In this article we will implement a data structure on top of a FoundationDB database. We will focus on how to test our implementation, using pytest. This page assumes familiarity with the python bindings.

Hello World

We'll need an instance of a `db` object which obeys the contract of the FoundationBD client. Assuming we have that, we can write a simple test like this

```
def test_simple(db):
    @fdb.transactional
    def simple(tr):
        tr[b'x'] = b'y'
        assert tr[b'x'] == b'z'
    simple(db)
```

We'll use a pytest fixture to provide a `db` implementation.

The simplest implementation would be this:

```
@pytest.fixture(scope='function')
def db():
    yield fdb.open()
```

Putting it all together, here's our "hello world" test:

```
$ ls
test_simple.py

$ cat test_simple.py
import pytest
import fdb
fdb.api_version(620)

@pytest.fixture(scope='function')
def db():
    yield fdb.open()

def test_simple(db):
    @fdb.transactional
    def simple(tr):
        tr[b'x'] = b'y'
        assert tr[b'x'] == b'z'
    simple(db)

$ pytest
...
tr = <fdb.impl.Transaction object at 0x7fe207915ba8>
```

```
    @fdb.transactional
    def simple(tr):
        tr[b'x'] = b'y'
>       assert tr[b'x'] == b'z'
E       AssertionError: assert b'y' == b'z'
E         At index 0 diff: b'y' != b'z'
E         Use -v to get the full diff

test_simple.py:13: AssertionError
================================================== 1 failed in 0.07s ======================================
```

Indeed 'y' != 'z', and the test is working.

Cleaning up the keyspace

If you want to start each test with an empty keyspace, you might implement your fixture like this instead:

```
@pytest.fixture(scope='function')
def db():
    @fdb.transactional
    def database_empty(tr):
        return len(list(tr.get_range(b'', b'\xff', limit=1))) == 0
    db = fdb.open()
    assert database_empty(db)
    yield db
    del db[b'':b'\xff']
```

This double checks that the database is empty at the beginning of the test (make sure you're using a test-only database), and clears the keyspace after the test.

Testing a "counted set" abstraction

A common design pattern is to record the number of key-pairs in a subspace in a separate key, and update it transactionally. This way you can get the count without doing a full range scan.

Here's an example implementation (counted_set.py)

```
import fdb
fdb.api_version(620)
import struct
```

```python
@fdb.transactional
def add_item(tr, set_name, item):
    keySubspace = fdb.directory.create_or_open(tr, (b'countedSet', set_name))
    if tr[keySubspace.pack((item, ))] == None:
        tr[keySubspace.pack((item, ))] = b''
        countSubspace = fdb.directory.create_or_open(
            tr, (b'countedSet', set_name, b'count'))
        tr.add(countSubspace.pack((b'count', )), struct.pack('<q', 1))


@fdb.transactional
def remove_item(tr, set_name, item):
    keySubspace = fdb.directory.create_or_open(tr, (b'countedSet', set_name))
    if tr[keySubspace.pack((item, ))] != None:
        del tr[keySubspace.pack((item, ))]
        countSubspace = fdb.directory.create_or_open(
            tr, (b'countedSet', set_name, b'count'))
        tr.add(countSubspace.pack((b'count', )), struct.pack('<q', -1))


@fdb.transactional
def cardinality(tr, set_name):
    countSubspace = fdb.directory.create_or_open(
        tr, (b'countedSet', set_name, b'count'))
    if tr[countSubspace.pack((b'count', ))] == None:
        return 0
    return struct.unpack('<q', bytes(tr[countSubspace.pack((b'count', ))]))[0]
```

The main invariant we want for this abstraction is that the cardinality is the same as the number of keys in the subspace.

We can test this invariant like so:

```python
def test_cardinality(db):
    @fdb.transactional
    def debug_cardinality(tr, set_name):
        s = fdb.directory.create_or_open(tr, (b'countedSet', set_name))
        assert cardinality(tr, set_name) == sum(1 for _ in tr[s.range()])
    for i in range(10):
        add_item(db, b'mySet', i)
    debug_cardinality(db, b'mySet')
    for i in range(10):
        remove_item(db, b'mySet', i)
    debug_cardinality(db, b'mySet')
```

Exercising the retry loop

A possible bug in the implementation of `add_item` is to increment the count even if the key is already present, but currently our test will not catch this bug.

We can use client buggify to increase our test coverage with a simple change to our test fixture.

```
@pytest.fixture(scope='function')
def db():
    @fdb.transactional
    def database_empty(tr):
        return len(list(tr.get_range(b'', b'\xff', limit=1))) == 0
    db = fdb.open()
    fdb.options.set_client_buggify_enable() # add this line
    assert database_empty(db)
    yield db
    fdb.options.set_client_buggify_disable() # add this line
    del db[b'':b'\xff']
```

Now our test will catch the bug (you may need to run the test a few times)

This will cause your transaction retry attempts to backoff as if the cluster were under high load, so to speed up your tests you may want to consider lowering the max retry delay in your db fixture (again make sure you're using a test-only database, where a lower retry delay is appropriate).