Contents

To build a secure application, it's not enough to only authenticate our users: we must still **authorize** their access to data and services.

Your application may support users with different roles - for example 'Admin' users may have full privileges, while 'HR' users may only work with information in their department. But you must also authorize against rogue or malicious users, since in most environments you must assume that either the client or the transport can be compromised.

DevForce provides role-based security authorization features to help.

Before we dive into a discussion of when and how authorization can be used, let's briefly review the *IPrincipal/ IIdentity* means of providing role-based authorization. Once logged in, by default a *UserBase* (or some custom IPrincipal implementation) is always available to all client and server code. On the client, the Thread.CurrentPrincipal is set to the logged in user (except in Silverlight), and the *EntityManager.Principal* property will return the logged in user. On the server, the *Thread.CurrentPrincipal* is also set for the calling user, and all methods and base classes provide either a Principal argument or property.

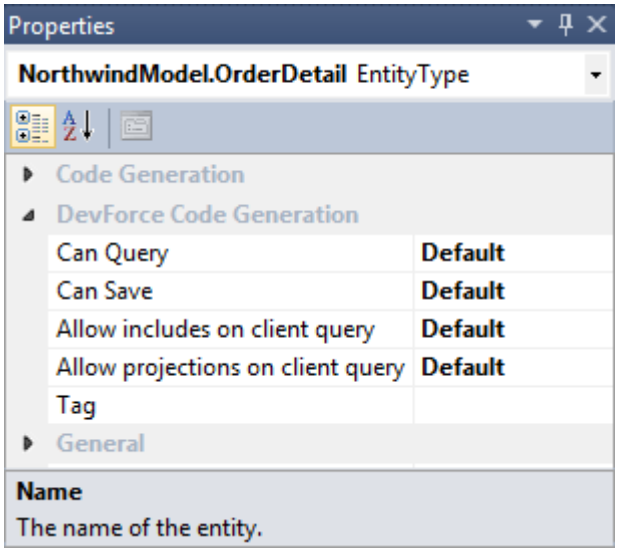
Having the principal always available means that both declarative and programmatic authorization can be performed. We discuss the attributes available for declarative authorization below, but it's also very easy to programmatically check authorization. For example, a query or save could include a simple check such as the following:

```
bool isAuthorized = Principal.IsInRole("admin");
```

```
Boolean isAuthorized = Principal.IsInRole("admin")
```

Entity-level authorization

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



The *CanQuery* and *CanSave* properties translate to *ClientCanQuery* and *ClientCanSave* attributes on the generated Entity class. The "*Allow includes*" and "*Allow projects*" properties combine to determine a *ClientQueryPermissions* attribute.

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

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

```
<IbEm.ClientCanQuery(False), IbEm.ClientCanSave(False)>
```

```
Partial Public Class OrderDetail
 Inherits IbEm.Entity
End Class
```

Turning off all direct query access may seem a bit draconian but you may have types that should really only be accessible on the server. Turning off all direct save access can be particularly useful, since many models have quite a bit of read-only data.

Why would you set *CanQuery* or *CanSave* to true? To override the default, which will be determined by the implementation of the "interceptor", *EntityServerQueryInterceptor* or *EntityServerSaveInterceptor*. (We'll discuss the interceptors in more detail below.) The default implementations of the interceptors allow all queries and saves, but if you've implemented custom interceptors you might find it useful to disable all access by default, and enable it only for entity types as needed.

Remember that each entity is defined as a partial class, making it easy to specify these attributes with code rather than in the designer. In the partial class you can be more particular and assign the permissions by role as seen in this example.

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

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

You may also decorate your entities with the RequiresAuthentication and RequiresRoles attributes. The default interceptors, when authorizing a query or save, will check these attributes before checking the *ClientCanQuery* and *ClientCanSave* attributes. *RequiresAuthentication* can be used to ensure that a guest user does not have any access to the entity, while *RequiresRoles* functions similarly to *ClientCanQuery* and *ClientCanSave* when used to require the user be a member of all roles specified.

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

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

A separate topic devoted to securing the query with attributes covers this approach in greater detail.

Query authorization

The *EntityServerQueryInterceptor*, discussed in depth in the query life cycle topic, is responsible for authorizing data retrieval. The interceptor allows you to authorize query access, add additional filters to a query, and to authorize the query results.

As we saw above, the default interceptor will authorize every query using the authorization attributes decorating the entity. To customize this behavior, and add additional authorization, you'll need to create a custom interceptor sub-classing the *EntityServerQueryInterceptor*.

In your custom interceptor you'll have a chance to:

- Override the default authorization - You can disable all retrieval unless specifically enabled for an entity or query.
- Perform programmatic authorization - The interceptor *Principal* property returns the *IPrincipal* from the *Login* and can be used to check roles, and additional information if a custom *IPrincipal* is used.
- Add filters to the query - You can inspect and modify the query prior to execution. For example, maybe a "UK" user can query orders from her country only - here's the place to add that logic.
- Authorize query results - Once the query has executed you have one last chance to look through the results to determine if the user is authorized to see them.

Through a custom interceptor you can also perform additional logging to audit user access and capture error details.

Save authorization

The *EntityServerSaveInterceptor* is responsible for authorizing modifications to data before they are committed to the data source. The interceptor is discussed in depth in the save life cycle topic.

As we saw above in the discussion of entity-level authorization, the default interceptor will authorize every save using the authorization attributes decorating the entity. To customize this behavior, and add additional authorization, you'll need to create a custom interceptor sub-classing the *EntityServerSaveInterceptor*.

In your custom interceptor you'll have a chance to:

- Override the default authorization - You can disable all saves unless specifically enabled for an entity.
- Perform programmatic authorization - The interceptor *Principal* property returns the *IPrincipal* from the *Login* and can be used to check roles, and additional information if a custom *IPrincipal* is used.
- Customize validation - Instance validation is performed by default, but you can fine tune the validation performed, and add additional verifiers not included on the client.

Through a custom interceptor you can also perform additional logging to audit user access and capture error details.

Field-level authorization

You might need to authorize user access to specific properties on an entity. Maybe all your users have read access to the Employee.Name property, but only users with the "Admin" role can modify it. There are several techniques within DevForce to control field-level (property-level) access to data.

- Scrub the queried data - You can override the *ExecuteQuery* method in the *EntityServerQueryInterceptor* to "scrub" the queried data before it is sent to the client. Note you can use this technique only for read-only data, since any modifications to the entity will cause the scrubbed result to be written to the database. For example:

```
protected override bool ExecuteQuery() {
 bool ok = base.ExecuteQuery();
 if (ok) {
  if (!this.Principal.IsInRole("Admin")) {
   if (Query.ElementType == typeof(Customer)) {
    foreach (var customer in this.QueriedEntities.Cast<Customer>()) {
     customer.CreditLimit = "***";
    }
   }
  }
 }
 return ok;
}
```

```
Protected Overrides Function ExecuteQuery() As Boolean
 Dim ok As Boolean = MyBase.ExecuteQuery()
 If ok Then
  If Not Me.Principal.IsInRole("Admin") Then
   If Query.ElementType = GetType(Customer) Then
For Each customer As var In Me.QueriedEntities.Cast(Of Customer)()
 customer.CreditLimit = "***"
Next
   End If
  End If
 End If
 Return ok
End Function
```

- Use property interceptors - You can define an interceptor to be called before or after a get or set, and you can define multiple chained interceptor actions. You can easily define interceptors via attributes, and you can create them programmatically. Within an interceptor you can check the *EntityManager.Principal* to determine the user's identity and roles. You can also log details on property access. Property interception is generally a client-side activity, since it's there that most property-level access and changes are performed, as such, don't overly rely on your property interceptors as a security mechanism. See the Property interceptors topic for detailed information.
- Customize validation - Validation of changes on the server is critical if you require authorization of property-level changes. Remember that on the server you can add verifiers not included on the client.
- Return part of an entity - Not a true security feature, and like property interceptors a client-side feature which malicious users can abuse, query projections (either anonymous or into a type) are a means of limiting a query to only the select properties.
- Model new entity types - In your entity model, use views, inheritance, entity splitting and other modeling features to define entities requiring special access requirements. Maybe you'll need an Employee entity and a SecureEmployee entity; in doing so you can determine the properties in each, perform custom validations, and control query and save authorization by entity type.

RSM authorization

You can and should add authorization checks to your remote server methods too. The *RequiresRoles* and *RequiresAuthentication* attributes can be used on remote methods, and you can add programmatic authorization. Let's look at a sample method:

```
[AllowRpc, RequiresRoles("admin")]
```

```
public static Object GetNumberOfOrders(IPrincipal principal, EntityManager entityManager, params Object[] args) {
  ...
}
```

```
<AllowRpc(), RequiresRoles("admin")> _
Public Shared Function GetNumberOfOrders(ByVal principal As IPrincipal, ByVal entityManager As EntityManager, ByVal ParamArray
args As Object()) As Object
    ...
End Function
```

Here we've added a *RequiresRoles* attribute to ensure that only users with the "admin" role will be able to invoke the method. But notice the principal argument to the method? We could just as easily use that principal to perform authorization, as shown below.

```
[AllowRpc]
public static Object GetNumberOfOrders(IPrincipal principal, EntityManager entityManager, params Object[] args) {
 if (!principal.IsInRole("admin")) {
   throw new PersistenceSecurityException("Access denied");
  }
 ..
}
```

```
<AllowRpc()> _
Public Shared Function GetNumberOfOrders(principal As IPrincipal, entityManager As EntityManager, ParamArray args As Object()) As
Object
  If Not principal.IsInRole("admin") Then
     Throw New PersistenceSecurityException("Access denied")
  End If
   ...
End Function
```

POCO authorization

Both declarative and programmatic authorization can be performed with POCO types.

All POCO CRUD methods in a service provider class - for query, insert, update and delete - can be decorated with the *RequiresRoles* or *RequiresAuthentication* attributes to control access to the method. You can also check the *Thread.CurrentPrincipal* directly to add programmatic authorization checks to these methods.

If using a custom *EntityServerPocoSaveAdapter*, you can add programmatic authorization to the *BeforeSave*, *InsertEntity*, *UpdateEntity* and *DeleteEntity* methods.

See the POCO topic for more information.