Contents

A discussion on development isn't complete without a mention of **testing**. We focus here primarily on unit and integration testing - activities often performed by the developer.

We know that some of you still aren't writing unit tests. You say it's too hard, or you don't have time.

It's no longer hard, since test tools are built into Visual Studio 2012, and if you're not happy with those there are free 3rd party tools such as NUnit, xUnit, and more. Surely you can find a test framework to suit you.

But you don't have the time? Yet you do have time to spend countless hours wading through your UI and code to isolate a single failing line of code, which could have been easily found and remedied, or maybe even avoided, if unit tests existed.

We'll offer two pieces of advice:

- Design with testability in mind - Are you using MVVM? If so, then creating unit tests for your view models is easy. Using the Repository pattern? Then a test or fake repository can help.
- Write unit tests as you go - Writing unit tests for new code is much easier than trying to add the tests later. Code coverage will be better, and tooling in Visual Studio 2012 makes creating test projects and tests easier.

Testing features in DevForce

DevForce can help make testing easier.

Test data

Unit tests shouldn't go to the database, and you don't want to fire up IIS for your *EntityServer*, yet you want data for your testing. You also want to refresh the data easily so that the tests don't corrupt it.

Most of the techniques for creating design time data also apply to unit tests. The *EntityManager* can be used in "offline" mode - created without connecting to the *EntityServer*, and the *DefaultQueryStrategy* set to *CacheOnly* so that no queries are sent to the server. Data can be loaded from a previously created *EntityCacheState*; or simple entities created, and then added or attached to the *EntityManager*. Your tests can then run queries and perform property navigation, without requiring a database or potentially asynchronous service calls.

Mocks and fakes

DevForce provides contextual discovery via the *CompositionContext*, which is particularly useful in creating custom fakes and mocks for testing.

Using *CompositionContext.Fake* and the *EntityServerFakeBackingStore*, you can test much of your code without ever touching a database.

Data source extensions

When you're ready to test against a database, you can dynamically target different databases using data source extensions. You likely use one database during development, and other databases in your staging, test and production environments. Data source extensions allow you to easily switch among these databases depending on the environment.

A note on Windows Store testing

While unit testing your application you likely won't need to communicate with the EntityService, but when running integration tests you may run into connectivity issues. You'll need to be sure you've added a loopback exemption for your test suite in order to call the EntityService. More information here: http://msdn.microsoft.com/en-us/library/windows/apps/hh780593.aspx.

Additional resources

Ward Bell on "The Art of Unit Testing"

Patterns and practices - Many of our patterns and practices for DevForce help with building more testable components.

MSDN documentation on creating unit tests