ADO.NET

Obiektowy dostęp do danych

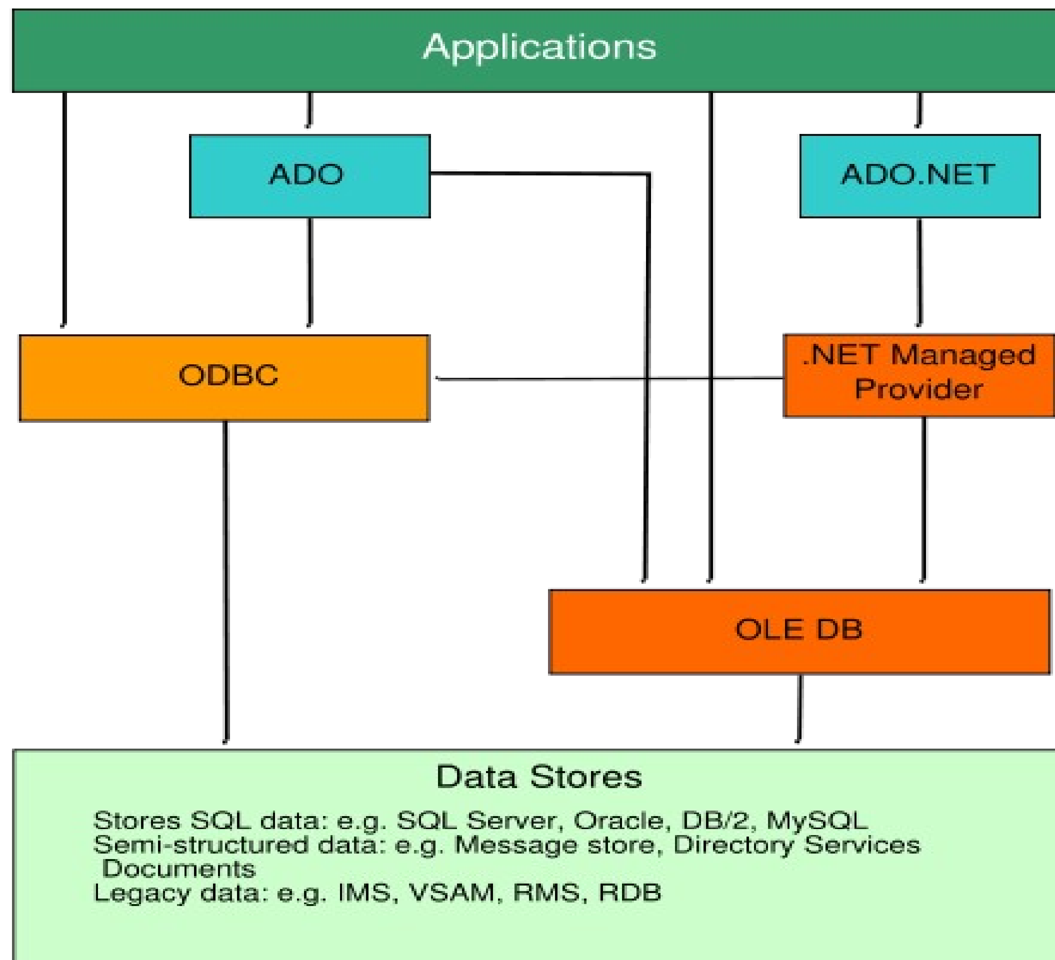
Przygotował Jakub Światły

Plan prezentacji

- Technologie dostępu do danych
- Cele i założenia ADO.NET
- Praca na danych podłączonych
- Praca na danych odłączonych
- Synchronizacja
- Obsługa XML
- Zastosowanie

Microsoft Data Access Components

źródło: http://en.wikipedia.org/wiki/Microsoft_Data_Access_Components



Open Database Connectivity

- Standardowe API do obsługi relacyjnych baz danych
- Niezależność od SZBD, języka, systemu operacyjnego
- Stara technologia – dopracowane sterowniki
- Pierwsze podejścia obiektowe:
  - DAO (przez JET)
  - RDO (bez JET)

OLE DB

- Krok w stronę COM
- Rowset – po prostu zbiór n-tek (wynik zapytania, tabela, widok, xls, tekst, rejestr)
- Wspólne mechanizmy zapytań, indeksowania, kursorów, bez konwersji danych

ActiveX Data Objects

- Nakładka na OLE DB, bez wskaźników, zarządzania pamięcią i czasem życia, gotowa do użycia w VB
- Recordset – kursor zdolny pracować z odłączonymi danymi i samodzielnie zapisywać zmiany do bazy
- Abstrakcja od stanu danych (podłączone czy odłączone)

Po co znowu coś innego?

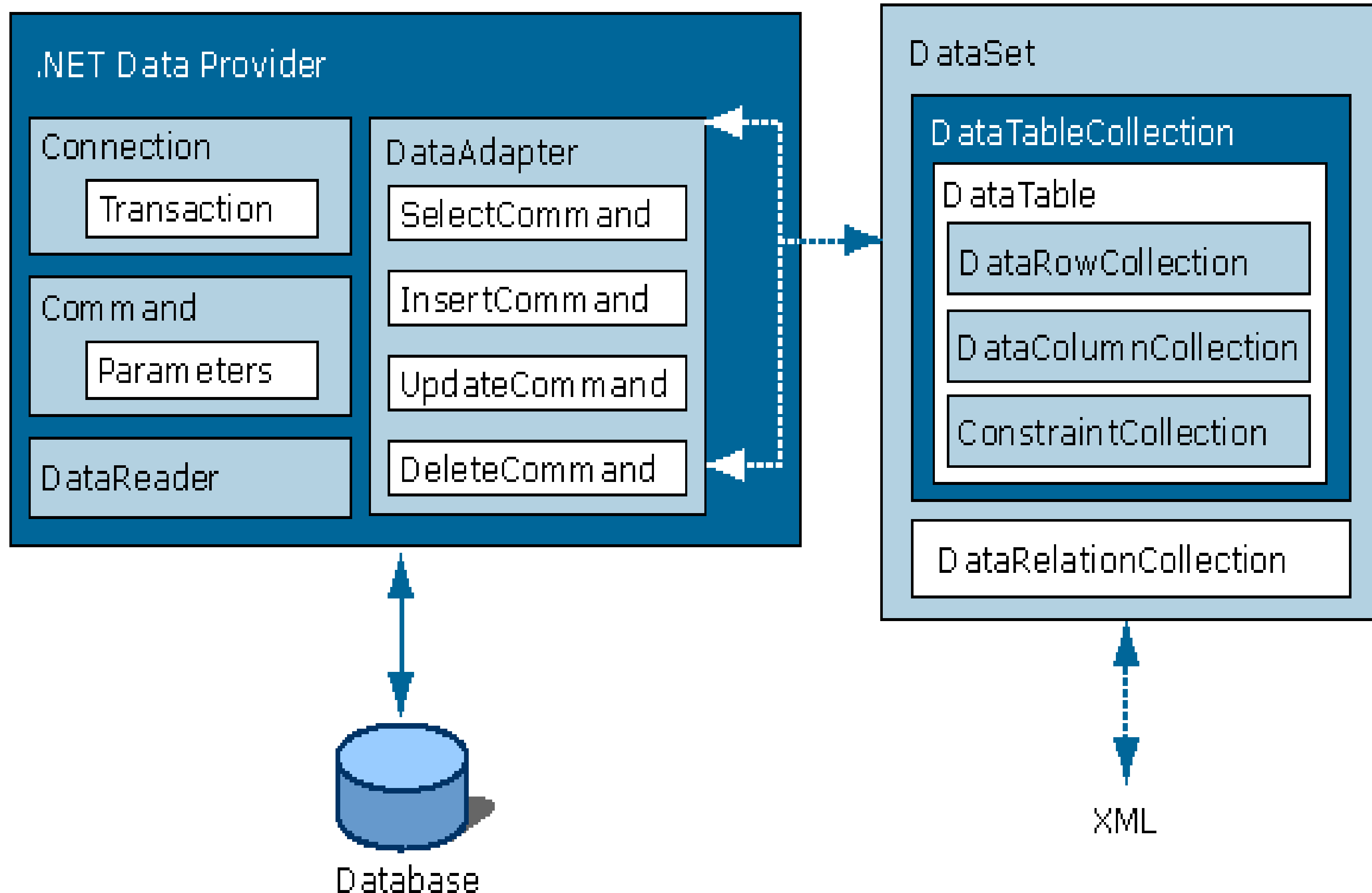
- Java wyparła COM
- Internet wymaga odłączonych danych
- Konieczna kontrola i elastyczność operacji UPDATE
- Popularność XML

ADO.NET - założenia

- Platforma .NET
- Silne rozgraniczenie danych podłączonych i odłączonych
- Wygodne, ale wysoce konfigurowalne przejścia między tymi stanami
- Solidna obsługa XML
- Zalety poprzednich wersji

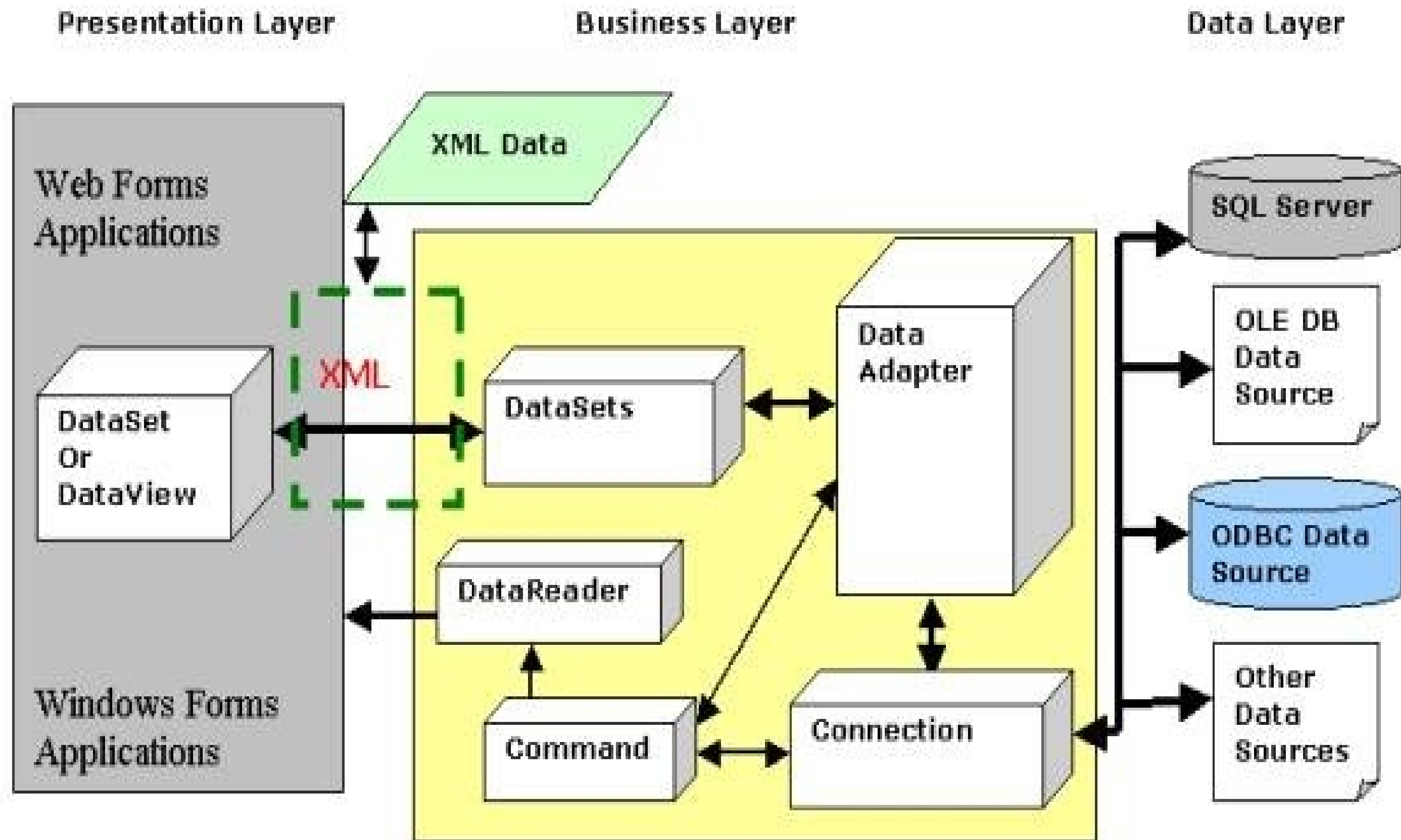
Architektura

źródło: http://www.dotnetforce.com



Architektura 2

źródło: http://www.vbdotnetheaven.com/



Dane Podłączone

- Połączenie
  - Connection
  - Transaction
- Odpytywanie bazy danych
  - Command
  - DataReader
  - Parameter
  - DataAdapter

Dostawcy danych w .NET

- Abstrakcyjny OleDb
  - przestrzeń System.Data.OleDb
  - Przedrostek OleDb
- MS SQL Server
  - Przestrzeń System.Data.SqlClient
  - Przedrostek Sql
- Abstrakcyjny ODBC
- Oracle
- SQL XML .NET

Połączenie

```
string connectionString = „Provider=[MSDAORA|
SQLOLEDB|...]; Data Source=MyServer; User Id
=kuba; Password=hasło”;

Connection cn = new OleDbConnection
(connectionString);

cn.Open();

cn.Close();
```

Połączenie

- Pula połączeń
- `cn.`**`CreateCommand`**`();`
- `cn.`**`BeginTransaction`**`();`
- `cn.`**`GetOleDbSchemaTable`**`(...);`

Odpytywanie bazy

- **Command** cmd = cn.CreateCommand();
- cmd.**CommandText** = „SELECT pleple FROM bleble”;
- cmd.**CommandText** = „[schUser].[dropUser]”;
- cmd.**CommandType** = „Text|StoredProcedure”;
- cmd.**ExecuteNonQuery**();
- cmd.**ExecuteScalar**();
- **DataReader** reader = cmd.**ExecuteReader**();
- cmd.**Dispose**();

DataReader

- rdr.**Read()**;
- rdr[„email”];
  - Lepiej int ordinal = rdr.GetOrdinal(„email”);
  - rdr.GetString(ordinal);
- rdr.**NextResult**() - batch queries (następny wynik wierszowy!)
- rdr.**Close();**

Co będzie jak nie zamkniemy rdr i chcemy zrobić coś nowego?

- Połączenie jest zajęte. Jakie były rozwiązania?
- Chronologicznie:
  - UBSQL → wyjątek
  - DAO / JET → niejawne nowe połączenie
  - RDO → wyjątek
  - ADO → niejawne nowe połączenie
  - ADO.NET → wyjątek

Parametry

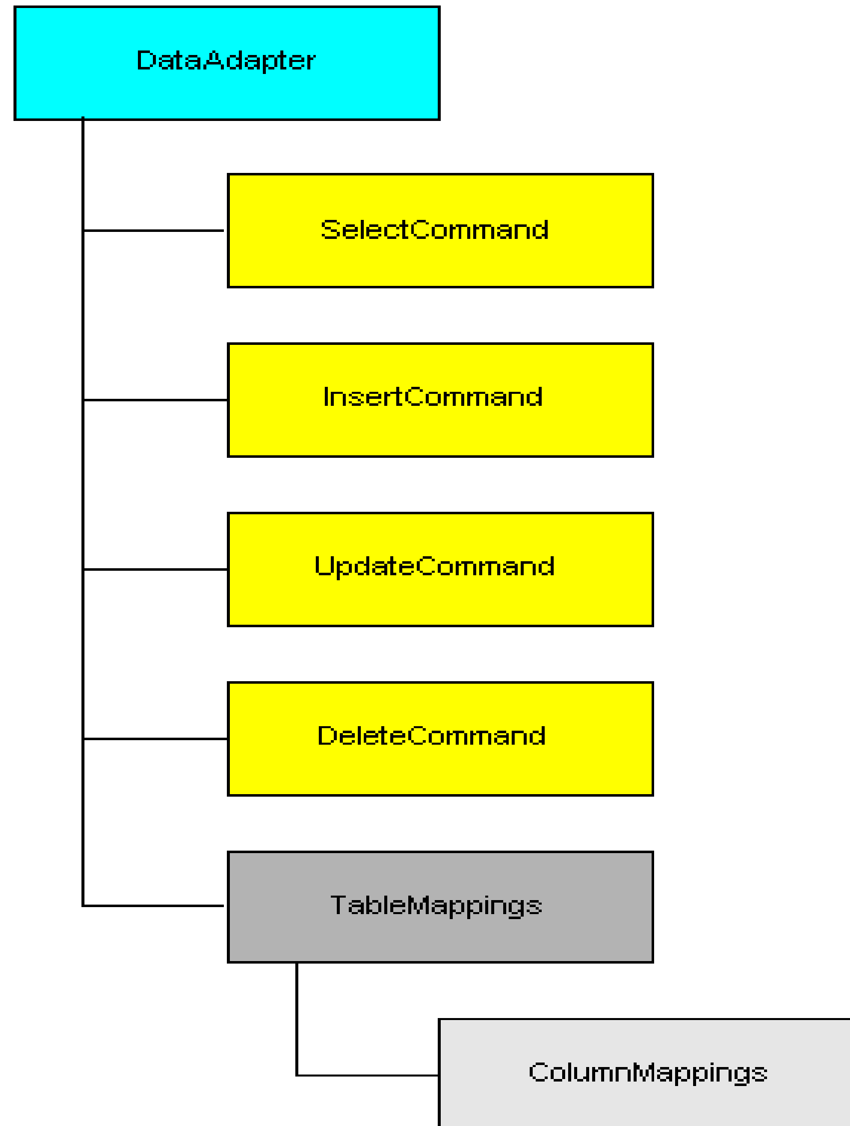
- `cmd.`**`Parameters.Add`**`(new `**`Parameter`**`(`**`SqlDataType.Int32`**`, „@userId”, 15);`
- Ustawianie kierunków
- Odczytywanie wyników (po przeczytaniu wszystkiego z rdr)

Odłączanie danych

- **string** strSQL = „SELECT * FROM Users”;
- **string** strConn = …;
- **DataAdapter** da = new **OleDbDataAdapter**(strSQL, strConn);
- **DataSet** ds = new **DataSet**();
- da.**Fill**(ds,”Users”);

DataAdapter

źródło: http://conferences.codegear.com/



DataAdapter

- Jeśli nie podamy nazwy tabeli → Table, Table1..
- A jeśli wywołamy `ds.Fill(da)` raz po razie?
  - Problem kluczy
- Jeśli chcemy sami zarządzać połączeniem:
  - `da.SelectCommand.Connection = cn;`
  - `cn.Open();`
  - `ds.Fill(da);`
  - `cn.Close();`

Stronicowanie wyników

- `int startingRow = 50;`
- `int rowsToRetreive = 10;`
- `int rowsRetrieved = da.`**`Fill`**`(ds, startingRow, rowsToRetrieve, „Users”);`

Niby zgrabnie, ale co jest pod spodem?

Odwzorowanie nazw

- **DataTableMapping** tblMap = da.**TableMappings**.**Add**(„Table”,”Users”);
- **DataColumnMapping** colMap = tblMap.**CollumnMappings**.**Add**(„UID”, „UserId”);

- `da.`**`FillSchema`**`(ds,`**`SchemaType.Source`**`, ”Users”);`

  `nazwy, typy, PK, AllowDbNull,Length`
- `Słaba wydajność`

Więzy

- **DataTable** tbl = ds.**Tables.Add**(„Users”);
- tbl.**Constraints.Add**(new **UniqueConstraint**(tbl.**Columns**[„UserId”]));
- tbl.**Constraints.Add**(new **ForeignKeyConstraint**(ds.**Tables**[„Roles”].**Columns**[„RoleId”],tbl.**Columns**[„RoleId”]));
- tbl.**PrimaryKey** = new **DataColumn**[] {tbl.**Columns**[„UserId”]};

Kolumny wyliczane

- **DataColumn** col = tbl.**Columns.Add**(„RowId”, typeof(int));
- col.**AutoIncrement** = true;
- col.**AutoIncrementSeed** = 1;
- col.**AutoIncrementStep** = 1;

- tbl.**Columns.Add**(„ItemTotal, typeof(Decimal), **„Quantity * UnitPrice**”);

Wstawianie wiersza

- **DataRow** newRow = ds.**Tables**[„Users”].**NewRow**();
- row[„UserName”] = „kuba”;
- ds.**Tables**[„Users”].**Rows**.**Add**(newRow);

- ds.**Tables**[„Users”].**LoadDataRow**(new object[] {„kuba”, „hasło”, 2}, **false**); //true → Unmodified)

Modyfikacja wiersza

- **DataRow** row = ds.**Tables**[„Users”]**.Rows.Find**(„kuba”);
- row[„UserName”] = „Kuba”;

cache:

- row.**BeginEdit**();
- row.[„UserName”] = „Kuba”;
- row.**EndEdit()**; //row.**CancelEdit()**;

Usuwanie i kasowanie

Usunięcie z bazy

- **DataRow** row = ds.**Tables**[„Users”].**Rows**.**Find**(„kuba”);
- row.**Delete**();

Usunięcie obiektu

- ds.**Tables**[„Users”].**Remove**(row);

Stan wiersza

- `row.`**`RowState`**

  wartości: (`enum` **`DataRowState`**)
- **`Unchanged`**
- **`Detached`**
- **`Added`**
- **`Modified`**
- **`Deleted`**

Wersje pól wiersza

- `row(„UserId”,` **`DataRowVersion.Current`**`));`
- **`Current`**
- **`Original`**
- **`Proposed`**
- **`Default`**
- `row(„UserId”) == row(„UserId”,` **`DataRowVersion.Default`**`);`

Relacje między tabelami

- **DataRelation** rel = new **DataRelation**(„RolesUsers”, ds.**Tables**[„Roles”].**Columns**[„RoleId”], ds.**Tables**[„Users”].**Columns**[„RoleId”]);
- ds.**Relationships.Add**(rel);
- foreach(**DataRow** userRow in roleRow.**GetChildRows**(„RolesUsers”)) {…}
- userRow.GetParentRow(„RolesUsers”)[„RoleName”];
- ds.**Tables**[„Users”].**Columns.Add**(„Role”, typeof(string), **Parent**(„RolesUsers”).**RoleName**);

Relacje między tabelami

- ForeignKeyConstraint.UpdateRule
- ForeignKeyConstraint.DeleteRule
- { Cascade, None, SetDefault, SetNull }

- Wiele do wielu → trzeba zrobić dwie relacje z tabelą łączącą

- relacje między tabelami z różnych źródeł
- walidacja

Wyszukiwanie wierszy - PK

- DataRow row = ds.Tables[„Users”].Rows.Find(„kuba”);

- object[] criteria = new object[] {123, 234};
- DataRow row = ds.Tables[„Users”].Rows.Find(criteria);

Wyszukiwanie wierszy - WHERE

- **string** strCriteria =

  „RoleName = 'user' AND” +

  „(AccountCreated > #01/01/2003#” +

  „OR City LIKE 'Nowy %')”);
- **DataRow**[] aRows = tbl.**Select**(strCriteria);

- Uwzględnianie stanu wierszy

Sortowanie – ORDER BY

- **string** strSortOrder = „UserId DESC”;
- aRows = tbl.**Select**(strCriteria, strSortOrder);

DataView

- DataView vue = DataView(tbl);
- vue.RowFilter = „Country = 'Spain'”;
- vue.GetEnumerator();

- vue.Sort = „Country”
- DataRowView[] aRows = vue.FindRows(„Spain”);

Silnie typowany DataSet

- Visual Studio może generować klasy dziedziczące po DataSet, DataTable, DataRow
- Właściwości ds takie jak nazwy tabel
- Właściwości wierszy takie jak nazwy kolumn
- Nazwane metody np. do tworzenia wierszy, ustawiania nulli, pobierania wierszy z relacji

- Można napisać bardziej efektywny kod bez typowania, ale różnice niewielkie

Synchronizacja DataSet

- **DataAdapter** da = new **OleDbDataAdapter**(strConn);
- da.**UpdateCommand** = ...;
- da.**DeleteCommand** = ...;
- da.**InsertCommand** = ...;
- da.**Update**(tbl);

Obiekty Command

- Procedury składowane albo tekst
- Mają kolekcję parametrów (Parameter), które za pomocą właściwości SourceColumn i SourceVersion są przybite do odpowiednich kolumn w tabeli i odpowiednich wersji wartości.
- Insert potrzebuje wartości aktualnych
- Delete wartości oryginalnych
- Update zarówno aktualnych jak i oryginalnych

Metoda Update

- Iteruje po wierszach
- Wybiera polecenie odpowiednio do stanu wiersza
- Ustawia wartości parametrów
- Próbuje wykonać polecenie i rejestruje ewentualne błędy

CommandBuilder

- Jeżeli polecenie Select:
  - Zwraca wyniki z jednej tabeli
  - Zwraca kolumnę, która jest kluczem głownym
- Wtedy CommandBuilder może wygenerować domyślne polecenia Update, Insert, Delete, na podstawie metadanych z wyników polecenia Select.
- Raczej unikać

Współbieżność - strategie

- Ostatni wygrywa – WHERE bazuje tylko na PK – CommandBuilder nie pomoże
- Pierwszy wygrywa – WHERE zawiera wszystkie kolumny (oryginalne wartości) – generowana przez CommandBuilder
- Rozłączne zmiany wiersza – WHERE zawiera kolumny zmodyfikowane – można napisać samemu, ale to dosyć trudne

Problem z więzami

- Trzeba o tym pomyśleć samemu
  - Wstawiać ojców
  - Wstawiać dzieci
  - Zmieniać ojców
  - Zmieniać dzieci
  - Usuwać dzieci
  - Usuwać ojców
- Pomaga Update(DataRow[]) w połączeniu z DataTable.Select(„”,””,DataViewRowState)

Obsługa XML

- Konwersja DataSet – XML i odwrotnie
- Wypisywanie i czytanie schematu XSD, można także wywnioskować schemat dynamicznie
- Na tworzony schemat XML można wpływać właściwościami

  DataSet.DataSetName,
  DataColumn.ColumnName,
  DataColumn.ColumnMapping (Attribute, Element, Hidden),

  DataRelation.Nested

XML

- Pliki różnicowe
- XmlDataDocument synchronizuje DataSet i dokument XML
- SQL XML .NET do obsługi interfejsu XML MS SQL Server (SELECT przez Xpath, UPDATE przez pliki różnicowe).
- Zastosowania: serializacja, XSLT

Podsumowanie

- DataSet jako pamięć podręczna
- Bezstanowość HTTP
- Do czego to się może przydać?
- Pytania