Contents

The *IdeaBlade.EntityModel.EntityListManager<T>* class is intended to provide "live-list" management capabilities to any list containing DevForce entities. The idea of a "live" or "managed" list is that the membership to the list is kept continuously updated based on changes to entities in an *EntityManager*'s entity cache. Filter expressions are used to determine the rules by which an entity is either included in or excluded from a specific list.

Creating an EntityListManager

Consider the following code:

```
Predicate<Employee> filter = (emp) => emp.City == "London";
_employeeEntityListManager = new EntityListManager<Employee>(_em1, filter, null);
bool refreshListWhenPlacedUnderManagement = true;
_employeeEntityListManager.ManageList(_salesReps, refreshListWhenPlacedUnderManagement);
```

```
Dim filter = New Predicate(Of Employee)(Function( _
  anEmployee As Employee) anEmployee.City = "London")
_employeeEntityListManager = New EntityListManager(Of Employee)(_em1, filter, Nothing)
Dim refreshListWhenPlacedUnderManagement As Boolean = True
_employeeEntityListManager.ManageList(_salesReps,  refreshListWhenPlacedUnderManagement)
```

This code sets up an *EntityListManager* to watch the cache for changes to Employees, or the insertion of new Employees. If any changed or new Employee is found to be based in London, a reference to that Employee will be added to the _salesReps list. If any Employee that was located in London is moved to another city, that entity will be removed from the list.

The second parameter to the *ManageList* call above, indicates whether you want the _employeeEntityListManager to clear the list and then scan the current "entity cache" and repopulate it based on the specified filter. A 'false' value would indicate that you want the list to be monitored from here on, but that you are confident of its initial population.

The only requirements for the list being managed, in this case: _salesReps, are that it

- implement System.Collections.IList; and
- contain instances of *IdeaBlade.EntityModel.Entity*.

A single *EntityListManager* can manage as many different lists as you wish. To put _employeeEntityListManager in charge of additional lists, you would simply invoke its *ManageList* method again for each desired list:

```
_employeeEntityListManager.ManageList(_telecommuters, false);
_employeeEntityListManager.ManageList(_fieldAgents, false);
```

```
_employeeEntityListManager.ManageList(_telecommuters, False)
_employeeEntityListManager.ManageList(_fieldAgents, False)
```

Of course, it only makes sense to do this when the same inclusion criteria apply to each targeted list.

Note that the *EntityListManager* is NOT 'watching' the list itself, it is only watching the EntityManager associated with the list and will insure that any changes to any entities within the EntityManager result is the addition or removal of entities from the list based on the specified filter.

This means that any changes that are made to the list **directly** do not result in any filtering action.

So as a general rule, you should usually avoid modifying a 'managed' list directly, and instead rely on its ability to stay synchronized with its associated EntityManager. The reason that the list does not 'watch' for changes to itself is that the IList interface does not provide any eventing mechanism that would allow an external component, such as the *EntityListManager*, to perform such a 'watch'. While some implementations of IList do offer such eventing we did not want to restrict the use of the *EntityListManager* working only with such lists.

EntityListManagers and the NullEntity

One exception to the general rule described above occurs when you want to add a NullEntity to a "managed" list. *NullEntities* are a special form of "detached" entities and do not reside in the cache, so there is no way that an *EntityListManager* will ever find one to either add to or be removed from a managed list. If you want a *NullEntity* in a managed list, you should manually add it. The *ListManager* will not remove it.

EntityListManagers and duplicates

The *EntityListManager* will not eliminate duplicates from a list. It will, however, insure that *it* does not add the same entity more than once. For example, suppose you direct the following statement against a list, _salesReps, that is already being managed to include Employees based in London:

```
_salesReps.AddRange(_entityManager.Employees.Where(e=>e.City == "London"));
```

```
_salesReps.AddRange(_entityManager.Employees.Where(e=>e.City == "London"))
```

You will end up with duplicate references to each of the London employees! Again, the general rule is that if you have a managed list, it is best not to attempt to populate it directly. Allow the *EntityListManager* to handle the work.

EntityListManagers and performance

*EntityListManager*s do create a certain amount of overhead, so be judicious in their use. It is also possible to narrow their scope of what they must monitor more than we did in our examples above. We instantiated our *EntityListManager* as follows:

```
var filter = new Predicate<Employee>(
 delegate(Employee anEmployee) { return anEmployee.City == "London"; });
EntityListManager<Employee> employeeEntityListManager =
 new EntityListManager<Employee>(_em1, filter, null);
```

```
Dim filter = New Predicate(Of Employee)(Function(anEmployee _
 As Employee) anEmployee.City = "London")
Dim employeeEntityListManager As New EntityListManager( _
 Of Employee)(_em1, Filter, Nothing)
```

The third argument, which we left null, is an array of *EntityProperty* objects. By leaving it null, we told the manager to submit any added or modified Employee to the test encoded in the filter Predicate. Suppose that, instead, we pass a list of properties of the Employee to this argument:

```
EntityListManager<Employee> employeeEntityListManager =
 new EntityListManager<Employee>(_entityManager, filter,
  new EntityProperty[] { Employee.CityEntityProperty });
```

```
Dim employeeEntityListManager As New EntityListManager( _
 Of Employee)(_em1, filter, New EntityProperty() {Employee.CityEntityProperty})
```

Now the *EntityListManager* will apply its test (about City being equal to London) only to an Employee whose City property, specifically, was modified. If you simply change only the Birthdate of an Employee already in the cache, the rule will not be evaluated. It can, after all, be safely assumed that said Employee would *already* be in the lists being managed if the value in its City property were “London”.

Coding more involved rules

In some of the examples above we passed an anonymous delegate to the constructor of the *Predicate* filter. That’s great for simple rules, but you can declare the predicate separately if you need to do something more involved. This also gives you a chance to name the rule, which can make your code more readable. Here’s a simple example:

```
private void SetUpEntityListManagerWithNamedDelegate() {
 // Identify Customer currently being edited by some process;
 // this is a stand-in.
 _currentCustomer = _em1.Customers.FirstOrNullEntity();
  EntityListManager<Order> orderEntityListManager =
 new EntityListManager<Order>(_em1, FilterOrdersByDate,
 new EntityProperty[] {
    Order.PropertyMetadata.OrderDate,
    Order.PropertyMetadata.Customer }
  );
}
/// <summary>
/// This rule gets the 1996 Orders for the current Customer
/// </summary>
/// <param name="pOrder"></param>
/// <returns></returns>
Boolean FilterOrdersByDate(Order pOrder) {
 return (pOrder.OrderDate.Value.Year == 1996 &&
    pOrder.Customer == _currentCustomer);
}
```

```
Private Sub SetUpEntityListManagerWithNamedDelegate()
 ' Identify Customer currently being edited by some process;
 ' this is a stand-in.
 _currentCustomer = _em1.Customers.FirstOrNullEntity()
 Dim orderEntityListManager As New EntityListManager(Of Order)(_em1, _
   AddressOf FilterOrdersByDate, New EntityProperty() { _
      Order.PropertyMetadata.OrderDate, Order.PropertyMetadata.Customer})
End Sub
''' <summary>
''' This rule gets the 1996 Orders for the current Customer
''' </summary>
''' <param name="pOrder"></param>
''' <returns></returns>
Private Function FilterOrdersByDate(ByVal pOrder As Order) As Boolean
 Return (pOrder.OrderDate.Value.Year = _
    1996 AndAlso pOrder.Customer.Equals(_currentCustomer))
End Function
```