Contents

In previous topics we've seen how to create a basic LINQ query, how to return part of an entity, how to include related entities, and more. Here are a few more **miscellaneous query tips**.

Create a Query with no EntityManager Attached

You're familiar with the auto-generated query properties in your domain-specific *EntityManager*, that's what you're using whenever you do something like the following:

```
var mgr = new NorthwindIBEntities();
var customerQuery = mgr.Customers;
```

```
Dim mgr = New NorthwindIBEntities()
Dim customerQuery = mgr.Customers
```

These queries are "for" that *EntityManager* instance. If you use one of the query extension methods such as *Execute* or *ExecuteAsync*, the query will be executed by the *EntityManager* on which the query was created.

```
var list = customerQuery.Execute();
```

```
Dim list = customerQuery.Execute()
```

It's often useful to create a query that can be easily used with any *EntityManager* however. Suppose your application requires multiple *EntityManagers* because you need separate editing contexts - separate "sandboxes" - for contemporaneous editing sessions. You know what queries you will need to support the sandbox scenarios. Because you will re-use the query among several *EntityManagers*, you don't want to tie the query to any particular *EntityManager*.

You can easily create a query without an *EntityManager*:

```
EntityQuery<Customer> query = new EntityQuery<Customer>();
```

```
Dim query As New EntityQuery(Of Customer)()
```

You then have a few choices for how you execute this query.

One is to use the query methods on the *EntityManager*:

```
var mgr = new NorthwindIBEntities();
var list = mgr.ExecuteQuery(query);
// ... and on another EM ...
var mgr2 = new NorthwindIBEntities();
var list2 = mgr2.ExecuteQuery(query);
```

```
Dim mgr = New NorthwindIBEntities()
Dim list = mgr.ExecuteQuery(query)
' ... and on another EM ...
Dim mgr2 = New NorthwindIBEntities()
Dim list2 = mgr2.ExecuteQuery(query)
```

You can also use the With extension method to target an *EntityManager*. You can use the *With* method for either an "unattached" query or one created for another *EntityManager*.

```
var query = manager.Customers;
var mgr2 = new NorthwindIBEntities();
var query2 = query.With(mgr2);
```

```
Dim query = manager.Customers
Dim mgr2 = New NorthwindIBEntities()
Dim query2 = query.With(mgr2)
```

If you execute a query without "attaching" it to an *EntityManager* in some way an exception will be thrown.

Existence queries: are there any entities that match?

If you need to determine whether one or more entities meets certain criteria without retrieving the entities the **Any** LINQ operator is a good choice.

```
string someName = "Some company name";
bool rc = manager.Customers.Any(c => c.CompanyName == someName);
```

```
Dim someName As String = "Some company name"
Dim rc As Boolean = manager.Customers.Any(Function(c) c.CompanyName = someName)
```

As an "immediate execution" query, to use this asynchronously you must use AsScalarAsync:

```
string someName = "Some company name";
bool rc = await  manager.Customers.AsScalarAsync().Any(c => c.CompanyName == someName);
```

```
Dim someName As String = "Some company name"
Dim rc as Boolean = Await manager.Customers.AsScalarAsync().Any(Function(c) c.CompanyName = someName)
```

The *Count* operator is also useful here, if instead of returning a boolean you want the total number matching the criteria.

Use FirstOrNullEntity

First, some explanation of *First*. The LINQ *First* operator, in all incarnations, will throw an exception if no items are found. Since you probably don't want your program to terminate for such a simple query, you're usually better off using either the standard LINQ *FirstOrDefault* or the DevForce extension *FirstOrNullEntity*.

FirstOrDefault will return the first item or its default value. For a reference type such as an entity the default value is null (Nothing in VB). It's often easier to work with null entities in DevForce, so if you instead use *FirstOrNullEntity* either the first item matching the selection criteria is returned, or the entity type's null entity.

As an immediate execution query, you must use *AsScalarAsync* to execute this query in asynchronous environments.

```
Employee emp = manager.Employees.FirstOrNullEntity(e => e.City == "Moscow");
// ... or ...
Employee emp = await manager.Employees.AsScalarAsync().FirstOrNullEntity(e => e.City == "Moscow");
```

```
Dim emp As Employee = manager.Employees.FirstOrNullEntity(Function(e) e.City = "Moscow")
' ... or ...
Dim emp as Employee = Await manager.Employees.AsScalarAsync().FirstOrNullEntity(Function(e) e.City = "Moscow")
```

First vs. Single

The LINQ *Single* operator returns the one and only element matching the selection criteria. If multiple elements match the criteria, it throws. If no elements match the criteria, it throws. This isn't some *diabolical DevForce design*, these are the rules of LINQ.

If you do decide to use *Single*, it's usually best to use either *SingleOrDefault* or for async only, the DevForce extension *SingleOrNullEntity*, to ensure that the query won't fail if no item is returned.

First vs. Take(1)

The LINQ *Take* operator is usually used to take one or more items. You can use *Skip* with *Take* to skip items before taking; this is how paging is done.

Take is not an immediate execution query, which can be good news in some environments, and doesn't use *AsScalarAsync* when executed asynchronously. It also always returns an *IEnumerable<T>*, so even a *Take(1)* will return an *IEnumerable* with the element. If no items matched the criteria then an empty enumeration is returned.

Query using an IN clause

If you've searched in vain for the LINQ equivalent to the SQL "In" clause, you can stop worrying. LINQ uses the *Contains* operator to implement a query with search criteria for a value in a list. (This is usually translated to a SQL "In" clause by the Entity Framework when the SQL is generated.)

```
var countryNames = new List<string> {"UK", "France", "Germany"};
var query = manager.Customers
          .Where(c => countryNames.Contains(c.Country));
```

```
Dim countryNames = New List(Of String) From {"UK", "France", "Germany"}
Dim query = manager.Customers.Where(Function(c) countryNames.Contains(c.Country))
```

There is one caveat here, however. Your *contains list* should be a List<T>, where "T" is a numeric type, a string, a DateTime or a GUID. Why this restriction? The list has to meet DevForce's requirements for known types. DevForce will automatically recognize these lists as known types without any extra effort on your part. If you need some other *List* then you will need to ensure it can be used in n-tier deployments. You also can't use an array, for example using string[] above will fail in an n-tier deployment. This is due to an arcane data contract naming issue, so don't say we didn't warn you.

Query with canonical functions

You can use the Entity Framework EntityFunctions class in your server-side code with data source only queries. *EntityFunctions* provides methods which map to canonical functions supported by all database providers.

For SQL Server databases, the SQLFunctions class can be used in server-side code with data source only queries.