Contents

You can write a **specialized named query** to reflect a particular business intent such as retrieving "gold customers."

A specialized named query typically supplements the default named query. It differs from the default named query in that it tends to restrict the range of queryable entities to a specific subset that represents an application domain concept. The **default** named query for *Customer* represents all customers. A **specialized** named query called *GetGoldCustomers* likely returns a highly-prized subset of those customers.

You can write as many specialized named queries as you need.

An example

Suppose your business makes special offers for "gold customers". While you could specify what makes a customer golden in the client query, for some reason you prefer to make that determination on the server. Let's write that specialized named query in a named query provider class:

```
// Customers who spent more than 10,000 this year
public IQueryable<Customer> GetGoldCustomers() {
    var thisYear = new DateTime(DateTime.Today.Year, 1, 1);
  return new EntityQuery<Customer>()
            .Where(c => 10000 < c.Orders
               .Where(o => o.OrderDate >= thisYear)
               .Sum(o => o.OrderDetails.Sum(od => od.Quantity * od.UnitPrice)));
}
```

```
' Customers who spent more than 10,000 this year
Public Function GetGoldCustomers() As IQueryable(Of Customer)
 Dim thisYear = New Date(Date.Today.Year, 1, 1)
 Return New EntityQuery(Of Customer)().Where(Function(c) 10000 < c.Orders _
   .Where(Function(o) o.OrderDate >= thisYear).Sum(Function(o) o.OrderDetails _
   .Sum(Function(od) od.Quantity * od.UnitPrice)))
End Function
```

All named query methods return an *IQueryable* or an *IEnumerable* of an entity type. *GetGoldCustomers* returns an *IQueryable* of *Customer*.

You could invoke this query with the following client statements:

```
query-B-GoldCustomers = new EntityQuery<Customer>("GoldCustomers")
                    .Where(c => c.StartsWith("B"));
myEntityManager.ExecuteQueryAsync(query-B-GoldCustomers, queryCallback);
```

```
query-B-GoldCustomers = New EntityQuery(Of Customer)("GoldCustomers") _
  .Where(Function(c) c.StartsWith("B"))
myEntityManager.ExecuteQueryAsync(query-B-GoldCustomers, queryCallback)
```

The named query's *EntitySet* name

Notice that the *EntitySet* name provided to the *EntityQuery* constructor is "GoldCustomers".

The *Customer* entity type's true *EntitySet* name is "Customers", not "GoldCustomers". "Customers" refers to all *Customer* entities in the data source. Only some of those customers are "gold customers." You can think of "GoldCustomers" as a subset of the "Customers" *EntitySet*.

As a practical matter, DevForce uses the "GoldCustomers" name to find the corresponding query method on the server. The *EntityServer* applies the DevForce query naming conventions to locate the query method, stripping the "Get" prefix from the *GetGoldCustomers* method name and matching the remaining "GoldCustomers" text to the *EntitySet* name in the client *EntityQuery*.

You can use the *Query* attribute to identify a query method that does not conform to the conventions as explained in the *query naming conventions* topic.

If the client query asks for a specialized named query, the *EntityServer* must be able to find the corresponding query method on the server; it will throw an exception if it can't find the method.

Merging the named query with the original client query

DevForce merges the client query with the named query by copying the LINQ clauses of the client query LINQ to the output of the named query method. The "re-composition" works something like this:

```
mergedQuery = GetGoldCustomers().Where(c => c.StartsWith("B"));
```

```
mergedQuery = GetGoldCustomers().Where(Function(c) c.StartsWith("B"))
```

The re-composed query returns "B" customers who spent $10,000 during the current calendar year.

Create a property for the named query

Notice the hard-coded the "GoldCustomers" string in the client query. That is a risky practice. It's a better idea to hide that magic string inside a dedicated, query-producing property or method.

The DevForce code generator does something similar when it creates *EntityQuery* factory properties for the entity model's custom *EntityManager*. The *Customers* property, shown here, is typical:

```
using IbEm = IdeaBlade.EntityModel;
...
public partial class NorthwindManager : IbEm.EntityManager {
    ...
   public IbEm.EntityQuery<Customer> Customers {
     get { return new IbEm.EntityQuery<Customer>("Customers", this); }
    }
    ...
}
```

```
Imports IbEm = IdeaBlade.EntityModel
...
Partial Public Class NorthwindManager
        ...
Inherits IbEm.EntityManager
  Public ReadOnly Property Customers() As IbEm.EntityQuery(Of Customer)
Get
Return New IbEm.EntityQuery(Of Customer)("Customers", Me)
End Get
         ...
  End Property
End Class
```

Notice that DevForce generated *NorthwindManager* as a partial class. You can extend it with *EntityQuery* factory properties for your specialized named queries as illustrated by this *GoldCustomers* property:

```
public partial class NorthwindManager {
    ...
   public IbEm.EntityQuery<Customer> GoldCustomers {
     get { return new IbEm.EntityQuery<Customer>("GoldCustomers", this); }
    }
    ...
}
```

```
Partial Public Class NorthwindManager
  ...
 Public ReadOnly Property GoldCustomers() As IbEm.EntityQuery(Of Customer)
   Get
     Return New IbEm.EntityQuery(Of Customer)("GoldCustomers", Me)
   End Get
 End Property
  ...
End Class
```

The DevForce *code generator* will write this property for you in a future release.

Now you can write a *GoldCustomers* query in your application code in the same manner as you would write a regular *Customer* query:

```
customersQuery = myEntityManager.Customers.Where(...);
```

```
...
goldCustomersQuery = myEntityManager.GoldCustomers.Where(...);
```

```
customersQuery = myEntityManager.Customers.Where(...)
...
goldCustomersQuery = myEntityManager.GoldCustomers.Where(...)
```

Specialized named queries in OData

You can use OData to invoke specialized named queries as long as you create a custom *EntityManager* factory property for that query as just explained.

You don't have to do anything to support OData queries that depend on the default named query. DevForce generated their factory properties automatically.

Specialized named queries are *DataSourceOnly*

A query rooted in a specialized named query must be executed using the *DataSourceOnly* *QueryStrategy*.

DevForce converts the default *Optimized QueryStrategy* to *DataSourceOnly* automatically.

The query is never remembered in the query cache, it ignores entities in cache, and it always fails if executed offline.

A query that is rooted in the default named query, on the other hand, can be remembered in the query cache, can be applied to the cache, and can execute even when the application is offline. Why the difference?

The default named query is associated with the entity type's *EntitySet*. That *EntitySet* represents the unrestricted view of all instances of the entity type. DevForce adopts this interpretation whether or not you've written a default named query, even if your implementation of the default named query in fact restricts query results to a handful of the possible entities.
Because DevForce assumes that the default root query embraces all entities of the type, it acts as if every entity in cache is fair game.

Please keep these semantics in mind when you write a default named query. The client application should be able to assume that a query for all Customers returns every Customer that the current user would ever be allowed to see.

DevForce gives the opposite interpretation to a specialized named query. Presumably you wrote a specialized named query for two reasons: (1) to limit the range of possible query results to a subset of the entity domain and (2) to hide that limiting logic from the client.

If you had wanted the client to know how it worked, you would have written the query as a client query. Instead, you chose to keep the logic a secret from the client by writing a specialized named query on the server. From the client perspective, a specialized named query is a black box.

DevForce cannot confidently reproduce the server-side behavior of the query on the client and does not try. DevForce concludes that the query must be executed on the server and only on the server. An attempt to force an *EntityManager* to evaluate the query on the client must fail with an exception.

Specialized named queries can take parameters

You can pass parameters to a specialized named query. This topic explains how and why.