**Riepilogo della sezione**

Accedere a un database relazionale con ADO.NET

ADO.NET è il nome della **tecnologia di accesso ai dati** che ci permette di collegarci a un database relazionale per leggere e scrivere dati grazie a query e comandi in linguaggio SQL. All'interno del database, i dati sono organizzati su tabelle, e ogni tabella è formata di righe e colonne. Ogni riga rappresenta un'entità (ad esempio un corso), mentre le colonne sono gli attributi di quell'entità (ad esempio il titolo, la descrizione, l'autore, ...).

Anche se esistono tante tecnologie database, ADO.NET è molto versatile perché è composto di classi base su cui poi ogni produttore può costruire il proprio provider. Ad esempio, Microsoft ha realizzato il provider System.Data.SqlClient che serve a collegarsi a un database SQLServer, ma anche Microsoft.Data.Sqlite che invece serve a collegarsi a un database Sqlite.

Per ogni tecnologia database esiste almeno un provider, che possiamo installare nel progetto come pacchetto NuGet. Perciò, se volessimo installare il provider per Sqlite, potremmo lanciare questo comando:

1. dotnet add package Microsoft.Data.Sqlite

A questo punto, nella nostra applicazione, abbiamo a disposizione varie classi per interagire con il database. Eccone alcune tra le più usate:

* **SqliteConnection** ci permette di stabilire una connessione al database;
* **SqliteCommand** serve ad inviare query e comandi SQL al database;
* **SqliteParameter** per usare l'input dell'utente in maniera sicura nelle nostre query e comandi;
* **SqliteDataReader** per leggere la tabella (o le tabelle) di risultati restituite dal database in seguito a una query SQL;
* **SqliteTransaction** per eseguire query e comandi in un contesto isolato e in maniera atomica (tutto o niente), così che i dati nel database restino sempre in uno stato consistente.

Queste classi sono presenti in tutti i provider ADO.NET e quindi, se impariamo a usarle con Sqlite, allora le sapremo già usare per qualsiasi altra tecnologia database. Quel che cambia è giusto il prefisso nel nome delle classi (SqliteConnection per Sqlite, SqlConnection per SQL Server, ecc...). Tuttavia, dobbiamo sempre tenere presente che ogni tecnologia database usa un proprio "dialetto" del linguaggio SQL e perciò ci servirà comunque del tempo per imparare a usarlo al suo massimo potenziale.

Classi "connesse" e "disconnesse"

Le classi che abbiamo appena visto sono anche chiamate classi "connesse" perché sono pensate per funzionare con una connessione al database. Poi esistono anche queste altre due classi, che invece sono definite "disconnesse" perché possono funzionare senza dover stabilire alcuna connessione. Infatti, le possiamo usare liberamente anche se nella nostra applicazione non esiste ancora alcun database.

* **DataTable** rappresenta una tabella dati, su righe e colonne. Grazie al suo metodo Load, è in grado di caricare dati da un SqliteDataReader (o da qualsiasi altro tipo di data reader), così che possiamo conservare i risultati nella memoria dell'applicazione anche dopo aver chiuso la connessione al database;
* **DataSet** è una collezione di DataTable e può anche descrivere le relazioni esistenti tra le DataTable, come ad esempio la relazione uno-a-molti che esiste tra un corso e le sue lezioni.

Insomma, le classi "disconnesse" sono in grado di imitare le stesse strutture dati che abbiamo in un database relazionale, così che possiamo usarle nell'applicazione anche dopo aver chiuso la connessione.

Qualche esempio completo

Vediamo qualche esempio tipico per impratichirci con l'uso di queste classi. Ecco la parte interessante in cui andiamo a leggere delle informazioni.

1. //In questa lista salveremo i titoli dei corsi letti dal database
2. var titles = new List<string>();
4. using (var conn = new SqliteConnection("Data Source=Data/MyCourse.db"))
5. {
6. conn.Open();
7. string query = "SELECT Title FROM Courses";
8. using (var cmd = new SqliteCommand(query, conn))
9. {
10. using (var reader = cmd.ExecuteReader())
11. {
12. while(reader.Read())
13. {
14. string title = Convert.ToString(reader["Title"]);
15. titles.Add(title);
16. }
17. }
18. }
19. }

La connessione al database andrebbe **aperta solo quando necessario e chiusa il più presto possibile**, proprio come abbiamo fatto nel precedente esempio. Perciò, se i risultati che abbiamo ottenuto dal database dobbiamo passarli a un altro componente della nostra applicazione, possiamo caricarli in una List<T> oppure in un DataTable e poi chiudere la connessione, come in questo esempio.

1. //Ora usiamo un DataTable per conservare i risultati
2. var dataTable = new DataTable();
4. using (var conn = new SqliteConnection("Data Source=Data/MyCourse.db"))
5. {
6. conn.Open();
7. string query = "SELECT Title FROM Courses";
8. using (var cmd = new SqliteCommand(query, conn))
9. {
10. using (var reader = cmd.ExecuteReader())
11. {
12. dataTable.Load(reader); //Qui carichiamo nel DataTable le informazioni dal SqliteDataReader
13. }
14. }
15. }

Possiamo anche inviare due o più query SQL con un unico oggetto SqliteCommand: ci basta separarle con punto e virgola. Ecco un esempio in cui recuperiamo sia i dati di un corso che l'elenco delle sue lezioni. Usiamo il ciclo do..while per creare tanti DataTable per quanti sono le tabelle di risultati restituite.

1. //Ora usiamo un DataSet, che conterrà più DataTable
2. var dataSet = new DataSet();
3. dataSet.EnableConstraints = false; //Fix per evitare un bug del provider Microsoft.Data.Sqlite
5. using (var conn = new SqliteConnection("Data Source=Data/MyCourse.db"))
6. {
7. conn.Open();
8. string query = "SELECT \* FROM Courses WHERE Id=5; SELECT \* FROM Lessons WHERE IdCourse=5";
9. using (var cmd = new SqliteCommand(query, conn))
10. {
11. using (var reader = cmd.ExecuteReader())
12. {
13. do
14. {
15. var dataTable = new DataTable();
16. dataSet.Add(dataTable);
17. dataTable.Load(reader);
18. } while (!reader.IsClosed); //Continuamo a ciclare finché il SqliteDataReader non viene chiuso
19. }
20. }
21. }

Prevenire la SQL Injection

Quando riceviamo l'input dell'utente, dovremmo sempre **sanitizzarlo**, per evitare che malintenzionati possano eseguire delle query e comandi SQL arbitrari. Per questo usiamo un oggetto SqliteParameter, che poi aggiungiamo al SqliteCommand come si vede nell'esempio seguente.

1. string title = "Il titolo del mio corso"; //Questo valore arriva dall'utente
2. using (var conn = new SqliteConnection("Data Source=Data/MyCourse.db"))
3. {
4. conn.Open();
5. string query = "SELECT \* FROM Courses WHERE Title=@title";
6. SqliteParameter = new SqliteParameter("title", title); //Qui creiamo il parametro assegnando nome e valore
7. using (var cmd = new SqliteCommand(query, conn))
8. {
9. cmd.Parameters.Add(parameter); //Qui aggiungiamo il parametro al command
10. using (var reader = cmd.ExecuteReader())
11. {
12. //OMISSIS
13. }
14. }
15. }

Distruggere correttamente gli oggetti IDisposable

Nei precedenti esempi abbiamo fatto ampio uso dei **blocchi using**. In questo modo ci assicuriamo che il metodo Dispose degli oggetti che implementano IDisposable venga sempre chiamato, anche se si dovesse verificare un'eccezione durante il loro utilizzo. È molto importante che tali oggetti vengano distrutti correttamente, altrimenti si potrebbero verificare sprechi di memoria o altre situazioni indesiderate.

Infatti, se non distruggessimo gli oggetti SqliteConnection, il **connection pool**, che è l'organo che si occupa di predisporre le connessioni, presto si saturerebbe di connessioni aperte e non ci concederebbe di aprirne altre, di fatto impedendoci di accedere al database.

Ecco uno screenshot di questo problema: un chiaro sintomo che in qualche punto dell'applicazione non stiamo distruggendo correttamente gli oggetti SqlConnection.

Immagine che contiene testo, Carattere, linea, schermata

Descrizione generata automaticamente

L'errore accenna a un "**timeout expired**" che si verifica perché la nostra applicazione rinuncia ad aspettare che il connection pool gli restituisca una connessione.

L'importanza di usare i metodi asincroni e le parole chiave async/await

Molti dei metodi delle classi di ADO.NET possiedono delle varianti asincrone. Ad esempio, la SqliteConnection, oltre al metodo Open dispone di OpenAsync.

I metodi asincroni esistono per le operazioni che fanno uso di **periferiche di I/O**, come il disco o la rete, che sono soggetti a latenze.

Usare i metodi asincroni è ideale perché migliora le prestazioni della nostra applicazione quando ci sono parecchi utenti contemporanei. Infatti, dato che aprire una connessione al database o inviare una query sono operazioni che richiedono un'attesa, il thread può rendersi disponibile durante quell'attesa per fare altro, ad esempio per iniziare a elaborare la richiesta di un altro utente. Poi, quando l'operazione asincrona si conclude, l'esecuzione può riprendere dall'istruzione successiva.

1. public async Task<List<string>> GetCourseTitlesAsync()
2. {
3. var titles = new List<string>();
5. using (var conn = new SqliteConnection("Data Source=Data/MyCourse.db"))
6. {
7. await conn.OpenAsync(); //Uso di un metodo asincrono
8. string query = "SELECT Title FROM Courses";
9. using (var cmd = new SqliteCommand(query, conn))
10. {
11. using (var reader = await cmd.ExecuteReaderAsync()) //Uso di un metodo asincrono
12. {
13. while(await reader.ReadAsync()) //Uso di un metodo asincrono
14. {
15. string title = Convert.ToString(reader["Title"]);
16. titles.Add(title);
17. }
18. }
19. }
20. }
21. return titles;
22. }

Come si vede nell'esempio, per poter sfruttare i metodi asincroni abbiamo dovuto apportare 3 modifiche.

1. Usare l'operatore **await** prima dell'invocazione del metodo asincrono;
2. Apporre la parola chiave **async** in corrispondenza della firma del nostro metodo;
3. Far restituire al metodo un Task<T> (oppure un Task, nel caso restituisse void).

Opzionalmente, possiamo aggiungere il suffisso "Async" al nome del metodo.