Contents

The server should only honor a query coming from a client if the client is properly authorized. The *EntityServerQueryInterceptor* is the most flexible way to evaluate and authorize a query but it requires code. You may be able to **satisfy your query security requirements declaritively by adding security attributes either to the entity type or to *named query methods***. This topic explains what those attributes are and how they work.

Security attributes

Security attributes are the easiest way to authorize client queries. You can add attributes to entity class definitions and to *named query methods*.

Here is a table summarizing query security attributes, followed by an explanation of how to add them, what they do, and when DevForce applies them.

Attribute	Summary
ClientQueryPermissions	Enable or disable the client's right to use certain query features such *Include* clauses and projections (*Select* clauses).
RequiresAuthentication	Whether client must be authenticated to execute this *named query* or refer to the adorned entity type in a query.
RequiresRoles	Which client roles are required in order to execute this *named query* or refer to the adorned entity type in a query.
ClientCanQuery	Whether the client can refer to this type in a query. This attribute can be applied to an entity type but not to a *named query method*.

ClientQueryPermissions

The *ClientQueryPermissions* attribute enables (or disables) certain query features. At the moment it controls two client query features:

1. *Include* clauses: an *Include* clause adds related entities to query results.
2. Projections: a "projection" is a query that uses the *Select* or *SelectMany* LINQ operation to change the type of the result.

The attribute specifies a flag enumeration, *ClientQueryPermissions*, that captures the permission combinations. The enum values are:

- Minimal (no Includes, no projections)
- AllowIncludes
- AllowProjections
- All (allow both includes, and projections)

You can apply this attribute to an entity type or to a *named query method*. Here is an example of the attribute applied to the *Orders* type.

```
[ClientQueryPermissions(ClientQueryPermissions.AllowProjections)]
public partial class Order{...}
```

```
<ClientQueryPermissions(ClientQueryPermissions.AllowProjections)>
Partial Public Class Order ...
```

When both the entity type and a *named query method* have conflicting attributes, the *named query* attribute takes precedence.

You can associate this attribute with specific user roles and you can apply the attribute several times in order to express different permissions for different roles. In the following *GetGoldCustomers* named query example, administrators can use Includes and Projections with this query but no one else can:

```
[ClientQueryPermissions(ClientQueryPermissions.All, "admin")]
[ClientQueryPermissions(ClientQueryPermissions.Minimal)]
public IQueryable<Customer> GetGoldCustomers() {...}
```

```
<ClientQueryPermissions(ClientQueryPermissions.All, "admin")> _
<ClientQueryPermissions(ClientQueryPermissions.Minimal)> _
Public Function GetGoldCustomers() As IQueryable(Of Customer) ...
```

This attribute is beta as of version 6.1.0. A known issue causes DevForce to improperly apply this attribute to LINQ statements returned by a *named query*.

RequiresAuthorization and *RequiresRoles*

These two attributes serve the same purpose as the like-named WCF RIA Services attributes.

RequiresAuthentication ensures that a user must be authenticated to query the entity or use the *named query method*.

RequiresRoles ensures that only users with specific roles can query the entity or use the *named query method*.

The *EntityServerSaveInterceptor* uses these same attributes to determine if the client is authorized to save the entity type.

ClientCanQuery

The *ClientCanQuery* attribute is similar to the *RequiresRoles* attribute but there are differences. You can only apply the *ClientCanQuery* to entity types, not *named queries*. It offers a little more precision when specifying which kinds of users can reference an entity type in a query.

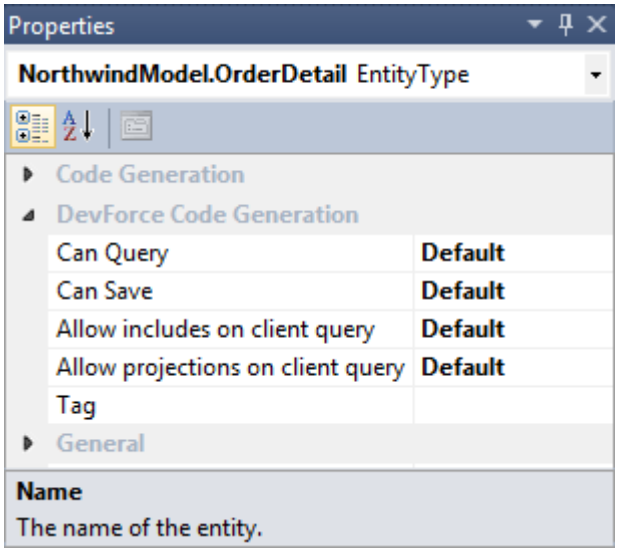
Example	Effect
ClientCanQuery(false)	No user can query with the type
ClientCanQuery(true)	All users can query with the type
ClientCanQuery(AuthorizeRolesMode.Any, role1, role2 ...)	A user with any of the mentioned roles can query with the type.
ClientCanQuery(AuthorizeRolesMode.All, role1, role2 ...)	Only a user with all of the mentioned roles can query with the type.

ClientCanQuery is *true* by default so you generally do not need to add this attribute to your entity classes. However, you can reverse the default (see the discussion of *DefaultAuthorization* below) which would mean that, by default, no client could query with this type. If you change the default in this way, you may want to add this attribute to those types which should be client queryable and set it to *true* explicitly.

The *ClientCanQuery* attribute is evaluated after *RequiresAuthentication* and *RequiresRoles*.

Add attributes to entity types in the EDM Designer

When you created your entity model, you may have noticed in the EDM Designer that each Entity has DevForce properties that govern the ability of the client to query and save:



The *CanQuery* property translates to the *ClientCanQuery* attribute on the generated entity class. The "*Allow includes*" and "*Allow projections*" properties combine to determine a *ClientQueryPermissions* attribute on the generated entity class.

The property values are tri-state: *True*, *False*, and *Default*. "*Default*" means "rely on the *EntityServerQueryInterceptor*'s default" as described below and DevForce won't generate the corresponding attribute. Because the "*Allow ...*" properties resolve to a single attribute; either both are "default" or neither is.

Suppose we kept the default values for the two "*Allow ...*" properties and disabled (made *false*) the client's ability to query this type. The generated class would look like this:

```
[IbEm.ClientCanQuery(false)]
public partial class OrderDetail : IbEm.Entity {}
```

```
<IbEm.ClientCanQuery(False)>
Partial Public Class OrderDetail
 Inherits IbEm.Entity
End Class
```

Turning off all direct query access may seem a bit draconian, but this is a valid approach for those types that you really only want to allow on the server.

Add attributes to the entity partial class

You don't have to rely on the EDM designer to generate the attributes. You can add them to the entity's partial class where you can be more particular and assign permissions by role as seen in this example.

```
[ClientCanQuery(AuthorizeRolesMode.Any, "Admin", "Sales")]
public partial class Order : IbEm.Entity {}
```

```
<ClientCanQuery(AuthorizeRolesMode.Any, "Admin", "Sales")>
Partial Public Class Order
 Inherits IbEm.Entity
End Class
```

You'll need the partial class if you want to add the RequiresAuthentication or RequiresRoles attributes.

```
[RequiresAuthentication]
public partial class OrderDetail : IbEm.Entity {}
```

```
<RequiresAuthentication>
Partial Public Class OrderDetail
 Inherits IbEm.Entity
End Class
```

Add attributes to *named query methods*

You can add any of the query security attributes to the definition of a named query method except *ClientCanQuery*. Here's and example restricting a specialized named query to adminstrators

```
[RequiresRoles("admin")]
public IQueryable<Customer> GetGoldCustomers() {...}
```

```
<RequiresRoles("admin")>
Public Function GetGoldCustomers() As IQueryable(Of Customer)...
```

Attributes and the *EntityServerQueryInterceptor*

The security attributes don't do anything on their own. They are metadata to be read and interpreted. Your code can read the metadata, perhaps to control the appearance and capabilities of the UI.

The DevForce *EntityServerQueryInterceptor* reads and interprets the metadata when it processes the query. You can derive from this class and override virtual methods to introduce custom logic of your own including custom query security logic.

EntityServerQueryInterceptor.AuthorizeQuery is the primary method to override. This method first calls *GetClientQueryPermission* to confirm that the client query is only using permitted features. Then it calls the *ClientCanQuery* method for each entity type involved in the original client query in order to verify that the client is authorized to refer to that type in a query. The default *ClientCanQuery* method discovers and applies the following attributes in sequence:

- *RequiresAuthentication*
- *RequiresRoles*
- *ClientCanQuery*

It bears repeating that this phase of the analysis applies to the **original** client query, not the *named query*.

Types specified inside the *named query* get a free pass. For example, you can prevent client queries from using the *Customer* type by setting *ClientCanQuery* to *false*. If a client submitted a query for *Customer*s, the server would reject it. But it is OK for the client to invoke the *GetGoldCustomers* named query even though its implementation involves the *Customer* type; a *named query* is written for and executed on the server and is presumed to be secure.

Types appearing in Include clauses are checked as well

For example, suppose you blocked query of the *OrderDetails* type by adorning that class with *ClientCanQuery(false)*. The client could not query for *OrderDetails* directly. The client would also be unable to acquire *OrderDetail* entities indirectly via a query for *Orders* that had an "OrderDetails" *Include* clause.

Defaults in the absence of attributes

The default *GetClientQueryPermission* method looks for a *ClientQueryPermissions* attribute for the query. If it can't find that attribute, it substitutes the *DefaultClientQueryPermissions*. The default is "*All*" meaning that all potential client query features are permitted. You can override this property to provide a more restrictive global setting such as "*Minimal*". Then the developer would have to enable query features explicitly on either the entity types or *named query methods*.

The *ClientCanQuery* method looks for a *ClientCanQuery* attribute on each type involved in the client query. If it can't find that attribute, it allows or disallows the query based on the value of the *DefaultAuthorization* property. That value is *true* by default, meaning that a type without a *ClientCanQuery* attribute can be queried. You can reverse this default by overriding this property and returning *false*; then every type must carry the *ClientCanQuery* attribute or it can't be referenced in a client query.

Summary of virtual authorization interception members

The following Authorization-related properties and methods are all virtual which means that you can override them in your custom derived *EntityServerQueryInterceptor*.

Member	Summary
AuthorizeQuery	The primary query authorization method; it orchestrates authorization and calls the other authorization members.
GetClientQueryPermissions	Get the permissions that govern certain query features.
DefaultClientQueryPermissions	Get the *ClientQueryPermissions* enum to use when no attribute is specified.
ClientCanQuery	Get whether the client can refer to the specified type in this query.
DefaultAuthorization	Get whether the type can be referenced in a query if it lacks a *ClientCanQuery* attribute.

You typically delegate to DevForce base class members, either before or after running your code ... but you don't have to do so. For example, within the interceptor you could ignore DevForce's interpretation of the attributes when you have better, perhaps contradicting, information available to you.

EntityServerQueryInterceptor and *named queries*

When the client submits a query that invokes a *named query*, DevForce authorizes the *named query* first. You cannot bypass the *RequiresAuthentication* and *RequiresRoles* attributes that adorn a *named query method*. If the *named query* fails attributed authorization, the query fails before reaching the *EntityServerQueryInterceptor*.

If the *named query* survives these attribute-based security checks, DevForce combines the output of the *named query method* with the original client query instructions and passes the product of that combination to the *EntityServerQueryInterceptor*.

The interceptor has access to the original client query and the original named query method *as described here* which means you can analyze the parts in your own custom interceptor.