**Contents**

Queries processed by the *EntityServer* pass through an instance of the DevForce *EntityServerQueryInterceptor*. You can alter the course of that processing by **creating a custom subclass of the *EntityServerQueryInterceptor*** and overriding its virtual properties and template methods. DevForce will discover the existence of your class automatically.

## Interceptor design guidelines

You don't have to write a custom interceptor class. The *EntityServer* will use the DevForce *EntityServerQueryInterceptor* if it doesn't find a custom subclass. Many production applications do include a custom interceptor.

The *EntityServerQueryInterceptor* class resides in the *IdeaBlade.EntityModel.Server* assembly which must be referenced by the project that contains your custom subclass.

You can write only one custom interceptor per composition context. It should be a public class and must have a public default, parameterless constructor (if it has a constructor).

Please keep your named custom interceptor stateless if possible. DevForce creates a new instance of this class for each query performed by the server so you generally don't have to worry about threading issues with instance state. If you decide to maintain static state, give great care to ensuring safe concurrent access to that state.

Avoid putting anything in the interceptor other than what is strictly necessary to achieve its purpose. The interceptor is a poor choice for a grab-bag of server-side features.

You don't have to override any of the template methods; the default base implementations all work fine. You may wish to be explicit in your custom class and override every template method; your override can simply delegate to the base implementation.

Make sure that the assembly containing your custom interceptor is deployed to the server such that it can be discovered. Assembly discovery is discussed *here*.

## Interceptor template methods

Most of the template methods provided by the *EntityServerQueryInterceptor* base class have no parameters because all of the relevant data are provided by properties and methods on each instance of the class. This also allows IdeaBlade to extend these base classes in the future without breaking custom developer code.

Many of the template methods described below return a boolean result with a base implementation that returns ‘true’. A return value of ‘false’ indicates that the operation should be cancelled. Query results include a flag to indicate a cancelled operation. Note that the base implementation of the authorization method does not return ‘false’. It treats an unauthorized query as an exception, not a cancellation.

| Method | Typical Uses |
|---|---|
| *AuthorizeQuery* | Determine if the user is authorized to make the request. |
| *FilterQuery* | Modify the query as needed. |
| *ExecuteQuery* | Can intercept immediately before and after the query is executed. Always call the base method to allow DevForce to execute the query. |
| *AuthorizeQueryResult* | Allows for authorization of types in the result set. |

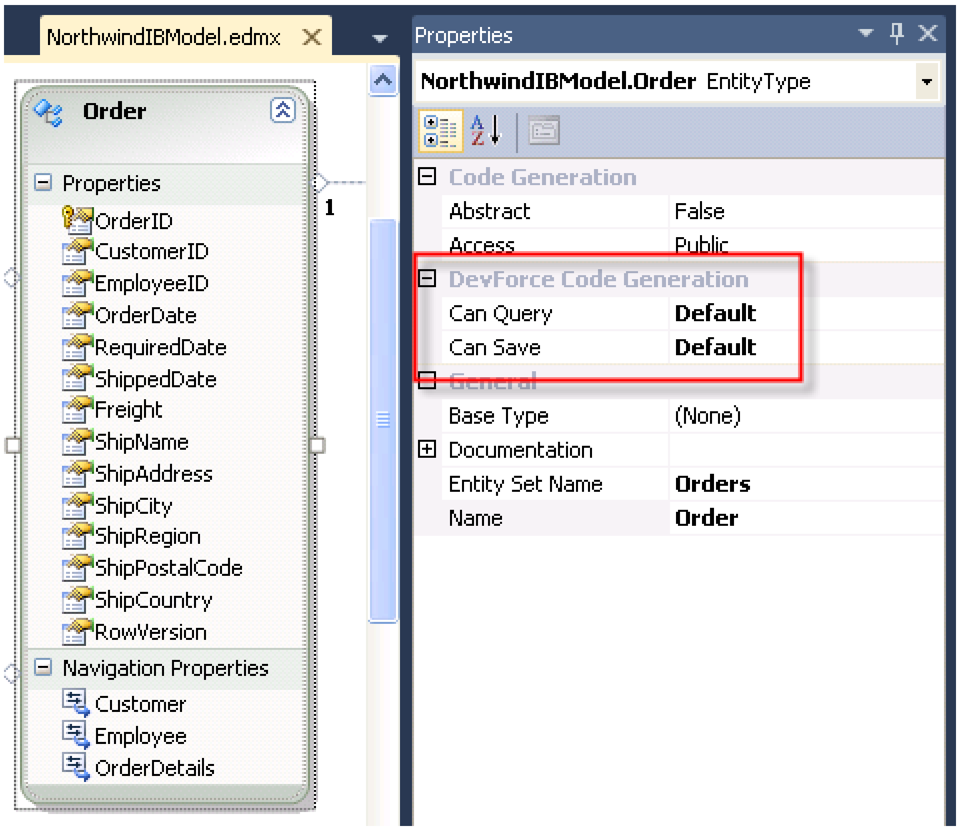
In addition to the above, there are supporting properties or methods that can be overridden. For data retrieval, these include

- *DefaultAuthorization*
- *ClientCanQuery*, and
- *ShouldAuthorizeQueryResult*

## Authorization

During data retrieval operations, the first interceptor method called is *AuthorizeQueryResult()*. The default implementation checks the *ClientCanQuery* property, passing it the target type of the query. *ClientCanQuery* in turn checks the *DefaultAuthorization* property.

If you select an Entity in the Entity Data Model Designer, you will see associated DevForce Code Generation properties named *Can Query* and *Can Save*. By default, these have their values set to the value *Default*; other options are *True* and *False*.


NorthwindIBModel.edmx

Order

Properties
- OrderID
- CustomerID
- EmployeeID
- OrderDate
- RequiredDate
- ShippedDate
- Freight
- ShipName
- ShipAddress
- ShipCity
- ShipRegion
- ShipPostalCode
- ShipCountry
- RowVersion

Navigation Properties
- Customer
- Employee
- OrderDetails

Properties

**NorthwindIBModel.Order** EntityType

| | |
|---|---|
| Code Generation | |
| Abstract | False |
| Access | Public |
| DevForce Code Generation | |
| Can Query | **Default** |
| Can Save | **Default** |
| General | |
| Base Type | (None) |
| Documentation | |
| Entity Set Name | **Orders** |
| Name | **Order** |

If you leave the default values unchanged and you also leave unchanged the value of the *DefaultAuthorization* property of your *EntityServerQueryInterceptor* class, then all entity types will be allowed in queries and saves. You can however override the settings in the EDMX file with code in your EntityServerQueryInterceptor.

```
/// <summary>
///
/// </summary>
/// <remarks>
/// Override this property to change the Default authorization result (whether
/// authorization succeeds if no Authorization attributes are found).
///
/// The default is true - allowing any entity type not specifically marked
/// with an authorization attribute to be queried.
/// </remarks>
protected override bool DefaultAuthorization {
 // Always require explicit authorization attributes on entities.
 get { return false; }
}
```

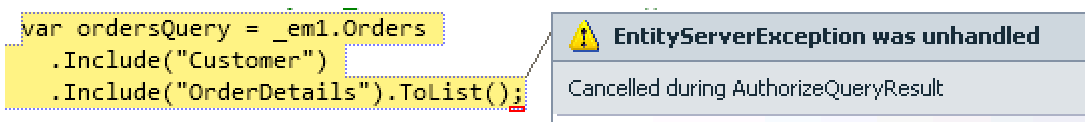
```
''' <summary>
'''
''' </summary>
''' <remarks>
''' Override this property to change the Default authorization result (whether
''' authorization succeeds if no Authorization attributes are found).
'''
''' The default is true - allowing any entity type not specifically marked
''' with an authorization attribute to be queried.
''' </remarks>
Protected Overrides ReadOnly Property DefaultAuthorization() As Boolean
 ' Always require explicit authorization attributes on entities.
 Get
   Return False
 End Get
End Property
```

You have just ordered that entities types are not authorized for queries, by default. An attempt to execute a query such as the following…

```
var customersQuery = _em1.Customers.ToList();
```

```
Dim customersQuery = _em1.Customers.ToList()
```

…now results in an exception, thrown in response to the call to base.AuthorizeQuery():



```
var ordersQuery = _em1.Orders
   .Include("Customer")
   .Include("OrderDetails").ToList();
```

EntityServerException was unhandled

Cancelled during AuthorizeQueryResult

If, however, you return to the EDM Designer and set the *Can Query* property on the Customer entity to *True*, then you will again be able to submit queries against the Customer type.

Suppose, having set DefaultAuthorization to false, you were to authorize the Order type (along with the Customer type), and then submit the following query:

```
var ordersQuery = _em1.Orders
 .Include("Customer")
 .Include("OrderDetails").ToList();
```

```
Dim ordersQuery = _em1.Orders _
 .Include("Customer") _
 .Include("OrderDetails").ToList()
```

The query is requesting Orders and Customers, both of which you have explicitly designated as queryable; but also OrderDetails, which you have *not* made queryable. Is the query executed?

If fact it is, as the *ClientCanQuery* test only looks at the target QueryableType of the query. To enforce a proscription on the retrieval of OrderDetails by way of Includes, you will need to implement logic to screen OrderDetails in an override of the *AuthorizeQueryResult()* interceptor, and also override the *ShouldAuthorizeQueryResult* property, setting its value to *true*:

```
protected override bool AuthorizeQueryResult() {
int countOrderDetails = this.QueriedEntities
 .Where(e => e.GetType() == typeof(OrderDetail)).Count();
if (countOrderDetails > 0) {
 return false;
 }
return true;
}
protected override bool ShouldAuthorizeQueryResult {
get {
 return true;
 }
}
```

```
Protected Overrides Function AuthorizeQueryResult() As Boolean
 Dim countOrderDetails As Integer = Me.QueriedEntities.Where _
   (Function(e) e.GetType() Is GetType(OrderDetail)).Count()
 If countOrderDetails > 0 Then
   Return False
 End If
 Return True
End Function
Protected Overrides ReadOnly Property ShouldAuthorizeQueryResult() As Boolean
 Get
   Return True
 End Get
End Property
```

Such logic would result in an exception upon the attempt to submit the query:

## Filtering the query

If the query *is* authorized, execution proceeds and the *FilterQuery* interceptor method is called. Here you may add filters conditions to the query such as these:

```
/// <summary>
///
```

```
/// </summary>
/// <returns></returns>
/// <remarks>
/// May be overridden to perform any logic before the query is submitted.
/// This can include adding to the QueryFilters collection or by modifying
/// the query directly using the Query.Filter method or the
/// QueryFilterCollection classes.
/// </remarks>
protected override bool FilterQuery() {
// Restrict Customer and Employee results to those based in the UK
QueryFilters.AddFilter<Customer>(q => q.Where(c => c.Country == "UK"));
 QueryFilters.AddFilter<Employee>(q => q.Where(e => e.Country == "UK"));
return true;
}
```

```
''' <summary>
'''
''' </summary>
''' <returns></returns>
''' <remarks>
''' May be overridden to perform any logic before the query is submitted.
''' This can include adding to the QueryFilters collection or by modifying
''' the query directly using the Query.Filter method or the
''' QueryFilterCollection classes.
''' </remarks>
Protected Overrides Function FilterQuery() As Boolean
 ' Restrict Customer and Employee results to those based in the UK
 QueryFilters.AddFilter(Of Customer)(Function(q) q.Where _
   (Function(c) c.Country = "UK"))
  QueryFilters.AddFilter(Of Employee)(Function(q) q.Where _
   (Function(e) e.Country = "UK"))
 Return True
End Function
```

## Executing the query

Next, the *ExecuteQuery()* interceptor method is called. That should include a call to *base.ExecuteQuery()*, which will actually execute the submitted query. You can insert logic both before and after the call to *base.ExecuteQuery()*.

## Virtual methods in more detail

Virtual methods listed in the order that they will be called -

**protected virtual bool AuthorizeQuery()**

Determines whether the query can be executed. The base implementation call performs an analysis of the query expression tree by calling the *ClientCanQuery* method defined below for each entity type used in the query to determine if any unauthorized types are being accessed. Note that this will not detect the case where a polymorphic query is executed against a base type and one of its subtypes has a *ClientCanQuery* restriction. In this case, the *ShouldAuthorizeQueryResult* property should be set to 'true' and the *AuthorizeQueryResult* method may be implemented. It is not usually necessary to override the *AuthorizeQueryResult* method; simply returning 'true' from *ShouldAuthorizeQueryResult* is usually sufficient.

**protected virtual bool FilterQuery()**

Provides a point at which the **Query** property can be modified before the query is executed. Optionally, the *QueryFilters* property can be utilized to add filters that will be applied when the query is actually executed, without actually modifying the *Query* property itself.

**protected virtual bool ExecuteQuery()**

Performs the actual execution of the query. The query itself may be modified before calling the base implementation of this method and logging or other post processing operations may be performed after the base implementation is called. Note that the base implementation must be called in order for the query to be executed. It is during the call to the base implementation that any *QueryFilters* defined earlier will be applied.

**protected virtual bool AuthorizeQueryResult()**

This method is only called if the *ShouldAuthorizedQueryResult* property returns 'true'. This method's base implementation actually walks through all of the entity types to be returned and throws an exception if the *ClientCanQuery* method below returns false for any of the specified types. The QueriedEntities collection mentioned below is also available and may be used to examine **every** entity being returned.

Other virtual properties and methods -

**protected virtual bool DefaultAuthorization { get; }**

This property defines the 'default' authorization behavior for any types that do not have a *ClientCanQuery* attribute defined. The base implementation returns 'true' which means that by default any type not otherwise restricted is queryable. By returning a 'false' here, any types that are not specifically marked as queryable will restricted.

**protected virtual bool ClientCanQuery(Type type)**

This method is called from the base implementation of both the *AuthorizeQuery* as well as the *AuthorizeQueryResult* methods. It may be overridden to add additional restrictions or to relax existing ones. If adding restrictions, make sure that the base implementation is called.

## Helper properties

**protected IEntityQuery Query { get; set; }**

The current query.  This can be modified anytime before query execution.

**protected EntityQueryFilterCollection QueryFilters { get; }**

A collection of Query filters that will automatically applied to every query. Use the **FilterQuery** method to add or remove filters from this collection.

**protected bool IsServerQuery { get; }**

Returns true if the 'current query' was issued on the server.  This can occur as a result of a query made from within this interceptor using the EntityManager mentioned below or as a result of a query invoked by a server implementation of an InvokeServerMethod call.

**protected EntityManager EntityManager { get; }**

An untyped EntityManager that can be used to query for additional information.  Note that this is not the original client-side EntityManager on which the query request was made. It is a 'preauthenticated' server-side EntityManager, and its cache is empty.  Because of its "preauthenticated" nature, there is **no** Principal attached to this entity manager and any operations that it performs bypass regular  authentication.  Because this EntityManager can only be accessed on the server from within code deployed to the server all operations that it performs are considered "priviledged".

Because this EntityManager is untyped, any entity type may queried with it.  Queries against this EntityManager are usually composed via a technique shown in the following example, ( the code below is assumed to be executing within the context of some EntityManagerQueryInterceptor method).  Note the use of the EntityQuery<T> constructor.

```
var newQuery = new EntityQuery<Customer>().Where(c => c.CompanyName.StartsWith("S"))
var custs = newQuery.With(EntityManager).ToList();
```

**protected IList<Object> QueriedEntities { get; }**

This property is only available after the query has executed and returns a list of every **entity** that was queried, whether directly or through an 'Include' or 'Query Inversion'.

For queries not returning entities - such as stored procedure queries returning complex types, queries returning POCO types, queries returning non-entity scalar types, and queries for anonymous types with primitive properties - the *QueriedEntities* collection will be empty.

Any modifications to this collection will be ignored.

**protected IPrincipal Principal { get; }**

The **IPrincipal** from the user session requesting this operation.

**protected void ForceResult(Object result, NavigationSet navSet = null, QueryInfo queryInfo = null)**

May be called to force the results of the query. The shape of the result being foreced must match that of the result of the query being executed.  This method can be called either before or after the execution of the query.
**protected bool ResultsForced { get; }**

Returns true if the ForceResult call above was made.