConcurrencyException* ausgelöst.

Diese Variante wird vom Assistenten umgesetzt. Eine Alternative bestünde darin, jede Tabelle der Datenbank mit einer Zeitstempelspalte (*TimeStamp*) auszustatten, in der immer das Datum der letzten Aktualisierung vermerkt ist. Werden die Änderungen in die Datenbank zurückgeschrieben, kann geprüft werden, ob in der Zwischenzeit jemand den Datensatz geändert hat, indem der originale Zeitstempel mit dem aktuellen in der Datenbank verglichen wird. Diese Variante macht die *Where*-Klausel effizienter, da nur noch zwei Spalten (Primärschlüssel und Zeitstempel) geprüft werden müssen.

Die dritte Option aktualisiert das *DataSet* nach einem *Update* oder *Insert*. Einige Datenbanken generieren einen Autowert für einen Primärschlüssel oder haben Standard- oder berechnete Werte für Spalten. Damit diese Änderungen am Datensatz, die auf der Seite der Datenbank stattfinden, auch im *DataSet* sichtbar werden, wird nach dem *Update* oder *Insert* ein *Select* ausgeführt. Damit sind die *Connection* und der *DataAdapter* konfiguriert.

**Das DataSet füllen**

Nachdem nun alles konfiguriert ist, kann das *DataSet* gefüllt werden. In der Methode *cmdCreateLoadDS\_Click* des Beispielpro-

jekts befindet sich der Code dafür. Wichtig ist, dass der *DataAdapter* erkennt, welche Spalte der Primärschlüssel ist, sonst werden beim erneuten Aufrufen der *Fill*-Methode die Datensätze an das Ende der Tabelle angefügt. Über die *FillSchema*-Methode lädt der *DataAdapter* das Schema einer Tabelle. Alternativ dazu kann die *MissingSchemaAction*-Eigenschaft gesetzt werden. Im Code-Beispiel erhält sie den Wert *AddWithKey*. Dadurch ermittelt der *DataAdapter* den Primärschlüssel der Tabelle und erzeugt einen für die *DataTable* im *DataSet*. Beim erneuten Aufruf der *Fill*-Methode gleicht der *DataAdapter* die Schlüssel ab und aktualisiert die bestehenden Zeilen. Allerdings hat er in den Tests nicht die Zeilen aus der *DataTable* entfernt, die in der Zwischenzeit von anderen Anwendern in der Datenbank gelöscht wurden.

Danach wird ein *DataGrid* an die *DataTable* gebunden. Es ermöglicht das einfache Bearbeiten der Tabelle. Sie können jetzt Datensätze im *DataGrid* ändern, löschen und hinzufügen.

```

dsKunde= new DataSet("Kunden");
daKunde.MissingSchemaAction= MissingSchemaAction.AddWithKey;
daKunde.Fill(dsKunde);
tblKunde = dsKunde.Tables["Kunde"];
dgContentLocal.DataSource = dsKunde;
dgContentLocal.DataMember = "Kunde";

```

### Die RowState-Eigenschaft

Abbildung 1 stellt *frmDS* dar. Im oberen Bereich befindet sich das *DataGrid*, in dem die Daten verändert werden können. Das *List-View* links unten zeigt die *RowState*-Eigenschaft jeder Zeile an. Für jede Zeile werden zwei Versionen gespeichert: die originale Version, die von der Datenbank geliefert wurde, und die Version, die den aktuellen lokalen Bearbeitungsstand enthält. Im *DataSet* wurde die Zeile mit der ID=1 geändert, die mit ID=2 gelöscht und die mit ID=13 neu hinzugefügt. Wird die *Update*-Methode des *DataAdapters* aufgerufen, dann kontrolliert dieser die *RowState*-Eigenschaft jeder Zeile und versucht, die geänderten Zeilen in die Datenbank zu schreiben.

Im *List-View* rechts unten sind die Originalwerte zu sehen. Dort ist auch der gelöschte Datensatz 3 noch zu finden. Auf die Originalwerte einer *DataRow* wird folgendermaßen zugegriffen:

```

dRow["ID",DataRowVersion.Original].ToString()

```

Auf der rechten Seite befinden sich drei Befehlsschaltflächen. *RejectChanges* überschreibt die aktuelle Version einer Zeile mit ihrer Originalversion und *AcceptChanges* überschreibt umgekehrt die Originalversion mit der aktuellen. Achtung: Diese beiden Methoden arbeiten nur lokal auf dem *DataSet* und haben keine Verbindung mit der Datenbank.

Die dritte Schaltfläche füllt das *DataSet* neu mit frischen Daten aus der Datenbank. Damit auch die gelöschten Datensätze aus dem *DataSet* entfernt werden, wird vor der *Fill*-Methode die *Clear*-Methode des *DataSets* aufgerufen.

### Die Datenbank wird aktualisiert

Unten auf dem Formular befindet sich eine Schaltfläche zum Aktualisieren der Datenbank. Über das Kontrollkästchen neben der Schaltfläche kann die Eigenschaft *ContinueUpdateOnError* des *DataAdapters* eingestellt werden. Hat sie den Wert *true*, werden Fehlermeldungen unterdrückt. Der Entwickler kann so nach

Abschluss der *Update*-Methode prüfen, ob Fehler aufgetreten sind, und entsprechend reagieren. Ist die Eigenschaft nicht auf *true* gesetzt, muss im *RowUpdated*-Ereignis geprüft werden, ob ein Fehler aufgetreten ist. Andernfalls wird ein Laufzeitfehler ausgelöst.

```

private void daKunde_RowUpdated(object sender, SqlRowUpdatedEventArgs e) {
    if(e.RecordsAffected==0 && !daKunde.ContinueUpdateOnError){
        //Problem Concurrency violation
        MessageBox.Show("daKunde_RowUpdated: Jemand hat den Datensatz
geändert!");
        e.Status=UpdateStatus.SkipCurrentRow;
    }
}

```

Werden die Fehler nicht im *RowUpdated*-Ereignis behandelt, muss das nach dem Aufruf der *Update*-Methode erfolgen (siehe Listing 1).

### Behandlung der Konflikte

Die Behandlung hängt ab von der Logik der Anwendung und dem Aufwand, der betrieben werden soll. Das Beste ist sicherlich, den Anwender zu informieren, dass der Datensatz in der Zwischenzeit gelöscht oder verändert wurde. In besonders sensiblen Bereichen könnte ein Konfliktlöser informiert werden, der als Schiedsrichter fungiert. In den meisten Situationen dürfte es aber ausreichen, dem Anwender die Entscheidung zu überlassen. Wurde der Datensatz geändert und nicht gelöscht, kann der Anwender anhand der verschiedenen Versionen entscheiden, welche Werte gültig sind, und die Datenbank dann mit seiner überarbeiteten Version aktualisieren. Enthält die Tabelle eine Spalte mit Namen und Telefonnummer desjenigen, der die letzte Änderung durchgeführt hat, kann der Anwender ihn anrufen und sich mit ihm abstimmen, bevor er seine Version der Werte in die Datenbank schreibt. In Abbildung 2 wird das Konfliktformular dargestellt. Es werden alle drei Versionen des Datensatzes angezeigt. Die zurzeit in der Datenbank aktuelle Version muss separat abgefragt werden. Der Anwender kann jetzt entscheiden, welche die richtige ist. Optimal wäre die Möglichkeit, basierend auf den drei Versionen eine ganz neue Version zusammenstellen zu können. Diese wird dann in die Datenbank geschrieben.

### Gelöschte Datensätze

Wurde der Datensatz in der Zwischenzeit gelöscht, muss überlegt werden, wie damit umgegangen werden soll. In den meisten Geschäftsanwendungen werden selten Datensätze gelöscht. Wenn also ein Datensatz, der in Bearbeitung ist, gelöscht wird, sollten die Arbeitsprozesse der Abteilung genauer betrachtet werden. So etwas sollte in der Praxis nicht vorkommen, aber es wird passieren. Nun könnte der gelöschte Datensatz wieder in die Datenbank eingefügt werden. Schwierig wird es, wenn Primärschlüssel automatisch (Autowert) generiert werden, was in den meisten Datenbanken der Fall ist. Fügen Sie den Datensatz erneut ein, erhält er einen neuen Primärschlüssel. Für abhängige Tabellen sollte das nicht so problematisch sein, da ein Datensatz erst gelöscht werden kann, wenn alle seine Kinder gelöscht wurden. Angenommen, dritte Anwender haben auch eine Version des Datensatzes in einem lokalen *DataSet*: Jetzt wird der Datensatz unter einem

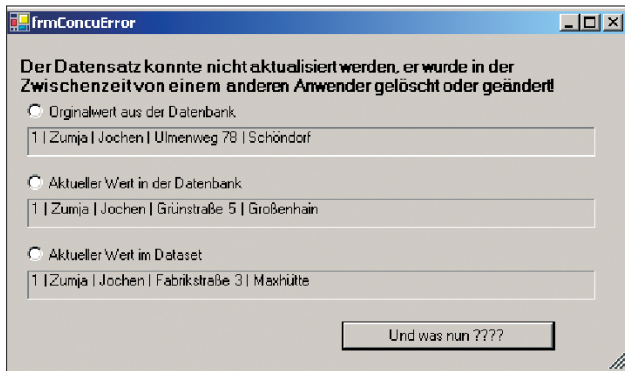


Abbildung 2 | Mehrere Anwender haben den gleichen Datensatz bearbeitet.

neuen Primärschlüssel eingefügt. Rufen weitere Anwender die *Update*-Methode auf und finden den Datensatz unter dem alten Primärschlüssel nicht, würden sie den Datensatz ebenfalls einfügen, jedesmal unter einem neuen Schlüssel. Zugegebenermaßen ist eine solche Situation sehr unwahrscheinlich. Tritt sie aber ein, gäbe es doppelte Datensätze, der Horror eines jeden Informationssystems. Abhilfe würden GUIDs als Primärschlüssel schaffen. Der Datensatz könnte unter seiner alten GUID wieder eingefügt werden. Vielleicht ermöglicht das eingesetzte Datenbanksystem auch das Einfügen von vorgegebenen Werten (dem originalen Schlüssel) in einer Autowert-Spalte. Einfacher ist es, dem Anwender mitzuteilen, dass seine Änderungen im Papierkorb verschwinden, da in der Zwischenzeit jemand anders seinen Datensatz gelöscht hat.

### Lösen der Konflikte

Wenn die Konflikte gelöst sind, kann die Datenbank aktualisiert werden. Achtung: In der Zwischenzeit könnte wieder jemand den Datensatz verändert haben. Am besten wird das *Update*- oder das *Delete*-Kommando des *DataAdapters* benutzt. Die *Update*-Methode des Adapters kann nicht einfach aufgerufen werden, da die Originalwerte der Zeile immer noch die sind, die jetzt nicht mehr in der Datenbank zu finden sind. Wenn die *Update*-Methode eingesetzt werden soll, müssen die Originalwerte der Zeile auf die momentan in der Datenbank aktuellen Werte gesetzt werden. Der Indexer (in VB die *Item*-Eigenschaft), über den auf die Originalversion einer Zeile zugegriffen werden kann, ist aber schreibgeschützt.

Ein Ausweg wäre, die in der Datenbank aktuellen Werte in die *DataRow (Current Version)* zu schreiben. Die Änderungen, die der Anwender zuvor gemacht hat, müssen vorher gesichert werden. Dann kann die *AcceptChanges*-Methode des *DataSets* aufgerufen werden. Diese Methode schreibt die *DataRowVersion.Current* in die *DataRowVersion.Original*-Version der *DataRow*. Anschließend werden die zwischengespeicherten Änderungen des Anwenders wieder in die *DataRowVersion.Current*-Version geschrieben. Wird jetzt die *Update*-Methode des *DataAdapters* aufgerufen, wird der Datensatz in der Datenbank aktualisiert, es sei denn, der Datensatz wurde in der Zwischenzeit bearbeitet. Dann geht genau das gleiche Spiel von vorn los, nur mit anderen Daten. Alternativ dazu können das *Update*- und das *Delete*-Kommando direkt aufgerufen werden. Das *DataSet* wird von diesen

### Installieren der Beispieldatenbank und des Beispielprojektes

Wenn Sie das Beispielprojekt ausführen möchten, benötigen Sie einen SQL Server. Führen Sie mit dem Query-Analyzer (iSQLw) auf der Datenbank das Skript *DatenbankSkript.sql* aus. Das Skript legt eine Datenbank *Kunde* und eine Tabelle mit dem Namen *Kunde* an. Verfügen Sie nicht über einen SQL Server, müssen Sie das Skript an Ihr Datenbanksystem anpassen. Dies sollte nicht schwer sein. Oder Sie legen die Tabelle von Hand an.

Dann müssen Sie in der Anwendungs-Konfigurationsdatei den Connection-String zu Ihrer Datenbank einstellen. Ändern Sie dazu die Datei *app.config*, die sich im Projektverzeichnis befindet. Beim Kompilieren wird die Datei *Gifhorn.exe.config* im gleichen Verzeichnis wie die EXE erzeugt.

Sie können den Code auch auf jeder anderen beliebigen relationalen Datenbank ausführen, das Prinzip ist stets das gleiche.

Zum Testen starten Sie eine Instanz der Anwendung in Visual Studio und eine zweite Instanz ohne VS.NET, indem Sie auf die kompilierte EXE-Datei doppelklicken. Jetzt ändern Sie in der zweiten Instanz einen Datensatz und aktualisieren die Datenbank. Ändern Sie jetzt in der Instanz, die Sie debuggen, den gleichen Datensatz und Konflikte werden angezeigt. Setzen Sie Haltepunkte an den wichtigen Stellen und betrachten Sie den Code in der Einzelschrittausführung.

Änderungen nicht beeinflusst und muss anschließend aktualisiert werden. Am besten wird es nach Abschluss aller Änderungen neu mit Daten gefüllt.

### Fazit

Das Framework bietet interessante Mechanismen, die den Anwendungsentwickler bei der Programmierung unverbundener Datenbank-Clients und bei der Behandlung der Konflikte, die dabei auftreten, unterstützen. Es zeigt sich, dass es keinen optimalen Weg gibt. Das *DataSet* bietet sehr gute Skalierungsmöglichkeiten. Die Datenbank liefert einige Daten und kann dann den Client vergessen und sich neuen Anfragen widmen. Erkauft wird dieser Vorteil mit einem höheren Programmieraufwand, und ein Update dürfte etwas langsamer ausgeführt werden.

Ebenso sollten Situationen vermieden werden, in denen ein Anwender stundenlang Datensätze bearbeitet und dann seine Änderungen nicht speichern kann, weil in der Zwischenzeit die Datensätze geändert wurden. In der Praxis werden Änderungen an mehreren Tabellen vorgenommen und das Lösen der Konflikte wird aufwändiger. Diese Probleme existierten schon vor .NET. Microsoft gibt dem Entwickler Hilfsmittel an die Hand, die die Lösung vereinfachen.

Wie immer spielen die Anforderungen an die Anwendung eine große Rolle: Wie wichtig ist Skalierbarkeit? Wie hoch ist das Budget? Wie wahrscheinlich ist das Auftreten von Konflikten und können diese vielleicht über Berechtigungen gelöst werden? Unter [2] können Sie die Beschreibung im MSDN nachlesen.

|||||

[1] Marcel Gnoth, Das Dataset – ein starker Typ, dotnetpro 5/2002, S. 32

[2] MSDN für Visual Studio .NET, suchen Sie nach: "Introduction to DataSet Updates"