Contents

The **default named query** is a server-side query method that represents the entire *EntitySet* for a given entity type. In this topic we discuss what a default named query is and how to write one.

The default named query is a server-side query method that represents the entire *EntitySet* for a given entity type. For example, the *EntitySet* for the Customer entity type is called “Customers”. When a client submits a query that is rooted in the “Customers” *EntitySet* and you’ve written a named query method for that *EntitySet*, such as *GetCustomers*, then that method is the **default named query**.

Conceptually the default named query means “*return to the client every instance of this type.*” You write this method on the server. Whether it in fact returns every instance of the type is up to you.

The following discussion begins with a client query for *Customers* when there is no named query. It identifies the root query to which you add filters, ordering, grouping, and selection. Then it shows two examples of a default named query that DevForce *EntityServer* substitutes for the root query when processing a query from the client.

A separate topic explains how to write a specialized named queries that can address a narrower business purpose.

A simple client query

Here is a query that you might write on the client to retrieve all customers in the database.

```
query = myEntityManager.Customers; // uses the generated query property
```

```
query = myEntityManager.Customers ' uses the generated query property
```

Customers is a factory property that returns an *EntityQuery* for *Customer* entities. The DevForce code generator added this property to the model’s custom *EntityManager* for your convenience. When you examine the generated code file, you’ll see that it was implemented as follows:

```
query = new EntityQuery<Customer>("Customers", myEntityManager);
```

```
query = New EntityQuery(Of Customer)("Customers", myEntityManager)
```

Locate the EntityQueries region of the generated entity class file.

The string, “Customers”, is the name of the *EntitySet* associated with the *Customer* entity type. DevForce interprets this to mean “the unrestricted set of all customers”.

The query is also associated with a particular *EntityManager*, the one in the *myEntityManager* variable. That means the query can be executed directly by writing something like this

```
query.ExecuteAsync(queryCallBack); // get the results in the callback
```

```
query.ExecuteAsync(queryCallBack) ' get the results in the callback
```

We don’t have to specify the *EntityManager* when we create the query. We might want to re-use the query with different *EntityManager* instances in which case we can write.

```
query = new EntityQuery<Customer>("Customers");
em1.ExecuteQueryAsync(query, queryCallback);
em2.ExecuteQueryAsync(query, queryCallback);
```

```
query = New EntityQuery(Of Customer)("Customers")
em1.ExecuteQueryAsync(query, queryCallback)
em2.ExecuteQueryAsync(query, queryCallback)
```

Query Root

The client query examples we’ve seen so far return all customers in the database. Of course you can restrict the query further by adding *Where* clauses such as one that returns only customers beginning with the letter “B”.

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

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

Both the original query and this filtered query begin with the same **root**, the *EntityQuery("Customers")*. This is the root of the query no matter what LINQ clauses are added, no matter what types are ultimately returned.

"Customers" is the name of the *EntitySet* that includes all *Customer* entities in the database. It defines the **default query**. Keep your eye on the root and the *EntitySet* name. If you create a *specialized named query* (as described elsewhere), you will create a different root that uses a different *EntitySet* name.

Add a do-nothing named query

Now let's write a default named query method in a named query provider class.

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

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

Named queries are methods that return an IQueryable or an IEnumerable of an entity type. *GetCustomers* returns an *IQueryable* of *Customer*.

The definition of this query method should seem familiar. The returned value is almost exactly the same as the EntityQuery we wrote on the client. It's only missing the *EntitySet* string ("Customers"). That's OK because DevForce infers the "Customers" entity set name automatically.

It also lacks an *EntityManager*. That is to be expected on the server; the query will be used by a different *EntityManager* instance every time.

DevForce can tell that this is a query method for the *Customer* type because it has the right signature. It has no parameters and returns an *IQueryable* of *Customer*. DevForce also determines that this is the **default named query**: the named query to use when the client sends a query specifying the "Customers" *EntitySet*.

The *EntityServer* found this method by applying the DevForce query naming conventions during its search. It stripped off the "Get" from the method name, leaving the word "Customers" which matches the name of the Customer type's *EntitySet*. You can use the *Query*

Merging the named query with the original client query

Once DevForce finds the named query that matches the client's query, it merges the two queries by copying the LINQ clauses of the client query to the output of the named query method. You can imagine DevForce doing something like the following with *query-B-Customers*:

```
// query-B-Customers = new EntityQuery<Customer>().Where(c => c.StartsWith("B");
mergedQuery = GetCustomers().Where(c => c.StartsWith("B");
```

```
' query-B-Customers = new EntityQuery<Customer>().Where(c => c.StartsWith("B");
mergedQuery = GetCustomers().Where(Function(c) c.StartsWith("B")
```

Then the *EntityServer* forwards the *mergedQuery* to the DevForce *EntityServerQueryInterceptor* or to your custom *EntityServerQueryInterceptor*

Add a do-something named query

The *GetCustomers* method we just wrote does exactly what DevForce would do anyway. There's no point in writing a default named query unless it adds value.

You do not have to write a default named query.

But we can write a *GetCustomers* method that adds value such as this one:

```
[RequiresRoles("admin")]
public IQueryable<Customer> GetCustomers() {
```

```
  return new EntityQuery<Customer>().Include("Orders");
}
```

```
<RequiresRoles("admin")>
Public Function GetCustomers() As IQueryable(Of Customer)
  Return New EntityQuery(Of Customer)().Include("Orders")
End Function
```

Now only administrators can issue a query for *Customers*, thanks to the *RequiresRoles attribute* . The query also automatically includes the *Order* entities related to the *Customers* returned by the query.

The result returned to the client will be further restricted to *Customers* whose names begin with “B” because that criterion was copied over from the client's *query-B-Customers* query.
Let’s try a different implementation:

```
public IQueryable<Customer> GetCustomers() {
    var user = System.Threading.Thread.CurrentPrincipal as User;
    EnsureValidUser(user);
   // only show the user’s own Customers
   return new EntityQuery<Customer>()
              .Where(c => c.UserID = user.ID);
}
```

```
Public Function GetCustomers() As IQueryable(Of Customer)
Dim user = TryCast(System.Threading.Thread.CurrentPrincipal, User)
EnsureValidUser(user)
  ' only show the user’s own Customers
  Return New EntityQuery(Of Customer)().Where(Function(c) c.UserID = user.ID)
End Function
```

In this one, the named query extracts the user from the server IPrincipal and limits the query to *Customer*s that belong to the client user only. The *EnsureValidUser* method (not shown) throws a useful exception if the *IPrincipal* is not a proper *User* object.

After merging with the client *query-B-Customers*, the query results contain only those *Customer*s whose names begin with “B” **and**

The client query does not change

You can add, modify, or delete the default named query without changing your client query. Client query syntax remains the same and the query is used the same way, with or without a default named query on the server. The presence, absence, or nature of the default named query is structurally and functionally transparent to the client.
Of course the default named query itself can change what entities are returned from the data store. That’s why you wrote it. The named query may even reject the client query and throw an exception. These behaviors are up to you.

The default named query applies to the cache

A client query that is rooted in the default named query can be applied to the local entity cache.
If the query’s *QueryStrategy* is *Optimized* or *DataSourceThenCache* or *CacheOnly*, the query will include cached entities. If the *EntityManager* is disconnected, the query applies to the cache. The *EntityManager* can also remember the query in its query cache, potentially avoiding a redundant trip to the server when the query is executed a second time.

This is not true for specialized named queries described elsewhere. They are always executed as *DataSourceOnly* queries and are never remembered in the query cache. They always fail if executed offline. The reason for this difference is explained in the topic on specialized named queries.

Only one default named query

You don’t have to write a default named query. If you do, there can be only one named query method per entity type. There can be only one named query method associated with an entity type’s *EntitySet* and that one method must not take parameters.

Another topic covers *parameterized named queries*. The default named query may not have parameters.