Contents

In order to provide **support for POCO queries** from your client-side *EntityManager*, a *POCO Service Provider* must be defined. The service provider contains the query methods which will access the data source.

Sample provider

The example below shows a simple POCO Service Provider which provides the ability to perform client-side queries against a collection of *State* POCO entities sourced from an XML file.

```
namespace DomainModel {
  [EnableClientAccess]
 public class PocoServiceProvider {
   public IEnumerable<State> GetStates() {
     if ( _states == null) {
       _states = ReadStatesData("states.xml");
      }
     return states;
    }
   private static IEnumerable<State> ReadStatesData(string embeddedFileName) {
      ...
    }
   private static List<State> _states = null;
  }
}
```

```
Namespace DomainModel
  <EnableClientAccess>
 Public Class PocoServiceProvider
   Public Function GetStates() As IEnumerable(Of State)
     If _states Is Nothing Then
       _states = ReadStatesData("states.xml")
     End If
     Return states
   End Function
   Private Shared Function ReadStatesData(ByVal embeddedFileName As String) As IEnumerable(Of State)
...
   End Function
   Private Shared _states As List(Of State) = Nothing
 End Class
End Namespace
```

Convention vs. attributes

DevForce will discover service provider query methods which correspond to entity queries by either convention-based method naming or by explicit attribute markup.

Finding providers by convention

The *GetStates()* method retrieves the data requested by a query (typically a query from the client).

The method name is important. It follows the same query naming conventions that DevForce uses to locate named query methods. In brief, DevForce looks for a method whose name begins with a known prefix and is followed by an *EntitySet*name. The *EntitySet* name can be anything that makes sense to you. Most developers use the plural of the POCO entity type name.

Here are the recognized prefixes:

- **Get**
- **Query**
- **Fetch**

- **Retrieve**
- **Find**
- **Select**

The *GetStates* method conforms to the convention: "Get" is the prefix; "States" is the *EntitySet* name.

The following query will be routed to the *GetStates* server-side query method when executed:

```
var statesQuery = new EntityQuery<State>("States", anEntityManager);
```

```
Dim statesQuery = New EntityQuery(Of State)("States", anEntityManager)
```

The Query attribute alternative

The server method doesn't have to follow the naming convention. It can be named as you please as long as it's adorned with the *Query* attribute.

```
[Query]
public IEnumerable<State> ReturnAllStates() {
  IEnumerable<State> states = ReadStatesData("states.xml");
return states;
}
```

```
<Query()> _
Public Function ReturnAllStates() As IEnumerable(Of State)
 Dim states As IEnumerable(Of State) = ReadStatesData("states.xml")
 Return states
End Function
```

You must adjust the client-side query to specify the full method name as the *EntitySet* name:

```
var statesQuery = new EntityQuery<State>("ReturnAllStates", anEntityManager);
```

```
Dim statesQuery = New EntityQuery(Of State)("ReturnAllStates", anEntityManager)
```

Query methods with parameters

Methods with parameters are supported as well. For example, an additional overload to the *GetStates()* method above that only returns states with a population size greater than a size passed in might be written as shown below. (Naturally, this would require that the internal *ReadStatesData()* method be modified as well.)

```
public IEnumerable<State> GetStates(long minPopulationSize) {
  IEnumerable<State> states = ReadStatesData("states.xml",
    minPopulationSize);
return states;
}
```

```
Public Function GetStates(ByVal minPopulationSize As Long) _
 As IEnumerable(Of State)
 Dim states As IEnumerable(Of State) = ReadStatesData("states.xml", _
    minPopulationSize)
 Return states
End Function
```

On the client side, parameters may be specified by using one of the *EntityQuery.AddParameter()* overloads. The following snippet calls the parameterized *GetStates()* method to return just those states with a population greater than one million people:

```
var query = new EntityQuery<State>("States", anEntityManager);
query.AddParameter(1000000);
```

```
Dim query = New EntityQuery(Of State)("States", anEntityManager)
query.AddParameter(1000000)
```

Any number of query parameters are permitted, and the standard .NET overload resolution rules apply. This means that the order and type of the parameters are checked to find appropriate matches, with type coercion occurring as required.

Parameterized query methods vs. LINQ

The following two queries will return the same results:

```
var query1 = new EntityQuery<State>("States", anEntityManager);
query.Where(state => state.Population > 1000000);
```

```
var query2 = new EntityQuery<State>("States", anEntityManager);
query.AddParameter(1000000);
```

```
Dim query1 = New EntityQuery(Of State)("States", anEntityManager)
query.Where(Function(state) state.Population > 1000000)
Dim query2 = New EntityQuery(Of State)("States", anEntityManager)
query.AddParameter(1000000)
```

Furthermore, in **both** cases, the restriction to those states with greater than one million population occurs on the **server,** not the client. So the question arises: is one to be preferred over the other? The answer usually depends upon how the server-side method itself is implemented.

In general, unless the server-side method can *internally* use the query parameter *to restrict its own query against some back-end data store*, query parameters have no advantage over LINQ query restrictions. In fact, LINQ queries are far more flexible and intuitive to work with under most circumstances. Nevertheless, there will be cases where a back-end data store's ability to optimize some queries will yield sufficient performance improvement to justify the use of query parameters.

For example, consider the Windows file system's ability to search for files, given a path and wildcards. While the same result could be accomplished via a server-side method that returned all of the files in the file system and then iterated over them to locate a desired set of files, it would likely be faster to call the file system directly with the path and wildcard restrictions provided via query parameters.

Returning IQueryable vs. IEnumerable

The *GetStates* POCO query returns an *IEnumerable* of *State*. That's fine for this example because there are only 50 state objects and they are sourced from an XML file. The impact of a State query on the server is small whether the client asks for one state or all states.

But what if the POCO is of type *Order* and we source orders from a database. If *GetOrders* returns an IEnumerable<Order>, every time the client queries for orders the server retrieves every order in the database. There could be millions of orders. Every client query would force the server to retrieve those millions of orders.

Even a filtered client query such as

```
var ordersQuery = new EntityQuery<Order>("Orders", anEntityManager).Where(o => o.OrderID == 42);
```

```
Dim ordersQuery = New EntityQuery(Of Order)("Orders", anEntityManager).Where(function (o) o.OrderID = 42)
```

causes the server to first retrieve every order from the database and then filter down to the one order with ID of 42. That's not going to perform well!

Suppose the Order data source can be queried with an *IQueryable* form of LINQ. Raw Entity Framework can do it with LINQ-to-Entities; there is LINQ for NHibernate.

You then write an *IQueryable<Order> GetOrders()* query method on the server that returns the LINQ query. DevForce appends the client's "Where" clause as before ... but now, when the query is executed on the server, the query is composed such that the data source *only returns the one order*.

The difference between *IEnumerable* and *IQueryable* bears repeating. Both example queries return a single order to the client. The supporting *IQueryable* server-side query method retrieves only *one order* from the database; the *IEnumerable* server-side query method retrieves *every order* from the database before filtering down to the one desired order. That's a huge performance - and scaling - improvement.

It pays to write POCO query methods that return *IQueryable<T>* when possible.

Include syntax is not supported for POCO queries

The *Include()* syntax on a POCO entity query is not currently implemented. The call will compile but will not do anything.