Contents

This topic covers how to **add** new entities to an EntityManager's cache, how to **remove** entities from the cache, and how to **attach** other entities to the cache such that they appear to have been queried rather than created.

When do entities get attached?

In order for entities to be managed by DevForce they must first be associated with an entitymanager. Entities will be associated with an EntityManager as a result of any of the following operations:

- A query.
- A navigation from one entity to another that in turn triggers a query.
- The creation of a new entity and its addition to an EntityManager.

An entity can only be associated with a single EntityManager at one time. Every entity has an associated *EntityState* that indicates whether a particular entity is new, modified, marked for deletion, or unmodified. The *EntityState* for an entity is automatically set and updated by the DevForce infrastructure.

Attaching entities to the EntityManager

Once an entity has been created, it must be attached to an entitymanager before it can be saved, validated, or participate in a number of other services that the EntityManager class offers. Entities may be added to the EntityManager in two ways; either via the EntityManager *EntityManager.AddEntity* method or via the EntityManager *EntityManager.AttachEntity* method. The *AttachEntity* method is actually the primary workhorse here and the *AddEntity* method actually delegates to it. The difference between these two methods is that the ***AddEntity*** method adds the entity to an EntityManager in an "Added" state, whereas ***AttachEntity*** can add an entity to an EntityManager in any of the following *EntityStates*, "Unchanged", "Added" or "Modified". The *AttachEntity* method, by default, attaches entities in an "Unchanged" state, but the method offers an optional parameter that allows you to specify the desired resulting *EntityState*.

The *AddEntities* and *AttachEntities* methods perform the same operations as described above but support the ability to attach collections of entities in a single call.

So why would we want to attach entities in different states?

EntityState	Use case
Added	Probably the most common case. Usually used after creating a *new* entity that we want to have inserted into the database on the next save. This is such a common case that the *AddEntity* method was created just to cover this case. Any time an entity is attached in this state DevForce also performs the task of generating and updating the entity with a 'temporary' primary key if the entity has been declared as participating in automatic id generation.
Unchanged	Less common case. Usually used when reattaching a previously *detached* entity and we don't want any persistence operation to occur on the entity.
Modified	Less comon case. Usually used when reattaching a previously *detached* entity and we want to be able to update the backend database with the values of this entity. Note that an entity that is attached in a *modified* state will preserve any of its "original values" for the purposes of determining which properties actually require modification on the database. These original values will not exist unless the entity was previously queried or saved and then *detached*.

In addition to the *EntityManager.AddEntity* method mentioned above, the *EntityAspect.AddToManager* method does exactly the same thing. The *AddToManager* method is provided because it is sometimes clearer, within an entity constructor to write code where the entity adds itself to a manager, instead of the reverse.

Attaching entity graphs

An entity graph is group of entities that are associated with one another via navigation properties. It is possible to 'wire up' a graph of entities before any of them have been attached to an EntityManager. When any one of these entities is then attached, all of the entities in the group will be attached as well, in whatever state was specified for the initial entity.

In addition, if we ever use a navigation property on an already attached entity to reference a detached entity, the detached entity will automatically get attache, along with any of its detached relations. In this case the detached entities will all be attached in an *added* state.

Detaching entities

Entities can be removed from an EntityManager in either of the following two ways:

- *EntityManager.RemoveEntity* or *EntityManager.RemoveEntities**
- *EntityAspect.RemoveFromManager*

Note that 'removing' or 'detaching' an entity from the EntityManager is NOT the same as deleting it. Removing an entity from an EntityManager marks the entity as 'detached'. Detaching is basically telling the EntityManager to 'forget' the entity.

Note that any dependent children of this entity will NOT be removed, they will be "orphaned", meaning that they will no longer have a parent but will still exist in the EntityManager.

Deleting, ***a completely different operation***, marks the entity as 'deleted' and schedules it for deletion during the next save operation at which point it will be removed from both the database and the EntityManager.

Detached entities can be reassociated with an EntityManager via either the AddEntity or AttachEntity methods described earlier.

There is one other issue related to removing or detaching entities from an EntityManager. It has to do with the state of the EntityManager's query cache after the removal of an entity. Because DevForce cannot associate a removal of an entity with the query or queries that produced it, by default, DevForce is forced to clear its *query cache* ***(not the entity cache)*** when a removal occurs. This may seem drastic, but basically the *query cache* is there to improve performance when DevForce knows that a query has already been processed and that it can return the same results running from cache as it would when querying from a database. When any entity is removed from the cache, DevForce loses the ability to make this guarantee. This is the default behavior, but you can tell DevForce not to clear the query cache by making use of the optional second parameter of the RemoveEntity/RemoveEntities/RemoveFromManager methods.

Note that the removall of Added entities will not trigger the default clearing of the query cache because these entities could never have been returned from a query.

Using the AttachEntity method for testing

Those of you who write tests and don't want those tests to touch the database will appreciate the ability of the AttachEntity method to be used in testing.

As you know, you sometimes need to write tests which rely upon interaction with the EntityManager. You want to populate a disconnected EntityManager with a small collection of hand-rolled stub entities. While such tests are integration tests because they rely on a dependency, we still want to make them easy to write and we want them to be fast. That means we don't want a trip to a database when we run them; we shouldn't need to have a database to run them.

I usually start by creating a test-oriented, disconnected EntityManager ... which can be as simple as the following:

```
var testManager = new EntityManager(false /* disconnected */ );
```

```
Dim testManager = New EntityManager(False) ' disconnected
```

The easiest way to get a stub entity is to "new" it up, set some of its properties, give it an *EntityKey*, and dump it in our testManager. When we're done it should appear there as an unchanged entity ... as if you had read it from the datastore.

The catch lies in the answer to this question: "How do I add the entity to the manager?"

In the absence of AttachEntity() method, you would have to use EntityManager.AddEntity(). But after AddEntity, the EntityState of the entity is always "Added". You want a state of "Unchanged" so you have to remember to call AcceptChanges (which changes the state to "Unchanged").

That's not too hard. Unfortunately, it gets messy if the key of the entity is auto-generated (e.g., mapped to a table whose id field is auto-increment) because DevForce automatically replaces your key with a temporary one as part of its auto-id-generation behavior.

We could explain how to work around this, but the AttachEntity() method already does what we need.

Let us elaborate here and compare it to some similar methods by calling out some facts about the following code fragment:

```
theEntityManager.AttachEntity(theEntity);
```

```
theEntityManager.AttachEntity(theEntity)
```

- theEntity's EntityKey ("the key") must be preset prior to the attach operation (which will not touch the key).
- An exception is thrown if an entity with that key is already in the cache.
- After attach, theEntity is in an "Unchanged" EntityState ("the state").
- theEntity is presumed to exist in the persistent store; a subsequent change and save will translate to an *update* statement.
- After a successful attach, a reference to theEntity is a reference to the entity with that key in the manager's EntityCache. Contrast this with the effect of anEntityManager.Imports(new [] {anEntity})" as discussed below.
- theEntity must be in the "Detached" state prior to the operation.
- An exception is thrown if theEntity is other than in "Detached" state prior to the operation.
- After attach, related entities are implicitly associated with theEntity automatically; for example, if anOrder with Id==22 is attached and there are OrderDetails with parent OrderId==22, then after the attach, anOrder.OrderDetails returns these details and any one of them will return 'anOrder' in response to anOrderDetail.Order.
- The sequence of attachments is not important; OrderDetails may be added prior to the parent Order.
- Attach has no effect on theEntityManager's QueryCache.

AddEntity behaves the same way as *AttachEntity* except as follows:

- After add, theEntity is in an "Added" state
- theEntity is presumed to be new and to be absent from in the persistent store; a save will translate to an *insert* statement.
- If the key for this type is auto-generated (e.g., backed by an auto-increment field in the database), the existing key will be set to a generated temporary key, replacing the prior key value.

The following is true regarding *detaching* anEntity:

- After detach, anEntity enters the "Detached" state no matter what its prior state.
- Detaching an Order does not detach its child OrderDetails - they remain "orphaned" in the cache.
- The sequence of detachments is not important; an Order may be detached prior to detaching its child OrderDetails.
- Detach has no effect on theEntityManager's QueryCache.

Difference between AttachEntities and ImportEntities

EntityManager.ImportEntities is another way of populating an EntityManager with a collection of entities that may have come from anywhere (including hand-rolled). Here's how you might "import" a single stub entity:

```
theEntityManager.ImportEntities(new [] {theEntity});
```

```
theEntityManager.ImportEntities( {theEntity})
```

ImportEntities differs from *AttachEntity* in that:

- It requires a MergeStrategy to tell it what to do if an entity with the same key as "theEntity" already exists in the cache.
- It merges "theEntity" into the cache based on the MergeStrategy
- It makes a clone of "theEntity" and adds that clone to the EntityCache ... unless "theEntity" happens to already be in the cache in which case it is ignored ... which means that
- Using our example and assuming that "theEntity" was not already in the manager, the entity instance in the cache is not the same as the entity instance you imported, although their keys are equal; the following is true:
  theEntity != theManager.FindEntity(theEntity.EntityAspect.EntityKey)
- A "clone" is a copy of an entity, equivalent to calling the following:

```
((ICloneable)theEntity).Clone();
```

```
CType(theEntity, ICloneable).Clone()
```

- This is a copy of the entity, not of its related entities.