ADO.NET

Interfaccia applicativa di programmazione (API) per l'accesso a database

Anno 2022/2023

Indice generale

1 Applicazioni che accedono ai database

I database sono il cuore delle attività che richiedono la gestione di informazioni: la catalogazione dei libri di una biblioteca, la realizzazione di un sito di *e-commerce*, la gestione delle prenotazioni per i voli di una compagnia aerea, etc. I dati necessari allo svolgimento di queste attività devono essere resi disponibili agli utilizzatori: utenti, impiegati, amministratori, etc.

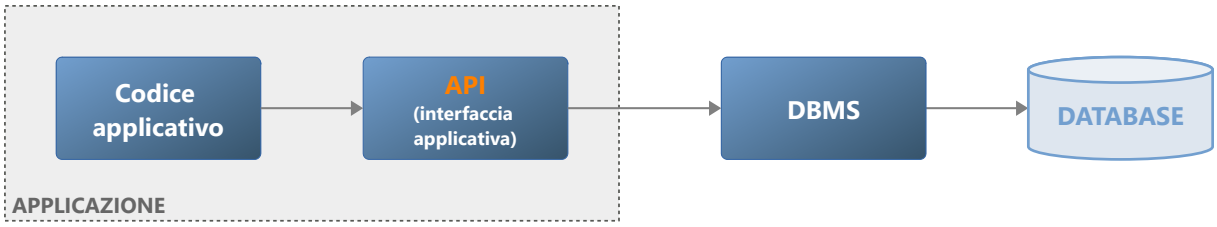
In generale non è possibile fornire agli utenti l'accesso diretto al database, per vari motivi:

1. Al di fuori degli amministratori, nessuno dovrebbe avere accesso alla struttura interna del database.
2. Gli utilizzatori (utenti del sito, impiegati, etc) non conoscono il funzionamento di un database relazionale, né conoscono il linguaggio SQL.
3. A categorie diverse di utilizzatori è necessario fornire prospettive diverse dei dati. Per l'utente di un sito musicale, il database è rappresentato da un catalogo di titoli, consultabile ma non modificabile. Un impiegato del sito dovrà avere la possibilità di aggiungere o eliminare titoli, verificare l'evasione degli ordini, etc.

In sostanza, la corretta implementazione di un database non garantisce la sua fruibilità da parte degli utenti finali. Tale obiettivo è realizzabile mediante delle applicazioni che si interpongono tra il database e gli utilizzatori.

1.1 Interfaccia tra applicazioni e DBMS

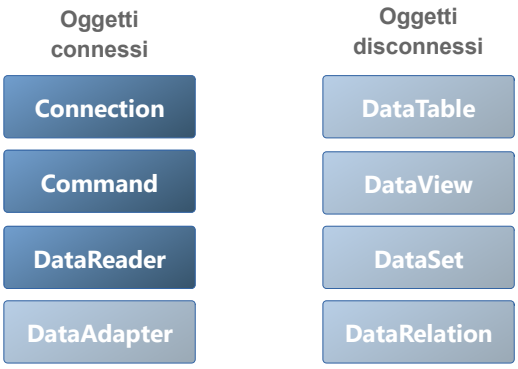
I programmi elaborano variabili semplici, vettori, collezioni, oggetti definiti dal programmatore, etc; i database memorizzano i dati in tabelle, secondo una struttura interna propria del DBMS, e manipolabili mediante istruzioni SQL. Per collegare i due mondi serve un'***interfaccia applicativa*** (API) tra il programma e il DBMS che consenta di recuperare le informazioni dal database ed effettuare modifiche su di esso. Questa *interfaccia applicativa* si avvarrà del linguaggio SQL, poiché è proprio questo il linguaggio utilizzato dai DBMS.



Esistono vari tipi di interfacce applicative. ADO.NET è una di queste.

2 Introduzione ad ADO.NET

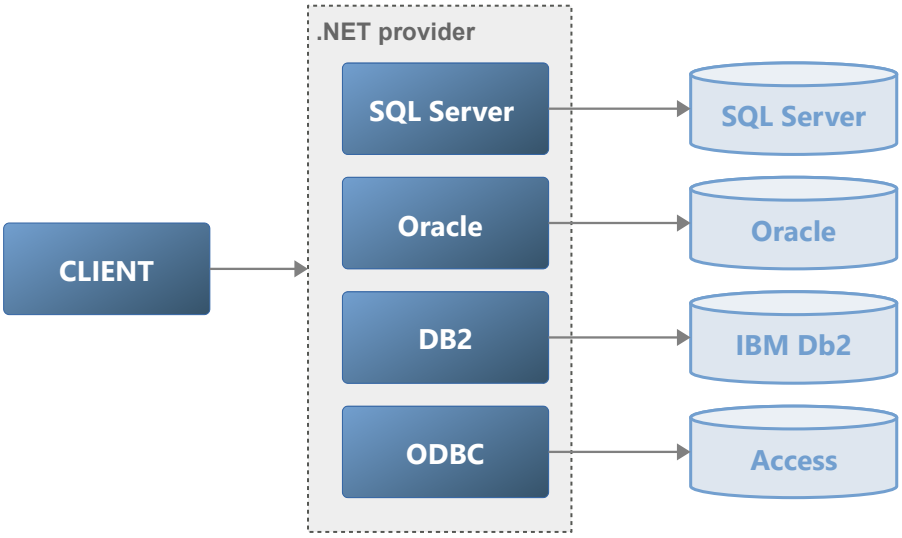
ADO.NET fornisce un insieme di oggetti che mettono l'applicazione in grado di dialogare con un DBMS. Gli oggetti sono stati progettati per collaborare tra loro e sono suddivisibili in due categorie: *oggetti connessi* e *oggetti disconnessi*.



I primi forniscono l'accesso al database; i secondi sono progettati per gestire i dati in memoria. Questo tutorial tratta soltanto gli oggetti connessi: *connection*, *command* e *datareader*.

2.1 ADO.NET Provider

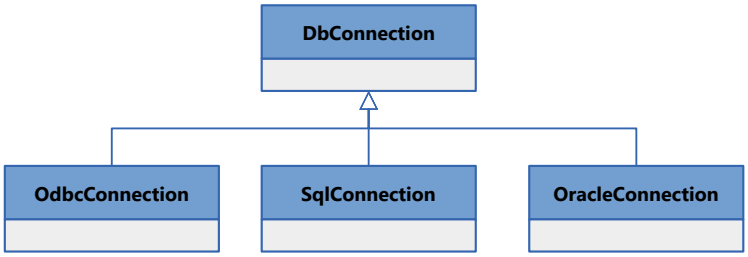
Ogni DBMS adotta un proprio modello di comunicazione con le applicazioni e dunque necessita di un insieme di oggetti specializzati: l'insieme di tali oggetti viene definito ***ADO.NET Provider***.



Ciascun *provider* fornisce una versione specializzata degli oggetti *connection*, *command* e *datareader*. Ad esempio, l'*SQL Server provider* definisce le classi `SqlConnection`, `SqlCommand`, etc. Il *provider* di Oracle definisce `OracleConnection`, `OracleCommand`, e via dicendo.

Le classi specializzate di ciascun *provider* derivano dalle classi base `DbConnection`, `DbCommand` e `DbDataReader`.

Il seguente schema mostra la gerarchia degli oggetti *connection*:



(Nel programma è opportuno utilizzare quanto più possibile le classi base, in modo da rendere il codice facilmente modificabile nel caso in cui fosse necessario adattarlo a diversi DBMS.)

2.1.1 ADO.NET Namespace

Le classi di ADO.NET sono definite all'interno di vari *namespace*:

1. Classi base e tipi comuni utilizzati da tutti i provider `System.Data` e `System.Data.Common`.
2. Provider SQL Server: `System.Data.SqlClient`
3. Provider Oracle: `System.Data.OracleClient`
4. Provider Odbc: `System.Data.Odbc`

Nel resto del tutorial utilizzerò prevalentemente l'*Odbc Provider* e i database **Access**. (Vedi APPENDICE I: istallare il software richiesto per l'installazione dei software necessari.)

3 Introduzione a *command*, *connection* e *datareader*

Segue una breve introduzione agli oggetti *connection*, *command* e *datareader*. Negli esempi si fa riferimento al database **Northwind.mdb** (nella versione italiana) e al *provider Odbc*.

3.1 *Connection*

Un oggetto *connection* fornisce un canale di comunicazione con il database. Il canale viene stabilito utilizzando la *stringa di connessione*, che definisce le informazioni necessarie per connettersi al database.

Supponi di avere il database **Northwind.mdb** memorizzato nel disco virtuale **U:**; la *stringa di connessione* richiesta dal *provider Odbc* è la seguente:

```
Driver={Microsoft Access Driver (*.mdb, *.accdb)} ; Dbq=U:/Northwind.mdb
```

È composta da due parti, separate dal carattere punto-e-virogola, che specificano il tipo del DBMS (chiave **Driver**) e il database (chiave **Dbq**). Il valore di **Driver** è sempre uguale, mentre **Dbq** richiede definisce il percorso completo del database.

Nota bene: in entrambi i casi, prima e dopo il carattere `=` non ci vogliono spazi.

3.1.1 Creazione di un oggetto *connection*

Il modo più immediato per fornire la *stringa di connessione* a un oggetto un *connection* è nell'istruzione di creazione:

```
using System.Data.Odbc;
using System.Data.Common;

const string cnStr = @"Driver={Microsoft Access Driver (*.mdb, *.accdb)};
                       Dbq=U:/Northwind.mdb";

DbConnection cn = new OdbcConnection(cnStr);
...
```

Alternativamente è possibile usare la proprietà `ConnectionString`:

```
DbConnection cn = new OdbcConnection();
cn.ConnectionString = cnStr;
...
```

3.1.2 Apertura della connessione

La creazione di un oggetto *connection* non produce alcun accesso al database. Prima di eseguire qualsiasi operazione, la connessione deve essere aperta con il metodo `Open()`:

```
DbConnection cn = new OdbcConnection(cnStr);
cn.Open();
...
```

Se `Open()` non è in grado di stabilire una connessione con il database, solleva un'eccezione. Supposto che nel sistema sia istallato il software necessario (APPENDICE I: istallare il software richiesto), i problemi più comuni sono due:

- C'è un errore nella parte iniziale della *stringa di connessione*: il testo dell'eccezione inizia per **ERROR [IM002]...**
- C'è un errore nel nome del database o nel suo percorso: il testo dell'eccezione inizia per **ERROR [HY000]...**

3.1.3 Chiusura della connessione

Una connessione dovrebbe restare aperta soltanto lo stretto necessario per eseguire le operazioni sul database, poi deve essere chiusa mediante `Close()`.

```
DbConnection cn = new OdbcConnection(cnStr);
cn.Open();

// ... una o più operazioni sul database

cn.Close();
```

3.2 Command

Qualsiasi operazione sul database è prodotta mediante l'esecuzione di un'istruzione SQL; a questo scopo è necessario usare un *command*. Per svolgere la propria funzione, un *command* ha bisogno di due cose: una connessione verso il database e l'istruzione SQL da eseguire.

```
DbConnection cn = new OdbcConnection(cnStr);

DbCommand cmd = cn.CreateCommand();
cmd.CommandText = "SELECT NomeSocietà, Paese FROM Clienti";
...
```

La prima istruzione crea il *command* a partire dall'oggetto *connection*. La seconda istruzione assegna alla proprietà `CommandText` la query SQL in grado di ottenere i nomi e le nazioni di tutti i clienti.

3.2.1 Eseguire un command

La creazione del *command* non determina la sua esecuzione. Innanzitutto è necessario aprire la connessione, quindi occorre eseguirlo; esistono tre metodi, `ExecuteReader()`, `ExecuteScalar()`, `ExecuteNonQuery()`, ognuno dei quali è adatto a un preciso scenario.

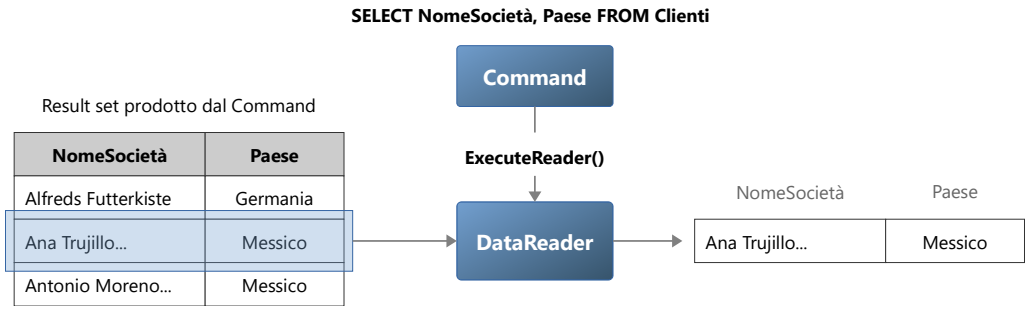
Il codice seguente mostra l'uso del metodo `ExecuteReader()`, che restituisce un *datareader*, attraverso il quale si può accedere all'insieme dei record (*result set*) restituiti dal DBMS:

```
DbConnection cn = new OdbcConnection(cnStr);
DbCommand cmd = cn.CreateCommand();
cmd.CommandText = "SELECT NomeSocietà, Paese FROM Clienti";
```

```
cn.Open();
DbDataReader dr = cmd.ExecuteReader();
// ... accesso ai dati
cn.Close();
```

3.3 Datareader

Il *datareader* rappresenta un cursore ***read-only forward-only***, cioè un oggetto che consente di scorrere una riga per volta le righe di un *result set* restituite dal DBMS.



I *datareader* non devono essere creati esplicitamente, poiché sono estituiti dal metodo `ExecuteReader()`.

3.3.1 Accesso ai dati del result set

L'accesso ai dati si ottiene facendo "avanzare" il *datareader* una riga per volta e indicizzando le colonne. Il metodo `Read()` fa avanzare il *datareader* alla riga successiva, restituendo `false` quando non ci sono più righe da leggere.

Nell'esempio seguente, di ogni riga viene visualizzata la colonna **NomeSocietà**:

```
DbConnection cn = new OdbcConnection(cnStr);

DbCommand cmd = cn.CreateCommand();
cmd.CommandText = "SELECT NomeSocietà, Paese FROM Clienti";
cn.Open();
DbDataReader dr = cmd.ExecuteReader();

while (dr.Read() == true) // leggi fintantoché ci sono righe da leggere
{
    string nomeSocietà = (string) dr["NomeSocietà"]; // non distingue maiuscole/minuscole
    Console.WriteLine(nomeSocietà);
}
cn.Close();
```

Nota bene:

- Il ciclo termina quando non ci sono più righe da leggere (`Read()` ritorna `false`).
- L'accesso alla colonna avviene mediante il suo nome, come se il *datareader* fosse un dizionario.
- Poiché l'indicizzatore del *datareader* è di tipo `object`, è necessario eseguire un cast al tipo `string`.

3.3.2 Accesso associativo, indicizzato, tipizzato alle colonne

Il *datareader* fornisce diversi modi di accesso alle colonne di una riga. Quello usato nell'esempio precedente è di tipo ***associativo non-tipizzato***: il nome della colonna viene usato come chiave per accedere al suo valore, restituito come tipo `object`. Il nome della colonna è da intendersi *case-insensitive* (senza distinzione tra maiuscole e minuscole).

Ciò detto, può essere utile usare l'operatore `nameof()` per evitare errori di scrittura:

```
string nomeSocietà = (string) dr["NomeSocietà"];
string nomeSocietà = (string) dr[nameof(nomesocietà)];
```

Un secondo modo è l'accesso ***indicizzato non-tipizzato***:, che prevede l'uso di un indice, come se il *datareader* fosse una lista di colonne:

```
while (dr.Read() == true)
{
    string nomeSocietà = (string) dr[0];
    Console.WriteLine(nomeSocietà);
}
```

Nota bene: l'ordine delle colonne è quello specificato nella clausola SELECT della query SQL.

Infine c'è l'accesso ***indicizzato tipizzato***: il valore della colonna viene ottenuto mediante un metodo `GetXXX()`, dove `XXX` è il tipo della colonna:

```
while (dr.Read() == true)
{
    string nomeSocietà = dr.GetString(0);
    clienti.Add(nomeSocietà);
}
```

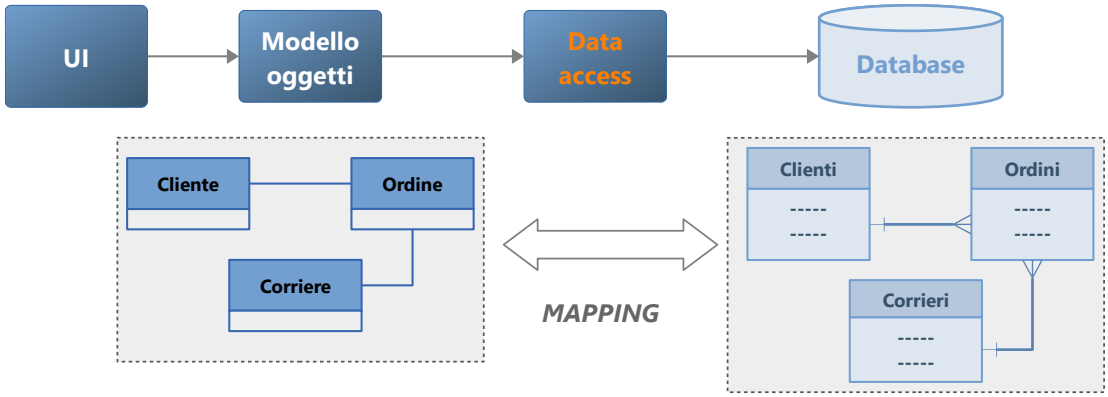
Qualunque sia il modo impiegato, se il nome o l'indice fanno riferimento a una colonna inesistente viene sollevata l'eccezione `IndexOutOfRangeException`.

4 Programmare con ADO.NET

Di seguito proporrò degli esempi concreti allo scopo di mostrare in azione le funzionalità precedentemente introdotte. Tutti gli esempi si riferiranno al database ***Northwind.mdb*** e al *provider Odbc*.

4.1 Struttura delle applicazioni di accesso ai database

Le applicazioni che si interfacciano con un database hanno spesso la seguente struttura:



È definito un *modello a oggetti* composto dalle classi che rappresentano le entità rilevanti per l'applicazione. Queste *mappano* (corrispondono) alle entità del modello *Entity-Relationship* del database e definiscono i campi corrispondenti alle colonne delle tabelle (non necessariamente tutte le colonne).

Poiché, però, i dati di queste entità sono memorizzati nel database, occorre uno "strato" (*layer*) di software che faccia da intermediario tra il *modello a oggetti* e il *modello relazionale* del database: il ***data access***.

Questa parte dell'applicazione svolge un duplice compito:

1. Mediante query SQL ottiene i dati dal database e crea le strutture dati appropriate basate sul *modello a oggetti*.
2. A partire dai dati del *modello a oggetti*, esegue le modifiche sul database.

Il *data access* consente al codice applicativo di elaborare oggetti e collezioni di oggetti senza dipendere da dove (il database) e come (il *modello relazionale*) i dati sono memorizzati.

4.1.1 Implementazione del data access

Negli esempi che seguono, il *data access* è rappresentato da un'unica classe: `Northwind`. Questa definisce internamente la *stringa di connessione* e gli oggetti *connection*, *command* e *datareader* necessari per svolgere il compiti richiesti.

Segue lo scheletro della classe, che definisce un costruttore all'interno del quale crea un oggetto *connection* per usarlo successivamente:

```
class Northwind
{
    const string cnStr = @"...";
    private DbConnection cn;

    public Northwind()
    {
        cn = new OdbcConnection(cnStr);
    }
    ...
}
```

4.1.2 Modello a oggetti

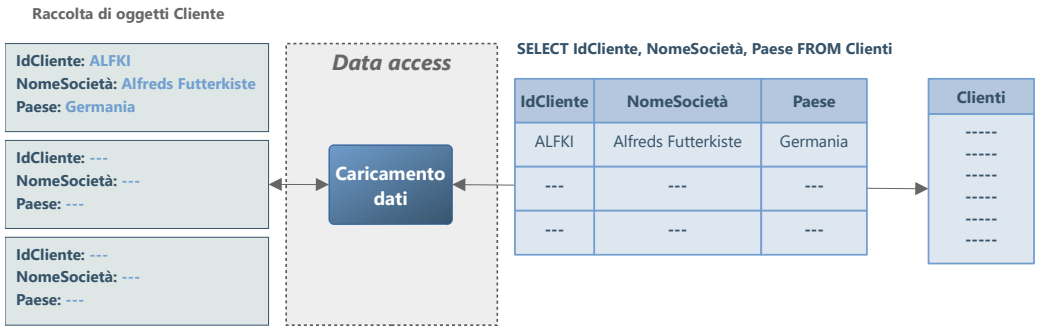
È composto dalle classi `Cliente`, `Corriere`, `Ordine`. Per ogni classe sono definiti soltanto i campi necessari alla corretta implementazione degli esempi.

```
public class Cliente
{
    public string IdCliente { get; set; }
    public string NomeSocietà { get; set; }
    public string Paese { get; set; }
}

public class Corriere
{
    public int IdCorriere { get; set; }
    public string NomeSocietà { get; set; }
    public string Telefono { get; set; }
}

public class Ordine
{
    public int IdOrdine { get; set; }
    public DateTime DataOrdine { get; set; }
    public string PaeseDestinatario { get; set; }
}
```

Nota bene: il *modello a oggetti* corrisponde a un sotto insieme del database, il quale contiene in realtà otto tabelle, ognuna delle quali ha molte colonne. È un approccio standard nel definire un *modello a oggetti*: si definiscono soltanto le classi e le proprietà necessarie a soddisfare i requisiti del problema.

5 Caricamento dati

È lo scenario più comune: viene interrogato il database mediante una query SQL allo scopo di ottenere i dati richiesti. Il *result set* viene elaborato per ottenere una raccolta di oggetti (o un oggetto soltanto).



5.1 Caricamento dei clienti

Nella classe `Northwind` implemento un metodo che restituisce la raccolta di tutti i clienti:

```
class Northwind
{
    ...
    public IEnumerable<Cliente> Clienti()
    {
        DbCommand cmd = cn.CreateCommand();
        cmd.CommandText = "SELECT IdCliente, NomeSocietà, Paese FROM Clienti";
        cn.Open();
        DbDataReader dr = cmd.ExecuteReader();
        while (dr.Read() == true)
        {
            var c = new Cliente
            {
                IdCliente= dr.GetString(0),
                NomeSocietà = dr.GetString(1),
                Paese = dr.GetString(2),
            };
            yield return c;
        }
        cn.Close();
    }
}
```

Nota bene: per accedere alle colonne della riga uso l'accesso *indicizzato-tipizzato* (3.3.2).

Segue un esempio d'uso del metodo:

```
Northwind db = new Northwind();
var clienti = db.Clienti();
foreach (var cliente in clienti)
{
    Console.WriteLine($"{cliente.NomeSocietà,-30} {cliente.Paese}");
}
```

5.2 Istruzioni SQL dinamiche e parametriche

Considera l'ipotesi di voler ottenere i clienti di una determinata nazione, inserita dall'utente. Il nome della nazione non può essere scritto nell'istruzione, altrimenti sarebbe sempre la stessa.

Si parla in questi casi di ***istruzioni SQL dinamiche*** o ***istruzioni SQL parametriche***.

5.3 Istruzioni SQL dinamiche

Un'istruzione SQL è una stringa; nulla vieta di comporre questa stringa usando una o più variabili, in modo che l'istruzione finale dipenda da parametri definiti esternamente.

Il seguente metodo restituisce l'elenco dei clienti residenti in una determinata nazione, ricevuta come parametro:

```
public IEnumerable<Cliente> ClientiPerNazione(string nazione)
{
    DbCommand cmd = cn.CreateCommand();

    cmd.CommandText = $"SELECT * FROM Clienti WHERE Paese = '{nazione}'";

    //... (il resto del metodo è uguale al precedente)
}
```

Ci sono alcune considerazioni da fare:

- Il simbolo ***** dopo la parola SELECT equivale a "tutte le colonne".
- Nel linguaggio SQL, le *costanti stringa* sono delimitate dall'apice singolo, non dalle virgolette.

5.4 Istruzioni SQL parametriche

Le *istruzione parametriche* forniscono una soluzione alternativa alle *istruzioni dinamiche*. Rispetto alle seconde sono più prestazionali e non soggette ai problemi di sicurezza. [1]

Un'*istruzione parametrica* implica l'uso di un *command parametrico*. L'istruzione utilizza dei *segnaposti* che, in fase di esecuzione, saranno sostituiti dai parametri del comando.

Segue una nuova versione del metodo che restituisce i clienti in base alla nazione di residenza:

```
public IEnumerable<Cliente> ClientiPerNazione(string nazione)
{
    DbCommand cmd = cn.CreateCommand();
    cmd.CommandText = $"SELECT * FROM Clienti WHERE Paese = ?";  // ? è un segnaposto

    var p = cmd.CreateParameter();      // crea un parametro
    p.Value = nazione;                  // assegna al parametro il valore della nazione
    cmd.Parameters.Add(p);              // aggiunge il parametro alla lista dei parametri

    //... (il resto del metodo è uguale al precedente)
}
```

1 Le istruzioni *SQL dinamiche* prestano il fianco a una tecnica di hacking chiama *SQL injection*.

Nota bene: nell'istruzione SQL, il simbolo **?** funge da *segnaposto* per la nazione. Il valore di quest'ultima viene assegnato al parametro `p` del *command*.

5.5 *Command* parametrici

Un *command parametrico* memorizza una collezione di parametri utilizzati per valorizzare i *segnaposti* di un'*istruzione SQL parametrica*. La creazione di un *command parametrico* segue il seguente pattern:

```
crea il command
imposta l'istruzione SQL (CommandText)
  crea il parametro
  imposta valore del parametro (eventualmente, imposta anche il tipo)
  aggiungi il parametro alla collezione del command
ripeti le istruzioni evidenziate per ogni parametro
```

5.5.1 Corrispondenza tra parametri e segnaposti

Per una corretta esecuzione ci dev'essere una corrispondenza tra i parametri del *command* e i *segnaposti* dell'istruzione SQL. Questa corrispondenza è posizionale: al primo *segnaposto* viene fatto corrispondere il primo parametro, al secondo *segnaposto* viene fatto corrispondere il secondo parametro, etc.

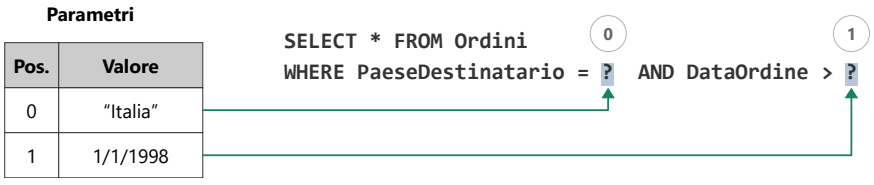
Supponi di voler caricare gli ordini destinati a una certa nazione ed effettuati dopo una certa data. È necessario eseguire l'istruzione SQL parametrica:

```
SELECT * FROM Ordini WHERE PaeseDestinatario = ? AND DataOrdine > ?
```

Occorre un *command parametrico* con due parametri, il primo per la nazione, il secondo per la data:

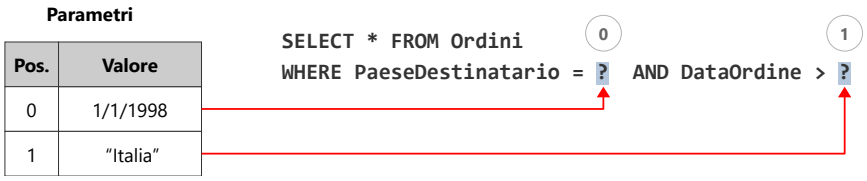
```
public IEnumerable<Ordine> OrdiniPerNazione(string nazione, DateTime data)
{
    DbCommand cmd = cn.CreateCommand();
    cmd.CommandText = $"...";

    var p = cmd.CreateParameter();          // nazione
    p.Value = nazione;
    cmd.Parameters.Add(p);

    p = cmd.CreateParameter();              // data
    p.Value = data;
    cmd.Parameters.Add(p);
    ...
}
```

Lo schema mostra la corrispondenza tra parametri e segnaposti:



Appare evidente che la posizione del parametro nella lista dei parametri del *command* è fondamentale. Ad esempio, il seguente codice produce una corrispondenza errata:

```
...
cmd.CommandText = $"...";

var p = cmd.CreateParameter();        // data
p.Value = data;
cmd.Parameters.Add(p);

p = cmd.CreateParameter();            // nazione
p.Value = nazione;
cmd.Parameters.Add(p);
...
```



L'esecuzione dell'istruzione produrrebbe un errore di esecuzione che indica una mancata corrispondenza tra i tipi. In casi più sfortunati, però, dove entrambi i parametri sono dello stesso tipo, l'istruzione SQL sarebbe eseguita correttamente, salvo restituire dati scorretti.

5.5.2 Tipi del parametri

Durante la creazione di un parametro è possibile specificare il suo tipo mediante l'enum `DbType`:

```
...
var p = cmd.CreateParameter();
p.DbType = DbType.String;
p.Value = nazione;
cmd.Parameters.Add(p);

p = cmd.CreateParameter();
p.DbType = DbType.DateTime;
p.Value = data;
cmd.Parameters.Add(p);
...
```

Si tratta di un'operazione utile in alcuni scenari; di base, comunque, è sufficiente assegnare il valore al parametro, sarà il *provider* a dedurre il tipo appropriato.

5.6 Caricamento di una singola entità in base alla chiave

È un'operazione molto comune, che produce una singola riga (oppure nessuna riga se la chiave specificata non esiste).

Il seguente metodo restituisce il corriere corrispondente alla chiave primaria specificata, o `null` se non esiste un corriere con quella chiave:

```
public Corriere Corriere(int idCorriere)
{
    DbCommand cmd = cn.CreateCommand();
    cmd.CommandText = $"SELECT * FROM Corrieri WHERE IdCorriere = ?";
    var p = cmd.CreateParameter();
    p.Value = idCorriere;
    cmd.Parameters.Add(p);    cn.Open();
    DbDataReader dr = cmd.ExecuteReader();

    Corriere corriere = null;
    if (dr.Read())                                  // è stata trovata una riga?
    {
        corriere = new Corriere()
        {
            IdCorriere = (int) dr["IdCorriere"],
            NomeSocietà = (string) dr["NomeSocietà"],
            Telefono = (string) dr["Telefono"],
        };
    }
    cn.Close();
    return corriere;
}
```

Nota bene: in questo non uso un ciclo `while` per far avanzare il *datareader*, poiché il risultato della query è zero righe o una riga.

5.7 Caricare un'entità correlata: proprietà di navigazione

Nel *modello a oggetti* usato finora (4.1.2) le classi non hanno alcun legame tra loro, diversamente dalle entità corrispondenti del modello *Entity-Relationship:*

Implementare le associazioni di un modello ER richiede del codice specifico. Di seguito mi limito a implementare l'associazione tra `Ordine` e `Corriere`. A questo scopo è necessario modificare la classe `Ordine`:

```
public class Ordine
{
    public int IdOrdine { get; set; }
```

```
    public DateTime DataOrdine { get; set; }
    public string PaeseDestinatario { get; set; }

    public int IdCorriere { get; set; }      // corrisponde alla colonna di chiave esterna
    public Corriere Corriere { get; set; }
}
```

La proprietà `Corriere` viene chiamata ***proprietà di navigazione***, poiché consente di accedere ("navigare") da un'entità all'entità associata.

Resta il fatto che, dopo aver caricato un ordine, la proprietà `Corriere` contiene `null`. Di seguito mostro come caricare un ordine e il corriere associato:

```
public Ordine OrdineECorriereAssociato(int idOrdine)
{
    DbCommand cmd = cn.CreateCommand();
    cmd.CommandText = $"SELECT * FROM Ordini WHERE IdOrdine = ?";
    var p = cmd.CreateParameter();
    p.Value = idOrdine;
    cmd.Parameters.Add(p);

    cn.Open();
    DbDataReader dr = cmd.ExecuteReader();
    Ordine ordine = null;

    if (dr.Read())
    {
        ordine = new Ordine()
        {
            IdOrdine = (int)dr["IdOrdine"],
            DataOrdine = (DateTime)dr["DataOrdine"],
            PaeseDestinatario = (string)dr["PaeseDestinatario"],
            IdCorriere = (int) dr["Corriere"],  // la chiave esterna ha un nome diverso
        };
        cn.Close();
        ordine.Corriere = Corriere(ordine.IdCorriere);
    }
    cn.Close();
    return ordine;
}
```

Alcune osservazioni:

- Alla proprietà `IdCorriere` viene assegnata la colonna **Corriere**. È l'unico caso del database **Northwind** in cui la colonna di chiave esterna di una tabella non ha lo stesso nome della chiave primaria della tabella associata.
- Per caricare il corriere di un ordine ho usato il metodo realizzato in precedenza.

Segue un esempio d'uso:

```
Northwind db = new Northwind();

var o = db.OrdineECorriereAssociato(10248);

Console.WriteLine($"{o.DataOrdine:d} {o.PaeseDestinatario} {o.Corriere.NomeSocietà}");
```

che produce:

```
04/07/1996 Francia Federal Shipping
```

6 Modifica dei dati

La modifica dei dati avviene eseguendo operazioni di inserimento, aggiornamento, eliminazione di uno o più record del database, effettuate con le istruzioni SQL INSERT, UPDATE e DELETE.

Queste istruzioni non sono interrogazioni e dunque non restituiscono un *result set*; per questo motivo vengono eseguite con il metodo `ExecuteNonQuery()` del *command*. Il metodo restituisce il numero di righe coinvolte dall'operazione, valore che può essere utilizzato per avere una conferma sulla sua corretta esecuzione.

6.1 Inserimento di un nuovo corriere

Occorre eseguire la seguente *istruzione SQL parametrica*:

```
INSERT INTO Corrieri (NomeSocietà, Telefono) VALUES (?, ?)
```

Nota bene: non ho specificato la colonna **IdCorriere**, poiché si tratta di una *colonna identità*, il cui valore viene generato automaticamente dal DBMS. (Access non impedisce che venga inserita dall'esterno anche la *colonna identità*, ma non è una pratica consigliabile.)

L'implementazione dell'operazione è realizzata dal metodo `InserisciCorriere()`, che riceve il corriere da inserire:

```
public void InserisciCorriere(Corriere c)
{
    DbCommand cmd = cn.CreateCommand();
    cmd.CommandText = " INSERT INTO Corrieri (NomeSocietà, Telefono) VALUES (?, ?)";

    DbParameter paNome = cmd.CreateParameter();
    paNome.Value = c.NomeSocietà;
    cmd.Parameters.Add(paNome);

    DbParameter paTel = cmd.CreateParameter();
    paTel.Value = c.Telefono;
    cmd.Parameters.Add(paTel);

    cn.Open();

    cmd.ExecuteNonQuery();

    cn.Close();
}
```

Nota bene: non uso il valore restituito dal metodo `ExecuteNonQuery()`, dato che, o l'inserimento ha successo (viene inserita una riga nella tabella **Corrieri**) o fallisce, nel qual caso viene sollevata un'eccezione.

Segue un esempio d'uso:

```
var db = new Northwind();

var corriere = new Corriere()
{
```

```
    NomeSocietà = "Speedy Express",
    Telefono = "12345678"
};

db.InserisciCorriere(corriere);
```

6.2 Recuperare il valore della colonna identità

Esistono scenari nei quali si desidera utilizzare immediatamente l'entità appena inserita per future operazioni. In questi casi sorge il problema di recuperare il valore della chiave primaria *identità*, poiché è generato dal DBMS e dunque è sconosciuto all'applicazione.

Con Access è possibile farlo eseguendo la query seguente subito dopo aver inserito un record e utilizzando la stessa connessione:

```
SELECT @@identity
```

Il recupero dell'ultimo valore *identità* generato è un'operazione di utilità generale, dunque è opportuno implementarla in un metodo a parte.

```
public int UltimoValoreIdentità(DbConnection cn)
{
    DbCommand cmd = cn.CreateCommand();
    cmd.CommandText = "SELECT @@identity";
    bool cnAperta = cn.State == ConnectionState.Open;

    if (cnAperta == false)
    {
        cn.Open();
    }

    int identità = (int)cmd.ExecuteScalar();

    if (cnAperta == false)
    {
        cn.Close();
    }
    return identità;
}
```

Gestendo la proprietà `State` dell'oggetto *connection* faccio sì che la connessione venga aperta soltanto se era chiusa e che, a operazione conclusa, sia lasciata nello stesso stato in cui era prima di iniziare. Lo scopo del codice è quello di utilizzare correttamente una connessione esistente.

6.2.1 Recupero della colonna IdCorriere

Occorre modificare il metodo `InserisciCorriere()`:

```
public void InserisciCorriere(Corriere c)
{
    DbCommand cmd = cn.CreateCommand();
```

```
    cmd.CommandText = " INSERT INTO Corrieri (NomeSocietà, Telefono) VALUES (?, ?)";

    // ... creazione del command parametrico

    cn.Open();
    cmd.ExecuteNonQuery();
    c.IdCorriere = UltimoValoreIdentità(cn);
    cn.Close();
}
```

6.3 Eliminazione di un corriere

Occorre eseguire la seguente istruzione SQL:

```
DELETE FROM Corrieri WHERE IdCorriere = ?
```

Il metodo `EliminaCorriere()` riceve come argomento la chiave primaria del corriere da eliminare:

```
public bool EliminaCorriere(int idCorriere)
{
    DbCommand cmd = cn.CreateCommand();
    cmd.CommandText = "DELETE FROM Corrieri WHERE IdCorriere = ?";

    DbParameter paId = cmd.CreateParameter();
    paId.Value = idCorriere;
    cmd.Parameters.Add(paId);

    cn.Open();
    int numRighe = cmd.ExecuteNonQuery();
    cn.Close();

    return numRighe == 1;
}
```

In questo caso verifico il valore restituito da `ExecuteNonQuery()` per stabilire che sia stata effettivamente eliminata una riga; in caso contrario il metodo restituisce `false`.

Qui non prendo in considerazione la possibilità che l'operazione possa fallire con un'eccezione, eventualità che negli scenari scenari realistici non può essere ignorata. Nell'esempio, l'operazione è destinata a fallire se si tenta di eliminare un corriere al quale sono associati degli ordini, poiché nel database è presente un vincolo di integrità referenziale tra **Corrieri** e **Ordini** che impedisce questo tipo di modifica.

Segue un esempio nel quale un corriere viene inserito e immediatamente eliminato:

```
var db = new Northwind();
var corriere = new Corriere()
{
    NomeSocietà = "Speedy Express",
    Telefono = "12345678"
};
```

```
db.InserisciCorriere(corriere);

var eliminato = db.EliminaCorriere(corriere.IdCorriere); // -> true
```

6.4 Aggiornamento dati

L'aggiornamento è un'operazione che si volge in due fasi. Nella prima fase i dati vengono caricati in memoria, ad esempio per visualizzarli e consentire all'utente di modificarli. Dopodiché, i dati modificati vengono inviati al database, sovrascrivendo quelli originali.

A dirigere questo scambio tra l'applicazione e il database è la chiave primaria del record da modificare.

6.4.1 Modifica di un corriere

Occorre eseguire la seguente istruzione SQL:

```
UPDATE Corrieri SET
    NomeSocietà = ?,
    Telefono    = ?
WHERE IdCorriere = ?
```

Nota bene:

- L'operazione è diretta a uno specifico corriere, identificato dalla sua chiave primaria.
- L'operazione richiede tutti i dati del corriere, indipendentemente dal fatto che soltanto alcuni di essi siano diversi da quelli memorizzati nel database.

(Il secondo punto non rappresenta un vincolo dell'istruzione UPDATE, che può essere usata anche per modificare una singola colonna di un record.)

L'operazione è realizzata dal metodo `ModificaCorriere()`, che riceve come argomento il corriere da modificare.

```
public bool ModificaCorriere(Corriere c)
{
    DbCommand cmd = cn.CreateCommand();
    cmd.CommandText = "UPDATE Corrieri SET NomeSocietà = ?, Telefono = ?
                       WHERE IdCorriere = ?";

    DbParameter paNome = cmd.CreateParameter();
    paNome.Value = c.NomeSocietà;
    cmd.Parameters.Add(paNome);

    DbParameter paTel = cmd.CreateParameter();
    paTel.Value = c.Telefono;
    cmd.Parameters.Add(paTel);

    DbParameter paId = cmd.CreateParameter();
    paId.Value = c.IdCorriere;
    cmd.Parameters.Add(paId);

    cn.Open();
```

```
    int numRighe = cmd.ExecuteNonQuery();
    cn.Close();

    return numRighe == 1;
}
```

Segue un esempio d'uso, nel quale un corriere viene inserito e immediatamente modificato:

```
var db = new Northwind();
var corriere = new Corriere()
{
    NomeSocietà = "Speedy Express",
    Telefono = "12345678"
};

db.InserisciCorriere(corriere);
corriere.Telefono = "0575-741867";
db.ModificaCorriere(corriere);
```

APPENDICE I: istallare il software richiesto

Per interfacciarsi con un database Access, nel sistema deve essere innanzitutto istallato **l'Access Database Engine**. Questo è istallato automaticamente insieme a Microsoft Access, ma può essere istallato anche indipendentemente.

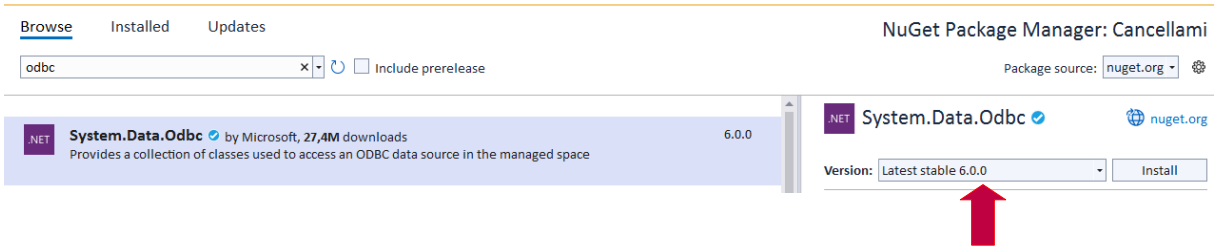
Esiste in versioni a 32 e 64 bit (soltanto una delle due può essere istallata) ed è disponibile sul G-DRIVE: (https://drive.google.com/drive/folders/1eGwXWZ-3Tr9EXXq4HH49r3QhiPOXv9MS?usp=sharing)

(Si può scaricare all'indirizzo: https://www.microsoft.com/en-us/download/details.aspx?id=54920.)

6.5 Istallare il *provider* ODBC

L'*Odbc Provider* deve essere istallato per ogni progetto che ne fa uso. Da Visual Studio lo si può fare dal **Nuget Package Manager**, oppure dalla **Package Manager Console**.

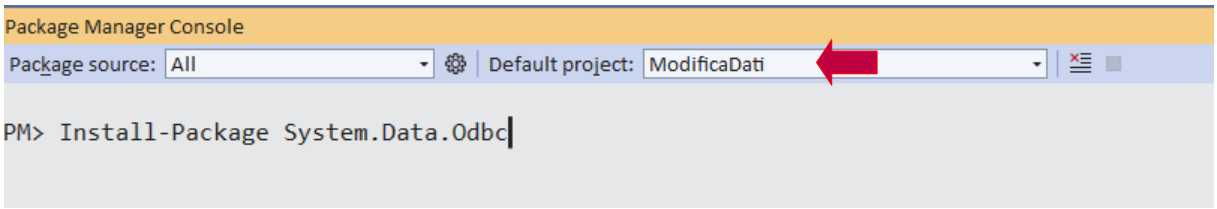
Esegui il **Nuget Package Manager** (*Click destro sul progetto* | *Manage NuGet Packages…*) e, nella scheda **Browse**, digita *Odbc* nella casella di ricerca:



Prima di istallare, verifica che la versione del pacchetto (freccia a destra) non sia superiore alla versione del .NET istallato nel computer. Se è questo il caso clicca sull'elenco a discesa e seleziona una versione precedente.

6.6 Uso della Package Manager Console

Esegui la *console* mediante **Tools | NuGet Package Manager | Package Manager Console** e inserisci il comando per istallare il *provider*:



Se la *solution* ha più progetti, verifica che sia selezionato il progetto giusto in *default project*.

Per istallare una specifica versione, usa l'opzione **-version** e specifica il numero di versione:

```
Install-Package System.Data.Odbc -version 6.0.0
```