Contents

You can define a **specialized named query** **that accepts parameters** to constrain the behavior of the named query.

Here is a parameterized named query that retrieves *Customer*s whose names begin with a prefix string.

```
public IQueryable<Customer> GetCustomersStartingWith(string prefix)
{
  return new EntityQuery<Customer>().Where(c => c.Name.StartsWith(prefix));
}
```

```
Public Function GetCustomersStartingWith(ByVal prefix As String) As IQueryable(Of Customer)
  Return New EntityQuery(Of Customer)().Where(Function(c) c.Name.StartsWith(prefix))
End Function
```

The client provides the prefix by adding a parameter to the client query:

```
var query = IbEm.EntityQuery<Customer>("CustomersStartingWith", myEntityManager);
query.Parameters.Add(new EntityQueryParameter(prefix));
```

```
Dim query = IbEm.EntityQuery(Of Customer)("CustomersStartingWith", myEntityManager)
query.Parameters.Add(New EntityQueryParameter(prefix))
```

Notice the *EntitySet* name, “CustomersStartingWith”, that corresponds to the specialized named query on the server.

These two lines make sense after close study but they are not easily grasped at a glance. It’s a good practice to wrap them in a method that you add to the model’s custom *EntityManager*, as illustrated here:

```
public partial class NorthwindManager {
   ...
  public IbEm.EntityQuery<Customer> GetCustomersStartingWith(string prefix)
   {
      var query = IbEm.EntityQuery<Customer>("CustomersStartingWith", this);
      query.Parameters.Add(new EntityQueryParameter(prefix));
     return query;
   }
   ...
}
```

```
Partial Public Class NorthwindManager
  Public Function GetCustomersStartingWith(ByVal prefix As String) _
    As IbEm.EntityQuery(Of Customer)
    Dim query = IbEm.EntityQuery(Of Customer)("CustomersStartingWith", Me)
     query.Parameters.Add(New EntityQueryParameter(prefix))
    Return query
  End Function
End Class
```

Now you can write a simple *GetCustomersStartingWith* query in your application code:

```
query = myEntityManager.CustomersStartingWith("B").Where(...);
```

```
query = myEntityManager.CustomersStartingWith("B").Where(...)
```

Multiple parameters, multiple overloads

A parameterized named query can have as many parameters as you require. Be sure that you add parameters to the client query that are of the same type and in the same order as the parameters of the named query method on the server.

You can specify the name of the parameter in the constructor of the *EntityQueryParameter*. That eases analysis of the query within your custom *EntityServerQueryInterceptor* but it doesn’t influence how DevForce matches the client query to the named query on the server. The .NET reflection API doesn't reveal method parameter names.

If you write two specialized named queries with the same method name, DevForce attempts to pick the best-fit overload by matching the parameters in the client query with the parameters of the candidate query methods.

When to use parameterized named queries

You may not need parameterized queries. It is usually easier and clearer to write a client-side query that incorporates the would-be-parameter information in the LINQ clauses.

Why not get the "B" customers with this garden variety *Customer* query?

```
query = myEntityManager.Customers.Where(c=>c.Name.StartsWith("B") && ...);
```

```
query = myEntityManager.Customers.Where(Function(c) c.Name.StartsWith("B") AndAlso ...)
```

When the *EntityServer* receives this query, it copies the LINQ clauses from the original client query to the output of the named query method as if you had written:

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

```
GetCustomers().Where(Function(c) c.StartsWith("B")
```

This is efficient and effective when the named query method (*GetCustomers* in this case) returns an *IQueryable*.

However, this approach can be extremely inefficient if the named query method returns an *IEnumerable* instead of an *IQueryable*. Such a method might cause the server to fetch an enormous number of entities from the database when, in fact, the client only wants a small subset of those entities.
True, the LINQ clauses from the client query reduce the transmitted results to just the entities the client requested. But this reduction occurs in server memory after the full set of entities had been retrieved and the damage done.

It's usually best to write named queries that return *IQueryable* when you can. Sometimes you can't. For example, you might have to implement the named query with Entity SQL (ESQL); ESQL queries return *IEnumerable*.

Here is a parameterized named query that uses DevForce ESQL to get *Customer*s whose names begin with a prefix string:

```
public IEnumerable<Customer> GetCustomersStartingWith(string prefix) {
    var param = new QueryParameter("prefix", prefix);
    var paramEsql = new ParameterizedEsql(
      "SELECT VALUE c FROM Customers AS c WHERE c.NAME LIKE '@prefix%'", param);
    var query = new PassthruEsqlQuery(typeof(Customer), paramEsql);
    var results = query.Execute().Cast<Customer>();
   return results;
}
```

```
Public Function GetCustomersStartingWith(ByVal prefix As String) As IEnumerable(Of Customer)
  Dim param = New QueryParameter("prefix", prefix)
  Dim paramEsql = New ParameterizedEsql( _
        "SELECT VALUE c FROM Customers AS c WHERE c.NAME LIKE '@prefix%'", param)
  Dim query = New PassthruEsqlQuery(GetType(Customer), paramEsql)
  Dim results = query.Execute().Cast(Of Customer)()
 Return results
End Function
```

The client invokes this server-side parameterized query method using the same custom client-side *myEntityManager.GetCustomersStartingWith* method we showed earlier:

```
query = myEntityManager.CustomersStartingWith("B").Where(...);
```

```
query = myEntityManager.CustomersStartingWith("B").Where(...)
```

The prefix parameter contributes to the ESQL query, reducing the number of *Customer*

Limitations

The default named query method cannot take parameters.

All parameterized named queries are "specialized" which means they are not remembered in the query cache, are not applied to entities in cache, and will fail if executed while the application is offline.

You cannot refer to a parameterized named query in an OData query.