ADO.NET

Introduzione all'accesso ai database mediante ADO.NET

Progetto di Informatica classe 5^{a}

Anno 2013/2014

Ambiente: .NET 4.0+/C# 4.0+

Indice generale

1 Introduzione: applicazioni che accedono ai database

I database sono al centro delle attività che richiedono la gestione di informazioni: la catalogazione dei libri di una biblioteca, l'organizzazione di un sito di e-commerce, la gestione delle prenotazioni per i voli di una compagnia aerea, etc. I dati necessari allo svolgimento di tali attività devono essere resi disponibili agli utilizzatori: utenti, impiegati, dirigenti, etc.

Naturalmente, non è possibile fornire l'accesso diretto al DBMS che gestisce il database; il motivo più ovvio è dato dal fatto che il DBMS gira su un computer distinto dai computer degli utilizzatori. Ma ci sono altri motivi:

1. per ragioni di sicurezza: al di fuori degli amministratori nessuno dovrebbe avere accesso alla struttura interna del database;
2. gli utilizzatori (utenti del sito, impiegati, dirigenti) non conoscono il funzionamento di un database relazionale, né conoscono il linguaggio SQL;
3. ad ogni tipo di utilizzatore è necessario fornire una prospettiva diversa 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 memorizzazione dei dati in un database non garantisce di per sé la loro fruibilità. Un simile obiettivo è realizzabile attraverso applicazioni che si interpongono tra il database e gli utilizzatori e che forniscono una rappresentazione adeguata al tipo di operazioni che essi devono compiere. In tal senso, esse vengono definite *applicazioni client* del database, o semplicemente *client.*

1.1 Interfaccia tra applicazioni e DBMS

Nella comunicazione tra una applicazione client e il DBMS vi sono due questioni da affrontare:

1. il modo in cui viene gestito lo scambio "fisico" dei dati tra applicazione e DBMS;
2. le funzionalità usate dall'applicazione per manipolare i dati memorizzati del database.

Il primo punto riguarda i protocolli e l'interfaccia di basso livello utilizzati da client e DBMS; dalla prospettiva che stiamo considerando, questo punto non è importante. Il secondo aspetto è invece fondamentale per la programmazione.

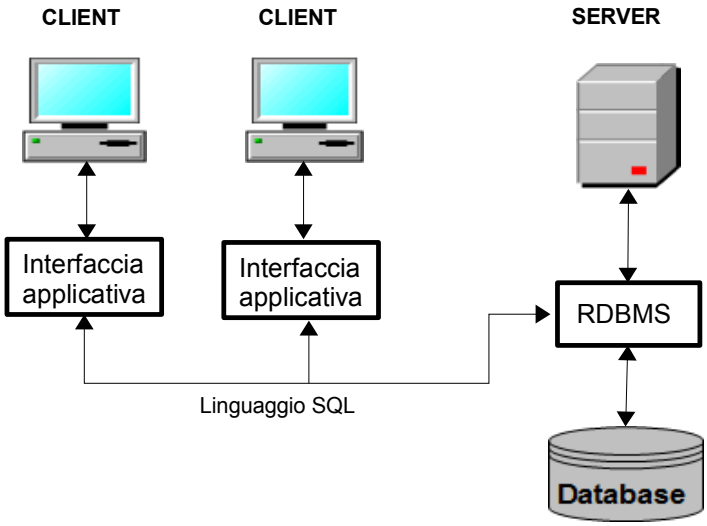
Un programma è in grado di elaborare dati memorizzati in variabili semplici, vettori, collezioni dinamiche, oggetti definiti dal programmatore, ecc. D'altra parte, i dati del database sono memorizzati in tabelle, secondo una struttura interna propria del DBMS, e manipolabili mediante istruzioni SQL[1]. Ciò che serve è una *interfaccia applicativa* tra il programma client e il DBMS che consenta di:

1. recuperare le informazioni dal database e di accedere ad esse;

1 Ovviamente qui stiamo parlando di database relazionali.

2. inviare al database i dati memorizzati nelle variabili del programma; in generale, consentire all'applicazione di effettuare modifiche sul database.

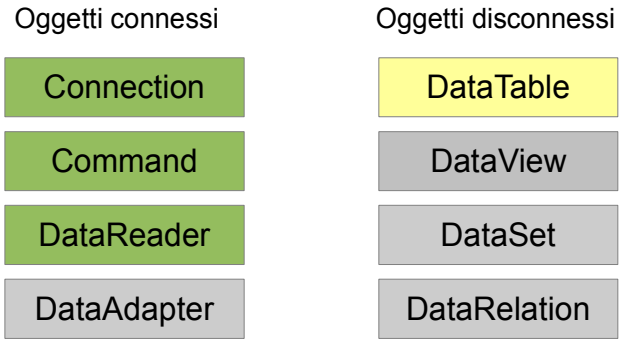
Tale interfaccia applicativa si avvarrà normalmente del linguaggio SQL, poiché è proprio questo il linguaggio utilizzato dal DBMS.



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

2 Introduzione ad ADO.NET

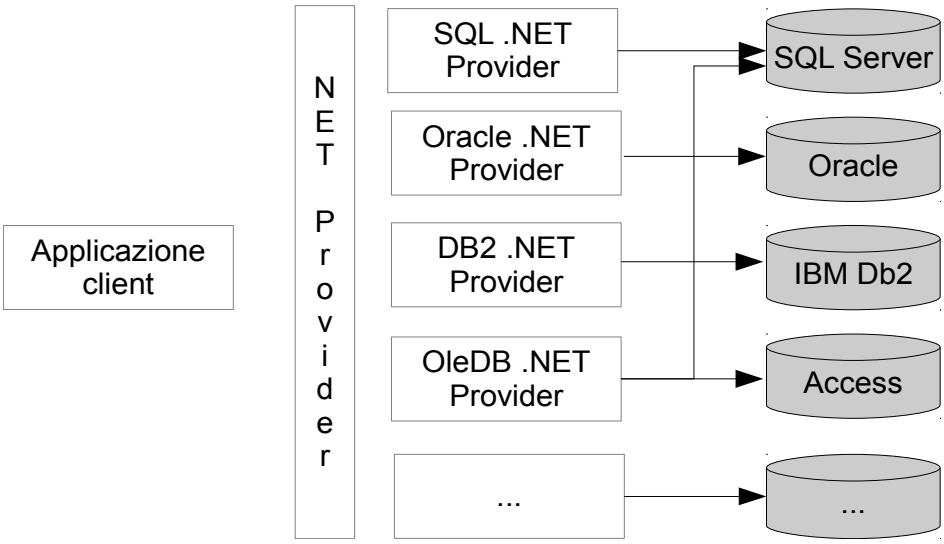
ADO.NET comprende un insieme di oggetti in grado di "dialogare" con un DBMS. Gli oggetti sono stati progettati per collaborare tra loro e sono suddivisibili in due categorie: gli *oggetti connessi* e gli *oggetti disconnessi*.



I primi forniscono l'accesso al database; i secondi sono progettati per gestire i dati in memoria, indipendentemente dalla loro provenienza. Questo tutorial tratta soltanto alcuni di essi, che nello schema appaiono in verde e in giallo.

2.1 ADO.NET Provider

Poiché ogni DBMS adotta un proprio modello di comunicazione con l'esterno, necessita di un insieme di oggetti connessi appositamente progettati per comunicare con esso: l'insieme di tali oggetti viene definito ***ADO.NET Provider***.



Ciascun provider implementa una versione specializzata delle classi base **DbConnection**, **DbCommand**, **DbDataReader**, etc. Ad esempio, il provider di SQL Server definisce le classi **SqlConnection**, **SqlCommand**, etc; il provider di Oracle definisce **OracleConnection**, **OracleCommand**, e via dicendo.

Esistono comunque dei provider particolari, i quali consentono di comunicare con più tipi di DBMS. Tra questi vi è l'OleDb Provider, che in grado di interfacciarsi con database Access, SQL Server, e altri.

2.1.1 ADO.NET *Namespace*

Le classi di ADO.NET sono definite all'interno di svariati *namespace.* Gli oggetti disconnessi sono definiti in **System.Data**. Gli oggetti connessi sono definiti nel namespace specifico del provider utilizzato. Ecco un elenco di namespace per i provider più comuni:

1. SQL Server: **System.Data.SqlClient**
2. Oracle: **System.Data.OracleClient**
3. OleDB: **System.Data.OleDb**

Le classi base degli oggetti connessi sono definite in **System.Data.Common**.

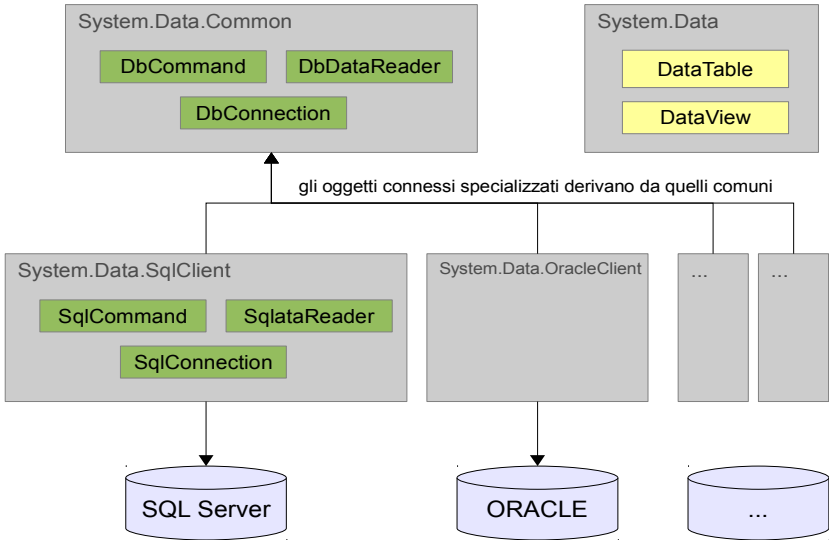
2.2 Architettura di ADO.NET

Gli oggetti connessi sono specializzati per comunicare con uno specifico DBMS e dunque non possono essere utilizzati con un DBMS diverso[2]. Ad esempio, SqlCommand e OracleCommand hanno lo stesso ruolo e caratteristiche identiche, ma uno può interfacciarsi soltanto con SQL Server, l'altro solo con Oracle.

In alcuni scenari ciò rappresenta un problema, poiché si desidera scrivere del codice applicativo in grado di adattarsi a DBMS diversi. Per questo motivo è opportuno utilizzare quanto più possibile le classi base, DbConnection, DbCommand e DbDataReader, le quali pur con quale limitazione, rendono possibile scrivere codice in grado di interfacciarsi con qualsiasi DBMS.

Di seguito è mostrato un schema dell'interfaccia applicativa di ADO.NET. Occorre sottolineare alcune cose:

1. Lo schema mostra vari provider, ma nella maggior parte delle applicazioni è necessario utilizzarne uno soltanto.
2. E' consigliabile, ma non obbligatorio, usare gli oggetti comuni.
3. Gli oggetti disconnessi sono utili, ma non essenziali, per la gestione dei dati nel codice applicativo.



2 Fanno parziale eccezione gli oggetti dell'OleDbProvider

3 Introduzione agli oggetti connessi

Segue una breve panoramica sulle funzionalità degli oggetti Connection, Command e DataReader. Negli esempi viene utilizzato il database Northwind.mdb e l'OleDb Provider.

3.1 Connection

Un oggetto Connection fornisce un canale di comunicazione con il database. E' l'unico oggetto strettamente necessario in qualsiasi applicazione ADO.NET, poiché l'accesso al database deve sempre avvenire attraverso un oggetto Connection.

3.1.1 Stringa di connessione

La proprietà fondamentale di un Connection è la ***stringa di connessione***. Questa definisce le informazioni sulla sorgente dati e consente di stabilire alcune caratteristiche applicate alla connessione.

Sorgente dati

Per "Informazioni sulla sorgente dati" si intendono le informazioni necessarie per accedere al database. Queste possono essere diverse in base al DBMS e al provider utilizzato.

Per accedere ad un database Access le informazioni essenziali sono:

1. l'OleDb provider utilizzato: parametro "Provider";
2. il percorso del file contenente il database: parametro "Data Source".

Segue una stringa di connessione per l'accesso al database Northwind.mdb situato nella cartella "D:\database":

```
Provider=Microsoft.ACE.OLEDB.12.0; Data Source=C:\Database\Northwind.mdb
```

Provider OLEDB

Per utilizzare il provider "Microsoft.ACE.OLEDB.12.0" è necessario aver prima installato il file AccessDataBaseEngine.exe, a 32 o 64 bit in base al SO installato nel computer.

3.1.2 Creazione di un oggetto Connection

Durante la creazione di un Connection è possibile specificare la stringa di connessione.

```
using System.Data;
using System.Data.Common;
using System.Data.OleDb

...
const string cnStr = @"Provider=Microsoft.ACE.OLEDB.12.0;
                       Data Source=D:\Database\VentoDelNord.mdb";
static void Main(string[] args)
{
    DbConnection cn = new OleDbConnection(cnStr);
```

```
    ...
}
```

Alternativamente, è possibile impostare la stringa di connessione mediante la proprietà **ConnectionString**:

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

3.1.3 Apertura della connessione

La creazione di un oggetto Connection non implica alcuna comunicazione con il database. Prima di eseguire qualsiasi operazione, la connessione deve essere aperta. A questo scopo esiste il metodo **Open()**:

```
OleDbConnection cn = new OleDbConnection(cnStr);
cn.Open();
```

Se non è in grado di stabilire una connessione con il database, Open() solleva un'eccezione.

3.1.4 Chiusura della connessione

Come buona regola, una connessione dovrebbe restare aperta soltanto lo stretto necessario per eseguire le operazioni richieste, poi deve essere chiusa mediante **Close()**.

```
DbConnection cn = new OleDbConnection(cnStr);
cn.Open();
... // qui vengono eseguite una o più operazioni sul database
cn.Close();
```

3.2 Command

Qualsiasi operazione sul database implica l'esecuzione di un'istruzione SQL; a questo scopo esistono i Command.

Per svolgere la propria funzione, un oggetto Command ha sostanzialmente bisogno di due cose: un oggetto Connection e l'istruzione SQL da eseguire.

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

DbCommand cmd = cn.CreateCommand();
cmd.CommandText = " SELECT Count(*) FROM Clienti WHERE Paese = 'Italia'";
```

Nota bene: la prima istruzione crea il Command attraverso il metodo **CreateCommand()** dell'oggetto Connection. La seconda imposta l'istruzione SQL da eseguire.

3.2.1 Esecuzione di un Command

La creazione dell'oggetto Command non determina la sua esecuzione. Innanzitutto è necessario aprire la connessione associata al Command, quindi occorre eseguirlo invocando uno dei tre metodi progettati a questo scopo, **ExecuteScalar()**, **ExecuteNonQuery()** e **ExecuteReader()**.

Il codice seguente mostra l'uso del metodo ExecuteScalar(); questo produce un singolo

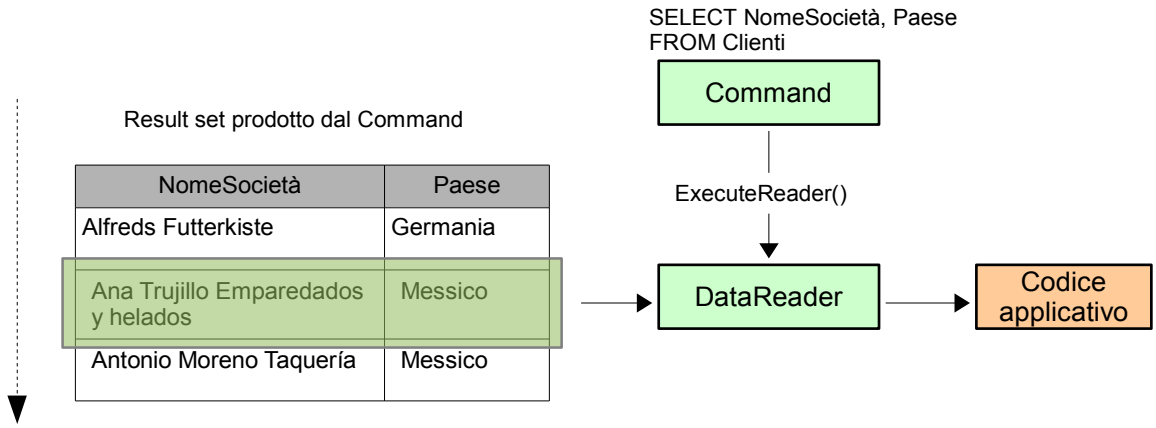
valore e dunque è adatto per l'esecuzione di query scalari:

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

DbCommand cmd = cn.CreateCommand();
cmd.CommandText = "SELECT Count(*) FROM Clienti WHERE Paese = 'Italia'";

cn.Open();
int numClientiItaliani = (int) cmd.ExecuteScalar();
cn.Close();
```

ExecuteScalar() produce un valore di tipo object e pertanto è necessario convertirlo al tipo appropriato.

3.3 DataReader

Il DataReader rappresenta un cursore ***read-only forward-only*** e cioè un oggetto che consente di scorrere una riga per volta tutte le righe di un result set.



Gli oggetti DataReader non devono essere istanziati esplicitamente; sono infatti creati come risultato del metodo **ExecuteReader()**. Il metodo esegue una query sul database e associa l'oggetto DataReader al result set prodotto.

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

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

cn.Open();
DbDataReader dr = cmd.ExecuteReader();
...   // qui viene usato il DdataReader
```

3.3.1 Accesso alle righe e alle colonne del result set

L'accesso ai dati si ottiene facendo "avanzare" il DataReader una riga per volta e accedendo alle colonne che si desidera gestire.

Il metodo che fa avanzare il DataReafer è **Read()**. Questo torna false quando non ci sono più righe da leggere.

Nell'esempio che segue, da ogni riga viene estratto il campo NomeSocietà e memorizzato in un oggetto List<string>:

```
...
cn.Open();
DbDataReader dr = cmd.ExecuteReader();
List<string> clienti = new List<string>();
while (dr.Read() == true)
{
    string nomeSocietà = dr["NomeSocietà"].ToString();
    clienti.Add(nomeSocietà);
}
cn.Close();
```

Il ciclo viene eseguito fintantoché ci sono righe nel result set. Per ogni riga, l'accesso ad una colonna viene eseguito specificandone il nome come chiave del DataReader; ciò produce un valore object che deve essere convertito in stringa.

Infine viene chiusa la connessione, operazione che provoca l'automatica chiusura del DataReader.

3.3.2 Accesso associativo, indicizzato e tipizzato alle colonne

Il DataReader fornisce diversi meccanismi di accesso alle colonne di una riga. Quello usato nell'esempio è di tipo ***associato non-tipizzato***. Il nome della colonna viene usato come chiave per accedere al suo valore, il quale viene ritornato sotto forma di object.

Un secondo meccanismo è quello ***indicizzato non-tipizzato***:

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

In questo caso viene usato un indice base zero per accedere alla colonna. L'ordine delle colonne è quello specificato nella clausola SELECT della query.

Infine, esiste l'accesso ***indicizzato tipizzato***, con il quale il valore della colonna viene ottenuto mediante un metodo che ritorna un valore del tipo appropriato:

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

Il DataReader definisce un metodo per ogni tipo di dato primitivo.

Qualunque sia il metodo utilizzato, se il nome o l'indice fanno riferimento ad una colonna inesistente, viene sollevata l'eccezione di **IndexOutOfRange**.

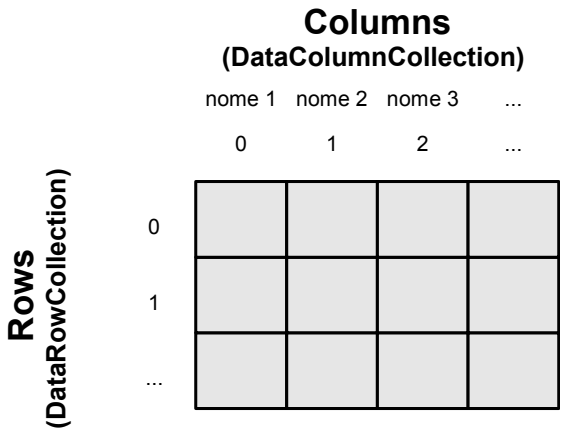
4 Introduzione agli oggetti disconnessi: DataTable

Gli oggetti disconnessi sono così definiti perché il loro funzionamento e ciclo di vita sono indipendenti dal fatto che vi sia una comunicazione con il database. Inoltre, essi sono unici all'interno di ADO.NET e non specifici di un determinato provider.

4.1 DataTable

Gli oggetti DataTable consentono di memorizzare insiemi di dati di natura tabellare. Benché un DataTable possa memorizzare dati provenienti da qualsiasi origine, esso rappresenta di solito il corrispondente in memoria del result set prodotto da una query.

Un DataTable possiede una collezione di righe (oggetti DataRow) e di colonne (oggetti DataColumn), esposte rispettivamente attraverso le proprietà Rows e Columns.



4.1.1 Caricare un result set su un DataTable

Per caricare un result set nel DataTable è possibile utilizzare il metodo **Load()**; questo richiede come argomento l'oggetto DataReader da utilizzare per scorrere le righe:

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

DbCommand cmd = cn.CreateCommand();
cmd.CommandText = "select NomeSocietà, Paese from Clienti";

cn.Open();
DbDataReader dr = cmd.ExecuteReader();

DataTable dtClienti = new DataTable();
dtClienti.Load(dr);...
```

Load(), prima crea la struttura interna necessaria per ospitare il result set, numero, nome e tipo appropriato delle colonne, quindi carica le righe. Infine, chiude automaticamente il DataReader.

4.1.2 Accesso alle righe e alle colonne del DataTable

L'accesso alle righe si ottiene indicizzando o scorrendo la proprietà Rows, che è una collezione di oggetti DataRow. L'accesso alle colonne si ottiene indicizzando una riga; è possibile farlo sia mediante un indice sia mediante il nome della colonna.

```
...
dtClienti.Load(dr);

foreach (DataRow row in dtClienti.Rows)
{
    Console.WriteLine("Nome: {0} | Paese: {1}", row[0], row[1]);
}
```

I valori di colonna ottenuti sono di tipo object e dunque potrebbe essere necessario convertirli nel tipo appropriato.

4.1.3 Accesso tipizzato ai valori di una riga

La classe DataRow espone due metodu, **Field<>()** e **SetField<>()**, che consentono di specificare il tipo di dato della colonna alla quale si accede. Ad esempio:

```
...// qui carica gli ordini dal database
dtOrdini.Load(dr);

foreach (DataRow row in dtOrdini.Rows)
{
    Console.WriteLine("Data ordine: {0}", row.Field<DateTime>(0));
}
```

4.1.4 Visualizzare un DataTable in un DataGridView

Negli scenari più semplici è sufficiente associare il DataTable alla proprietà DataSource del DataGridView. Quest'ultimo provvederà automaticamente a generare la struttura di visualizzazione necessaria (numero, nome e tipo delle colonne).

```
private void Form1_Load(object sender, EventArgs e)
{
    ...
    dtClienti.Load(dr);
    dgwClienti.DataSource = dtClienti;
}
```

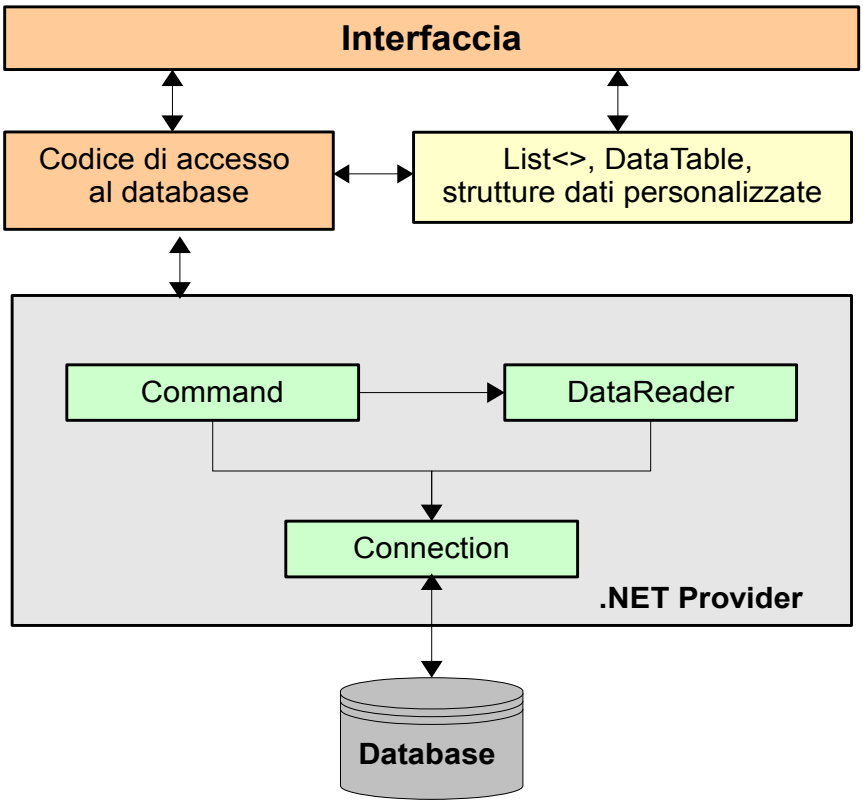
5 Programmare con ADO.NET

In questa sezione prenderemo in considerazione dei semplici scenari di accesso ad un database, mostrando in azione le funzionalità principali degli oggetti precedentemente introdotti.

Utilizzeremo il più possibile gli oggetti connessi comuni, in modo da rendere il codice facilmente adattabile a DBMS diversi.

5.1 Struttura delle applicazioni di accesso ai database

Una tipica applicazione di accesso ai database è strutturata secondo lo schema mostrato in figura.



Il codice di accesso dal database svolge due compiti:

1. recupera i dati e li carica in memoria, in strutture dati predefinite (array, List<>, DataTable), o progettate dal programmatore.
2. esegue le modifiche sul database.

Gli esempi che seguiranno rispetteranno questo schema.

5.2 Gestire globalmente le connessioni

Per semplificare il codice, le connessioni saranno gestite mediante una classe helper.

```
public static class Database
{
    public static string cnStr = "...";
    public static DbConnection CreaConnessione()
    {
        return new OleDbConnection(cnStr);
    }
}
```

5.3 Caricamento dei dati in una collezione di oggetti

Da una parte ci sono i dati memorizzati in tabelle, secondo la struttura di un database relazionale; d'altra si desidera implementare un modello ad oggetti corrispondente.

Qui ci limiteremo ad un scenario semplificato, che prevede il caricamento dei corrieri in un elenco di oggetti.

5.3.1 Caricamento in memoria dei corrieri

Il primo passo è definire la classe Corriere. Questa “mappa” l'entità Corriere del modello E/R e definisce tanti attributi quante sono le colonne della tabella Corrieri che si desidera gestire.

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

Successivamente, è opportuno definire una classe che implementi il *codice di accesso ai dati* e abbia il compito di eseguire le operazioni sul database.

```
public class GestioneCorrieri
{
    public static List<Corriere> CaricaCorrieri()
    {
        List<Corriere> corrieri = new List<Corriere>();
        DbConnection cn = Database.CreaConnessione();
        DbCommand cmd = cn.CreateCommand();
        cmd.CommandText = "SELECT idCorriere, NomeSocietà, Telefono FROM Corrieri";
        cn.Open();
        DbDataReader dr = cmd.ExecuteReader();
        while (dr.Read() == true)
        {
            Corriere c = new Corriere();
            c.IdCorriere = dr.GetInt32(0);
            c.NomeSocietà = dr.GetString(1);
            c.Telefono = dr.GetString(2);
```

```
            corrieri.Add(c);
        }
        cn.Close();
        return corrieri;
    }
}
```

5.3.2 Uso del metodo CaricaCorrieri()

Una tipica funzionalità da implementare è quella di visualizzare i corrieri su un ListBox:

```
List<Corriere> corrieri = GestioneCorrieri.CaricaCorrieri();
lstCorrieri.DisplayMember = "NomeSocietà";
lstCorrieri.Items.AddRange(corrieri.ToArray());
```

5.4 Istruzioni SQL parametriche

Negli scenari realistici, l'effettiva istruzione SQL da eseguire dipende spesso dall'input dell'utente. Si parla in questo caso di ***istruzioni SQL parametriche***. Queste contengono uno o più segnaposti, i quali devono essere valorizzati prima che l'istruzione venga eseguita.

Ecco un esempio di query parametrica:

```
SELECT NomeSocietà FROM Clienti
WHERE Paese = @Paese
```

@Paese funge da variabile, il cui valore (la nazione del cliente) deve essere specificato prima di eseguire l'istruzione. Per questo è necessario l'uso di un ***Command parametrico***.

Istruzioni SQL dinamiche

E' possibile utilizzare anche **istruzioni SQL dinamiche**. In sostanza si tratta di costruire la stringa che rappresenta l'istruzione concatenando le parti fisse con quelle che dipendono dall'input dell'utente.

5.4.1 Creazione di un Command parametrico

Un Command parametrico definisce internamente una collezione di parametri, utilizzati per valorizzare i segnaposti di un'istruzione SQL parametrica.

Il tipo dei parametri dipende dal provider utilizzato (**OleDbParameter**, **SqlParameter**, etc); esiste comunque il tipo comune DbParameter, che può essere usato negli scenari più semplici.

La creazione di un Command parametrico segue di solito la seguente scaletta:

```
// crea il command
// imposta il testo del comando (istruzione SQL)
// crea il parametro
// imposta proprietà parametro: nome, tipo, valore...
// aggiungi parametro alla collezione del command
// ripeti le istruzioni evidenziate per ogni parametro
```

5.4.2 Caricare gli ordini evasi dal corriere selezionato

E' opportuno implementare questa operazione come metodo della classe GestioneOrdini. Il metodo riceve come argomento l'id del corriere selezionato ed esegue l'istruzione SQL:

```
SELECT DataOrdine, Destinatario FROM Ordini
WHERE Corriere = @IdCorriere
```

Il metodo ritorna un DataTable contenente gli ordini richiesti.

```
public class GestioneOrdini
{
    public static DataTable OrdiniByCorriere(int idCorriere)
    {
        string sql = " SELECT DataOrdine, Destinatario FROM Ordini" +
                     " WHERE Corriere = @IdCorriere";

        DbConnection cn = Database.CreaConnessione();
        DbCommand cmd = cn.CreateCommand();
        cmd.CommandText = sql;

        DbParameter paId = cmd.CreateParameter();
        paId.ParameterName = "@IdCorriere";
        paId.DbType = DbType.Int32;
        paId.Value = idCorriere;
        cmd.Parameters.Add(paId);

        cn.Open();
        DbDataReader dr = cmd.ExecuteReader();
        DataTable dtOrdini = new DataTable();
        dtOrdini.Load(dr);
        cn.Close();
        return dtOrdini;
    }
}
```

Ci sono alcune considerazioni da fare sul codice di creazione del parametro:

1. l'uso di un oggetto comune (DbParameter) richiede l'impostazione delle singole proprietà. Gli oggetti specializzati (SqlParameter, OldDbParameter, etc) possono essere impostati già durante la loro creazione;
2. non tutte le proprietà devono essere valorizzate; ciò dipende dal provider utilizzato. Nel caso specifico sarebbe stato sufficiente impostare la sola proprietà **Value**.

Uso del metodo di caricamento degli ordini

Si presuppone che l'utente, selezionando un corriere nel ListBox, possa visualizzare gli ordini in un DataGridView cliccando su un apposito bottone.

```
private void btnOrdini_Click(object sender, EventArgs e)
{
    Corriere c = lstCorrieri.SelectedItem as Corriere;
    if (c == null)
```

```
        return;

    DataTable dtOrdini = GestioneOrdini.OrdiniByCorriere(c.IdCorriere);
    dgwOrdini.DataSource = dtOrdini;
}
```

5.4.3 Criterio di associazione dei parametri ai segnaposti corrispondenti

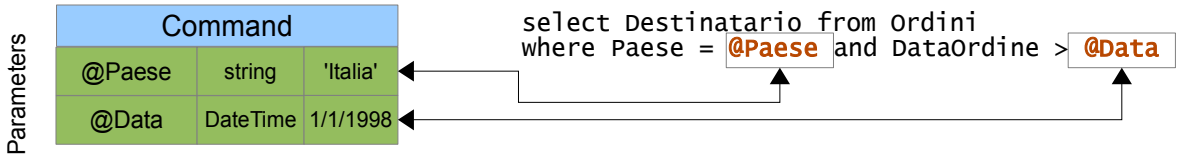
Il criterio di associazione dei parametri ai segnaposti dell'istruzione SQL dipende dal provider utilizzato. Questo può essere ***posizionale*** o ***basato sul nome***.

Associazione posizionale

Usato dal provider OleDb, utilizza la posizione dei parametri e dei segnaposti, ignorando i loro nomi. Il primo parametro del Command viene associato al primo segnaposto, il secondo parametro al secondo segnaposto, e così via.

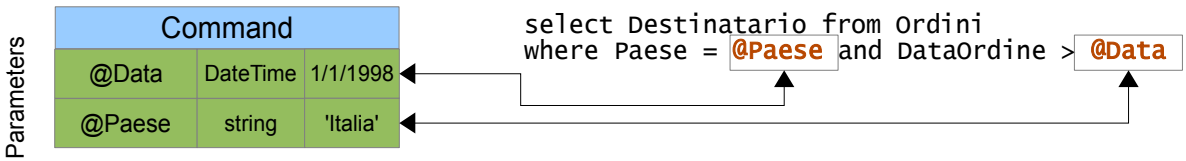
Di seguito vengono forniti due esempi di associazione di un Command parametrico a una query parametrica. Nel primo la corrispondenza è corretta.

Corretta corrispondenza tra parametri e segnaposti



Nel secondo esempio, il primo parametro, una data, viene associato al primo segnaposto, una stringa.

Errata corrispondenza tra parametri e segnaposti



Un simile errore non viene rilevato e può produrre risultati imprevedibili.

Nome parametri posizionali

Poiché il nome di parametri e segnaposti viene ignorato, qualunque combinazione di caratteri è corretta. Uno standard molto utilizzato prevede l'uso del simbolo '?' come segnaposto nelle istruzioni SQL. Per i parametri, invece, è sempre opportuno utilizzare un nome significativo.

Criterio di associazione basato sul nome

E' usato dai provider più evoluti, come SQL Server, Oracle, etc, ed è basato unicamente sulla corrispondenza tra i nomi dei parametri e quelli dei segnaposti.

Prendendo come riferimento questo criterio, entrambi gli esempi precedenti sarebbero stati corretti.

5.5 Modifica dei dati del database

In questa categoria rientra l'esecuzione di istruzioni INSERT, UPDATE e DELETE. Poiché queste prevedono una modifica del database vi sono due considerazioni da fare:

1. l'applicazione deve avere i permessi necessari per eseguire una modifica dei dati;
2. benché possibile, non ha senso utilizzare un DataReader, dato che l'esecuzione del Command non produce un result set.

Il punto 1) riguarda l'amministrazione del DBMS, il quale deve garantire che i vari client abbiano i privilegi necessari per eseguire le operazioni loro richieste (in realtà la questione è generale e non riguarda soltanto le operazioni di modifica). Per quanto riguarda i database basati su file (come Access e SQLite) è necessario che l'applicazione abbia l'autorizzazione di scrittura sul file contenente il database.

5.5.1 Esecuzione delle istruzioni di modifica

Per queste istruzioni, il Command prevede un metodo apposito: **ExecuteNonQuery()**. Il metodo ritorna un valore intero che indica il numero di righe coinvolte nell'esecuzione del comando. Il valore può essere utilizzato per avere una conferma sulla corretta esecuzione dell'operazione.

5.6 Eliminazione del corriere selezionato

Occorre eseguire la seguente istruzione SQL:

```
DELETE FROM Corrieri
WHERE IdCorriere = @IdCorriere
```

L'implementazione dell'operazione viene realizzata da un nuovo metodo della classe GestioneCorrieri, EliminaCorriere(). Questo richiede come argomento l'id del corriere da eliminare.

```
public static bool EliminaCorriere(int idCorriere)
{
    string sql = "DELETE FROM Corrieri WHERE IdCorriere = @IdCorriere";
    DbConnection cn = Database.CreaConnessione();
    DbCommand cmd = cn.CreateCommand();

    DbParameter paId = cmd.CreateParameter();
    paId.ParameterName = "@IdCorriere";
    paId.DbType = DbType.Int32;
    paId.Value = idCorriere;

    cmd.Parameters.Add(paId);
```

```
    cmd.CommandText = sql;
    cn.Open();
    int numRighe = cmd.ExecuteNonQuery();
    cn.Close();
    return numRighe == 1;
}
```

Verificando che il comando abbia effettivamente eliminato una riga, il metodo ritorna true se il corriere è stato eliminato, false in caso contrario.

Errore nell'esecuzione della query

In questo e nei precedenti esempi non viene presa in considerazione la possibilità che l'operazione possa fallire. Si tratta di una questione molto importante, che negli scenari realistici non può essere ignorata, ma che qui non viene considerata per mantenere gli esempi di facile comprensione.

Nel caso in oggetto, l'operazione è destinata a fallire se si tenta di eliminare un corriere al quale sono collegati degli ordini. Nel database è presente infatti un vincolo di integrità referenziale tra Corrieri e Ordini che impedisce questo tipo di modifica. Un programma realistico dovrebbe intercettare questo errore, informando l'utente.

5.6.1 Uso del metodo di eliminazione del corriere

Si presuppone che l'utente selezioni un corriere nel ListBox.

```
private void btnEliminaCorriere_Click(object sender, EventArgs e)
{
    Corriere c = lstCorrieri.SelectedItem as Corriere;
    if (c == null)
        return;

    GestioneCorrieri.EliminaCorriere(c.IdCorriere);
    MessageBox.Show("Il corriere è stato eliminato");
}
```

Nota bene: non viene presa in considerazione la possibilità che il metodo ritorni false, dato che l'id del corriere selezionato è sicuramente corretto. L'eventuale fallimento dell'operazione può dipendere da altri motivi, uno dei quali indicato nella nota precedente.

5.7 Inserimento di un nuovo corriere

Occorre eseguire la seguente istruzione SQL:

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

Nota bene: non viene specificata la colonna IdContatore, poiché si tratta di una colonna identità, il cui valore viene automaticamente generato dal DBMS.

L'implementazione dell'operazione viene realizzata da un nuovo metodo della classe GestioneCorrieri, InserisciCorriere(). Questo richiede come argomento il corriere da inserire.

```
public static void InserisciCorriere(Corriere c)
{
    string sql = " INSERT INTO Corrieri(NomeSocietà, Telefono)" +
                 " VALUES (@NomeSocietà, @Telefono)";
    DbConnection cn = Database.CreaConnessione();
    DbCommand cmd = cn.CreateCommand();

    DbParameter paNome = cmd.CreateParameter();
    paNome.ParameterName = "@NomeSocietà";
    paNome.DbType = DbType.String;
    paNome.Value = c.NomeSocietà;
    cmd.Parameters.Add(paNome);

    DbParameter paTel = cmd.CreateParameter();
    paTel.ParameterName = "@Telefono";
    paTel.DbType = DbType.String;
    paTel.Value = c.Telefono;
    cmd.Parameters.Add(paTel);

    cmd.CommandText = sql;
    cn.Open();
    cmd.ExecuteNonQuery();
    cn.Close();
}
```

5.7.1 Uso del metodo di inserimento corriere

Si presuppone che l'utente inserisca nome e telefono mediante TextBox.

```
private void btnInserisciCorriere_Click(object sender, EventArgs e)
{
    Corriere c = new Corriere();
    c.NomeSocietà = txtNomeSocietà.Text
    c.Telefono = txtTelefono.Text
    GestioneCorrieri.InserisciCorriere(c);
    MessageBox.Show("Il corriere è stato inserito");
}
```

5.7.2 Recuperare il valore della colonna identità

Esistono scenari nei quali si desidera utilizzare immediatamente l'elemento appena inserito per future operazioni. Ad esempio, nel caso precedente si può immaginare di voler aggiungere immediatamente il nuovo corriere al ListBox senza dover ricaricare l'intero elenco dal database.

In questi casi sorge il problema di recuperare il valore delle colonne identità, poiché questo è generato automaticamente dal DBMS e dunque sconosciuto all'applicazione.

Ogni DBMS usa delle tecniche specifiche sia per gestire le colonne identità sia per recuperarne i valori. Con SQL Server e Access è possibile farlo eseguendo la query[3]:

3 In realtà con SQL Server questa tecnica ha qualche limitazione, che però può essere trascurata nei semplici scenari che stiamo considerando.

```
SELECT @@identity
```

Metodo di recupero dell'ultimo valore identity generato

Il recupero dell'ultimo valore identità generato rappresenta un'operazione di utilità generale, dunque è opportuno implementarla nella classe helper Database.

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

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

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

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

Vi sono diverse osservazioni da fare.

1. Il metodo riceve l'oggetto Connection da utilizzare per eseguire il comando. E' una scelta opportuna, dato che di solito si desidera recuperare il valore identity utilizzando la stessa connessione usata dall'istruzione INSERT.
2. Mediante la proprietà **State** il metodo verifica se la connessione è già aperta; in caso contrario l'apre.
3. Dopo aver eseguito la query, il metodo chiude la connessione, ma soltanto se è stata aperta in precedenza. In caso contrario, lascia la responsabilità al metodo chiamante.

Recupero della colonna IdCorriere

Occorre modificare il metodo InserisciCorriere():

```
public static void InserisciCorriere(Corriere c)
{
    string sql = " INSERT INTO Corrieri(NomeSocietà, Telefono)" +
                 " VALUES (@NomeSocietà, @Telefono)";

    DbConnection cn = Database.CreaConnessione();
    DbCommand cmd = cn.CreateCommand();

    ...  // qui prepara il Command

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

Uso del corriere appena inserito

Dopo aver inserito il nuovo corriere questo viene immediatamente aggiunto al ListBox.

```
private void btnInserisciCorriere_Click(object sender, EventArgs e)
{
    Corriere c = new Corriere();
    c.NomeSocietà = txtNomeSocietà.Text
    c.Telefono = txtTelefono.Text

    GestioneCorrieri.InserisciCorriere(c);
    MessageBox.Show("Il corriere è stato inserito");
    lstCorrieri.Items.Add(c);
}
```

5.8 Aggiornamento dati

L'aggiornamento è un'operazione che si volge in due fasi.

Nella prima, l'utente seleziona l'elemento da modificare: un corriere, un ordine, i dati di un cliente, etc. L'applicazione risponde visualizzando i dati attuali e consentendo all'utente la loro modifica. Infine, i dati aggiornati vengono inviati al database, sovrascrivendo quelli originali.

A “dirigere” questo scambio tra applicazione e database è la chiave primaria dell'elemento da modificare.

Di seguito sarà applicata questo procedimento nella modifica di un corriere.

5.8.1 Caricamento del corriere da modificare

Si presuppone che l'utente selezioni nel ListBox il corriere da modificare. Occorre eseguire la seguente query.

```
SELECT NomeSocietà, Telefono FROM Corrieri
WHERE IdCorriere = @IdCorriere
```

L'implementazione dell'operazione viene realizzata da un nuovo metodo della classe GestioneCorrieri, GetCorriere(). Questo richiede come argomento l'id del corriere da caricare.

```
public static Corriere GetCorriere(int idCorriere)
{
    string sql = " SELECT NomeSocietà, Telefono FROM Corrieri" +
                 " WHERE IdCorriere = @IdCorriere";

    DbConnection cn = Database.CreaConnessione();
    DbCommand cmd = cn.CreateCommand();

    DbParameter paId = cmd.CreateParameter();
    paId.ParameterName = "@IdCorriere";
    paId.DbType = DbType.Int32;
    paId.Value = idCorriere;

    cmd.Parameters.Add(paId);
    cmd.CommandText = sql;
    cn.Open();
    DbDataReader dr = cmd.ExecuteReader();

    Corriere c = null;
    if (dr.Read() == true)
    {
        c = new Corriere();
        c.IdCorriere = idCorriere;
        c.NomeSocietà = dr.GetString(0);
        c.Telefono = dr.GetString(1);
    }
    cn.Close();
    return c;
}
```

Non c'è alcun ciclo, dato che il result set contiene una o nessuna riga, condizione che viene verificata nella if(). Alternativamente, è possibile testare la proprietà ***HasRows***, che indica se il result set contiene almeno una riga.

Uso del metodo GetCorriere()

In uno scenario realistico, la modifica del corriere avverrebbe in un form apposito. Qui ci si limita a mostrare l'invocazione del metodo.

```
... // qui viene selezionato l'id del corriere
Corriere c = GestioneCorrieri(idCorriere);
txtNomeSocietà.Text = c.NomeSocietà;
txtTelefono.Text = c.Telefono;
...
```

5.8.2 Scrittura dei dati aggiornati

Si tratta di eseguire la seguente istruzione SQL:

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

L'operazione viene realizzata da un nuovo metodo della classe GestioneCorrieri: ModificaCorriere(). Questo richiede come argomento il corriere da modificare.

```
public static bool ModificaCorriere(Corriere c)
{
    string sql = " UPDATE Corrieri SET" +
                  " NomeSocietà = @NomeSocietà, "+
                  " Telefono = @Telefono "+
                " WHERE IdCorriere = @IdCorriere";

    DbConnection cn = Database.CreaConnessione();
    DbCommand cmd = cn.CreateCommand();

    DbParameter paNome = cmd.CreateParameter();
    paNome.ParameterName = "@NomeSocietà";
    paNome.DbType = DbType.String;
    paNome.Value = c.NomeSocietà;
    cmd.Parameters.Add(paNome);

    DbParameter paTel = cmd.CreateParameter();
    paTel.ParameterName = "@Telefono";
    paTel.DbType = DbType.String;
    paTel.Value = c.Telefono;
    cmd.Parameters.Add(paTel);

    DbParameter paId = cmd.CreateParameter();
    paId.ParameterName = "@IdCorriere";
    paId.DbType = DbType.Int32;
    paId.Value = c.IdCorriere;
    cmd.Parameters.Add(paId);

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

    return numRighe == 1;
}
```

Uso del metodo ModificaCorriere()

Si può supporre di eseguire la modifica in risposta all'azione di conferma da parte dell'utente, utilizzando lo stesso oggetto corriere precedentemente caricato dal database. Qui ci si limita a mostrare l'invocazione del metodo.

```
... // 'c' contiene i dati originali, compreso l'id
c.Nomesocietà = txtNomeSocietà.Text;
c.Telefono = txtTelefono.Text;
GestioneCorrieri.ModificaCorriere(c);
MessageBox.Show("Il corriere è stato modificato");
...
```