ADO.NET

1. Was ist ADO.NET?

1.1 Die ADO.NET-Architektur

connected ⟷ disconnected

Data Provider: Klassen zum Zugriff auf / Bearbeiten von Daten aus Datenquellen
Data Set: Lokaler Puffer für Daten auf dem Client / der Mittelschicht

1.2 Der Zugriff auf verschiedene Datenbanksysteme

erfolgt über datenbankspezifische Treiber der Hersteller (Data Provider)

1.3 Wichtige Klassen des Data Providers

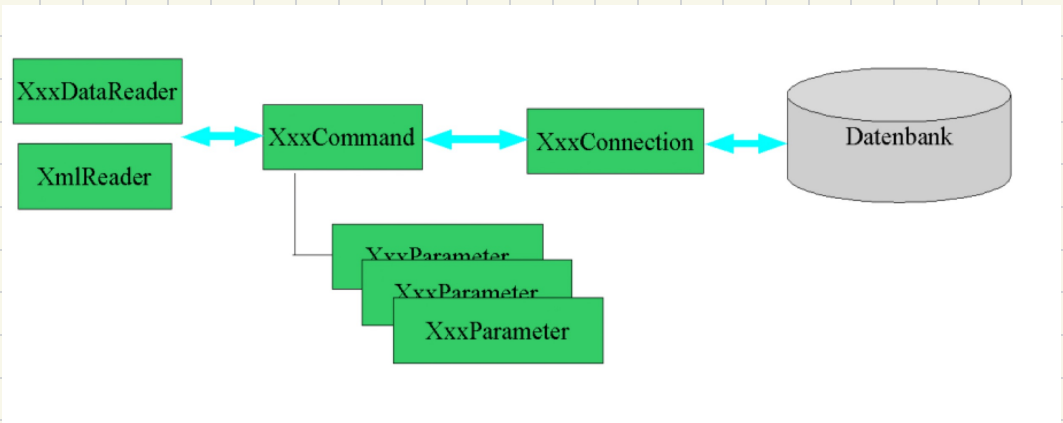
connected	disconnected
connection	dataadapter
transaction	dataset
command	
Parameter	
datareader	

2. Arbeiten im Connected-Modell

Client hält eine Ständige Verbindung zur Datenquelle

Vorteile:
- aktuellste Daten
- Datenkonsistenz

Nachteile:
- hohe Anforderungen an die Hardware (DB-Server, Netzwerk, …)



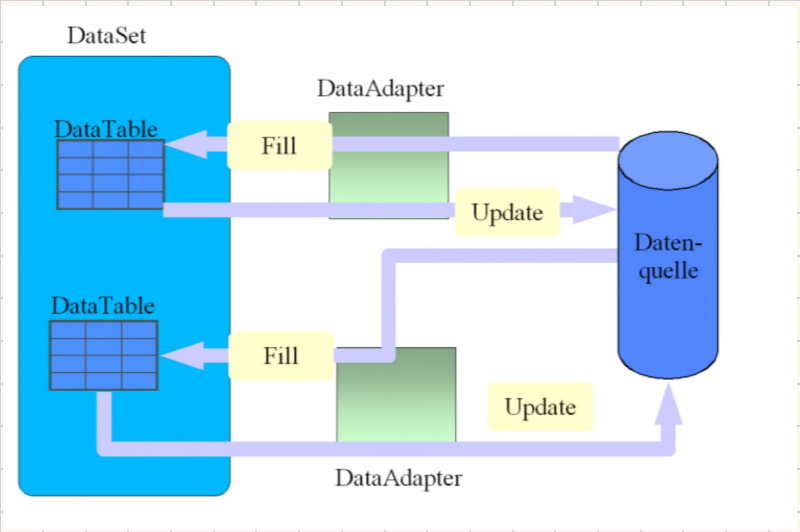
3. Arbeiten im Disconnected-Modell

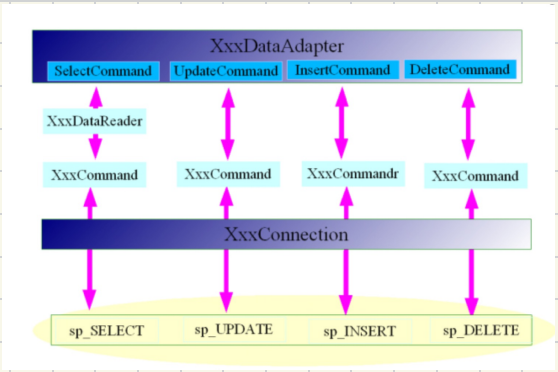
Die Clientapplikation speichert die Daten im DataSet (lokaler Puffer) und hält keine ständige Verbindung zur Datenquelle

Vorteile:
- Datenbank/Netzwerkressourcen werden geschont (tausende Clients möglich)
- Skalierbarkeit

Nachteile:
- Probleme mit der Datenkonsistenz
- Konkurrierender Zugriff

3.1 Ein Überblick über die verwendeten Daten





4. Das Entity Framework (ORM)

ORM (Object Relational Mapping) erzeugt ein Mapping zwischen Daten einer relationalen Datenbank und Objekten in einem Programm.

4.2 Das Entity Framework

Das Entity Framework ermöglicht den Datenzugriff aus dem .NET Framework auf relationale Datenbanken.

Code First <-------------> Database First

Die DbContext Klasse dient als Verbindungspunkt zwischen Anwendung und Datenbank.

Karl-Peter-Obermaier-Schule
Staatliche Berufsschule 1 Passau
Staatliche Fachschulen für Elektrotechnik und Maschinenbautechnik

UNTERRICHTSPROJEKT FUNEVENTS

TEIL II
DESIGN UND REALISIERUNG DES EVENT MANAGEMENT SYSTEMS (EMS)

MODUL 1A
DATENZUGRIFF UNTER .NET

Datenzugriff unter .NET

1 WAS IST ADO .NET? 1
1.1 Die ADO .NET - Architektur 2
1.2 Der Zugriff auf verschiedne Datenbanksysteme 2
1.3 Wichtige Klassen des Datenbank - Providers 3
2 ARBEITEN IM CONNECTED MODELL 4
2.1 Ein Überblick über die verwendeten Klassen 4
2.1.1 Das Connection - Objekt 4
2.1.2 Das Command - Objekt 5
2.1.3 Das DataReader - Objekt 5
2.2 Ein Codebeispiel zum Connected Modell 6
3 ARBEITEN IM DISCONNECTED MODELL 7
3.1 Ein Überblick über die verwendeten Klassen 7
3.1.1 Das DataAdapter - Objekt 8
3.1.2 Das DataSet - Objekt 9
3.2 Ein Codebeispiel zum Disconnected Modell 10
4 DAS ENTITY FRAMEWORK (ORM) 11
4.1 ORM Definition 11
4.2 Das Entity Framework 11
4.2.1 Code First vs. Database First Approach 12
4.2.2 Data Annotations 12
4.2.3 Die DbContext Klasse 12

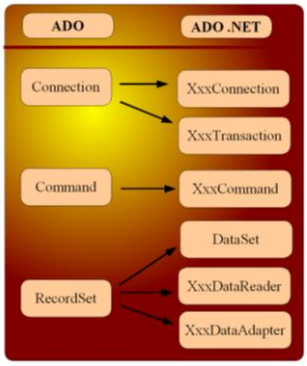
1 Was ist ADO .NET?

☞ ***Nennen Sie die Gründe für die Weiterentwicklung von ADO zu ADO .NET!***

☞ ***Welche Vorteile bietet die n- tier Architektur?***

ADO .NET ist ein Satz von Klassen, die mit Daten arbeiten. Sie stellen folgende Funktionalitäten zur Verfügung:

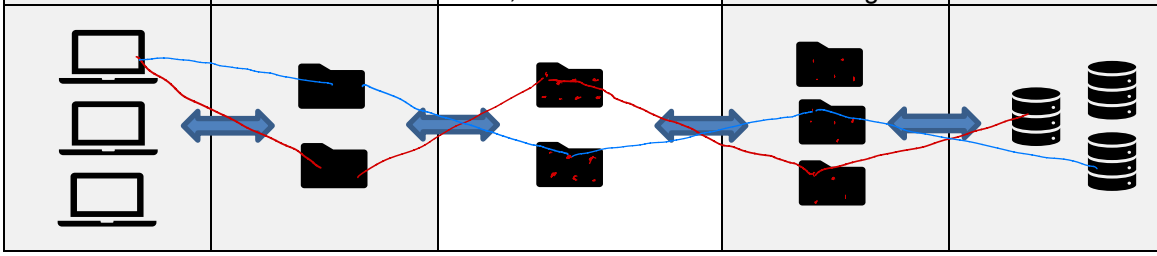
- Ein weiterentwickeltes, flexibleres Nachfolgemodell von ADO (ActiveX Data Objects).
- Ein System, das auch bei unterbrochenen Umgebungen eine logische Datenmanipulation ermöglicht (**disconnected model**).
- Ein Programmiermodell mit umfangreicher XML - Unterstützung
- Ein Satz von Klassen, Interfaces und Datenstrukturen zum Datenzugriff innerhalb des .NET – Frameworks.



Klassen mit zu großer Funktionalität wie beispielsweise RecordSet oder Connection sind unter ADO .NET in mehrere Klassen aufgegliedert.

Die größte Umstellung besteht im Bereich des RecordSet - Handlings. Während bisher die per SQL - Statement gewonnen Daten in einem RecordSet - Objekt abgelegt wurden, stehen jetzt gleich mehrere Vorgehensweisen des Datenzugriffs und der Datenmanipulation zur Verfügung. Die Motivation zur Einführung der neuen Schnittstelle ist dabei von folgenden Überlegungen geprägt.

- ADO, OLEDB und ODBC wurden für enge und dauerhafte Verbindung zwischen Fat-Client (Logik und Anzeige) und Datenbankserver (Datenhaltung) im klassischen **Zweischichtmodell** konzipiert (two tier architecture).
- XML entwickelt sich zum universellen Eingabeformat.
- Web-Applikationen benötigen eine andere Architektur als die bisher von ADO verwendete. Eine **lose Kopplung** (loosely coupled) zwischen Applikation und Daten ist bei **Drei**- oder **Mehrschichtmodellen** erwünscht (three tier/ n- tier architecture).

Präsentation		**Logik**	**Daten**	
Anzeige, z.B. Browser auf dem Client	**Anzeigelogik**, z.B. Webserver	**Geschäftslogik**, z.B. Application Server, Microservices	**Dateninterface**, z.B. Connection Pooling	**Datenhaltung**, z.B. Datenbank Farm
				

1. Zugriff
2. Zugriff

Bei der n- tier Architektur können die Komponenten der einzelnen Schichten unabhängig voneinander entwickelt werden. Durch die lose Kopplung landen hintereinander gestellte Clientabfragen nicht unbedingt auf dem gleichen Webserver, Application Server oder DB Server (Lastverteilung, Skalierbarkeit).

1.1 Die ADO .NET - Architektur

☞ ***Worin unterscheiden sich connected und disconnected Modell in ADO .NET?***

☞ ***Erklären Sie die Begriffe DataProvider und DataSet aus ADO .NET!***

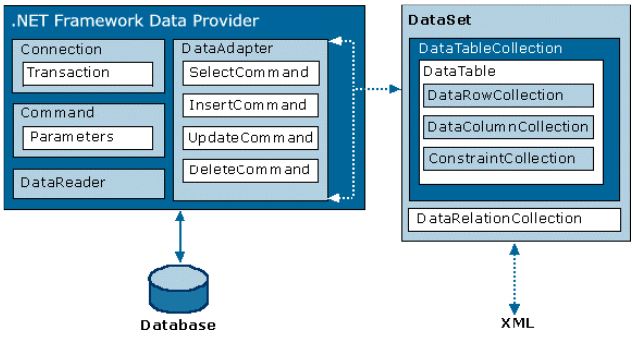
Das ADO .NET - Objektmodell sieht eine explizite Unterscheidung zwischen dauerhafter Verbindung (connected) und nicht dauerhafter Verbindung (disconnected) des Clients mit der Datenbank vor. Die Grundlage für beide Verbindungsmodelle bilden dabei die **Data-Provider** sowie das **DataSet**.

DataProvider

- stellen den Zugriff auf die Datenbank her,
- führen Befehle auf der Datenbank aus,
- holen Ergebnisse aus der Datenbank heraus und
- stehen hauptsächlich für den connected - Zugriff auf Datenbanken.

DataSets

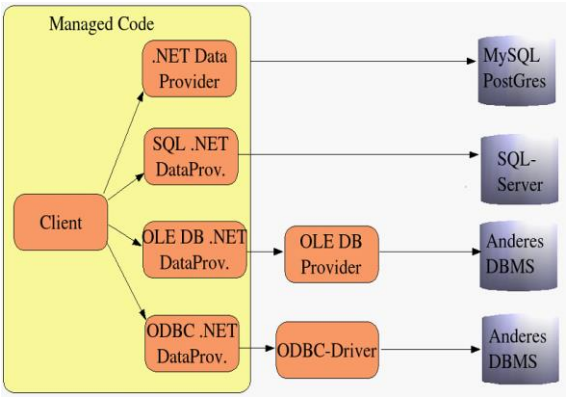
- dienen als lokaler Speicher von Daten auf dem Client oder der Mittelschicht,
- können aus unterschiedlichsten Quellen stammen (Datenbank, XML - Files) und
- bieten die Funktionalität für disconnected Datenobjekte, sind also als Cache für Daten zu betrachten.



1.2 Der Zugriff auf verschiedne Datenbanksysteme

☞ ***Beschreiben Sie den Zugriff auf verschiedene Datenbanksysteme!***

☞ ***Weshalb sollten der OLE DB und ODBC .NET – Data Provider nach Möglichkeit nicht eingesetzt werden?***



Der Zugriff des Clients auf verschiedene Datenbanksysteme erfolgt im .NET – Framework über datenbankspezifische Treiber. Im Idealfall verfügt der Datenbankhersteller über einen nativen Treiber. Sollte dies nicht der Fall sein, kann über sog. OLE – DB oder ODBC .NET – Data Provider eine Brücke zu existierenden OLEDB bzw. ODBC - Datenbanktreibern geschlagen werden.

Da die Verwendung von OLE DB und ODBC Treibern durch die zusätzliche Schicht beim Datenzugriff Performancenachteile mit sich bringt, sollte man diesen Weg nur als letzte Lösung betrachten. Mittlerweile verfügen alle wichtigen Datenbanken über native Datenbanktreiber für das .NET Framework.

1.3 Wichtige Klassen des Datenbank - Providers

☞ ***Beschreiben Sie die Aufgaben der wichtigsten Data Provider Klassen!***

☞ ***Welche Klassen finden im connected, welche im disconnected Modell ihren Einsatz?***

Der Datenbank - Provider stellt die eigentliche Verbindung zur Zusammenarbeit mit einer Datenbank her. Im Objektmodell von ADO .NET wird dies mit Hilfe von folgenden Klassen bewerkstelligt:

Connection

- Sie erstellt bzw. löst Verbindungen zu einer bestimmten Datenbank.
- Über die Connection kann ein **Transaction – Objekt** erzeugt und eine Transaktion gestartet werden.

Command

- Es Dient dem Erstellen und Ausführen von Befehlen (z.B. SQL - Abfragen) in der Datenbank und
- kann zum Ausführen von Stored Procedures sowie dem Anlegen von **Parameter Objekten** genutzt werden.

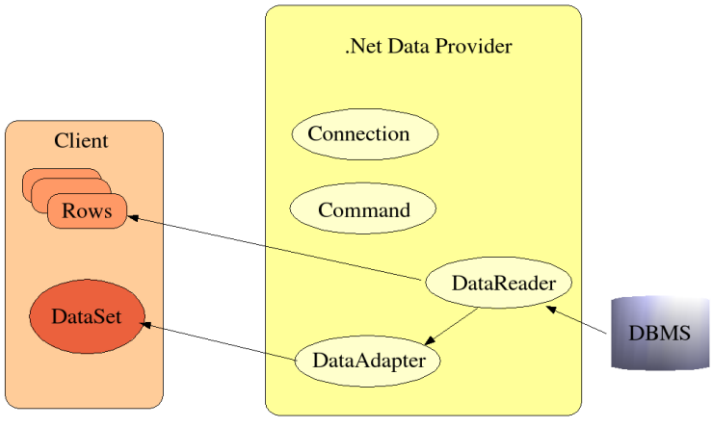
DataReader

- stellten einen direkten, sequentiellen (forward - only), lesenden Zugriff auf die Daten der Datenbank zur Verfügung. Das Lesen von Daten über einen DataReader erfolgt im connected Modell sehr schnell.

DataAdapter

- sind Abgeleitet von DataReader,
- erzeugen und füllen Objekte der Klasse **DataSet** im disconnected Modell. DataSets erlauben dabei einen flexibleren Zugriff auf Daten als DataReader – Objekte.

Folgende Grafik verdeutlicht die prinzipielle Zusammenarbeit der Data Provider Klassen im connected und disconnected Modell:



Connection und Command Objekte finden sowohl im disconnected als auch im connected Model von ADO .NET ihren Einsatz.

Während beim **connected Model** Daten über einen DataReader als Rows zum Client übertragen und mit Command Objekten Daten in der Datenbank bearbeitet werden können, arbeitet man im **disconnected Modell** mit einem DataAdapter und einem DataSet. Der DataAdapter dient als Verbindung zwischen dem DataSet (lokaler Puffer der Daten auf dem Client) und der Datenbank. Nur wenn Daten über ihn geladen, eingefügt, modifiziert oder gelöscht werden besteht eine direkte Verbindung zur Datenbank.

2 Arbeiten im Connected Modell

☞ ***Erläutern Sie die Vor- und Nachteile des Connected Modells!***

☞ ***Nennen Sie Beispiele zum Einsatz des Connected Modells!***

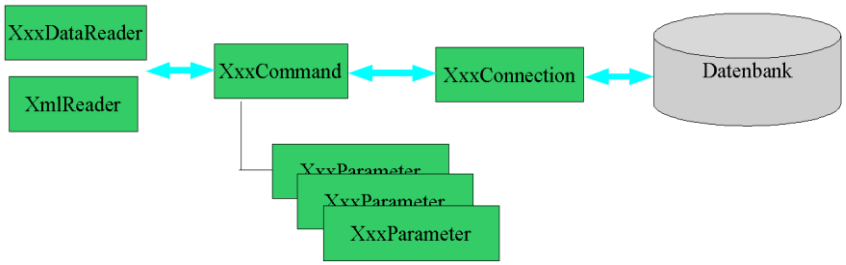
Beim Connected Modell besitzt die Clientanwendung einen ständigen Zugriff auf die Datenbank. Das Lesen, Schreiben, Ändern und Löschen von Daten erfolgt mit Hilfe von Command Objekten direkt in der Datenbank. Neben dem Vorteil immer mit den aktuellsten Daten zu Arbeiten und kaum Probleme mit der Datenkonsistenz zu haben werden in diesem Modell aber hohe Anforderungen an die Ressourcen der Datenbank Hardware gestellt, da eine einzige Clientapplikation unter Umständen bereits mehrere Verbindungen zur Datenbank benötigt.

Das Connected Modell eignet sich beispielsweise für Clientapplikationen, die auf Rechnern innerhalb eines Firmennetzwerks ausgeführt werden.

2.1 Ein Überblick über die verwendeten Klassen

☞ ***Erklären Sie, wann ein DataReader eingesetzt werden kann!***

Beim Connected Modell wird die Verbindung zur Datenbank über ein Connection Objekt hergestellt. Benötigt man nur Lesezugriff auf die Daten, so führt man einen DataReader über ein zugehöriges Command Objekt aus.



Sollen Daten auch geändert, eingefügt oder gelöscht werden können, so erfolgt die mit Hilfe von Command Objekten über SQL – Statements oder gespeicherte Prozeduren in der Datenbank. Command Objekte enthalten dazu oftmals Parameter Objekte in einer Parameters – Collection.

2.1.1 Das Connection - Objekt

☞ ***Beschreiben Sie die Aufgaben des Connection Objekts!***

Das Connection - Objekt stellt mit Hilfe des ConnectionStrings die Verbindung zur Datenbank her. Es handelt die Änderung und das Schließen einer Verbindung und handhabt die Transaktionssteuerung über ein Transaction Objekt mit der Datenbank.

Wichtige Methoden des Connection – Objektes sind:

- Open( ): öffnet eine Verbindung zu einer Datenbank
- Close( ): schließt eine geöffnete Datenbankverbindung

2.1.2 Das Command - Objekt

☞ ***Beschreiben Sie die wichtigsten Methoden und Properties des Command Objekts!***

Command - Objekte stehen stellvertretend für die SQL-Befehle, die an eine Datenbank geschickt werden können. Sie nutzen ein vorhandenes Connection - Objekt, um ihren SQL - Befehl an die Datenbank zu senden oder eine gespeicherte Prozedur aufzurufen. Command - Objekte verfügen über Parameterlisten, mit denen der SQL-String zum Beispiel **WHERE -** Bedingungen genau definieren sowie die Rückgabewerte der Datenbank aufnehmen kann.

Wichtige Properties des Command - Objektes sind:

- **Connection**: speichert die Verbindung zur Datenbank als Connection Objekt
- **CommandText**: enthält den Query - String oder den Namen der gespeicherten Prozedur
- **CommandType**: legt die Art des CommandTextes fest(StoredProcedure, SQL - Query, ..)
- **CommandTimeout**: Anzahl der Sekunden bis zum TimeOut

Wichtige Methoden des Command – Objektes sind:

- **ExecuteNonQuery( )**: Wird verwendet, wenn ein SQL - Statement keine Daten zurück gibt (z.B. Updates, Inserts oder Deletes).
- **ExecuteReader( )**: liefert ein DataReader – Objekt zum Lesen von Daten eines SELECT – Befehls
- **ExecuteScalar( )**: wird für SELECT - Statements benutzt, welche nur einen einzigen Wert zurück liefern sollen

2.1.3 Das DataReader - Objekt

☞ ***Beschreiben Sie den Zugriff auf die Datensätze über einen DataReader!***

☞ ***Welche Einschränkungen bestehen beim Benutzen eines DataReaders?***

DataReader - Objekte erstellt und initialisiert man über die ExecuteReader( ) Methode eines Command - Objekts. Die entsprechenden SQL - Befehle (SELECT - Anweisungen) werden an die Datenbank gesendet und die Ergebnismengen in ein DataReader- Objekt verpackt. Über den DataReader steht den nutzenden Klassen ein schneller, forward only, read only Zugang zu den Resultsets zur Verfügung.

Wichtige Datenzugriffsmethoden des DataReaders sind:

- über den **Feldnamen** oder den **Index** des Resultsets: z.B. myReader[“Feldname“] bzw. myReader[2].
- **GetXXX(int SpaltenIndex)**: DataReader liefern Werte vom Typ Object zurück, welche vor Gebrauch gecastet werden müssen. Zum Lesen bereits konvertierter Daten verfügt der DataReader vorgefertigte GetXXX(...) Methoden für alle Standard Datentypen (z.B. myReader.GetString(2)).
- **Read( )**: bewegt den Reader zum nächsten Datensatz und liefert true zurück, solange noch Datensätze vorhanden sind.
- **NextResult( )**: schaltet bei SQL – Abfragen mit mehreren Resultsets auf das nächste Resultset weiter.

2.2 Ein Codebeispiel zum Connected Modell

☞ ***Analyieren Sie die Objekte und eingesetzten Methoden des Quelltextbeispiels!***

Das folgende Quelltextbeispiel zeigt den Zugriff auf Daten einer Datenbank über ein DataReader – Objekt unter ADO.NET:

```
01  // Zugriff im Connected Modell per DataReader
02  using System.Data.SqlClient;
03  …
04  class DataReaderExample
05  {
06      public static void Main( )
07      {
08          // Connection Objekt erstellen
09          SqlConnection cn =
10          new SqlConnection("data source=localhost;initial catalog=example;integrated security=SSPI");
11          // Command Objekt erstellen und initialisieren
12          SqlCommand cmd = new SqlCommand( )
13          cmd.Connection = cn;
14          cmd.CommandText = "SELECT name, age FROM employees";
15          // Datenbankverbindung oeffnen
16          cn.Open( );
17          // Initialisieren des DataReaders mit ExecuteReader( )
18          SqlDataReader dr = cmd.ExecuteReader( );
19          // Solange Datensaetze vorhanden sind den naechsten lesen
20          while (dr.Read( ))
21          {
22             System.Console.WriteLine("Name:{0},Age:{1}",dr.GetString(0), dr.GetInt32(1));
23          }
24          // DataReader und Datenbankverbindung schliessen
25          dr.Close( );
26          cn.Close( );
27      }
28  }
```

Nach dem Erzeugen eines Connection – Objekts wird ein Command Objekt erstellt. Die Properties Connection und CommandText des Command – Objektes sind zu initialisieren.

Mit der Open( ) Methode öffnet man die Verbindung zur Datenbank.

Das Ausführen der SQL – Abfrage über die ExecuteReader( ) Methode des Connection – Objektes liefert ein DataReader – Objekt zurück.

In einer while – Schleife durchläuft man im Anschluss daran alle Datensätze und den Namen und das Alter der Employees auf dem Bildschirm aus.

Am Ende müssen noch der DataReader und die Datenbankverbindung geschlossen werden um keine unnötigen Ressourcen der Datenbank zu beanspruchen.

3 Arbeiten im Disconnected Modell

☞ ***Erläutern Sie die Vor- und Nachteile des Disconnected Modells!***

☞ ***Nennen Sie Beispiele zum Einsatz des Disconnected Modells!***

Beim Disconnected Modell besitzt die Clientanwendung keine ständige Verbindung zur Datenbank. Die bearbeiteten Daten liegen in einer Art Cache, dem DataSet. Nur wenn aktuelle Daten eingelesen oder Änderungen mit INSERT, UPDATE oder DELETE in der Datenbank durchzuführen sind wird eine Verbindung zur Datenbank hergestellt.

Da die Clientanwendungen nicht ständig mit der Datenbank verbunden sind werden die Datenbank und Netzwerkressourcen geschont. Clientanwendungen mit tausenden von gleichzeitigen Nutzern sind möglich.

Neben diesen Vorteilen können beim disconnected Modell aber auch Probleme mit der Datenkonsistenz auftreten wenn z.B. von einem Benutzer ein Datensatz gelöscht wird, der gerade von einem anderen Benutzer bearbeitet wird.

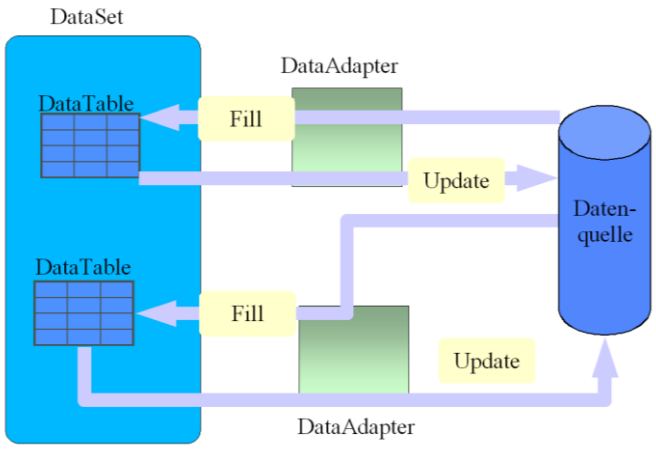
Das disconnected Modell eignet sich zum Beispiel für Internet Warenhäuser mit tausenden von Kunden, die auf die Daten der Waren zugreifen oder Bestellungen verschicken. Ebenso können Außendienstmitarbeiter Teile ihrer Kundendaten unterwegs mit einem Laptop bearbeiten und abends in der Firma mit der Datenbank abgleichen.

3.1 Ein Überblick über die verwendeten Klassen

☞ ***Beschreiben Sie die Zusammenarbeit von DataAdapter, DataSet und DataTables im disconnected Modell!***

Kernstück des Disconnected Modells sind die Klassen DataSet und DataAdapter. Der DataAdapter bildet dabei die Verbindung zwischen der Datenbank und dem DataSet in der Clientapplikation. Das DataSet wiederum enthält die zu bearbeitenden Daten als lokaler Cache in einer oder mehreren DataTable – Objekten gespeichert. Zwischen den Tabellen des DataSets können Relationen hergestellt werden, welche die Einhaltung der referentiellen Integrität der Daten garantieren. In der Regel existiert ein DataAdapter pro DataTable im DataSet.

Die Aufgabe des DataAdapter

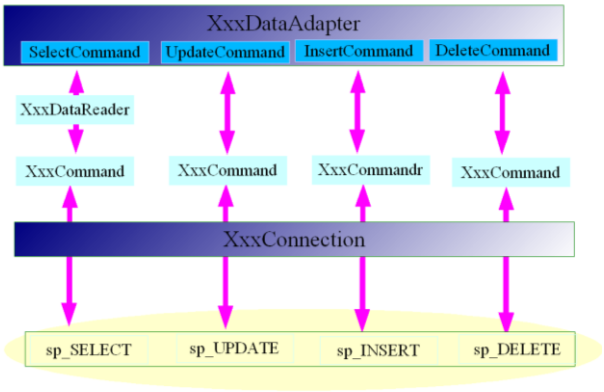


Benötigt man nun die aktuellsten Daten aus der Datenbank, so stellt der DataAdapter in der Fill( ) Methode die Verbindung zur Datenbank her und überträgt die Daten in das DataTable Objekt.

Sollen neue, geänderte oder gelöschte Datensätze aus einer Tabelle des DataSets an die Datenbank zurück geschrieben werden, so erfüllt diese Aufgabe wiederum der DataAdapter über seine gespeicherten Insert-, Update- und DeleteCommand – Objekte mit der Update( ) Methode.

3.1.1 Das DataAdapter - Objekt

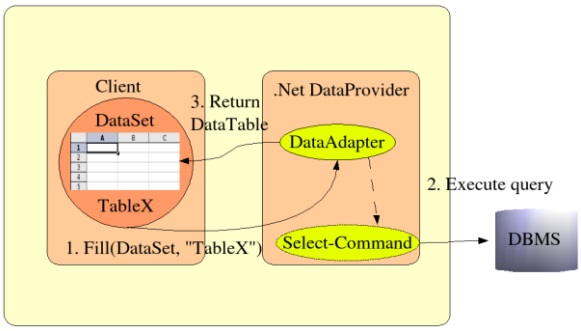
- ***Nennen Sie die vier Command – Objekte eines DataAdapters und erläutern Sie deren Aufgabe!***
- ***Wie werden Daten in ein DataSet eingelesen?***
- ***Wie viele DataAdapter arbeiten in der Regel mit einem DataSet zusammen?***



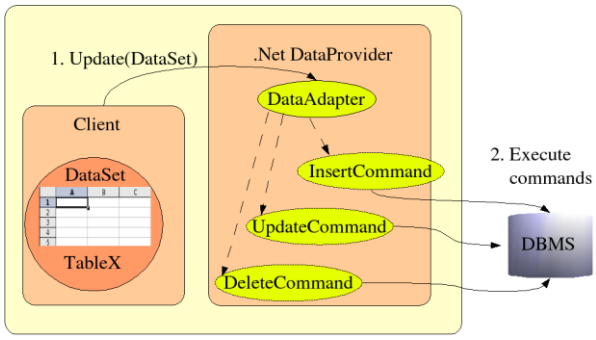
Während ein DataReader - Objekt nur einen lesenden Zugriff auf die Datenbank zulässt, verfügen DataAdaper - Objekte über vier weitere Command - Objekte, die auch einen schreibenden Zugriff auf die Datenbank zur Verfügung stellen.

Der DataAdapter dient insbesondere als Bindeglied der so genannten DataSet - Objekte zur eigentlichen Datenbank. Für die Verbindung zur Datenbank speichert er daher in seiner Connection Property ein Connection – Objekt.

Hauptaufgabe des DataAdaper-Objektes ist es, der lokalen Anwendung so genannte DataSet - Objekte zur Verfügung zu stellen. Mit Hilfe der Fill( ) Methode wird dazu das **SelectCommand** des DataAdapters ausgeführt, welches über ein SQL – SELECT Statement oder eine gespeicherte Prozedur Daten aus der Datenbank in das DataSet schreibt, mit denen die Anwendung dann lokal arbeiten kann. Die Verbindung zur Datenbank ist nach dem Einlesen der Daten zu schließen (Disconnected Modell). Verfügt ein DataSet über mehrere DataTables, so sind in der Applikation in der Regel auch mehrere DataAdapter implementiert, ein DataAdapter pro Tabelle.



Alle möglichen Änderungen (löschen, ändern, einfügen), die im DataSet innerhalb der lokalen Anwendung auf dem Client durchgeführt worden sind, werden über den DataAdapter und seine **Insert**-, **Update**- und **DeleteCommand** - Objekte wiederum in die Datenbank überführt.



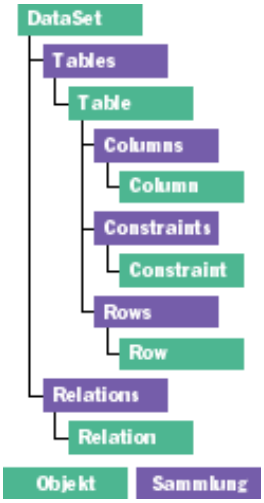
Dazu verwendet man die Update(...) Methode des DataAdapters und übergibt als Parameter nach Bedarf das gesammte DataSet oder einzelne darin enthaltene DataTables. Die drei Commands führen daraufhin die entsprechenden Änderungen über SQL – Statements oder stored procedures in der Datenbank aus.

3.1.2 Das DataSet - Objekt

☞ ***Erläutern Sie die Aufgaben des DataSets in einer Clientapplikation!***

☞ ***Welche Collections sind in einem DataSet enthalten?***

Damit die Clientapplikation unabhängig von einer Datenbankverbindung arbeiten kann benötigt diese einen Puffer für die Daten. Diesen lokalen Datencache bietet das DataSet. Entsprechend dem disconnected Modell ist ein DataSet nicht ständig mit einer Datenbank verbunden. Nur beim Lesen von Daten oder dem Ausführen von Anweisungen in der Datenbank wird eine Verbindung über die DataAdapter hergestellt (connect, execute statement, disconnect).



Zur Abbildung eines relationalen Datenspeichers besitzt das DataSet zwei Collections. Über die **Relations – Collection** werden Verbindungen zwischen DataTable - Objekten hergestellt, welche für die Navigation zwischen Parent und Child Tabellen einzusetzen sind. In der **Tables - Collection** speichert das DataSet die Tabellen, welche sich wiederum aus Zeilen und Spaltenobjekten in den entsprechenden Collections zusammensetzen. Diese DataTables können auf physische Tabellen in der Datenbank gemappt sein.

Zur Einhaltung der referentiellen Integrität bei der Arbeit im DataSet besitzt jede Tabelle die Möglichkeit Constraints in ihrer **Constraints - Collection** zu definieren. Das Löschen von Einträgen in der Mastertabelle, für die Daten in einer abhängigen Tabelle existieren, kann damit verhindert werden. Das DataSet ist daher mit einer relationalen Datenbank vergleichbar, die im Arbeitsspeicher liegt.

Neben der Verwaltung der Daten besitzt ein DataSet weiterhin Methoden, mit denen man Daten oder Schemainformationen wie enthaltene Tabellen, Spalten, Relationen und Constraints im XML Format einlesen oder abspeichern kann.

Zu den wichtigen Properties eines DataSets zählen

- **Tables**: eine Sammlung aller enthaltenen Tabellen,
- **Relations**: eine Sammlung aller enthaltenen Relationen zwischen den Tabellen,
- **EnforceConstraints**: gibt an, ob bei Aktualisierungen Einschränkungsregeln beachtet werden oder nicht,
- **HasErrors**: gibt an, ob in Zeilen des DataSets Fehler nach einer Aktualisierung vorliegen.

Wichtige Methoden des DataSets sind

- **HasChanges( )**: prüft, ob seit dem letzen Laden der Daten oder dem letzten Aufruf von AcceptChanges( ) Daten im DataSet geändert worden sind.
- **GetChanges( )**: liefert ein DataSet zurück, in dem nur die geänderten Datensätze gespeichert sind. Somit muss bei einem Update nicht das gesammte DataSet an die Datenbank zurückgeschrieben werden.
- **AcceptChanges( )**: Übernimmt alle Änderungen, die am DataSet seit dem letzten Ladevorgang oder dem letzten Aufruf von AcceptChanges vorgenommen wurden.
- **ReadXml( )**: liest Daten im XML Format in das DataSet ein,
- **WriteXml( )**: schreibt Daten aus einem DataSet im XML Format in eine Datei.

3.2 Ein Codebeispiel zum Disconnected Modell

☞ ***Analyieren Sie die Objekte und eingesetzten Methoden des Quelltextbeispiels!***

Das folgende Quelltextbeispiel zeigt den Zugriff auf Daten einer Datenbank über ein DataSet und ein DataReader – Objekt unter ADO.NET:

```
01  // Zugriff im disconnected Modell per DataSet
02  using System.Data.SqlClient;
03  …
04  class DataSetExample
05  {
06     public static void Main( )
07     {
08        // Connection – Objekt erstellen
09        SqlConnection cn =
10           new SqlConnection("data source=localhost;initial catalog=example;integrated security=SSPI");
11        // Command Objekt erstellen und initialisieren
12        SqlCommand cmd = new SqlCommand( );
13        cmd.Connection = cn;
14        cmd.CommandText = "SELECT name, age FROM employees";
15        // DataAdapter erstellen und initialisieren
16        SqlDataAdapter da = new SqlDataAdapter( );
17        da.SelectCommand = cmd;
18        // DataSet erstellen
19        DataSet ds = new DataSet( );
20        // Datenbankverbindung oeffnen
21        cn.Open( );
22        // Daten in DataSet einlesen
23        da.Fill(ds, "NameUndAlter");
24        // Datenbankverbindung schliessen
25        cn.Close( );
26     }
27  }
```

Nach dem Erzeugen eines Connection – Objekts wird ein Command Objekt erstellt. Die Properties Connection und CommandText des Command – Objektes sind zu initialisieren.

Dem DataAdapter ordnet man das SelectCommand zum Lesen der Daten aus der Datenbank zu. Mit der Open( ) Methode öffnet man nach der Erstellung eines DataSet – Objektes die Verbindung zur Datenbank .

Das Ausführen der SQL – Abfrage über den DataAdapter in dessen Fill( ) Methode schreibt die Daten in das zuvor generierte DataSet.

Am Ende ist nur noch die Datenbankverbindung zu schließen (disconnected Modell). Die Daten befinden sich jetzt im lokalen Puffer zur Bearbeitung.

4 Das Entity Framework (ORM)

Mit dem Entity Framework existiert nun auch ein objektorientierter Zugang zu Datenquellen im .NET Framework und .NET Core.

4.1 ORM Definition

☞ ***Erklären Sie den Begriff ORM!***

Ein ORM (Object-Relational-Mapping) ist eine Programmiertechnik, die es ermöglicht, Operationen an einer relationalen Datenbank in einer objektorientierten Programmiersprache auszuführen, ohne dass man sich explizit mit SQL-Anweisungen auseinandersetzen muss. ORM Tools erstellen für jede Tabelle in der Datenbank eine Klasse und für jede Zelle ein Objekt. Diese Objekte, die Daten aus der Datenbank repräsentieren, können konventionell im Programm genutzt werden, wie jedes andere Objekt, um mit den Daten aus der Datenbank zu interagieren (Mapping).

ORMs sind besonders nützlich, wenn man mit verschiedenen Datenbanken arbeitet, da sie es ermöglichen, den Code unabhängig von der verwendeten Datenbank zu schreiben. Sie können auch dazu beitragen, die Leistung von Anwendungen zu verbessern, indem sie sicherstellen, dass optimierte SQL-Anweisungen generiert werden, um die Daten aus der Datenbank zu laden.

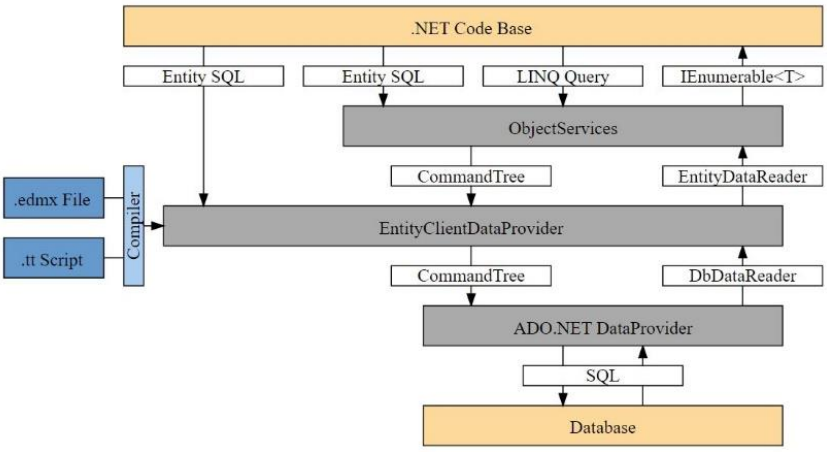
Außerdem lässt ORM die Abfragen in Form von Methodenaufrufe zu schreiben, was die Entwicklung beschleunigt. Beispiele für ORMs sind Hibernate für Java oder Entity Framework für .NET.

4.2 Das Entity Framework

☞ ***Erläutern Sie den Zweck des Entity Frameworks!***

Das Entity Framework ist ein ORM für .NET, der es Entwicklern ermöglicht, die Daten aus der relationalen Datenbank in einem .NET-Programm zu verwenden und die Datenbankabfragen in Form von LINQ Funktionen zu schreiben. Diese Methodenaufrufe werden automatisch in SQL Abfragen übersetzt und ausgeführt.

Das Entity Framework (.NET) bietet dazu Funktionen wie eine visuelle Designeroberfläche zum Definieren von Datenmodellen und Mapping zwischen objektorientierten Klassen und relationalen Tabellen. Es enthält auch Unterstützung für das lazy loading von Daten, bei dem Daten erst dann aus der Datenbank geladen werden, wenn sie tatsächlich benötigt werden. Ebenso bestehen Funktionen zum Verwalten von Änderungen an Daten, die in der Anwendung vorgenommen wurden, um sie in die Datenbank zu übernehmen.



4.2.1 Code First vs. Database First Approach

☞ ***Beschreiben Sie den Unterschied zwischen dem Code First und dem Database First Ansatz!***

Das Entity Framework bietet zwei Hauptansätze, um das Programm und die Datenbank zu verbinden.

Bei der **Code First Approach** wird der Code als Hauptquelle für die Definition der

Datenbankstruktur verwendet. Dies bedeutet, dass der Entwickler Klassen in der Anwendung schreibt, die das Datenmodell darstellen und Entity Framework verwendet diese Klassen, um die Datenbankstruktur zu erstellen.

Der Vorteil dieses Ansatzes besteht darin, dass der Entwickler vollständig Kontrolle über die Struktur der Datenbank hat und diese strukturieren kann, wie es für die Anwendung am besten geeignet scheint. Der Nachteil ist, dass der Entwickler möglicherweise mehr Zeit damit verbringen muss, die Datenbankstruktur manuell zu definieren.

Bei der **Database First Approach** wird die vorhandene Datenbank als Hauptquelle für die Definition des Datenmodells verwendet. Dies bedeutet, dass das Entity Framework die Struktur der vorhandenen Datenbank analysiert und Klassen in der Anwendung generiert, die das Datenmodell darstellen.

Der Vorteil dieses Ansatzes besteht darin, dass der Entwickler schnell mit der Entwicklung von Anwendungsfunktionen beginnen kann, ohne sich Gedanken über die manuelle Definition der Datenbankstruktur machen zu müssen. Der Nachteil ist, dass der Entwickler möglicherweise weniger Kontrolle über die Struktur des Datenmodells hat und es schwieriger sein kann, Änderungen an der Datenbankstruktur vorzunehmen.

4.2.2 Data Annotations

☞ ***Welchen Zweck erfüllen Data Annotations?***

Data Annotations sind .NET Attribute, die verwendet werden, um die Struktur und das Verhalten von Klassen und Eigenschaften im Code zu definieren. Häufig verwendete Annotations sind z.B. Key, ForeignKey(…) oder Required.

```
public class User
{
    [Key]
    public int CustomerId { get; set; }

    [MaxLength(50)]
    public string Name { get; set; }

    [Required]
    public string Email { get; set; }
}
```

Data Annotations werden häufig in Verbindung mit der Code First Approach verwendet, um das Entity Framework beim Generieren einer Datenbank aus dem Code heraus zu leiten. Sie können jedoch auch in Verbindung mit der Database First Approach verwendet werden, um das Verhalten von Klassen im Code genauer zu konfigurieren.

4.2.3 Die DbContext Klasse

☞ ***Erläutern Sie die Bedeutung der DbContext Klasse!***

☞ ***Welche Methoden enthält DbContext für Datenbankoperationen?***

☞ ***Wie wird der DbContext initialisiert?***

Der DbContext ist eine Klasse in Entity Framework, die als Verbindungspunkt zwischen der Anwendung und der Datenbank dient. Er verwaltet die Entitäten (die Klassen, die das Datenmodell darstellen) und die Beziehungen zwischen ihnen und führt die Datenbankoperationen aus, die von der Anwendung angefordert werden. Um einen DbContext zu verwenden, muss man ihn in der Anwendung instanziieren und ihm eine Verbindungszeichenfolge

zur Datenbank bereitstellen. Danach kann man Methoden wie Add, Remove und SaveChanges aufrufen, um Datenbankoperationen auszuführen, und die Entitäten abfragen, indem man LINQ-Abfragen verwendet.

Um einen DbContext in Entity Framework zu verwenden, muss man zunächst eine Klasse erstellen, die von der DbContext-Klasse abgeleitet wird. In dieser Klasse können die Entitätsklassen definiert werden, die das Datenmodell der Anwendung darstellen.

```
public class MyDbContext : DbContext
{
    public MyDbContext(DbContextOptions<MyDbContext> options)
        : base(options)
    {
    }

    public DbSet<User> Users { get; set; }
}
```

Vor der Verwendung des DbContext in der Anwendung muss dieser zunächst instanziiert werden. Das kann auf verschiedene Arten geschehen. Eine Möglichkeit nutzt die **DbContextOptionsBuilder** Klasse zur Initialisierung:

```
var contextOptions = new DbContextOptionsBuilder<MyDbContext>()
    .UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Test")
    .Options;

using var dbContext = new MyDbContext(contextOptions)
```

Alternativ dazu kann man auf **Dependency Injection** zurückgreifen, um den DbContext in der Anwendung bereitzustellen. Das folgende Beispiel zeigt, wie man den DbContext mit dem Dependency Injection-Container von ASP.NET Core registriert:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<MyDbContext>(options =>
        options.UseSqlServer("connection_string"));
}
```

Die Initialisierung von DbContext in einer Controller-Klasse erfolgt dann über den Konstruktor der Controller Klasse:

```
public class HomeController : Controller
{
    private readonly MyDbContext _dbContext;

    public HomeController(MyDbContext dbContext)
    {
        _dbContext = dbContext;
    }

    public IActionResult Index()
    {
        var entities = _dbContext.Entities.ToList();
        return View(entities);
    }
}
```