Ziffect Documentation

Release 0.0.1

Marcus Henry Ewert

February 06, 2019

Contents

1 Introduction **3**
1.1 Motivation . . . 3

2 Coding with effect **5**
2.1 TypeDispatchers are just classes . . . 14

3 Coding with ziffect **17**
3.1 Summary . . . 22
3.2 Future Work . . . 22

4 API **25**
4.1 ziffect . . . 25
4.2 ziffect.matchers . . . 25

5 Indices and tables **27**

Python Module Index **29**

Contents:

CHAPTER 1

Introduction

1.1 Motivation

The motivation for `ziffect` was an inner sensation that effect was slightly incomplete, and with the help of zope.interface and pyrsistent it could be made a lot better.

In order to justify this library I will attempt to write the same bit of code using pure `Effect` and using pure `ziffect`

CHAPTER 2

Coding with effect

Let's walk through an example to illustrate my grievances with the `effect` library. For starters, let's say we are using `effect` to interact with a database. Reading values from and writing values to a database are certainly operations that have side-effects, so we believe this to be a good candidate use case for our new toy.

Aside

Apologies for this rather long example, I just wanted to walk through a sufficiently complex scenario as a matter of proving to myself that this library adds value.

For sake of example I will assume we are using a simple revision-based document store (perhaps a wrapper on CouchDB). This document store has a simple synchronous python API that consists of merely `db.get(doc_id, rev=LATEST)` and `db.put(doc_id, rev, doc)`. As this is a fictional API, rather than giving a full spec, I will demonstrate how it works with a simple demo of functionality:

```
>>> # Make a new db.
>>> db = DB()
>>> # Create an id for a doc we'll work with.
>>> my_id = uuid4()

>>> # Getting a doc that doesn't exist is an error:
>>> db.get(my_id)
DB Response<NOT_FOUND>

>>> # Putting revision 0 for a doc that doesn't exist succeeds:
>>> db.put(my_id, 0, {'cat': 0})
DB Response<OK rev=0>

>>> # `get`ing a doc gets the latest version:
>>> db.get(my_id)
DB Response<OK rev=0 {"cat": 0}>

>>> # Attempting to put a document at existant revision is an error:
>>> db.put(my_id, 0, {'cat': 12})
DB Response<CONFLICT>

>>> # Instead `put` it at the next revision:
>>> db.put(my_id, 1, {'cat': 12})
DB Response<OK rev=1>

>>> # `get`ing a doc gets the latest version:
>>> db.get(my_id)
```

```
DB Response<OK rev=1 {"cat": 12}>

>>> # But old revisions can still be gotten:
>>> db.get(my_id, 0)
DB Response<OK rev=0 {"cat": 0}>
```

Using this system, we will try to implement a piece of code that will execute a change on a document in the database. This code should take as inputs:

- A `DB` instance where the document is stored.
- The `doc_id` of the document that is to be changed within the database.
- A pure function to execute on the document.

The code will get the document from the database, execute the pure function on the document, and put it back in the database. If the `put` fails, then the code should get the latest version of the document, execute the pure function on the latest version of the document, attempt to `put` it again, and repeat until it succeeds.

For good measure, this code can return the final version of the document.

So let's take a stab at implementing this piece of code. We are using effect, so I guess that means we want to put `db.get` and `db.put` behind intents and performers, and then we want to create a function that returns an "effect generator" that can be performed by a dispatcher.

Aside

I'm still pretty new to `effect`, and playing around with how to do good design in this paradigm. You may notice this in my tenative design desisions. If you have any recommendations on how I could do it better, tell me on github as an issue filed against ziffect.

```
from effect import TypeDispatcher, sync_performer

class GetIntent(object):
  def __init__(self, doc_id, rev=LATEST):
    self.doc_id = doc_id
    self.rev = rev


def get_performer_generator(db):
  def get(dispatcher, intent):
    return db.get(intent.doc_id, intent.rev)
  return get


class UpdateIntent(object):
  def __init__(self, doc_id, rev, doc):
    """
    Slightly different API that the DB gives us, because we need to update a
    document below rather than just put a new doc into the DB.

    :param doc_id: The document id of the document to put in the database.
    :param rev: The last revision gotten from the database for the document.
      This update will put revision rev + 1 into the db.
    :param doc: The new document to send to the server.
    """
    self.doc_id = doc_id
    self.rev = rev
    self.doc = doc
```

```
def update_performer_generator(db):
  def update(dispatcher, intent):
    intent.rev += 1
    return db.put(intent.doc_id, intent.rev, intent.doc)
  return update


def db_dispatcher(db):
  return TypeDispatcher({
    GetIntent: sync_performer(get_performer_generator(db)),
    UpdateIntent: sync_performer(update_performer_generator(db)),
  })
```

Okay, so now we have the `Effect` -ive building blocks that we can use to create our implementation:

```
from effect import Effect
from effect.do import do

@do
def execute_function(doc_id, pure_function):
  result = yield Effect(GetIntent(doc_id=doc_id))
  new_doc = pure_function(result.doc)
  yield Effect(UpdateIntent(doc_id, result.rev, new_doc))
```

We still don't technically have what we set out for, as this effect generator only takes two arguments, not the underlying db. So we'll add one more convenience function that we can play around with on the interpreter:

```
from effect import (
  sync_perform, ComposedDispatcher, base_dispatcher
)

def sync_execute_function(db, doc_id, function):
  dispatcher = ComposedDispatcher([
    db_dispatcher(db),
    base_dispatcher
  ])
  sync_perform(
    dispatcher,
    execute_function(
      doc_id, function
    )
  )
```

The implementation of `execute_function` should fairly obviously have bugs, but it's a good enough implementation that we can convince ourselves that the happy case works:

```
>>> db = DB()
>>> doc_id = uuid4()
>>> doc = {"cat": "mouse", "count": 10}
>>> db.put(doc_id, 0, doc)
DB Response<OK rev=0>

>>> def increment(doc_id):
...     return sync_execute_function(
...         db,
...         doc_id,
...         lambda x: dict(x, count=x.get('count', 0) + 1)
...     )
```

```
>>> increment(doc_id)
>>> db.get(doc_id)
DB Response<OK rev=1 {"cat": "mouse", "count": 11}>

>>> increment(doc_id)
>>> db.get(doc_id)
DB Response<OK rev=2 {"cat": "mouse", "count": 12}>

>>> increment(doc_id)
>>> db.get(doc_id)
DB Response<OK rev=3 {"cat": "mouse", "count": 13}>
```

In the interest of test driven development, at this point we want to write our unit tests. They should fail, then we'll fix the implementation of `execute_function`, write more unit tests, etc.

```
from effect.testing import perform_sequence

class DBExecuteFunctionTests(TestCase):

  def test_happy_case(self):
    doc_id = uuid4()
    doc_1 = {"test": "doc", "a": 1}
    doc_1_u = {"test": "doc", "a": 2}
    seq = [
      (GetIntent(doc_id),
        lambda _: DBResponse(status=DBStatus.OK, rev=0, doc=doc_1)),

      (UpdateIntent(doc_id, 0, doc_1_u),
        lambda _: DBResponse(status=DBStatus.OK)),
    ]
    perform_sequence(seq, execute_function(
        doc_id, lambda x: dict(x, a=x.get("a", 0) + 1)
      )
    )

  def test_sad_case(self):
    doc_id = uuid4()
    doc_1 = {"test": "doc", "a": 1}
    doc_1_u = {"test": "doc", "a": 2}
    doc_2 = {"test": "doc2", "a": 5}
    doc_2_u = {"test": "doc2", "a": 6}
    seq = [
      (GetIntent(doc_id),
        lambda _: DBResponse(status=DBStatus.OK, rev=0, doc=doc_1)),

      (UpdateIntent(doc_id, 0, doc_1_u),
        lambda _: DBResponse(status=DBStatus.CONFLICT)),

      (GetIntent(doc_id),
        lambda _: DBResponse(status=DBStatus.OK, rev=1, doc=doc_2)),

      (UpdateIntent(doc_id, 1, doc_2_u),
        lambda _: DBResponse(status=DBStatus.OK)),
    ]
    perform_sequence(seq, execute_function(
        doc_id, lambda x: dict(x, a=x.get("a", 0) + 1)
      )
    )
```

Now a few iterations of TDD:

```
>>> run_test(DBExecuteFunctionTests)
FAILURE(test_happy_case)
Traceback (most recent call last):
  File "<interactive-shell>", line 17, in test_happy_case
  File "effect/testing.py", line 115, in perform_sequence
    return sync_perform(dispatcher, eff)
  File "effect/_sync.py", line 34, in sync_perform
    six.reraise(*errors[0])
  File "effect/_base.py", line 78, in guard
    return (False, f(*args, **kwargs))
  File "effect/do.py", line 121, in <lambda>
    error=lambda e: _do(e, generator, True))
  File "effect/do.py", line 98, in _do
    val = generator.throw(*result)
  File "<interactive-shell>", line 6, in execute_function
  File "effect/_base.py", line 150, in _perform
    performer = dispatcher(effect.intent)
  File "effect/testing.py", line 108, in dispatcher
    intent, fmt_log()))
AssertionError: Performer not found: <GetIntent object at 0x7fff0000>! Log follows:
{{{
NOT FOUND: <GetIntent object at 0x7fff0000>
NEXT EXPECTED: <GetIntent object at 0x7fff0001>
}}}
...
```

First bug: Intents need to have valid `__eq__` implementations. Also let's give them a `__repr__` that makes them slightly less hard to work with.

```
class GetIntent(object):
  def __init__(self, doc_id, rev=LATEST):
    self.doc_id = doc_id
    self.rev = rev

  def __eq__(self, other):
    return (
      type(self) == type(other) and
      self.doc_id == other.doc_id and
      self.rev == other.rev
    )

  def __repr__(self):
    return 'GetIntent<%s, %s>' % (
      rev_render(self.rev), self.doc_id)


class UpdateIntent(object):
  def __init__(self, doc_id, rev, doc):
    self.doc_id = doc_id
    self.rev = rev
    self.doc = doc

  def __eq__(self, other):
    return (
      type(self) == type(other) and
      self.doc_id == other.doc_id and
      self.rev == other.rev and
```

```
      self.doc == other.doc
    )

  def __repr__(self):
    return 'UpdateIntent<%s, %s, %s>' % (
      rev_render(self.rev),
      self.doc_id,
      repr(self.doc)
    )
```

Rerun the tests:

```
>>> run_test(DBExecuteFunctionTests)
FAILURE(test_sad_case)
Traceback (most recent call last):
  File "<interactive-shell>", line 41, in test_sad_case
  File "effect/testing.py", line 115, in perform_sequence
    return sync_perform(dispatcher, eff)
  File "effect/testing.py", line 463, in consume
    [x[0] for x in self.sequence]))
AssertionError: Not all intents were performed: [GetIntent<LATEST, f456150c-d4ba-5b09-a3fc-7ce3a7dbe9
...
```

Cool, now that we have a failing test, lets improve our implementation to handle the case where the DB was updated while we were running:

```
@do
def execute_function(doc_id, pure_function):
  done = False
  while not done:
    original_doc = yield Effect(GetIntent(doc_id=doc_id))
    new_doc = pure_function(original_doc.doc)
    update_result = yield Effect(
      UpdateIntent(doc_id, original_doc.rev, new_doc))
    done = (update_result.status == DBStatus.OK)
```

Rerun the tests:

```
>>> run_test(DBExecuteFunctionTests)
[OK]
```

Okay, so that all seems reasonable. This style of testing reminds me a lot of mocks. I am creating a canned sequence of expected inputs and return values for my dependencies, and running my code under test using this canned dependency.

Aside

I'm sure you can search the internet for debates of mocks versus fakes and find out more about the issues that some people have with mocks. In my view, two of the best arguments against mocks are:

- Does the mock sufficiently behave like a real implementation so that the test is meaningful? This is particularly pertinent in python, because something simple like, "your mock does not return the correct type of value" might mean that your unit test fails to catch a `TypeError` that will always happen with the real implementation.
- Mocks create tests that are tightly tied to the implementation of the code under test; if the implementation is changed, the test must also be modified. Consider, for instance, if we add a 2nd GetIntent to the beginning of the implementation, it should not change the correctness, but the test would now fail without modification. Specifically the sequence that is passed to perform_sequence would need a second GetIntent call at the beginning of the sequence.

Personally, I think mocks do have a place in unit tests like the one above. Specifically you are interfacing with an API that can return different values for the same inputs, and you need to force some external state change at a specific time in order to force the different inputs.

There are other strategies to do similar testing, but as long as you have a solid, simple interface to mock, I believe that form of testing gets the most bang for your buck.

Let's build on our existing implementation. Let's say after using this code for awhile we realize that the DB commands can also return a `NETWORK_ERROR`. We are going to take the simple policy of retrying any attempt that results in a `NETWORK_ERROR`. We are not going to bother with exponential back-off or any other nice-to-have right now, just a dead simply retry.

Aside

Assuming that `NETWORK_ERRORS` can happen before or after an operation is complete, this has some interesting ramifications. Our implementation of `execute_function()` will be an at-least-once implementation, where it guarantees that the function you specified will have occured at least once on the doc_id specified. A poorly timed `NETWORK_ERROR` after a successful update will cause our code to retry the update, get a conflict, and cycle through the code again.

In response to some of the fears about using mocks, lets utilize an `InMemoryDB` fake and a `NetoworkErrorDB` fake in the next implementation. This will force our tests to actually test in the performers in conjunction with the other code. We are still using `perform_sequence` to inject the fakes in a mock-like manner mind you.

```
class NetworkErrorDB(object):
  def get(self, doc_id, rev=LATEST):
    return DBResponse(status=DBStatus.NETWORK_ERROR)

  def put(self, doc_id, rev, doc):
    return DBResponse(status=DBStatus.NETWORK_ERROR)

class DBExecuteNetworkErrorTests(TestCase):

  def test_network_error(self):
    doc_id = uuid4()

    db = InMemoryDB()
    update_performer = update_performer_generator(db)
    get_performer = get_performer_generator(db)

    bad_db = NetworkErrorDB()
    bad_update_performer = update_performer_generator(bad_db)
    bad_get_performer = get_performer_generator(bad_db)

    db.put(doc_id, 0, {"test": "doc", "a": 1})
    doc_1 = {"test": "doc", "a": 1}
    doc_1_u = {"test": "doc", "a": 2}
    seq = [
      (GetIntent(doc_id), lambda i: bad_get_performer(None, i)),

      (GetIntent(doc_id), lambda i: get_performer(None, i)),

      (UpdateIntent(doc_id, 0, doc_1_u),
       lambda i: bad_update_performer(None, i)),

      (UpdateIntent(doc_id, 0, doc_1_u),
```

```
        lambda i: update_performer(None, i)),
    ]
    perform_sequence(seq, execute_function(
        doc_id, lambda x: dict(x, a=x.get("a", 0) + 1)
      )
    )
```

Test Failure:

```
>>> run_test(DBExecuteNetworkErrorTests)
ERROR(test_network_error)
Traceback (most recent call last):
  File "<interactive-shell>", line 36, in test_network_error
  File "effect/testing.py", line 115, in perform_sequence
    return sync_perform(dispatcher, eff)
  File "effect/_sync.py", line 34, in sync_perform
    six.reraise(*errors[0])
  File "effect/_base.py", line 78, in guard
    return (False, f(*args, **kwargs))
  File "effect/do.py", line 120, in <lambda>
    return val.on(success=lambda r: _do(r, generator, False),
  File "effect/do.py", line 100, in _do
    val = generator.send(result)
  File "<interactive-shell>", line 6, in execute_function
  File "<interactive-shell>", line 36, in <lambda>
AttributeError: 'NoneType' object has no attribute 'get'
...
```

The `NETWORK_ERROR` on the get is causing issues...

```
@do
def execute_function(doc_id, pure_function):
  done = False
  while not done:
    original_doc = None
    while original_doc is None:
      original_doc = yield Effect(GetIntent(doc_id=doc_id))
      if original_doc.status == DBStatus.NETWORK_ERROR:
        original_doc = None
    new_doc = pure_function(original_doc.doc)
    update_result = yield Effect(
      UpdateIntent(doc_id, original_doc.rev, new_doc))
    done = (update_result.status == DBStatus.OK)
```

Run the test again:

```
>>> run_test(DBExecuteNetworkErrorTests)
FAILURE(test_network_error)
Traceback (most recent call last):
  File "<interactive-shell>", line 36, in test_network_error
  File "effect/testing.py", line 115, in perform_sequence
    return sync_perform(dispatcher, eff)
  File "effect/_sync.py", line 34, in sync_perform
    six.reraise(*errors[0])
  File "effect/_base.py", line 78, in guard
    return (False, f(*args, **kwargs))
  File "effect/do.py", line 121, in <lambda>
    error=lambda e: _do(e, generator, True))
  File "effect/do.py", line 98, in _do
```

```
    val = generator.throw(*result)
  File "<interactive-shell>", line 7, in execute_function
  File "effect/_base.py", line 150, in _perform
    performer = dispatcher(effect.intent)
  File "effect/testing.py", line 108, in dispatcher
    intent, fmt_log()))
AssertionError: Performer not found: GetIntent<LATEST, 9515f7cf-8e34-c0f0-49ab-ddee515684b5>! Log fo
{{{
sequence: GetIntent<LATEST, 9515f7cf-8e34-c0f0-49ab-ddee515684b5>
sequence: GetIntent<LATEST, 9515f7cf-8e34-c0f0-49ab-ddee515684b5>
sequence: UpdateIntent<1, 9515f7cf-8e34-c0f0-49ab-ddee515684b5, {'a': 2, 'test': 'doc'}>
NOT FOUND: GetIntent<LATEST, 9515f7cf-8e34-c0f0-49ab-ddee515684b5>
NEXT EXPECTED: UpdateIntent<0, 9515f7cf-8e34-c0f0-49ab-ddee515684b5, {'a': 2, 'test': 'doc'}>
}}}
...
```

The `NETWORK_ERROR` on the update is causing issues...

```
@do
def execute_function(doc_id, pure_function):
  done = False
  while not done:
    original_doc = None
    get_intent = GetIntent(doc_id=doc_id)
    while original_doc is None:
      original_doc = yield Effect(get_intent)
      if original_doc.status == DBStatus.NETWORK_ERROR:
        original_doc = None
    new_doc = pure_function(original_doc.doc)
    update_result = None
    update_intent = UpdateIntent(doc_id, original_doc.rev, new_doc)
    while update_result is None:
      update_result = yield Effect(update_intent)
      if update_result.status == DBStatus.NETWORK_ERROR:
        update_result = None
    done = (update_result.status == DBStatus.OK)
```

```
>>> run_test(DBExecuteNetworkErrorTests)
FAILURE(test_network_error)
Traceback (most recent call last):
  File "<interactive-shell>", line 36, in test_network_error
  File "effect/testing.py", line 115, in perform_sequence
    return sync_perform(dispatcher, eff)
  File "effect/_sync.py", line 34, in sync_perform
    six.reraise(*errors[0])
  File "effect/_base.py", line 78, in guard
    return (False, f(*args, **kwargs))
  File "effect/do.py", line 121, in <lambda>
    error=lambda e: _do(e, generator, True))
  File "effect/do.py", line 98, in _do
    val = generator.throw(*result)
  File "<interactive-shell>", line 15, in execute_function
  File "effect/_base.py", line 150, in _perform
    performer = dispatcher(effect.intent)
  File "effect/testing.py", line 108, in dispatcher
    intent, fmt_log()))
AssertionError: Performer not found: UpdateIntent<1, c2d99fe7-48e7-9846-a601-ce405b5baedf, {'a': 2,
{{{
sequence: GetIntent<LATEST, c2d99fe7-48e7-9846-a601-ce405b5baedf>
```

```
sequence: GetIntent<LATEST, c2d99fe7-48e7-9846-a601-ce405b5baedf>
sequence: UpdateIntent<1, c2d99fe7-48e7-9846-a601-ce405b5baedf, {'a': 2, 'test': 'doc'}>
NOT FOUND: UpdateIntent<1, c2d99fe7-48e7-9846-a601-ce405b5baedf, {'a': 2, 'test': 'doc'}>
NEXT EXPECTED: UpdateIntent<0, c2d99fe7-48e7-9846-a601-ce405b5baedf, {'a': 2, 'test': 'doc'}>
}}}
...
```

For those of you who are familiar with `Effect`, you probably noticed pretty early in this post what the error is about. My implementation of the `update_performer` modifies the intent that is passed in when it is called. Specifically it increments the revision of the intent in place before passing it to the underlying call to `db.put`. With this implementation of how we handle NETWORK_ERRORS we are re-using the same intent with the next performance of update. The second run of `update` is unaware that the first one already incremented `rev`, so it is incremented a second time. This is the source of our bug.

Effect recommends against mutating intents, but there is not any mechanism that enforces it. Luckily, depending on your code it might be sort of rare to re-use intents. If you do happen to re-use intents though, and you have not been diligent about never mutating them, you might be vulnerable to some pretty pesky bugs to track down.

The quick fix is simply not to modify intent in the function:

```
def update_performer_generator(db):
  def update(dispatcher, intent):
    return db.put(intent.doc_id, intent.rev + 1, intent.doc)
  return update
```

```
>>> run_test(DBExecuteNetworkErrorTests)
[OK]
```

This for now pretty much wraps up my implementation using pure `Effect`, but there is one last observation I'd like to make:

2.1 TypeDispatchers are just classes

Look at db_dispatcher:

```
def db_dispatcher(db):
  return TypeDispatcher({
    GetIntent: sync_performer(get_performer_generator(db)),
    UpdateIntent: sync_performer(update_performer_generator(db)),
  })
```

This is a chunk of python that describes what functions to execute when a certain identifier (type of intent) occurs. At some later point during the program some values will be passed to one of the code chucks associated with one of the identifiers.

It is sort of a funny way of describing it, but to me this describes a class definition. The intents are bundles of arguments, the type of the intents are the names of the methods, and the `TypeDispatcher` instance represents an object that is an instance of that type.

Think about attempting to create a `TypeDispatcher` that can perform the same effects as the objects returned by `db_dispatcher`, but rather than performing db interactions just writes an object to a file or reads an object from a file:

```
_FILEPATH = '/tmp/datastore'

def _get_stored_obj():
  return json.load(open(_FILEPATH, "r"))
```

```
def _store_obj(obj):
  return json.dump(obj, open(_FILEPATH, "w"))

def file_update_performer(intent):
  file_store = _get_stored_obj()
  obj_revs = file_store.get(intent.doc_id, [])
  if len(obj_revs) != intent.rev:
    return DBResponse(status=DBStatus.CONFLICT)
  file_store[doc_id] = obj_revs
  obj_revs.push(intent.doc)
  _store_obj(file_store)

def file_get_performer(dispatcher, intent):
  file_store = _get_stored_obj()
  if intent.rev < LATEST:
    return DBResponse(status=DBStatus.BAD_REQUEST)
  try:
    return DBResponse(
      status=DBStatus.OK,
      rev=intent.rev,
      doc=file_store[intent.doc_id][intent.rev]
    )
  except KeyError:
    return DBResponse(
      status=DBStatus.NOT_FOUND
    )
  except IndexError:
    return DBResponse(
      status=DBStatus.NOT_FOUND
    )

def file_dispatcher():
  return TypeDispatcher({
    GetIntent: sync_performer(file_get_performer),
    UpdateIntent: sync_performer(file_update_performer),
  })
```

This feels a lot like implementing another class that implements the same interface. It is just writing performers for a specific intent types (`GetIntent` and `UpdateIntent`) rather than writing methods with specific names.

If you put a bunch of dispatchers together using a `ComposedDispatcher` it is similar to subclassing, in that you are adding more performers to the same namespace, just like adding more methods to the same class. There even is the ability to overload since ComposedDispatchers prefer earlier dispatchers over later dispatchers.

CHAPTER 3

Coding with ziffect

The `ziffect` takes the idea of `TypeDispatchers` as a core part of the design. Similar to zope interfaces, you start coding with ziffect by specifying an interface that you will implement. It also builds upon `pyrsistent` `PClass` s, and thus adds type-checking at intent creation time.

```
from uuid import UUID
from six import text_type
import ziffect

@ziffect.interface
class DBInterface(object):

  def get(doc_id=ziffect.argument(type=UUID),
          rev=ziffect.argument(type=int, default=LATEST)):
    pass

  def update(doc_id=ziffect.argument(type=UUID),
             rev=ziffect.argument(type=int),
             doc=ziffect.argument(type=dict)):
    pass
```

This specifies the interface to the DB that we intend to implement. So when we write performers, we just write a class that implements the interface:

```
@ziffect.implements(DBInterface)
class ZiffectDB(object):
  def __init__(self, db):
    """
    :param db: The underlying db to make calls to.
    """
    self.db = db

  def get(self, doc_id, rev):
    return self.db.get(doc_id, rev)

  def update(self, doc_id, rev, doc):
    rev += 1
    return self.db.put(doc_id, rev, doc)
```

Note that this bit of code is supposed to encompass both the `TypeDispatcher` as well as the performers from earlier.

Then when we go to actually implement our function, we need to be able to create effects representing the methods on our interface. To do that we use `ziffect.effects`. When you pass ziffect.effects a `ziffect` interface it returns

an object that has all the same methods as the interface and generates effects representing the intent of having those methods called on some other implementation:

```
from effect.do import do

@do
def execute_function(doc_id, pure_function):
  db_effects = ziffect.effects(DBInterface)
  result = yield db_effects.get(doc_id=doc_id)
  new_doc = pure_function(result.doc)
  yield db_effects.update(doc_id=doc_id,
                          rev=result.rev,
                          doc=new_doc)
```

Again we need a nice little wrapper if we are going to attempt to use this tool interactively. Note that `ziffect` also can create dispatchers for you. The `ziffect` dispatcher is created using `ziffect.dispatcher`. It takes a dict that maps `ziffect` interfaces to objects that provide that interface. This is effectively choosing the implementation of the interface that will be used to perform effects created from `ziffect.effects` -style effect generators.

```
from effect import (
  sync_perform, ComposedDispatcher, base_dispatcher
)

def sync_execute_function(db, doc_id, function):
  dispatcher = ComposedDispatcher([
    ziffect.dispatcher({
      DBInterface: ZiffectDB(db)
    }),
    base_dispatcher
  ])
  sync_perform(
    dispatcher,
    execute_function(
      doc_id, function
    )
  )
```

Running the same interactive test that we ran on our effect implementation:

```
>>> db = DB()
>>> doc_id = uuid4()
>>> doc = {"cat": "mouse", "count": 10}
>>> db.put(doc_id, 0, doc)
DB Response<OK rev=0>

>>> def increment(doc_id):
...     return sync_execute_function(
...         db,
...         doc_id,
...         lambda x: dict(x, count=x.get('count', 0) + 1)
...     )

>>> increment(doc_id)
>>> db.get(doc_id)
DB Response<OK rev=1 {"cat": "mouse", "count": 11}>

>>> increment(doc_id)
>>> db.get(doc_id)
DB Response<OK rev=2 {"cat": "mouse", "count": 12}>
```

```
>>> increment(doc_id)
>>> db.get(doc_id)
DB Response<OK rev=3 {"cat": "mouse", "count": 13}>
```

Again the happy case works right out of the box. Once again we'll continue with test-driven development. For starters, I'll demonstrate directly how we can use the same tools we used when testing `effect` to test with `ziffect` .

```
from effect.testing import perform_sequence

class DBExecuteFunctionTests(TestCase):

  def test_happy_case(self):
    db_intents = ziffect.intents(DBInterface)
    doc_id = uuid4()
    doc_1 = {"test": "doc", "a": 1}
    doc_1_u = {"test": "doc", "a": 2}
    seq = [
      (db_intents.get(doc_id=doc_id),
       lambda _: DBResponse(status=DBStatus.OK, rev=0, doc=doc_1)),

      (db_intents.update(doc_id=doc_id,
                         rev=0,
                         doc=doc_1_u),
       lambda _: DBResponse(status=DBStatus.OK)),
    ]
    perform_sequence(seq, execute_function(
        doc_id, lambda x: dict(x, a=x.get("a", 0) + 1)
      )
    )

  def test_sad_case(self):
    db_intents = ziffect.intents(DBInterface)
    doc_id = uuid4()
    doc_1 = {"test": "doc", "a": 1}
    doc_1_u = {"test": "doc", "a": 2}
    doc_2 = {"test": "doc2", "a": 5}
    doc_2_u = {"test": "doc2", "a": 6}
    seq = [
      (db_intents.get(doc_id=doc_id),
        lambda _: DBResponse(status=DBStatus.OK, rev=0, doc=doc_1)),

      (db_intents.update(doc_id=doc_id, rev=0, doc=doc_1_u),
        lambda _: DBResponse(status=DBStatus.CONFLICT)),

      (db_intents.get(doc_id=doc_id),
        lambda _: DBResponse(status=DBStatus.OK, rev=1, doc=doc_2)),

      (db_intents.update(doc_id=doc_id, rev=1, doc=doc_2_u),
        lambda _: DBResponse(status=DBStatus.OK)),
    ]
    perform_sequence(seq, execute_function(
        doc_id, lambda x: dict(x, a=x.get("a", 0) + 1)
      )
    )
```

Now to run the test and fix as needed:

```
>>> run_test(DBExecuteFunctionTests)
FAILURE(test_sad_case)
Traceback (most recent call last):
  File "<interactive-shell>", line 45, in test_sad_case
  File "effect/testing.py", line 115, in perform_sequence
    return sync_perform(dispatcher, eff)
  File "effect/testing.py", line 463, in consume
    [x[0] for x in self.sequence]))
AssertionError: Not all intents were performed: [_Intent(doc_id=UUID('3a80d1fb-b1b0-35b7-bd12-39ccdbb
...
```

We have the expected error of not doing a get in the case of receiving a conflict notification.

Aside

Obviously the fact that all of those intents are named `_Intent` is less than desireable. `ziffect` is a work in progress, and long term I hope to make all of the meta attributes (`__name__` and the like) on the auto-generated intents much more usable.

Fixing the error by doing a full implementation:

```
@do
def execute_function(doc_id, pure_function):
  db_effects = ziffect.effects(DBInterface)
  done = False
  while not done:
    original = yield db_effects.get(doc_id=doc_id)
    new_doc = pure_function(original.doc)
    result = yield db_effects.update(doc_id=doc_id,
                                     rev=original.rev,
                                     doc=new_doc)
    done = (result.status == DBStatus.OK)
```

```
>>> run_test(DBExecuteFunctionTests)
[OK]
```

Okay, so already we have had a marginally easier time working with `ziffect`. We did not have to write quite as much boiler plate code defining intents and creating dispatchers, and the intents that `ziffect` created for us had reasonable `__repr__` and `__eq__` implementations so we did not have to deal with that ourselves.

For completeness, we'll continue on with the addition of the `NETWORK_ERROR` retries as we have done previously.

```
#@ziffect.implements(DBInterface)
class NetworkErrorDB(object):
  def get(self, doc_id, rev=LATEST):
    return DBResponse(status=DBStatus.NETWORK_ERROR)

  def put(self, doc_id, rev, doc):
    return DBResponse(status=DBStatus.NETWORK_ERROR)


class DBExecuteNetworkErrorTests(TestCase):

  def test_network_error(self):
    doc_id = uuid4()
    db_intents = ziffect.intents(DBInterface)

    db = InMemoryDB()
```

```
    bad_db = NetworkErrorDB()

    good_impl = ZiffectDB(db)
    bad_impl = ZiffectDB(bad_db)

    db.put(doc_id, 0, {"test": "doc", "a": 1})
    doc_1 = {"test": "doc", "a": 1}
    doc_1_u = {"test": "doc", "a": 2}
    seq = [
      (db_intents.get(doc_id=doc_id), bad_impl.get),

      (db_intents.get(doc_id=doc_id), good_impl.get),

      (db_intents.update(doc_id=doc_id, rev=0, doc=doc_1_u),
       bad_impl.update),

      (db_intents.update(doc_id=doc_id, rev=0, doc=doc_1_u),
       good_impl.update),
    ]
    ziffect.perform_sequence_destructed_args(
      seq, execute_function(
        doc_id, lambda x: dict(x, a=x.get("a", 0) + 1)
      )
    )
```

Note

```
>>> run_test(DBExecuteNetworkErrorTests)
ERROR(test_network_error)
Traceback (most recent call last):
  File "<interactive-shell>", line 38, in test_network_error
  File "<interactive-shell>", line 294, in perform_sequence_destructed_args
    effect_generator)
  File "effect/testing.py", line 115, in perform_sequence
    return sync_perform(dispatcher, eff)
  File "effect/_sync.py", line 34, in sync_perform
    six.reraise(*errors[0])
  File "effect/_base.py", line 78, in guard
    return (False, f(*args, **kwargs))
  File "effect/do.py", line 120, in <lambda>
    return val.on(success=lambda r: _do(r, generator, False),
  File "effect/do.py", line 100, in _do
    val = generator.send(result)
  File "<interactive-shell>", line 7, in execute_function
  File "<interactive-shell>", line 38, in <lambda>
AttributeError: 'NoneType' object has no attribute 'get'
...
```

So we have to actually add the retries on `NETWORK_ERROR` s:

```
@do
def execute_function(doc_id, pure_function):
  db_effects = ziffect.effects(DBInterface)
  done = False
  while not done:
    original = None
    while original is None:
      original = yield db_effects.get(doc_id=doc_id)
      if original.status == DBStatus.NETWORK_ERROR:
```

```
        original = None
    new_doc = pure_function(original.doc)
    result = None
    while result is None:
      result = yield db_effects.update(doc_id=doc_id,
                                       rev=original.rev,
                                       doc=new_doc)
      if result.status == DBStatus.NETWORK_ERROR:
        result = None
    done = (result.status == DBStatus.OK)
```

And we've completed our implementation:

```
>>> run_test(DBExecuteNetworkErrorTests)
[OK]
```

3.1 Summary

Hopefully, that example was sufficient to demonstrate the benifits of using `ziffect` instead of `effect` directly, although there certainly is some room for criticism:

1. ***Most of the benefits of `ziffect` come fro using `pyrsistent` to make** intents. If you just have a codebase-wide policy of using `pyrsistent` to make intents, you would not have to add the dependency on `ziffect`.* This

   is probably true, and it certainly is the case the `ziffect` has made some decisions in favor of ease-of-use over flexability. Nonetheless, I think `ziffect` also comes with a model of code that is cleaner and easier to maintain long term. Specifically, sandboxing performers behind interfaces makes it easier to identify which performers concern a specific system of side effects, and provide a clear interface to fake out if you want a fake implementation for testing.

2. *''ziffect'' peformers do not get a ''dispatcher'' argument, how am I supposed to write performers that dispatch other ''Events''.* This is certainly true, `ziffect` does not allow for as flexible performers because it does not pass the dispatcher in. I'm still trying to figure out how to think about the dispatcher argument, and processing ideas of what the API should look like.

   Sometimes `dispatcher` feels like dependency injection to me. For instance, if you are writing a performer and you want to ensure that something is logged before and after you do some operation, you might use the dispatcher that is handed in to dispatch some `Log` events. You just want to ensure the `Log` intent is handled, but the implementation is determined at runtime by what dispatcher you have.

   Other times, `dispatcher` is just providing an interface for performers that are schedulers. For instance, you could have an `in_parallel` intent, which would simply use the dispatcher to dispatch all of the events at once, and then aggregate the events to a single event before concluding the event they are performing. This feels subtly different than the other use of `dispatcher` to me.

   As I figure out how to reconcile these two uses of `dispatcher` and determine if they are fundamentally different or effectively the same, I'll be extending the `ziffect` API to support these performers.

3.2 Future Work

- Lots of error handling tests. I'd like to add tests for common coding mistakes, and ensure the errors raised are actionable for the programmer.

- Actual integration with `zope.interface`, presently the test matcher is a lie, and actually integrating with `zope.interface` would allow for the creation of proxy implementations.
- Utilities, like a function that takes a `ziffect` interface and a provider of that interface, and returns an implementation of that interface that logs before and after that function finishes.
- `txziffect` or equivalent.

CHAPTER 4

API

4.1 ziffect

The ziffect module.

ziffect.**interface**(*wrapped_class*)
Class decorator to wrap ziffect interfaces.

Parameters **wrapped_class** – The class to wrap.

Returns The newly created wrapped class.

ziffect.**effects**(*interface*)
Method to get an object that implements interface by just returning effects for each method call.

Parameters **interface** – The interface for which to create a provider.

Returns A class with method names equal to the method names of the interface. Each method on this class will generate an Effect for use with the Effect library.

class ziffect.**argument**
Argument type

TODO(mewert): fill the rest of this in.

4.2 ziffect.matchers

The ziffect.matchers module, filled with convenient testtools matchers for use with ziffect.

ziffect.matchers.**Provides**(*interface*)
Matches if interface is provided by the matchee.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Z

ziffect, 25
ziffect.matchers, 25

A

argument (class in ziffect), 25

E

effects() (in module ziffect), 25

I

interface() (in module ziffect), 25

P

Provides() (in module ziffect.matchers), 25

Z

ziffect (module), 25
ziffect.matchers (module), 25