Contents

A **named query method** returns either an ***IQueryable*** **or an** ***IEnumerable*** of an entity type. This topic explains the implications of each choice.

The query that the DevForce *EntityServer* actually processes is the result of copying LINQ clauses from the original client query and attaching them to the output of the named query method. The output of the named query can be either an *IQueryable* or an *IEnumerable* of an entity type. The difference is important.

What's the difference?

An ***IQueryable*** is an expression tree. The LINQ clauses from the original client query are added to that expression tree, effectively redefining the query *before* it is executed

An ***IEnumerable***, on the other hand, is a delegate, an impenetrable function. The LINQ clauses added from the original client query take effect *after* the delegate has been executed. Those clauses filter, sort, group, and page the in-memory entities produced by the delegate.

An IQueryable that is also an EntityQuery<T> can also have "Include" operations performed against it. The "Includes" can be performed on either the client or the server in this case. Note, however, that IEnumerables and IQueryables that are not also instances of an EntityQuery<T> do not support using the "Include" operator.

Performance implications

The difference can profoundly affect performance on the server and on the database. An example may help clarify. Imagine …

- there are 10,000 customers in the database.
- only 100 of them have names that begin with the letter “B”.
- the client send a query that retrieves only the “B” customers.
- you’ve written a default named query called *GetCustomers()*.

The *EntityServer* will merge the client and named queries yielding something akin to this:

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

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

Suppose the *GetCustomers* named query method returns *IQueryable*

```
public IQueryable<Customer> GetCustomers() {
  return new EntityQuery<Customer>();
}
```

```
Public Function GetCustomers() As IQueryable(Of Customer)
  Return New EntityQuery(Of Customer)()
End Function
```

When the re-composed query executes, it will retrieve 100 *Customer* entities from the database.

Suppose instead that the *GetCustomers* named query method returns *IEnumerable* as in this contrived example:

```
public IEnumerable<Customer> GetCustomers() {
  return new EntityQuery<Customer>().ToList();
}
```

```
Public Function GetCustomers() As IEnumerable(Of Customer)
  Return New EntityQuery(Of Customer)().ToList()
End Function
```

The merged query has not changed. It still has a *Where* clause that filters for the “B”s. But this time, the server fetches all 10,000 customers from the database thanks to the *ToList()* method. Only after the 10,000 customers arrive on the server will the *Where* clause kick in and filter the results down to the 100 customers the client actually wanted.

The client receives only 100 customers with either *GetCustomers* implementation. It neither knows nor cares how the server did its job. But with the *IEnumerable* version of *GetCustomers*, the *EntityServer*

When to use IEnumerable

Clearly *IEnumerable* is a poor choice for *Customer* queries. It might be a good choice for small, stable entity lists.

Imagine that your business is very popular. Many clients are constantly online, badgering the server repeatedly with requests for your product catalog. Your catalog has only 100 highly sought after items that don't change very often … perhaps a few times per day. You'd like to avoid the pressure of unnecessary trips to the database by caching the product list on the *EntityServer*. You can do that with a named *Product* query.

```
public IEnumerable<Product> GetProducts() {
  return ThreadSafeProductCatalog.GetLatestList(); // periodically self-updates
}
```

```
Public Function GetProducts() As IEnumerable(Of Product)
  Return ThreadSafeProductCatalog.GetLatestList() ' periodically self-updates
End Function
```

Consider this different scenario. You have a specialized named query method that can't be implemented as an *IQueryable*. For some reason you had to implement it with Entity SQL (ESQL); ESQL queries return *IEnumerable*.

Here is a contrived DevForce ESQL version of *GetCustomersStartingWithB*

```
public IEnumerable<Customer> GetCustomersStartingWithB() {
     var query = new PassthruEsqlQuery(
       typeof(Customer),
       "SELECT VALUE c FROM Customers AS c WHERE c.NAME LIKE 'B%'");
     var results = query.Execute().Cast<Customer>();
    return results;
}
```

```
Public Function GetCustomersStartingWithB() As IEnumerable(Of Customer)
 Dim query = New PassthruEsqlQuery(GetType(Customer), _
        "SELECT VALUE c FROM Customers AS c WHERE c.NAME LIKE 'B%'")
 Dim results = query.Execute().Cast(Of Customer)()
Return results
End Function
```

See the parameterized named queries topic for an example of a parameterized ESQL named query.