Contents

The most common and flexible method of composing a query is to **use LINQ** syntax with the *EntityQuery*.

DevForce LINQ is a comprehensive implementation with unique capabilities:

- can execute synchronously and asynchronously.
- applies to remote data source, local cache, or both simultaneously.
- works in 2-tier mode when the client has line-of-sight access to the database **or** in n-tier mode when a remote server mediates between clients and the database.
- can be composed statically inline with other code **or** dynamically to accommodate user-defined query criteria that can only be known at runtime.

Every valid EntityFramework LINQ query is also a valid DevForce LINQ query. The range and power of DevForce LINQ querying may best be appreciated by taking a tour of the EntityFramework's MSDN "101 LINQ Samples" web page. Every sample works in DevForce, 2-tier or n-tier, whether sent to the database or applied to local cache.

Every EntityFramework entity type can be queried with DevForce LINQ: all forms of inheritance, complex types, anonymous types, all association cardinalities (including many-to-many).

The DevForce LINQ query story begins with *EntityQuery*.

The EntityQuery

DevForce provides the *EntityQuery<T>* class to support LINQ syntax. The *EntityQuery<T>* implements .NET's IQueryable<T> interface and offers a complete implementation of LINQ functionality, including multiple overloads for all of the following standard LINQ operators:

All, *Any*, *Average*, *Cast*, *Concat*, *Contains*, *Count*, *DefaultIfEmpty*, *Distinct*, *ElementAt*, *Except*, *First*, *FirstOrDefault*, *GroupBy*, *Join*, *Last*, *LastOrDefault*, *OfType*, *OrderBy*, *OrderByDescending*, *Select*, *SelectMany*, *Single*, *SingleOrDefault*, *Skip*, *SkipWhile*, *Take*, *TakeWhile*, *ThenBy*, *ThenByDescending*, *Sum*, *Union* and *Where*.

Because an *EntityQuery* is most commonly used to query an Entity Data Model, DevForce is subject to the same restrictions which the Entity Framework places on such queries. For more information, see Supported and Unsupported LINQ Methods (LINQ to Entities).

Note that while an *EntityQuery* may be restricted from running against the Entity Framework on the backend, the same query can always be executed locally against the *EntityManager's* cache regardless of any backend restrictions. Similarly, POCO queries, are also not subject to these restrictions. This is why some of the operators listed above, such as *ElementAt*, *Last*, *SkipWhile*, and *TakeWhile*, among others, are still provided, even though they cannot be handled by the Entity Framework. In practice, these restrictions tend to be a minimal hindrance because there is usually another method or overload that can accomplish the same result.

Several additional DevForce-specific operators are offered as well, such as FirstOrNullEntity and support for building untyped LINQ queries dynamically.

EntityQuery basics

Entity queries, like all LINQ queries, can be composed, executed and enumerated in a variety of stepwise ways. Consider, for example, the following query:

```
var customersQuery =
  from cust in manager.Customers
  where cust.ContactTitle == "Sales Representative"
  orderby cust.CompanyName
  select cust;
```

```
Dim customersQuery =
  From cust In manager.Customers
```

```
  Where cust.ContactTitle = "Sales Representative"
  Order By cust.CompanyName
 Select cust
```

The same query can also be written using LINQ method-based syntax as shown below:

```
var customersQuery = manager.Customers
  .Where(c => c.ContactTitle == "Sales Representative")
  .OrderBy(c => c.CompanyName)
```

```
Dim customersQuery = manager.Customers _
  .Where(Function(c) c.ContactTitle = "Sales Representative") _
  .OrderBy(Function(c) c.CompanyName)
```

Whether to use query or method syntax is your choice. Both syntaxes provide the same functionality, with minor differences. We generally use method, sometimes called fluent, syntax in our samples and snippets, but that's because it's the syntax we're most comfortable with. If you're a VB developer, query syntax can be much easier to write and read.

The above returns an *EntityQuery<Customer>*. It's often easier when working with generic types (especially nested generic types), to use an implicitly typed variable, such as shown above. This isn't required, and you can still use explicit type variables too.

Creating a query

You may have wondered in looking at the above samples what manager.Customers referred to, and why you could append LINQ methods to it. When DevForce generates the code for your entity model it includes these helper properties on the *EntityManager*. Here's what *Customers* looks like:

```
public IbEm.EntityQuery<Customer> Customers {
 get { return new IbEm.EntityQuery<Customer>("Customers", this); }
}
```

```
Public ReadOnly Property Customers() As IbEm.EntityQuery(Of Customer)
 Get
  return new IbEm.EntityQuery(Of Customer)("Customers", Me)
 End Get
End Property
```

DevForce does this to make it easy to compose more complex LINQ queries without having to explicitly construct the *EntityQuery* from scratch every time. You're not limited to using these helper properties, but most developers find them useful.

Basic tasks

The *customersQuery* we showed above may look a bit daunting if you're new to LINQ. Queries can be as simple or complex as you need. Here are a few more samples of simple common queries.

Get all entities of a type

When you need to retrieve all instances of a type, all *Employees* for example, you need only provide a simple *EntityQuery* without restriction or selection methods:

```
var query = manager.Employees;
```

```
Dim query = manager.Employees
```

Or in query syntax:

```
var query = from emp in manager.Employees
          select emp;
```

```
Dim query = From emp In manager.Employees
          Select emp
```

You can execute the query in any of the ways described below. Remember that if there are potentially many instances of the type it's usually not a good idea to bring all of them into the entity cache. If you have 10,000 products for example, you rarely need them all loaded into memory.

Simple property filter

You'll often want to retrieve a subset of entities based on filter criteria applied to simple properties of the entity.

For example, a query to retrieve all customers in the UK:

```
var ukQuery = from cust in manager.Customers
              where cust.Country == "UK"
              select cust;
```

```
Dim ukQuery = From cust In manager.Customers
              Where cust.Country = "UK"
              Select cust
```

Query execution

As mentioned earlier, the DevForce *EntityQuery<T>* implements the *IQueryable* interface, which means that the execution of the query is deferred until one of the following operations is performed on the query

- *ToList* is called on the query.
- The query is enumerated in a foreach statement.
- One of the EntityManager *ExecuteQuery*, *ExecuteQueryAsync*, *TryExecuteQuery* or *TryExecuteQueryAsync* methods is called for the query.
- An immediate execution method is called on the query. These methods include *First*, *Single*, *Count* along with several others.
- The *AsScalarAsync()* method is called followed by a call to an immediate execution method.

Note that Silverlight and Windows Store applications, because of their asynchronous nature, can only make use of two of these mechanisms:

- Calling one of the *ExecuteQueryAsync* or *TryExecuteQueryAsync* method overloads.
- Calling the *AsScalarAsync()* method on the query followed by a call to an immediate execution method.

In the example below the addition of a call to *ToList()* forces DevForce to execute the query immediately:

```
List<Customer> customersList = manager.Customers
  .Where(c => c.ContactTitle == "Sales Representative")
  .OrderBy(c => c.CompanyName)
  .ToList();
```

```
Dim customersList As List(Of Customer) = manager.Customers _
  .Where(Function(c) c.ContactTitle = "Sales Representative") _
  .OrderBy(Function(c) c.CompanyName) _
  .ToList()
```

As does the *FirstOrNullEntity* call in the example below.

```
Customer firstCustomer = manager.Customers
  .Where(c => c.ContactTitle == "Sales Representative")
  .OrderBy(c => c.CompanyName)
  .FirstOrNullEntity();
```

```
Dim firstCustomer As Customer = manager.Customers _
  .Where(Function(c) c.ContactTitle = "Sales Representative") _
  .OrderBy(Function(c) c.CompanyName) _
  .FirstOrNullEntity()
```

The same queries executed asynchronously, would look like this.

```
var query = manager.Customers
  .Where(c => c.ContactTitle == "Sales Representative")
  .OrderBy(c => c.CompanyName);
var customers = await query.ExecuteAsync();
```

```
Dim query = manager.Customers _
  .Where(Function(c) c.ContactTitle = "Sales Representative") _
  .OrderBy(Function(c) c.CompanyName)
Dim customers = Await query.ExecuteAsync()
```

and

```
var query = manager.Customers
  .Where(c => c.ContactTitle == "Sales Representative")
  .OrderBy(c => c.CompanyName);
Customer cust = await query.AsScalarAsync().FirstOrNullEntity();
```

```
Dim query = manager.Customers _
  .Where(Function(c) c.ContactTitle = "Sales Representative") _
  .OrderBy(Function(c) c.CompanyName)
```

```
Dim cust as Customer = Await query.AsScalarAsync().FirstOrNullEntity()
```

In the above you may have noticed that instead of calling either the *ExecuteQuery* or *ExecuteQueryAsync* methods on the *EntityManager*, we called execute methods directly on the query itself. **Query extension** methods such as *Execute* and *ExecuteAsync* give you additional flexibility in how you execute a query. These are equivalent to the *EntityManager* methods, except with one important distinction. If a query is created without an *EntityManager* and executed with a query extension method, then that query will fail if you do not assign an *EntityManager*.

You can find more information on asynchronous queries here.

Logging & debugging

The DevForce debug log can be used to see more information regarding which queries are executed and when. Information about every query sent to the EntityServer will be written to the debug log. You can also log the generated SQL for queries using the *shouldLogSqlQueries* attribute on the logging element in your config file.

Learn more

For more information on the **basics** of LINQ queries see 101 LINQ query examples and its corresponding code solution, the Query Explorer.