Contents

LINQ expressions are strongly typed and static. They are easy to compose when you know the type of thing you want to query at compile time. But they are difficult to create if you don't know the type until the application runs. Many applications let the user specify the entity type to retrieve, the criteria to filter with, the properties to sort and group by. DevForce **dynamic query building** components help you **build and execute LINQ queries on the fly based on information supplied at runtime**.

The problem

"LINQ" stands for "**Language Integrated** Query". LINQ is designed to be written in the same manner as other code, rubbing shoulders with procedural statements in the body of your application source files. C# and Visual Basic are statically typed languages so the main pathway for writing LINQ queries is to compose them statically as strongly typed query objects that implement the *IQueryable<T>* interface. Standard LINQ queries, and almost of the LINQ examples in the DevForce Resource Center, are static and strongly typed. Here is a typical example:

```
IQueryable<Customer> customers = _myEntityManager.Customers;
var query = customers.Where(c => c.CompanyName = "Acme");
```

```
Dim customers As IQueryable(Of Customer) = _myEntityManager.Customers
Dim query = customers.Where(Function(c) c.CompanyName = "Acme")
```

The *_myEntityManager.Customers* expression returns an instance of a query object that implements *IQueryable<Customer>*. The strongly typed nature of the expression is what allows IntelliSense and the compiler to interpret the remainder of the statement.

What if 'T' (Customer in this case) is not known until runtime; from a compiler perspective we no longer have an *IQueryable<T>* but have instead just an *IQueryable*. And *IQueryable*, unfortunately, does not offer any of the extension methods that we think of a standard LINQ; in other words, no *Where*, *Select*, *GroupBy*, *OrderBy*, *Any*, *Count* etc, methods.

How do we write a query when we don't have the type available at runtime?

```
var typeToQuery = typeof(Customer);
IQueryable<???> someCollection = << HOW DO I CREATE AN IQUERYABLE OF "typeToQuery"?>> ;
var query = somecollection.Where( << HOW DO I COMPOSE A LAMBDA EXPRESSION WITHOUT A COMPILE TIME TYPE >> );
```

```
Dim typeToQuery = GetType(Customer)
IQueryable(Of ???) someCollection = << HOW DO I CREATE AN IQUERYABLE OF "typeToQuery"?>>
Dim query = somecollection.Where(<< HOW DO I COMPOSE A LAMBDA EXPRESSION WITHOUT A COMPILE TIME TYPE >>)
```

A second issue can occur even if we know the type of the query but we have a number of 'where conditions' or predicates, that need to be combined. For example

```
Expression<Func<Customer, bool>> expr1 = (Customer c) => c.CompanyName.StartsWith("A");
Expression<Func<Customer, bool>> expr2 = (Customer c) => c.CompanyName.StartsWith("B");
```

```
Dim expr1 As Expression(Of Func(Of Customer, Boolean)) = _
  Function(c As Customer) c.CompanyName.StartsWith("A")
Dim expr2 As Expression(Of Func(Of Customer, Boolean)) = _
  Function(c As Customer) c.CompanyName.StartsWith("B")
```

It turns out that we can use either of these expressions independently, but it isn't obvious how to combine them.

```
var query1 = myEntityManager.Customers.Where(expr1); // ok
var query2 = myEntityManager.Customers.Where(expr1); // ok
var queryBoth = myEntityManager.Customers.Where(expr1 && expr2) // BAD - won't compile.
```

```
Dim query1 = myEntityManager.Customers.Where(expr1) ' ok
Dim query2 = myEntityManager.Customers.Where(expr1) ' ok
Dim queryBoth = myEntityManager.Customers.Where(expr1 AndAlso expr2) ' BAD - won't compile.
```

The ESQL option

You can construct queries on the fly using DevForce "pass-thru" Entity SQL (ESQL). ESQL queries are strings that look like standard SQL, differing primarily (and most obviously) in their references to entity types and properties rather than tables and columns. You can build ESQL queries by concatenating strings that incorporate the runtime query critieria you gleaned from user input.

Most developers prefer to construct LINQ queries with DevForce for two reasons:

1. Building syntactically correct ESQL strings can be more difficult to do correctly than writing DevForce dynamic queries. The dynamic query approach affords a greater degree of compiler syntax checking and the structure of the query is more apparent. You often know *most* of the query at design time; when only some of the query is variable you can mix the statically typed LINQ statements with the dynamic clauses ... as described in the topic detail.
2. ESQL queries must be sent to and processed on the server. You cannot apply ESQL queries to the local entity cache. LINQ queries, on the other hand, can be applied to the database, to the cache, or to both. This is as true of dynamic LINQ queries as it is of static queries. For many developers this is reason enough to prefer LINQ to ESQL.