Contents

Write a **named query** method **on the server** when you want explicit control over how the server interprets a LINQ query received from the client.

Named queries make it easy to dictate the server-side implementation of a query. You can write several named queries for a particular entity type which means you can offer clients a menu of query alternatives that each express a distinctive intent.

A named query is a convenient way to consolidate in a single method the logic to:

- authorize the query
- modify the query, perhaps by adding filters or limiting the amount of data returned
- add *Include* clauses that package related entities in the query result

Named queries are methods that return an IQueryable or an IEnumerable of an entity type. The type can be any *serializable* class produced by almost any means imaginable. Most named queries will involve Entity Framework entity types produced by an Entity Framework query.

There are two basic kinds of named query: the default named query and specialized named queries.

The default named query

The **default named query** is a named query that represents all entities of a particular type in the database. The signature of the default named query for the *Customer* entity type might be as follows:

```
public IQueryable<Customer> GetCustomers();
```

```
public IQueryable(Of Customer) GetCustomers()
```

Various *GetCustomers* implementations are considered in the "default named query" topic. For the moment it's worth noting that

- the method returns an *IQueryable* (or *IEnumerable*) of a specific type
- there are no parameters
- the method name consists of a "*Get*" prefix followed by an *EntitySet* name, usually the plural form of the entity type name; you'll learn how to name a query method elsewhere.

When the *EntityServer* receives a *Customer* client query that corresponds to this named query, it builds a new query by copying the client query LINQ clauses and adding them to the output of the named query method.

All named queries are *composable* in this way. For example, DevForce translates a client query for *Customers beginning with the letter 'B'* such as this one:

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

```
myEntityManager.Customers.Where(Function(c) c.StartsWith("B")
```

into something like:

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

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

The *EntityServer*

Default named queries are optional

You *do not have* to write a default named query method. When the DevForce *EntityServer* receives a client query and there is no default named query method on the server, DevForce processes it directly using its own default *get-all-entities* query as the root.

If you decide to write a default named query, you can write only one per entity type.

Specialized named queries

The developer may wish to write one or more specialized named query methods for a particular entity type. Each method is associated with a particular business purpose and, accordingly, returns a different set of entities. For example, you might have one specialized named query for "*Gold Customers*" and another specialized named query returning "*the current user's Western Region Customers*".

Specialized named query methods can take parameters, unlike the default named query which must be parameterless.

The *EntityServer* must be able to find the specialized named query method if the client query asks for it.

Write a class for the named query methods

Named queries are instance methods of a class. DevForce can find them if the class resides in a discoverable assembly.

There isn't much special about that class and you can have as many such classes as you like. There are only a few requirements.

1. it must be public
2. it must have a default constructor (or no constructor)
3. it must be adorned with the *EnableClientAccess* attribute.
4. it must reside in an assembly that the DevForce *EntityServer* will discover.

Please keep your named query provider classes stateless if possible. If it must have state, give great care to ensuring safe concurrent access to that state.

Please avoid putting anything in this class other than what is strictly necessary to achieve its purpose. The named query provider class is a poor choice for a grab-bag of server-side features.

Here is an example of a provider class.

```
[EnableClientAccess]
public class NamedQueryServiceProvider {
 public IQueryable<Customer> GetCustomers() {...}
  }
```

```
<EnableClientAccess>
Public Class NamedQueryServiceProvider
 Public Function GetCustomers() As IQueryable(Of Customer)
  ...}
 End Function
```

You must put this *NamedQueryServiceProvider* class in an assembly deployed on the server. The domain model project would do; the domain model assembly is always discoverable and deployed to the server. If you do not want the named query methods exposed on the client, add it to a separate, full .NET class library project that you only deploy to the server.

Assembly discovery is discussed *here*.

Named query security

We trust you are authenticating clients before you accept their queries. Even so, you may be uncomfortable simply executing every query a client sends you, even if the client has been authenticated. You may prefer to inspect, modify, and possibly reject a query.

The easiest approach is to add query security attributes to the method definitions. The DevForce *EntityServerQueryInterceptor* is affords the most complete and custom control over the query and its interpretation. You can inherit from our base version of that class and override one or more of its virtual methods. The DevForce *EntityServer* calls these methods as it processes the query. Your overrides can terminate the query, replace or modify the query, or even alter the query results.

Such an interceptor remains the most powerful query management mechanism in DevForce. All queries – named and unnamed – pass through the interceptor. Anything you can do in a named query you can do in the interceptor. Many developers will combine named queries and query interception for a balance of convenience and power.

Learn more

Other subordinate topic pages explain in greater detail how to write named queries and use them appropriately.