Contents

The changes-set of entities to be saved are processed by the *EntityServer* which passes the change-set through an instance of the DevForce *EntityServerSaveInterceptor*. You can alter the course of that processing by **creating a 'custom' subclass of the *EntityServerSaveInterceptor*** and overriding its virtual properties and template methods. DevForce can 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 *EntityServerSaveInterceptor* if it doesn't find a custom subclass. Many production applications do include a custom interceptor.

The *EntityServerSaveInterceptor* 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. 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

The *EntityServerSaveInterceptor* class contains a number of template methods that you can override to modify the base behavior of the class. These methods are executed at key points within the server-side part of the save process; you override them in your subclass to perform custom interventions. They enable you to perform operation both before and after the physical save to the database (or equivalent backing store).

Most of the template methods 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 the base class 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 "true" allows the save to continue. A return value of "false" causes the save to exit with an exception. There is also a flag available that may be used to indicate a "canceled" operation.

Note that while the base implementation of the authorization method does not return ‘false’, it will throw an exception it it detects an unauthorized save. It treats an unauthorized save as an exception, not a ‘cancellation’.

protected virtual bool *AuthorizeSave*()

Override to control whether the user is authorized to perform the save. The base implementation walks all of the types involved in the save and calls the *ClientCanSave* method defined below for each to determine if any unauthorized types are being accessed. An EntityServerException will be thrown with a *PersistenceFailure* value of "Authorization" if any unauthorized types are encountered. You can bypass authorization by simply returning "true" without calling the base implementation.

protected virtual bool *ValidateSave*()

Override to extend (or remove) server-side validation of the data to be saved. The base implementation of this method will perform "instance validation" for each entity being saved. If any verification fails, an *EntityServerException* will be thrown with a *PersistenceFailure* type of "Validation". The *VerifierEngine* property is available in order to discover what validations will be performed. You can bypass validation by simply returning "true" without calling the base implementation.

protected virtual bool *ExecuteSave*()

Override to intercept the save immediately before and after the save request is executed. The entities involved in the save 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 save to be executed. The *EntityManager* property defined below contains all of the entities to be saved and its contents may be modified at any time prior to the base implemention of the *ExecuteSave* method being called.

protected virtual bool *ClientCanSave*(Type type)

Override to control which types are authorized to be saved. This method is called from the base implementation of the *AuthorizeSave*. It may be overridden to add additional restrictions or to relax existing ones. If adding restrictions, make sure that the base implementation is called.

protected virtual void *OnError*(Exception e, PersistenceFailure failureType)

May be overridden to log errors. No logging is performed by the default implementation.

Virtual Properties

protected virtual bool *DefaultAuthorization* { get; }

Override this property to change the Default authorization of whether or not authorization succeeds if no Authorization attributes are found. This property defines the default authorization behavior for any types that do not have a *ClientCanSave* attribute defined. The base implementation returns "true" which means that by default any type not otherwise restricted is saveable. By returning a "false" here, any types that are not specifically marked as saveable will restricted.

Other properties

In addition to the above, the following protected read only properties are also available.

Property	Property type	Used for
Context	object	Gets or sets a custom context object for this operation.
EntityManager	EntityManager	Returns an *EntityManager* holding the entities to be saved. Note that this is not the original *EntityManager* on which the *EntityManager.SaveChanges()* call was made. This property can be very useful when overriding the *ExecuteSave* method. Additional entities that need to be saved can be added to this EntityManager or entities that should not be saved can be removed before calling *base.ExecuteSave()*.
IsServerSave	boolean	Returns true if the save was issued on the server. This can occur as a result of an *InvokeServerMethod* call. This is useful because you typically do not need to reauthorize a save where the request for the save originates on the server.
Principal	IPrincipal	The *IPrincipal* from the user session requesting this operation.
SaveOptions	SaveOptions	Returns the *SaveOptions* provided in the *SaveChanges* call.
VerifierEngine	VerifierEngine	Returns the *VerifierEngine* which will be used for server-side validation.

- Note that the EntityManager returned by the EntityManager property is NOT the original client-side EntityManager on which the save request was made. It is a 'preauthenticated' server-side EntityManager, and its cache is consists of the entities being saved. Because of its "preauthenticated" nature, there is **no** Principal attached to this EntityManager 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, additional entities of 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 EntityManagerSaveInterceptor 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();
```