Contents

We've seen how DevForce supports discovery by context, something particularly useful in testing. DevForce also contains built-in support for **faking** using this facility.

There are a few definitions of faking, but let's use one from Martin Fowler in his post "Mocks Aren't Stubs": *"**Fake** objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in memory database is a good example)."*

Faking with CompositionContext.Fake

CompositionContext.Fake is a composition context in which "fake" implementations of several important DevForce interfaces are provided to simulate end-to-end query and save operations against a real backing store. Faking is also useful in facilitating model-first development, and providing data and functionality for demo applications.

CompositionContext.Fake components include:

1. **Id generation** with the *FakeIdGenerator* to allow both temporary ids for new entities and translation to "real" ids when a save is performed.
2. An **in-memory data store** with the *EntityServerFakeBackingStore*.
3. **Query** functionality with the *EdmQueryExecutorFake*.
4. **Save** functionality with the *EdmSaveExecutorFake*.

These are not stubs. They are rich implementation that provide functionality almost identical to that of their "real" counterparts. Faked queries and saves return virtually the same results you would expect from a real data store, but instead use a fake data store, called the *EntityServerFakeBackingStore* . Once you populate this store, you can issue standard queries and saves from an *EntityManager* created for the context without any additional code.

Note that other DevForce extensions, such as *IEntityLoginManager*, are not faked, so either DevForce defaults or your custom implementations will be used for those.

You can create multiple named fake contexts, and custom contexts based on the standard *CompositionContext.Fake*. Each unique context uses its own backing store. With a custom fake context you can supply additional fake functionality of your own.

Simple Faking

At the simplest, all you need to do is create an *EntityManager* for the default fake context and issue queries and saves against it. Of course, unless you've pre-populated the fake backing store queries will return no data until you've first saved some entities.

Here we create a new *EntityManager* and populate the backing store with a few test entities (remember to add a using statement to *IdeaBlade.Core.Composition*):

```
 var em = new DomainModelEntityManager(compositionContextName: CompositionContext.Fake.Name);
 PopulateBackingStore(em);
```

```
Dim em As New DomainModelEntityManager(compositionContextName:= CompositionContext.Fake.Name)
 PopulateBackingStore(em)
```

```
private void PopulateFakeBackingStore(EntityManager em) {
 var customer = new Customer { CompanyName = "Test Company" };
 em.AddEntity(customer);
 var employee = new Employee { FirstName= "Fred", LastName = "Smith"};
 em.AddEntity(employee);
 for (int i = 0; i < 5; i++) {
   var salesOrder = new OrderSummary { Employee=employee, Customer = customer};
   em.AddEntity(salesOrder);
 }
 em.SaveChanges();
}
```

```
Private Sub PopulateFakeBackingStore(ByVal em As EntityManager)
  Dim customer = New Customer() With {.CompanyName = "Test Company"}
   em.AddEntity(customer)
  Dim employee = New Employee() With {.FirstName = "Fred", .LastName = "Smith"}
   em.AddEntity(employee)
  For i As Integer = 0 To 4
     Dim salesOrder = New OrderSummary() With {.Employee = employee, .Customer = customer}
      em.AddEntity(salesOrder)
  Next
   em.SaveChanges()
 End Sub
```

Once the backing store is populated with test data we can then use the *EntityManager* as usual. Here we clear it first, to ensure that the *EntityManager* is querying from the backing store.

```
em.Clear(); // clear cache; no memory of prior entities
var query = em.Customers.Include(c => c.OrderSummaries)
  .Where(c => c.OrderSummaries.Any(os => os.Employee.FirstName == "Fred"));
```

```
 em.Clear() ' clear cache; no memory of prior entities
Dim query As em.Customers.Include(Function(c) c.OrderSummaries).Where _
 (Function(c) c.OrderSummaries.Any(Function(os) os.Employee.FirstName = "Fred"))
```

Working directly with a fake backing store

You can access a fake backing store via a fake CompositionContext, for example:

```
var store EntityManager.CompositionContext.GetFakeBackingStore();
```

```
Dim store = EntityManager.CompositionContext.GetFakeBackingStore()
```

Local vs. remote stores

Different backing stores are used in 2-tier vs. n-tier applications, *EntityServerFakeBackingStore.Local* and *EntityServerFakeBackingStore.Remote* . The *Local* store, which is not available in Silverlight and Windows Store applications, allows you to perform synchronous queries and saves when not using a remote *EntityServer*. The *Remote* store is used with n-tier applications and any others using a remote *EntityServer*. Generally, you won't be concerned whether the store in use is local or remote unless you're working directly with it, for example to clear or populate it.

Store methods

The entities stored and retrieved by the *EntityServerFakeBackingStore* are actually held in an *EntityCacheState* . The *EntityServerFakeBackingStore* provides methods to allow you to work directly with both it and the underlying *EntityCacheState*. You can use *Save* or *SaveAsync* to save the store to a file or stream, and *Restore* or *RestoreAsync* to restore data into the store. *Clear* or *ClearAsync* can be used to clear the store (for example to clear the store between tests).

Note that only the asynchronous versions of these methods are available in async environments such as Silverlight and Windows Store applications.

The *EntityCacheState* makes it easy to load test data into the backing store, possibly from a test database. The *EntityCacheState* can be initially loaded from an *EntityManager*, and allows you to save to and restore from a file or stream.