Contents

Navigation properties that return collections (e.g., *anEmployee.Orders*) are always of the type *RelatedEntityList<T>*. The *RelatedEntityList* in turn provides *Add* and *Remove* methods that may be used to **add or remove related entities**.

Add

The *Add()* method takes a parameter of the type contained by the collection. For example, here we add a new Order to an Employee's *Orders* collection:

```
Order anOrder = new Order();
anOrder.OrderDate = DateTime.Today;
anOrder.FreightCost = Convert.ToDecimal(999.99);
anEmployee.Orders.Add(anOrder);
```

```
Dim anOrder As New Order()
anOrder.OrderDate = Date.Today
anOrder.FreightCost = Convert.ToDecimal(999.99)
anEmployee.Orders.Add(anOrder)
```

Invoking *Add()* adds the supplied item to the collection. If the relation between the parent and child types is 1-to-many and the supplied item is currently associated with a different parent, then *Add()* simultaneously removes it from the corresponding collection of the other parent. The equivalent result on table rows in a relational database is that the child entity's foreign key value is changed.

Note in the above snippet that we did not need to set the parent SalesRep property of the new Order:

```
anOrder.SalesRep = anEmployee; //don't need this; Add() will handle it
```

```
anOrder.SalesRep = anEmployee  ' don't need this; Add() will handle it
```

Invocation of the *Add()* method on *anEmployee.Orders* produced the equivalent result.

Regardless of whether the child entity is added to the parent collection or the parent navigation property is set on the child, the associated foreign key property on the child is also set. This is true even when an entity is in an *Added* state and contains a temporary id.

Remove

Remove() also takes a parameter of the type contained by the collection. It dissociates the indicated instance from the collection's parent. Speaking again of the equivalent result on table rows in a relational database, the child entity's foreign key value is set to null; however, if the foreign key property is also part of the entity's primary key the value is unchanged.

```
anEmployee.Orders.Remove(anOrder);
```

```
anEmployee.Orders.Remove(anOrder)
```

Note that while Remove unassigns the Order from the target Employee, removing it from the collection returned by the navigation property, **it does not remove it from the cache or mark it for deletion**. If you want the Order removed from the cache or deleted from the back-end datastore you must call *EntityAspect.Remove()* or *EntityAspect.Delete()*. Rememember that removing an item from the cache is not the same thing as deleting the item; if you want the item permanently deleted from the database be sure to call *Delete*.

Add and Remove on many-to-many navigation properties

You will also use *Add()* and *Remove()* on many-to-many navigation collections.

A many-to-many relationship is one in which two entities are linked by a many-to-many join table that has "no payload", that is, no columns other than the two foreign keys (which also form a composite primary key. An example (from the NorthwindIB sample database) would be an Employee linked to a Territory by means of an *EmployeeTerritory* table whose composite primary key consists of the two foreign keys *EmployeeId* and *TerritoryId*, and which has no other columns.

With these "no payload" many-to-many associations, the join table is not included in your entity model. Continuing the Employee, EmployeeTerritory, and Territory example, only the Employee and Territory entities are included in your entity model; the join table is abstracted away.

When you call *Add()* and *Remove()* on a many-to-many navigation property, DevForce performs the necessary housekeeping to ensure that when these changes are saved to the database the join table is updated as needed with the appropriate insertions and deletions.

Because many-to-many relationships do not include a foreign key, *Add* and *Remove* are the only way to modify these relationships.

For example, the following will create a new linking entry for the EmployeeTerritory relationship:

```
var employee = mgr.Employees.Include("Territories").First(e => e.EmployeeID == 1);
var territory = mgr.Territories.First(t => t.TerritoryDescription == "Terra Incognita");
employee.Territories.Add(territory);
```

Although neither entity above has changed, the *EntityManager* is aware of the new linking entry and when *SaveChanges* is called the entry will be added to the link table in the database.

To remove the association:

```
var employee = mgr.Employees.Include("Territories").First(e => e.EmployeeID == 1);
var territory = employee.Territories.First(t => t.TerritoryDescription == "Terra Incognita");
employee.Territories.Remove(territory);
```

Here again, the *EntityManager* is aware of the removed linking entry and will delete the entry from the link table in the database when *SaveChanges* is called.

Clear

Calling *Clear()* on a *RelatedEntityList* is the equivalent of calling *Remove* for every item in the collection. For one-to-many relationships, the foreign key property of child items is reset to null, and the items are marked as *Modified*. For many-to-many relationships, the link entries are deleted. In both cases, *EntityManager.HasChanges* will register that pending changes are waiting to be saved to the database.

Read only navigation properties

By default navigation properties return a collection which may be modified. You can set the *RelatedEntityList* to be read only via the EDM Designer. In the designer, select the navigation property, then set "Is collection read only" in the Model Properties window to *True*.