ADO.NET

There are many object-relational model (ORM) solutions available, but there are also plenty of applications with custom data layer implementations using pure ADO.NET. For performance reasons, some designs are meant to directly pull data, skipping the ORM subsystem entirely, but even if you are using an ORM tool, ADO.NET is still relevant. Under the hood of every ORM sits ADO.NET

Many of the classes used in ADO.NET implement the `IDisposable` interface (which we saw in Chapter 4 when dealing with unmanaged resources), and their lifespans should be limited to prevent resource leakage. ADO.NET deals with many unmanaged components, and so there's some quite straightforward good advice to bear in mind: explicitly wrap Connections, Commands, and Adapters with `using` statements. Open connections as late as you can, and explicitly close them as soon as you can. The `using` statement will ensure that the `Dispose` method is called when the object goes out of the scope of the `using` block.

```
public DataTable GetData()
{
   using (var connection = new SqlConnection(ConnectionString))
   using (var command = new SqlCommand(storedProcedure, connection))
   using (var adapter = new SqlDataAdapter(command))
   {
     DataTable dataTable = new DataTable("Table");
     adapter.Fill(dataTable);
     return dataTable;
   }
}
```

Listing 5.7: Implementations of simple best practices with ADO.NET.

There are also certain objects that should never be used for caching purposes. While caching data can often improve your application performance, nothing from the `System.Data` namespace should ever be cached. It may seem natural to cache `DataTables`, `DataAdapters`, or `DataSets`, (and, indeed, it *can* be done) but these can easily grow out of hand and negate any benefits you might have gained from caching.

These objects are very "heavy," in that they have a lot of extra data and overhead associated with them. Naturally, you should always use the lightest-weight object that you can. Cache data with simple arrays, and populate these arrays with simple objects (ideally these will have simple read/write properties for the fields that are needed). Such objects are often referred to as POCO objects (Plain Old CLR Objects).

LINQ

LINQ (Language-Integrated Query) makes the promise of allowing you to query objects in memory with the same ease as querying a database. For the most part, LINQ lives up to the hype and, if you are at all familiar with SQL, the syntax for LINQ will feel natural.

Another promise of LINQ is improved memory management but, as with everything we're discussing in this chapter, it may also have the opposite affect and dramatically increase the memory footprint if you're not careful. The improved memory management rests in the potential of delayed execution and having to load entire results lists into memory at the same time. LINQ uses the `yield` statement we saw in Chapter 4 to avoid having to return all of the data until the `IEnumerable<T>` or `IQueryable<T>` is expanded.

LINQ to Objects are extensions built on `IEnumerable<T>` that allow you to query arrays and collection types in memory, and they are implemented in much the same way described in the `yield` section of Chapter 4. The extensions typically take delegates as parameters to be called during the iteration of the `IEnumerable<T>`. Thanks to features such as the

`yield` statement, lists can be processed without bringing the whole list into memory at a time. Clearly this means that, depending on your query, your memory footprint may well be reduced.

Other forms of LINQ, such as LINQ to Entity Framework, utilize the `IQueryable<T>` interface. The extensions for this interface take expression trees as parameters and pass them off to the `IQueryProvider`, which is then responsible for interpreting the expression trees to another language, such as T-SQL.

`IQuerytable<T>` defines `IEnumerable<T>` in its contract, which leads to this problem I'm about to describe, and which some developers encounter in their code. Listing 5.8 demonstrates a LINQ to Entity Framework call implemented using the extension method syntax.

```
Order[] pastDueAccounts = null;
DateTime dueDate = DateTime.Today.AddDays(-7);
using (var context = new Context())
{
   pastDueAccounts = context.Accounts
     .Where(account => account.DueDate < dueDate).ToArray();
}
```

Listing 5.8: Implementing a LINQ to Entity Framework call.

This code is pulling the past week's orders out of the database. One may be tempted to refactor the lambda expression in the `Where` clause. After all, it does represent a filter, and it is usually a good thing to encapsulate logic that may change or be used in multiple places.

```
public bool PastDueAccount(Account account)
{
   return account.DueDate < DateTime.Today.AddDays(-7);
}
```

Listing 5.9: Refactoring the lambda expression.

With such a method, it seems natural to rewrite our original query as:

```
Order[] pastDueAccounts = null;
using (var context = new Context())
{
   pastDueAccounts = context.Accounts
     .Where(account => PastDueAccount(account))
     .ToArray();
}
```

Listing 5.10: Rewriting the original LINQ to Entity Framework call.

This logic is actually incorrect, but it may not be caught until after an application has been deployed and the database has had time to accumulate data. Let me explain why.

With the original method, the compiler generated an expression tree and the database was queried for a limited amount of data. On the other hand, the refactored example pulls all of the orders from the database and then filters them in memory. The SQL that is generated will actually exclude the filtering restrictions, which will not be interpreted until *after* the SQL is run and the results returned.

Instead of having the filter use indexes in the database and call up a small subset of the data, no indexes will be used, and the entire contents of the table will be retrieved. For as long as the database has a small number of records, this will not cause a problem, but it will quickly cause *substantial* problems as the database volume grows.

Why do we get this behavior? The compiler implicitly converts lambda expressions to either expression trees or delegates, depending on the context in which they are defined, but named methods are not converted to expression trees. They therefore don't match any of the extension methods for `IQueryable<T>`, and are not incorporated into the SQL that is sent to the database. Instead, they are treated as delegates and passed to `IEnumerable<T>`, and the delegate operates on the objects returned from the SQL call.

This problem can be avoided by returning a lambda expression from the `Where` method, meaning we can rewrite our `PastDueAccount` method as shown in Listing 5.11.

```
public Expression<Func<Order, bool>> PastDueAccount
{
   get
   {
     DateTime startDate = DateTime.Today.AddDays(-7);
     return order => order.TransactionDate > startDate;
   }
}
```

Listing 5.11: Filtering function implemented as an expression.

This way, the lambda expression will be passed to the LINQ provider, and its logic will be incorporated into the final query.

The difference is subtle. The LINQ provider does not know what to do with a delegate like the first implementation of the `PastDueAccount` method. It only knows how to deal with expression trees, and will ignore the delegate. Once the expression tree is parsed by the LINQ provider and the resultant SQL executed on the database, then `IEnumerable` knows exactly how to deal with the delegate and will parse through the results after the query is run.

The end result will be the exact same data, but your computer, the network, and even the database will have to do substantially more work if you get it wrong.