Un framework per le operazioni con i database
da http://escher07.altervista.org

In questo documento viene illustrata in sintesi una libreria di funzioni utili per le operazioni con i database. La libreria è contenuta nel file sorgente DBUtility.cs e compilata come DLL la utilizzo correntemente (con alcune piccole differenze insignificanti in questo contesto) come framework per piccole implementazioni in uso presso clienti.

Premetto che è un framework nato esattamente al contrario rispetto a quello che prescriverebbero i "sacri testi" di programmazione ad oggetti : prima ho scritto l'applicazione poi via via ho spostato nel framework del codice.

Per cui, al di là degli inevitabili errori ci sono di sicuro problemi di standardizzazione (es. variabili stringa che in qualche caso si chiamano strVariabile, in altri sVariabile etc...) e di eleganza per lo meno non uniforme.

Detto ciò e detto pure che è un framework utilizzato essenzialmente con Database Oracle ed Access ritengo che possa essere almeno un buon punto di appoggio per chi voglia fare una DLL da utilizzare per lo stesso scopo in ambito veramente professionale e/o migliorare e completare il lavoro esistente, ad esempio aggiungendo il supporto (appena accennato) per SQL server o ODBC.

Generalità

Il framework è costituito dal namespace DBUtility che incorpora essenzialmente quattro classi, ovvero:

Classe	Utilizzo
DBSession	Gestione transazioni con DB
DBConvert	Conversioni
DBSQLStrings	Tutto quello che riguarda la creazione di comandi SQL
DBImpExp	Import ed Export

Nel sorgente che viene reso disponibile il framework è dato insieme ad una applicazione contenitore (namespace Container) corrispondente al Form1. Questa applicazione di fatto serve solo per testare le routine del framework prima di incapsularle in una DLL e referenziarle nei "veri" progetti. Come esempio ho messo nella Form1 la chiamata alle funzioni di crittografia ma solo così, per metterci qualcosa.

DBSession

E' per così dire la base del tutto. Il suo costruttore riceve in input tre parametri che sono i seguenti:

- Stringa di Connessione
- Numero Massimo di Record da Estrarre
- Tipo di DB

Questi parametri possono essere anche assegnati direttamente attraverso le proprietà pubbliche:

- `strConnessioneCript`
- `nrMaxRecords`
- `strDBType`

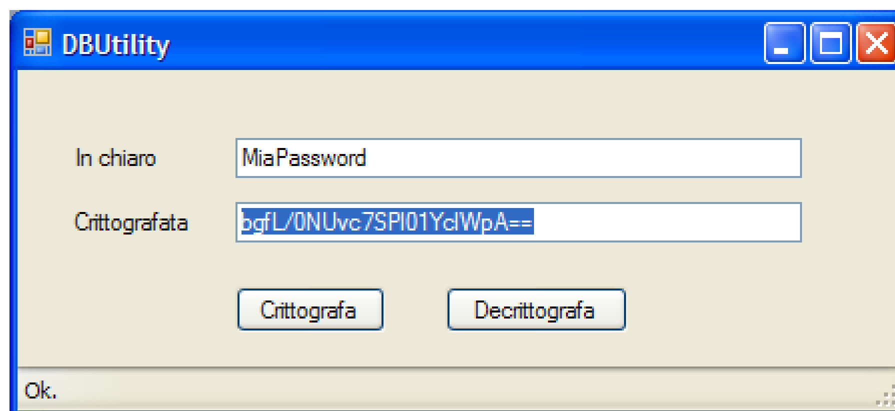
Un'osservazione sulla stringa di connessione. Tipicamente questa deve contenere lo UID e la PWD dell'utente di DB con cui si fa l'accesso. Anche se questa finisce nel caso di applicazioni ASPX nel Web.Config e quindi non può essere prelevato o visibile dall'utente ritengo comunque "brutto" avere una password in chiaro in un file di configurazione.

Per questo è stato previsto che la password sia scritta nel web config crittografata e che il framework provveda a de crittografarla per poi farne l'uso vero e proprio. Da notare che l'algoritmo di decrittazione scatta se nella stringa di connessione è presente la parola "Password" altrimenti vuol dire che non c'è niente da decrittare.

Per capirsi quindi la DBSession viene istanziata ad esempio così:

```
strConnection = "Data Source=MioOracleAlias;User
Id=MioUtente;Password=bgfL/0NUvc7SPl01YclWpA==;Integrated Security=no" ;
strDBType = "Oracle"
nrMaxRecords = Int32.MaxInt;
DBSession DBS = new DBSession(strConnection, nrMaxRecords, strDBType);
```

Dove ovviamente:



Notiamo il costruttore senza argomenti, pensato per utilizzo in applicazioni WEB, dove se ci sono i parametri in oggetto vengono prelevati dal Web.Config.

L'oggetto DBSession è ora istanziato ma per adesso non fa niente. Con questo però è possibile fare varie cose fra cui le principali sono queste:

- recuperare dei dati e trasferirli in un DataSet (che conserva i tipi) o DataView (tutti i dati rappresentati come stringhe);
- lanciare un comando SQL senza immagazzinare i dati da qualche parte gestita a programma (es. casi di INSERT,UPDATE, DELETE);

Per farlo occorre innanzitutto valorizzare un ulteriore parametro pubblico della istanza DBSession che è la `strSQLCommand`. Questa è solo una stringa per cui non è da questa che

il framework può riconoscere con che DB ha a che fare. Questa stringa sarà formattata secondo il particolare dialetto di SQL che implementa il nostro DB se non si vogliono degli errori, tutto qui.

Occorre poi invocare uno dei seguenti metodi pubblici della DBS:

Metodo	Utilizzo
LeggiTabellaDati	Utilizza la stringa SQL precedentemente definita come comando di SELECT e pone il result set contemporaneamente nelle proprietà TabellaDati (tipo DataSet) e VistaDati (Tipo DataView).
Esegui Comando	Utilizza la stringa SQL precedentemente definita come comando da lanciare al DB.

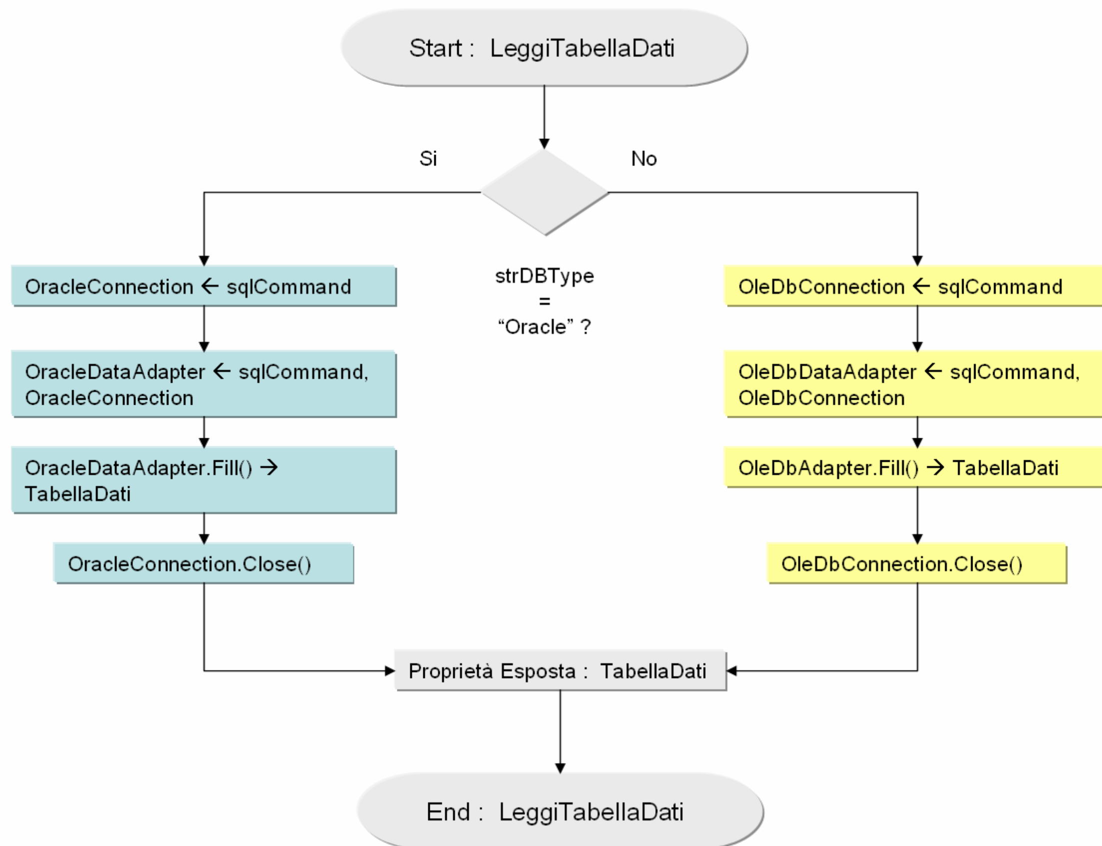
Per ottenere i risultati di cui sopra. Entrambi i comandi operano in modo simile e secondo lo standard ADO.NET ovvero:

- Istanziano un oggetto Connessione e lo inizializzano con la `strSQLCommand`;
- Istanziano rispettivamente un oggetto Adattatore o un Esecutore e lo inizializzano con la Connessione e `strSQLCommand`;
- Aprono la connessione con metodo Connection.Open();
- Utilizzano rispettivamente l'Adattatore o l'esecutore per riempire il DataSet (DataAdapter.Fill) o per lanciare il comando (Command.ExecuteNonQuery())

Ora, il Framework ADO.NET prevede un oggetto Connessione, Adattatore ed Esecutore per tipo di DB. Ovvero: non esistono gli oggetti Connection, Adapter e Commando ma esistono ad esempio gli oggetti

Database	Connessione	Adattatore	Esecutore
Oracle	OracleConnection	OracleDataAdapter	OracleCommand
SQL Server	SqlConnection	SqlDataAdapter	SqlCommand
Access	OleDbConnection	OleDbDataAdapter	OleDbCommand

Fortunatamente tutti gli adattatori ritornano lo stesso tipo DataSet. Per cui è pensabile procedere come segue (ipotesi semplificativa per il grafico, solo due alternative Oracle ed Access):



Fra gli oggetti accessibili della DBSession ce ne sono anche altri due che per così dire si muovono ad un livello superiore. Incapsulano cioè la comunissima operazione di verifica se un comando SQL di SELECT restituisce:

- nessuna riga
- una sola riga
- più di una riga

I metodi in questione (che restituiscono tipo Int) sono VerificaEsistenza e VerificaEsistenzaSQL che differiscono fra loro essenzialmente perché :

- nel primo caso si passa il campo (uno solo) da controllare (con la routine che sceglie il codice SQL a seconda del DBType)
- nel secondo caso si passa il comando SQL (o meglio la sua parte FROM e WHERE visto che la parte SELECT è sempre uguale) ovvero è a monte che questo (relativo al controllo di quanti campi si vuole) va formato correttamente a seconda del DB.

Tranne alcuni aspetti marginali la trattazione di DBSession è tutta qui. Da notare che questa classe è interamente multi DB con l'aggiunta di altri tipi di DB (purchè supportati da ADO.NET ovviamente) che è un'operazione abbastanza semplice.

DBConvert

Nelle operazioni coi DB sono fondamentali tutta una serie di funzioni che aiutano nella conversione dei tipi, specie abbastanza "robuste" da gestire le eccezioni come ad esempio i vari null.

In può cose abbastanza banali ma alla fine utilissime per non riscrivere mille volte la stessa cosa tipo conversioni true/false in "S","N" o 0/1 oppure funzioni stringa che prevengano i bug di SQL Injection tipo eliminazioni di apici o doppi apici.

A tutto questo provvede questa classe le cui funzioni mi sembrano abbastanza autoesplicative.

Solo alcune note, prima di chiudere questo paragrafo.

Le funzioni che operano la conversione stringa->data ipotizzano che la stringa sia in sempre in formato italiano (DD/MM/YYYY) : questa condizione andrà controllata chiaramente a livello di applicativo.

Fra le funzioni che operano sulle stringhe ho inserito anche la Distanza di Levenstein (`StringDistance`) di cui in apposito post sul sito.

In questa classe sono presenti anche le funzioni Cifra e Decifra che implementano la crittografia a chiave simmetrica AES. Il codice è stato prelevato dal sito di Microsoft e riportato qui con adattamenti minimi per cui rimando a quello e alla documentazione in rete per eventuali approfondimenti.

Notiamo infine che in alcune funzioni (es. AssegnaDefaults) è presente una caratteristica di questo framework : la gestione semplificata dei tipi. L'ambiente .NET mette a disposizione una serie molto vasta di tipi, ovvero ad esempio per gli interi int, Int16, Int32, Int64 etc... Stesso discorso per i vari DB, con i tipi che si chiamano in modo diverso a seconda del DB. L'idea è allora quella di ricondursi a tre grandi famiglie, ovvero:

- Date
- Stringhe
- Numeri

Contraddistinte dai descrittori "Date", "String" e "Numeric". In base a questi (e non ai reali tipi) poi prendere le decisioni. Si introducono certo delle semplificazioni ma in questo modo una gestione multi DB è almeno pensabile.

Anche per questo oggetto vale la "generalità" del precedente, anche se per un utilizzo un po' più ampio di dovrebbero aggiungere alcune funzioni quali l'eliminazione dei doppi apici (o meglio di caratteri generici) etc....

DBSQLStrings

Questo oggetto si occupa della costruzione di comandi SQL da valori di campi o array di valori. Siccome l'SQL è legato al DB la classe DBSQLString è quella in cui sarebbe più da lavorare per estenderne l'interoperabilità ad altri DB.

Un esempio chiarirà le cose : la funzione Conv riceve in ingresso una prima stringa che contiene il valore della varibile (es. "12", "12.3", "11/06/2008"...) ed una seconda che descrive il tipo semplificato ("Number", "Date"...). Restituisce una stringa che corrisponde al pezzo di SQL che va usato per gestirla.

Nel caso di date utilizza la funzione Oracle TO_DATE, che dà errore negli altri DB. Da tenere presente che comunque il grosso della complicazione è quasi sempre legato alle date!

Ancora: la GeneraCondizioniWhere è una funzione molto comoda che a partire da una matrice tipo questa:

Campo	Tipo	Valore Low	Valore High
Cod_articolo	String	null	null
Val_prezzo	Number	1000	2000
Dat_fine	Date	null	31/12/2008

Restituisce una stringa come questa:

WHERE (Val_prezzo>=1000)
AND (Val_prezzo<=2000)
AND (Dat_fine <= TO_DATE('31/12/2008','DD/MM/YYYY')

Utilizza la precedente Conv(.) e quindi anche questa va bene solo su Oracle di sicuro per il "maledetto" TO_DATE...

La "DecodificaRigaCfg" è una funzione che con l'SQL direttamente ha poco a che vedere. In pratica non fa altro che spezzare una stringa come questa:

"NomeParametro=ValoreParametro;"

in due stringhe contenenti appunto il nome del parametro ed il suo valore. In genere tutti i file di configurazione hanno questo formato e quindi con un algoritmo come questo (in "quasi C#"):

```
String NomeParametro;
String ValoreParametro;

for each Linea in Testo
{
    DecodificaRigaCfg (Linea, NomeParametro, ValoreParametro);
    switch (NomeParametro)
        {
        Case: "TipoDb"
                strDYType= ValoreParametro;
        Case: "NumeroRecords"
```

```
            strDYType= Convert.ToInt32(ValoreParametro);
        }
}
```

...si valorizzano tutte le variabili interne necessarie.

Operando con questa logica abbiamo alcune funzioni che a partire da una datarow e da una stringa di configurazione che dice quali sono i campi che costituiscono la una primary key riempiono un array list coi valori corrispondenti ovvero generano una rappresentazione lineare della medesima.

Per capirsi: se la Datarow è questa

Campo	**Tipo**	**Valore**
Prg_anno	Integer	2008
Prg_doc	Integer	143
Dat_inserimento	Date	11/06/2008
Txt_Note	Memo	"Con la seguente ordinanza il Sindaco stabilisce..."

...con PKString="Prg_anno;Prg_Doc":

- ArrayListConValori restituisce [2008, 143]
- StringaDaPK restituisce "2008/143"

Infine esistono tutta una serie di routines a tipo di ritorno string per la generazione di codice SQL quali GeneraSQLInsert, GeneraSQLUpdate etc... Queste si basano su un input come ad esempio questo:

```
string strDestTableName,
ArrayList alNomeCampoDest,
ArrayList alSqlCampoDest
```

A parte la presenza ovvia del nome della tabella di destinazione, abbiamo degli array contenenti rispettivamente l'elenco dei campi destinazione e l'elenco delle relative istruzioni SQL. Il funzionamento è del tutto simile alla GeneraCondizioniWhere.

Il fatto che queste funzioni partono non dai dati ma dalle loro stringhe SQL (es. non da una variabile di none Dat_inserimento, di tipo DateTime e di valore 11/06/2008 ma da una coppia di stringhe "Dat_inserimento" e "TO_DATE('11/06/2008','DD/MM/YYYY')") sposta a monte (ovvero alla costruzione dell'array alSqlCampoDest) la questione della dipendenza dal DB. Di per sé quindi sono DB indipendent.

<u>DBImpExp</u>

Questa classe è dedicata essenzialmente all'import al momento da Excel ed Access. Sono presenti due routines ovvero ImportFromExcel ed ImportFromAccess.

Entrambe utilizzano ADO.NET ed in particolare l'approccio "DBSession" di cui in precedenza. In pratica le due routine differiscono solo per la composizione della connection string a partire dai parametri "naturali" dei due casi ovvero:

- Nome File (completo di path) e Nome Foglio per il Caso XLS
- Nome File (completo di path) per il caso MDB

La definizione completa delle medesime è questa:

```
public static int ImportFromAccess(string MDBFileName, string MDBSql, ref string
MDBMessage, ref DataTable DataTableToFill)
public static int ImportFromExcel(string XLSFileName, string XLSSheetName, ref
string XLSMessage, ref DataTable DataTableToFill)
```

Il tipo di ritorno (int) è legato al codice di errore che è 0 se tutto è andato bene. Entrambe le routines restituiscono (attraverso il corrispondente parametro passato per indirizzo) un DataTable "`DataTableToFill`".

Questa tabella sarà poi utilizzabile per l'import vero e proprio che ad esempio può avvenire scorrendola ed utilizzando riga riga il controllo del caso (es. si decide di importare solo se nel DB destinazione quella riga non c'è) e il DBSession.Command del caso.

Notiamo che abbiamo inserito in questa sezione solo Excel ed Access perché le corrispondenti funzionalità per i database "veri" sono già in DBSession.LeggiTabellaDati. Stesso discorso vale per l'import dentro detti DB da eseguirsi a partire da un recordset costruito secondo le esigenze con DBSession.EseguiComando.