Ziffect Documentation

Release 0.0.1

Marcus Henry Ewert

January 29, 2016

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.

1.2 Using Effect and Limitations

Let's walk through an example to illustrate my grievances with the effect library. For starters, lets 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 Effect, sync_performer, TypeDispatcher

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


def get_performer_generator(db):
  @sync_performer
  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):
  @sync_performer
  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: get_performer_generator(db),
    UpdateIntent: 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 sync_perform, ComposedDispatcher, base_dispatcher
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))


def sync_execute_function(db, doc_id, function):
  """
  Convenience wrapper to perform :func:`execute_function` on a database from
  an interactive terminal.
  """
  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.

API

2.1 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.

2.2 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 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index