Contents

Navigation properties can be loaded asynchronously as well as synchronously. In asynchronous environments such as Silverlight, Windows Store and mobile applications, all property navigation is done asynchronously, but you can enable asynchronous property navigation in other environments too.

EntityReferenceStrategy

You use the *EntityReferenceStrategy* to define how reference properties are loaded.

The *EntityManager.DefaultEntityReferenceStrategy* can be set on the manager to control all property navigation within the *EntityManager*. If you haven't set the *DefaultEntityReferencyStrategy*, then the *EntityReferenceStrategy.Default* is used. There are three predefined *EntityReferenceStrategy* choices, but you can also define your own. The predefined choices are:

EntityReferenceStrategy.Default - Lazily load the navigation property. Will be synchronous in .NET applications and asynchronous in other environments.
EntityReferenceStrategy.NoLoad - Suppress property loading. This is most frequently used when the reference data is already available in cache and you don't want a data source query to be executed.
EntityReferenceStrategy.DefaultAsync - Lazily load the navigation property, asynchronously.

Asynchronous property navigation presents a challenge: property navigation by its nature must return an immediate result. If the query needs to execute asynchronously, then this immediate result must have some way of indicating that it is not (yet) a real result and that the 'real' result is coming. This introduces the concepts of a *PendingEntity* and a *PendingEntityList*.

Scalar properties and the PendingEntity

A scalar navigation property is one which returns a single instance of the reference entity and not a collection of entities. For example, Order.Customer is a scalar navigation property.

If a scalar navigation property is loaded asynchronously it will be a *pending entity* until loaded. You can check if an entity instance is *pending* via the *EntityAspect.IsPendingEntity* property.

When a pending entity is finally loaded into cache, the *PendingEntityResolved* event will be fired.

For example:

```
 Order anOrder = await manager.Orders.AsScalarAsync().FirstOrDefault();
// Access the customer property:
var customer = anOrder.Customer;
bool isPending = customer.EntityAspect.IsPendingEntity;
if (isPending) {
   customer.EntityAspect.PendingEntityResolved += (o, e) => {
      MessageBox.Show("Your customer is here!");
   };
 }
```

Collection properties and the PendingEntityList

A navigation property returning a collection will return a *PendingEntityList* when it must be loaded asynchronously. Once loaded, the *PendingEntityListResolved* event is fired.

```
 Customer customer = await manager.Customers.AsScalarAsync().FirstOrDefault();
// Access the orders:
var orders = customer.Orders;
bool isPending = orders.IsPendingEntityList;
if (isPending) {
   orders.PendingEntityListResolved += (o,e) => {
      MessageBox.Show("Your orders have arrived!")
   };
 }
```

Deferred retrieval

When does the EntityManager fetch *myOrder*'s line items from the data source?

We might have written DevForce to fetch them automatically when it fetched *myOrder*. But if DevForce were to get the line items automatically, why stop there? It could get the customer for the order, the sales rep for the order, and the products for each line item.

Those are just the immediate neighbors. It could get the customer's headquarter address, the sales rep's address and manager, and each product's manufacturer. If it continued like this, it might fetch most of the database.

Retrieving the entire graph is obviously wasteful and infeasible. How often do we want to know the manager of the sales rep who booked the order? Clearly we have to prune the object graph. But where do we prune? How can we know in advance which entities we will need and which we can safely exclude?

We cannot know. Fortunately, we don't have to know. We don't have to know if we can be certain of continuous connection to the data source. If we expect the application to run offline, we'll have to anticipate the related entities we'll need and pre-fetch them. We'll get to this issue later. We keep it simple. We use an entity query to get the root entities (such as *myOrder*). Then we use entity navigation to retrieve neighboring related entities as we need them.

This just-in-time approach is called deferred retrieval (also known as "lazy instantiation", "lazy loading", "Just-In-Time [JIT] data retrieval", and so on).