Contents

There will be points where the need arises to **intercept and modify a query while it is executing**. DevForce provides an extension method, *Filter()*, that can be used to superimpose one or more independently defined filter conditions upon an existing query.

Filter extension method

Filter() differs from *Where()* in two ways

- 1) Filters can be safely applied to queries that do not 'fit' the type of the query being processed. In these cases the Filter is simply ignored.
- 2) Filters operate on any subselects within the query in addition to result type of the query. Where clause's operate in a chained fashion on whatever result type immediately precedes them.

Filter()'s primary motivating use case is the need to apply query interception, either client or server-side, to submitted queries; although it is perfectly possible to use it in other contexts. In the case of query interception, there is a need to work with 'untyped' queries, or queries where both the return type as well as the source type of a query is very likely unknown at runtime, so there is a need to be able to apply a possibly large number of 'Filter's to any query that comes though and only have the 'applicable' filters apply to each query. Additionally, there is often a need to modify 'intermediate' portions of the query in addition to just restricting its final result.

For example, suppose your application's database includes data for customers worldwide, but that a given Sales Manager only works with data for customers from his region. Instead of baking the region condition into every query for Customers throughout your application, you could implement an *EntityServerQueryInterceptor* that imposes the condition upon any query for customers made while that Sales Manager is logged in.

The usefulness of *Filter()* becomes even more apparent when you need to apply filters in a global way for more than one type.

There are several overloads of *Filter()*, one that takes a *Func<T>* and another that takes an *EntityQueryFilterCollection* (each of whose members is a *Func<T>*). Let's look at some examples:

```
var query = _em1.Territories.Where(t => t.TerritoryID > 100);
var newQuery = query.Filter((IQueryable<Territory> q) =>
  q.Where(t => t.TerritoryDescription.StartsWith("M")));
```

```
Dim query = _em1.Territories.Where(Function(t) t.TerritoryID > 100)
Dim newQuery = query.Filter(Function(q As IQueryable(Of Territory)) _
  q.Where(Function(t) t.TerritoryDescription.StartsWith("M")))
```

In this example we have used the overload of Filter which takes as its argument a Func delegate. Said delegate takes an IQueryable<T> -- essentially a list of items of type T – and returns an IQueryable<T>. The IQueryable<T> that goes in is the one defined by the variable *query*, defined as

```
_em1.Territories.Where(t => t.TerritoryID > 100)
```

```
_em1.Territories.Where(Function(t) t.TerritoryID > 100)
```

The one that comes out is the one that went in minus those Territories whose Description property value begins with the letter "M".

In the first example, above, our filter applies to the query's root type, Territory. We aren't limited to that: we can also apply filters to other types used in the query. Consider the following:

```
var q1 = _em1.Customers.SelectMany(c => c.Orders
  .Where(o => o.ShipCity.StartsWith("N")));
var q1a = q1.Filter((IQueryable<Order> q) =>
  q.Where(o => o.FreightCost > maxFreight));
```

```
Dim q1 = _em1.Customers.SelectMany(Function(c) c.Orders. _
  Where(Function(o) o.ShipCity.StartsWith("N")))
Dim q1a = q1.Filter(Function(q As IQueryable(Of Order)) q. _
  Where(Function(o) o.FreightCost > MaxFreight))
```

The root type for this query is *Customer*, but the query projects *OrderSummaries* as its output, and it is against *OrderSummaries* that we apply our filter. This time the filter imposes a condition upon the values of the *OrderSummary.Freight property*. Without the filter we would have retrieved all *OrderSummaries* having a *ShipCity* whose name begins with "N"; with the filter, not only must the name begin with "N", but the Freight property value must exceed the value *maxFreight*.

Let's look at another example of filtering one some type other than the query's root type:

```
var q1 = _em1.Customers.Where(c => c.Orders.Any
  (o => o.ShipCity.StartsWith("N")));
var q1a = q1.Filter((IQueryable<Order> q) =>
  q.Where(o => o.FreightCost > maxFreight));
{{/code}}
```

```
Dim q1 = _em1.Customers.Where(Function(c) c.Orders.Any _
  (Function(o) o.ShipCity.StartsWith("N")))
Dim q1a = q1.Filter(Function(q As IQueryable(Of Order)) q. _
  Where(Function(o) o.FreightCost > MaxFreight))
```

In the absence of the filter, the above query would retrieve Customer objects: specifically, Customers having at least one Order whose *ShipCity* begins with the letter "N". The filter potentially reduces the set of Customers retrieved by imposing an additional condition on their related *OrderSummaries* (again, on the value of their Freight property).

Now let's look at a use of *Filter()* involving conditions on more than a single type.

```
var eqFilters = new EntityQueryFilterCollection();
eqFilters.AddFilter((IQueryable<Customer> q) =>
  q.Where(c => c.Country.StartsWith("U")));
eqFilters.AddFilter((IQueryable<Order> q) =>
q.Where(o => o.OrderDate < new DateTime(2009, 1, 1)));
var q0 = _em1.Customers.Where(c => c.Orders.Any
  (o => o.ShipCity.StartsWith("N")));
var q1 = q0.Filter(eqFilters);
```

```
Dim eqFilters = New EntityQueryFilterCollection()
eqFilters.AddFilter(Function(q As IQueryable(Of Customer)) q. _
  Where(Function(c) c.Country.StartsWith("U")))
eqFilters.AddFilter(Function(q As IQueryable(Of Order)) q. _
  Where(Function(o) o.OrderDate < New Date(2009, 1, 1)))
Dim q0 = _em1.Customers.Where(Function(c) c.Orders.Any( _
  Function(o) o.ShipCity.StartsWith("N")))
Dim q1 = q0.Filter(eqFilters)
```

In the above snippet, we instantiate a new *EntityQueryFilterCollection*, to which we then add two individual filters, each of which is a *Func<T>*. The first filter added imposes a condition on the Customer type; the second imposes a condition on the *OrderSummary* type. Note that we could now apply these filters to any query whatsoever. If the targeted query made use of the Customer type, the condition on Customers would apply; if it made use of the *OrderSummary* type, the condition on *OrderSummaries* would apply. If it made use of both, as does our example q0, both conditions would apply.

A filter is also applied directly to any clause of a query that returns its targeted type. Thus, the effect of the two filters defined above, applied against query q0, is to produce a query that would look like the following if written conventionally:

```
var q0 = _em1.Customers
  .Where(c => c.Country.StartsWith("U"))
  .Where(c => c.Orders
  .Where(o => o.OrderDate < new DateTime(2009, 1, 1))
  .Any(o => o.ShipCity.StartsWith("N")));
```

```
Dim q0 = _em1.Customers. _
  Where(Function(c) c.Country.StartsWith("U")). _
  Where(Function(c) c.Orders. _
  Where(Function(o) o.OrderDate < New Date(2009, 1, 1)). _
  Any(Function(o) o.ShipCity.StartsWith("N")))
```