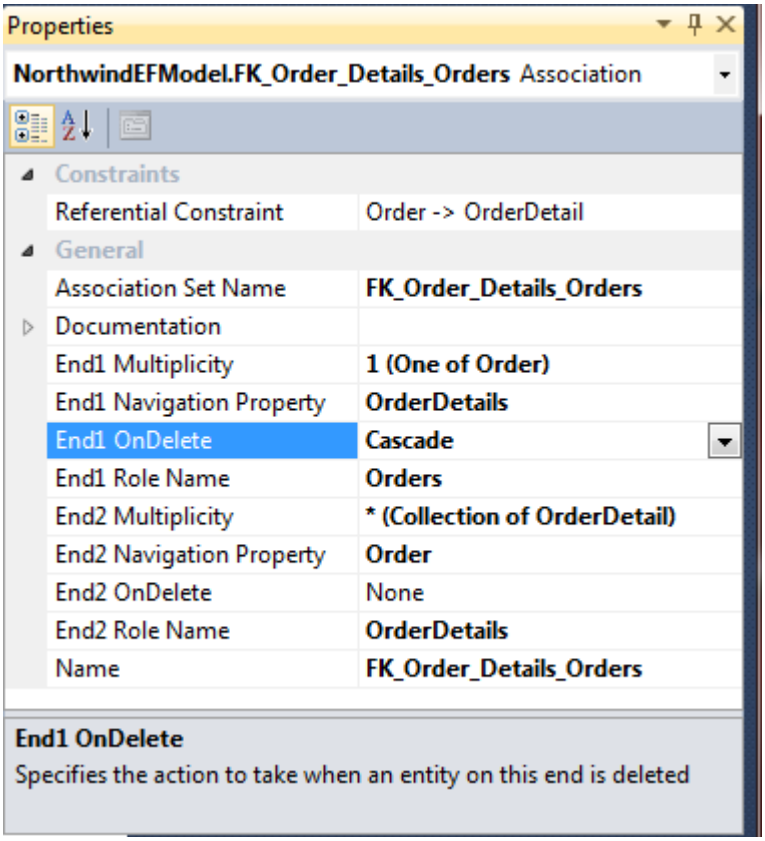
Contents

The ability to declaratively state that the delete of one entity should automatically cause the deletion of some its related "dependent" entities is a relatively common and very useful tool in the development of a complex domain model. In DevForce and in the Entity Framework these are called **cascaded deletes** and the Entity Framework Designer offers support for declaratively marking up an entity relationship as having a delete "cascade" rule that applies to it.

Telling EF about a cascaded delete

Within the Entity Framework Designer, every relationship has its own property page that looks something like the following.



Cascaded deletes are configured by setting either of the "End1 OnDelete" or "End2 OnDelete" combo boxes to "Cascade".

What this markup actually does in both DevForce and the Entity Framework (EF) is tell EF to expect that the "database" will perform its own form of cascaded delete for this relationship. i.e. that the database has some mechanism for insuring that when the records mapped to the parent entity are deleted that any "dependent" records will be automatically deleted as well. Let's call this a database "delete rule" for future reference.

Who does what?

To be clear, DevForce (and EF) does **NOT** take responsibility for cascading the delete in the database. DevForce does insure that the deletion of any principal entity within an *entitymanager* automatically forces the automatic deletion of its related entities also within the entity manager. Recall that deletion in this context is actually marking the entity for deletion so that it will actually be deleted from the database during the next *SaveChanges* call. In other words, DevForce will attempt to synchronize the *EntityManager* with the "expected" state of the database after a delete by understanding that the delete will cause side effect on the database. But...

DevForce does not retrieve all the dependent entities when an entity is deleted and then issue deletes for them as well: It only deletes (marks as deleted) any dependent entities that are already in memory.

Restated, DevForce expects that deleting the principal entity in the database, will delete all it's dependents in the database. So it makes, what should be, a redundant delete request for any dependent entities that it knows about, i.e. that are already loaded into the *EntityManager*'s entity cache.

The rule

This leads a basic rules about cascaded deletes.
If you add an Cascade delete rule to the model, you MUST have a corresponding *delete rule* in the database.

It is possible to break this rule if you are willing to insure that anytime that you delete a principal entity that all of its dependent entities have already been loaded into the *EntityManager* as well. We do NOT recommend this.

A problem

There is one interesting, rare, and annoying failure that can occur with this process.

Assume that you have relationships that looks something like:

Category –> Product –> Order

with cascading deletes set up so that deleting a Category deletes its Products which in turn deletes its Orders.

Then DevForce will, in some rare cases, be unable to synchronize an *EntityManager* with the database when you delete a Category.

This can occur if you have an Order loaded that is related to a Category via an **unloaded** Product, and you delete the Category. Under these conditions, DevForce won't know to delete the Order.
This means the Order will remain in the *EntityManager* in the unchanged state, despite it having been deleted in the database.