Contents

In this segment, we introduce automated testing a DevForce Code model. The lessons apply to all DevForce entity models, whether built "Code First" or "Database First" with an EDM.

We feel automated testing should be an essential part of any development regime. We recommend testing as you go rather than bolting it on at the end. In that respect, we've waited too long. The future evolution of the CodeFirstWalk sample will include tests and often rely on tests to explain each step.

In this lesson, we'll build some preliminary tests that run green:

- **Platform:** WPF
- **Language:** C#
- **Download:** Code First Walkthrough

Establish an MS-Test project for Model testing

1. **Add | New Project | Test Project**

2. Call it **CodeFirstWalk.Model.Test**.

3. **Add a reference to the model project**, *CodeFirstWalk.Model*.

4. **Add DevForce references:**

- *IdeaBlade.Core*
- *IdeaBlade.EntityModel*
- *IdeaBlade.EntityModel.Server*
- *IdeaBlade.Linq*
- *IdeaBlade.Validation*

5. **Set "Copy Local=True"** in the Properties window for all of them.

6. **Set "Specific Version=False"** for all of them.

7. **Disable PostSharp analysis** for the test project.

- Remove the *RequiresPostSharp.** file if present.

8. **Delete the generated constructor and *TestContext*** stuff because not used.

9. **Delete the "Additional test attributes" region** when you understand the comments.

10. **Rename *TestMethod1 -> DoNothing*.**

11. **Change its body** to *Assert.Fail("Fail on purpose");* .

12. **Run all tests [Ctrl-R, A]** … and **see the FAILURE** report in the Test Results window

These Visual Studio keyboard shortcuts are timesavers:
[Ctrl-R, A] **–** Run all tests.
[Ctrl-R, T] **–** Run the test my mouse is in; run the tests in the test class my mouse is in.
[Ctrl-R, Ctrl-A] **–** Debug all test(s)
[Ctrl-R, Ctrl-T] **–** Debug test(s) selected as in [Ctrl-R, T]

13. **Delete** *DoNothing*.

First offline EntityManager Test

We will do all testing for a long while with offline *EntityManagers*.

Most testing **can** be done with offline *EntityManagers*.

Most testing **should** be done with offline *EntityManagers*. Most tests don't need to involve a server or the Entity Framework or a database. These are downstream potential points of test failure that have nothing to do with the subject of your test. We will need to involve them – to use a connected *EntityManager* – when we write end-to-end integration tests. But here we are only interested in the interaction of application code with the model.

The *EntityManager* is an essential dependency when testing most entity model functionality. But we can limit the risk of downstream failures by keeping the *EntityManager* offline and working out of its cache of entities … which we control completely during test setup.

1. **Rename the class file** to *When_exploring_offline_ProductEntityManager*.

   Visual Studio should offer the opportunity to rename the class as well; say "Yes". If it doesn't …

2. **Rename the class** to *When_exploring_offline_ProductEntityManager*.

3. **Add public Manager property** returning the type of the model's *EntityManager* which is *ProductEntities* in our example. The property is public because we'll probably relocate this material for reuse in other test classes down the road.

4. **Add** *TestInitialize* method at the top, method runs before every test in the test class

5. **Assign** *Manager* inside *TestInitialize* with a new instance of *ProductEntities* that is disconnected.
The current state of the code should be this:

```
public ProductEntities Manager { get; set; }
[TestInitialize]
public void TestInitialize()
{
    Manager = new ProductEntities(shouldConnect: false);
}
```

Add "offline-optional" constructor to ProductEntities

The base *EntityManager* class has numerous optional constructors, many with optional parameters. We haven't needed any of them to date. Now we need to implement one of them (or a part of it).

1. **Go to the Model** file and find *ProductEntities*.

2. **Add** a constructor with optional *shouldConnect* parameter:

```
public ProductEntities(bool shouldConnect = true) : base(shouldConnect) { }
```

Make sure test manager never fetches or saves

We know the manager is offline when we create it. We don't want a test to get sneaky and connect it. Let's fail any test that tries to query from or save to the database.

1. **Return** to the test project which can now compile.

2. **Add** *using IdeaBlade.EntityModel;* .

3. **Add Fetching and Saving event handlers** that fail the test when called:

```
[TestInitialize]
public void TestInitialize()
{
    Manager = new ProductEntities(shouldConnect: false);
    Manager.Fetching += (s, e) => Assert.Fail("Manager tried to fetch entities.");
    Manager.Saving += (s, e) => Assert.Fail("Manager tried to save.");
}
```

4. **Add test for save blocking.**

```
[TestMethod]
[ExpectedException(typeof(EntityManagerSaveException))]
public void Fails_if_you_try_to_save()
{
    Manager.Connect(); // no-no
    Manager.SaveChanges(); // should catch and fail
}
```

Notice how the *ExpectedException* MS-Test attribute asserts that the body of the test will throw an exception of a particular type.

5. **Run the test** by placing the mouse somewhere in the test and pressing [Ctrl-R, T].

Write your first EntityManager test

1. **Write and execute the test** *Can_query_added_supplier* as follows:

```
[TestMethod]
public void Can_query_added_supplier()
{
    var supplier = new Supplier {CompanyName = "Test Supplier"};
    Manager.AddEntity(supplier);
    Assert.AreSame(supplier, Manager.Suppliers.FirstOrDefault());
}
```

Congratulations! Your first real DevForce test demonstrates that you can add a new entity to cache and get it back as if you were querying from the database.

Of course this is not a test of the Manager's actual ability to query the database. We can test that elsewhere. Our immediate goal is to test code that "thinks" it is querying for entities and is doing something useful with them afterward. We're testing (a) how the code responds to querying outcomes and (b) whatever it is that the code is doing with the entity.

Test entity business logic

Let's try testing the small bits of business logic that we added to the *Supplier* entity in our model:

- The *CompanyName* property returns an uppercase value, no matter what its internal value.
- *CompanyName* is required and has a maximum length of 20 characters.

1. **Write and call test of the uppercasing property interceptor.**

```
[TestMethod]
public void Then_Supplier_CompanyName_is_uppercased_on_get()
{
    const string companyName = "lower case name";
    var detachedSupplier = new Supplier {CompanyName = companyName};
    Assert.AreEqual(companyName.ToUpper(), detachedSupplier.CompanyName);
}
```

Property interceptors operate whether or not the entity is attached to an *EntityManager* so we didn't even bother adding the supplier to the *Manager*.

However, property validation **is disabled** until the entity becomes attached, as we see in this test:

```
[TestMethod]
public void Then_detached_Supplier_CompanyName_is_not_required()
{
    var supplier = new Supplier { CompanyName = "Supplier" };
    // Manager.AddEntity(supplier); // comment out to keep supplier detached
```

```
    supplier.CompanyName = string.Empty;
    var msg = GetFirstValidationErrorMessage(supplier);
    Assert.IsNull(msg, "Did not expect the validation error: " + msg);
}
```

The validation test requires the following *GetFirstValidationErrorMessage* helper method:

```
public string GetFirstValidationErrorMessage(IEntity entity)
{
    var firstError = entity.EntityAspect.ValidationErrors.FirstOrDefault();
    return (null == firstError) ? null : firstError.Message;
}
```

Notice how the helper method acquires instance validation error information by way of the *EntityAspect* propertythat we added to our model's *BaseEntity* class.

2. **Run the test**. If the "Required" rule were functioning, we would detect a validation error when we set the *CompanyName* to the empty string. The *Assert* confirms that the entity is unaware of the error.

You can validate a detached entity explicitly if you wish. The property validation **on set** is disabled for detached entities but the rule is still there to be enforced … as shown in the lesson appendix.

After attaching a Supplier to the Manager, the required and length validation rules are applied when the property is set as these tests show.

```
[TestMethod]
public void Then_attached_Supplier_errs_when_CompanyName_set_StringEmpty()
{
    var supplier = new Supplier {CompanyName = "Supplier"};
    Manager.AddEntity(supplier);
    supplier.CompanyName = string.Empty;
    var msg = GetFirstValidationErrorMessage(supplier);
    Assert.IsNotNull(msg,
            "Expected a validation error when CompanyName set to empty string.");
    Assert.IsTrue(msg.Contains("required"),
            "Error message did not contain 'required'; was " + msg);
}
[TestMethod]
public void Then_attached_Supplier_errs_when_CompanyName_set_null()
{
    var supplier = new Supplier {CompanyName = "Supplier"};
    Manager.AddEntity(supplier);

    supplier.CompanyName = null;
    var msg = GetFirstValidationErrorMessage(supplier);
    Assert.IsNotNull(msg,
            "Expected a validation error when CompanyName set to null.");
    Assert.IsTrue(msg.Contains("required"),
            "Error message did not contain 'required'; was " + msg);
}
[TestMethod]
public void Then_attached_Supplier_errs_when_CompanyName_set_too_long()
{
    var supplier = new Supplier { CompanyName = "Supplier" };
    Manager.AddEntity(supplier);
    supplier.CompanyName = "Supplier name with more than 20 characters";
    var msg = GetFirstValidationErrorMessage(supplier);
    Assert.IsNotNull(msg,
        "Expected a validation error when CompanyName > 20 chars.");
    Assert.IsTrue(msg.Contains("20"), // looking for "cannot be longer than 20 character(s)
        "Error message did not contain '20'; msg = " + msg);
}
```

Prepare test entities with a "Data Mother"

We often write a battery of tests that work with a bunch of related test entities. We will want to pretend that these test entities either are in the database already or were freshly queried. We don't need – or want - an actual database of test entities to fulfill our testing intentions.

We could code these test entities within the body of a test … as we've been doing so far. That's often a good practice because it keeps our tests from relying on far away code that we can't see.

But writing the same test entity setups over and over is tedious setup and can distract attention from the test purpose. It's usually better to delegate this kind of setup to a helper method … or such a method in a helper class which is sometimes known as a "Data Mother".

1. **Write this public *PopulateTestManager* method**.

```csharp
// Add test entities to the Manager
public void PopulateTestManager(EntityManager manager)
{
    testSupplier = new Supplier { CompanyName = "Test Supplier" };
    manager.AttachEntity(testSupplier); // attached unchanged as if from query
    testCategory = new Category {CategoryId = 123, CategoryName = "Test Category"};
    manager.AttachEntity(testCategory); // attached unchanged as if from query
}
public Supplier testSupplier;
public Category testCategory;
```

The treatment of *testSupplier* and *testCategory* is almost the same as in our earlier tests. The important difference is the call to *AttachEntity()* instead of *AddEntity()*. **AttachEntity()** puts the entity in cache in an "*Unchanged*" state, as if it had been queried from the database.

2. **Write a test** to confirm that the *testCategory* is configured as we expect. Put it above *PopulateTestManager* then run it.

```csharp
[TestMethod]
public void Then_testCategory_is_valid()
{
    PopulateTestManager(Manager);
    var cat = Manager.Categories.First();
    Assert.AreSame(testCategory, cat,
        "query didn't return the testCategory");
    var state = cat.EntityAspect.EntityState;
    Assert.AreEqual(state, EntityState.Unchanged,
        "unexpected category EntityState, " + state);
    Assert.IsTrue(cat.CategoryId > 0,
        "CategoryId appears to be a temporary id, " + testCategory.CategoryId);
}
```

3. **Add some test products** to *PopulateTestManager*. After revision, it looks like this:

```csharp
// Add test entities to the Manager
public void PopulateTestManager(EntityManager manager)
{
    testSupplier = new Supplier { CompanyName = "Test Supplier" };
    manager.AttachEntity(testSupplier); // attached unchanged as if from query
    testCategory = new Category { CategoryId = 123, CategoryName = "Test Category" };
    manager.AttachEntity(testCategory); // attached unchanged as if from query
    testProduct1 = new Product
                {
                    ProductName = "Product 1",
                    Category = testCategory,
                    Supplier = testSupplier,
                    ProductId = 1, // must set after navigation property!
                };
    testProduct2 = new Product
                {
                    ProductName = "Product 2",
                    Category = testCategory,
                    Supplier = testSupplier,
                    ProductId = 2, // must set after navigation property!
                };
    // Setting a Product's navigation property pulls it into cache as an ADDED entity
    Manager.AcceptChanges(); // makes everything in cache appear as if queried.
}
public Supplier testSupplier;
public Category testCategory;
public Product testProduct1;
public Product testProduct2;
```

The additional definitions of the test products are straightforward. There are three notable DevForce effects addressed in this implementation:

- DevForce pulls the products into the Manager's cache as entities in "added" state *immediately* after the code first sets a navigation property with an entity in cache (*Category*).
- DevForce sets the product keys with temporary key values *immediately* upon entering the cache; therefore, if we want our own test key values, we have to re-set the product keys **after** the product enters the cache.
- Because the products are in the "added" state, we must flip them to "*Unchanged*" state when we're done adding them … which we do by accepting all changes pending in the Manager.

4. **Write a test** to confirm that the *testProduct1* is configured as we expect. Put it above *PopulateTestManager* then run it.

```csharp
[TestMethod]
public void Then_testProduct_is_valid()
{
  PopulateTestManager(Manager);
  var state = testProduct1.EntityAspect.EntityState;
  Assert.AreEqual(state, EntityState.Unchanged,
    "unexpected testPoduct EntityState, " + state);
  Assert.IsTrue(testProduct1.ProductId > 0,
    "ProductId appears to be a temporary id, " + testProduct1.ProductId);
  Assert.AreSame(testCategory, testProduct1.Category, "unexpected testProduct Category");
  Assert.AreSame(testSupplier, testProduct1.Supplier, "unexpected testProduct Supplier");
}
```

In this "Prepare Test Entities" segment, the only testing we did is of our ability to write test entities. Yes, it's a good idea to validate the assumptions you have about your test data. And in the process, we've provided clues for testing facts about individual entities and their relationships to each other.

But until we use these entities to test our application, we've wasted time. Treat this work as a promise of future utility, a promise we'll redeem in an upcoming segment.

Lesson Appendix: validate on-demand

We showed earlier that property validation – automatic validation when a property is set – is enabled only for entities attached to an *EntityManager*. But you can always validate any object, attached or not, by acquiring a DevForce *VerifierEngine* and passing the object into its Execute method.

In the following test, we acquire a *VerifierEngine* from the Manager and use it to validate a detached *Supplier* instance with a bad *CompanyName*. A comment shows that we could have created a new *VerifierEngine* if we preferred.

```csharp
[TestMethod]
public void Can_validate_detached_Supplier()
{
  var supplier = new Supplier { CompanyName = "Supplier" };
  // Manager.AddEntity(supplier); // commented out to keep supplier detached
  supplier.CompanyName = string.Empty;
  // var engine = new IdeaBlade.Validation.VerifierEngine();
  var engine = Manager.VerifierEngine;
  var validationResults = engine.Execute(supplier);
  var firstErr = validationResults.FirstOrDefault(r => r.IsError);
  Assert.IsNotNull(firstErr, "Expected a validation error");
  var msg = firstErr.Message;
  Assert.IsTrue(msg.Contains("required"),
        "Error message did not contain 'required'; was " + msg);
}
```