Emit Documentation

Release 0.3.0

Brian Hicks

January 28, 2013

CONTENTS

Emit is a library that hooks into distributed systems like Celery or RQ (or just local memory) to provide subscriptions and notifications for your functions. It is designed to make processing streams of information a whole lot easier.

You may want to start at *Getting Started*. Other highlights include integration with *Celery* and *RQ* and Emit's *multi-language* capabilities.

Contents:

CHAPTER

ONE

GETTING STARTED

1.1 Installing

You can install emit with `pip`:

```
pip install emit
```

1.2 Quickstart

For a sampler, we're going to make a simple command-line application that will take and count all the words in a document, giving you the top 5.

Put the following into *graph.py*

```
from __future__ import print_function
from collections import Counter
from emit import Router
import sys

router = Router()


def prefix(name):
    return '%s.%s' % (__name__, name)


@router.node(('word',), entry_point=True)
def words(msg):
    print('got document')
    for word in msg.document.strip().split(' '):
        yield word


WORDS = Counter()


@router.node(('word', 'count'), prefix('words'))
def count_word(msg):
    print('got word (%s)' % msg.word)

    global WORDS
```

```
28      WORDS.update([msg.word])
29
30      return msg.word, WORDS[msg.word]
31
32  if __name__ == '__main__':
33      router(document=sys.stdin.read())
34
35      print()
36      print('Top 5 words:')
37      for word, count in WORDS.most_common(5):
38          print('    %s: %s' % (word, count))
```

(incidentally, this file is available in the project directory as `examples/simple/graph.py`.)

Now on the command line: `echo "the rain in spain falls mainly on the plain" | python graph.py`. You should get some output that looks similar to the following

```
got document
got word (the)
got word (rain)
got word (in)
got word (spain)
got word (falls)
got word (mainly)
got word (on)
got word (the)
got word (plain)

Top 5 words:
    the: 2
    on: 1
    plain: 1
    mainly: 1
    rain: 1
```

1.2.1 Breaking it Down

First, we need to construct a router:

```
router = Router()
```

Since we're keeping everything in-memory, we don't need to specify anything to get this to work properly. It should "Just Work(TM)".

Next, we define a function to split apart a document on spaces to get words:

```
@router.node(('word',), entry_point=True)
def words(msg):
    print('got document')
    for word in msg.document.strip().split(' '):
        yield word
```

`Router` provides a decorator (`node`). The first argument is the fields that the decorated function returns. These are wrapped in a message and passed around between functions.

We don't specify any subscriptions on this function, since it really doesn't need any. In fact, it's an entry point, so we specify that instead. This specifically means that if you call the router directly it will delegate to this function. There can be multiple functions with `entry_point` set to true on a given `Router`.

If the decorated function is a generator, each yielded value is treated as a separate input into the next nodes in the graph.

Splitting the document into parts is only as useful as what we can do with the words, so let's count them now:

```
WORDS = Counter()
@router.node(('word', 'count'), prefix('words'))
def count_word(msg):
    print('got word (%s)' % msg.word)

    global WORDS
    WORDS.update([msg.word])

    return msg.word, WORDS[msg.word]
```

There's a little less going on in this function. We just update a `Counter` builtin, and then return the word and the count to be passed down the graph. In real life, you'd probably persist this value in a database to allow multiple workers to process different parts of the stream.

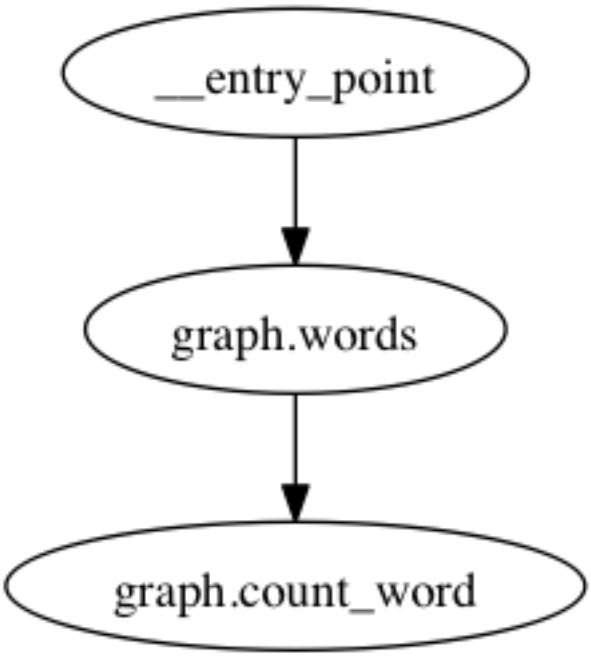
In non-entry nodes, the second argument of `router.node` is a string or list of functions to subscribe to. These need to be fully qualified when you're using Celery, but for now they're fine.

Now that we've defined both functions, it's time to send some data into our graph:

```
router(document=sys.stdin.read())
```

Calling this graph is easy, since we defined a function as an entry point. You can call any of the functions (or the router itself) by using keyword arguments or passing a dictionary.

In the end, data flows through the graph like this:



1.3 Next Steps

Now that you've got this under your belt, check out how to integrate your graph with *RQ* or *Celery*.

CHAPTER

TWO

DISTRIBUTING WORK

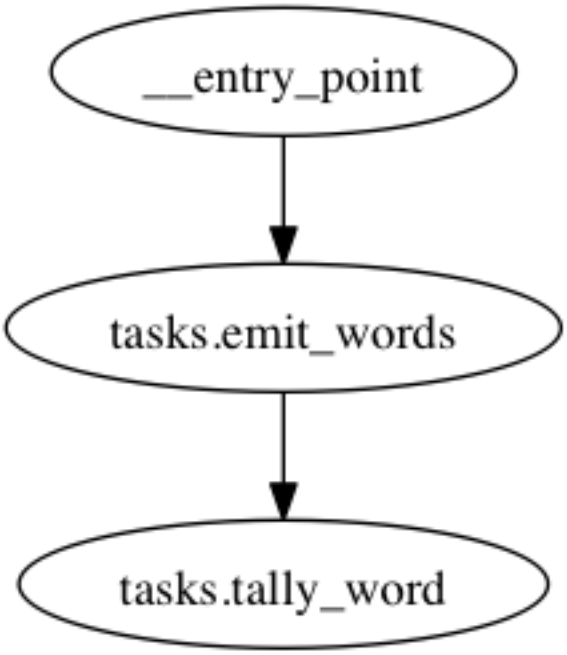
Emit follows the philosophy that routing execution of tasks over the network is best handled by an external library. Currently, there are two integrations: *RQ* and *Celery*.

In addition, you may want to *write your own* for an as-of-yet unknown backend.

2.1 Contents

2.1.1 Using RQ to Distribute Processing

RQ is a module that makes distributed processing easy. It's similar to Celery, but simpler and only for Python and Redis. We'll be using the same example as we did in the Celery example.



Setting up RQ

Create an `app.py` file for your RQ Router initializaition code to live in:

```
'simple rq app'
from redis import Redis
from emit import RQRouter

import logging

router = RQRouter(redis_connection=Redis(), node_modules=['tasks'])

logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG)
```

The `RQRouter` class only needs to know what Redis connection you want to use. The rest of the options are specified at the node level.

Next we'll define (in `tasks.py`) a function to take a document and emit each word:

```
@router.node(('word',), entry_point=True)
def emit_words(msg):
    for word in msg.document.strip().split(' '):
        yield word
```

Without any arguments, RQ tasks will go to the 'default' queue. If you don't want to mess with queues, this will *just work*.

If you want to set some attributes, however, you can:

```
@router.node(('word', 'count'), subscribe_to='tasks.emit_words', queue='words')
def tally_word(msg):
    redis = Redis()
    return msg.word, redis.zincrby('celery_emit_example', msg.word, 1)
```

Enqueued functions for this node will be put on the "words" node. You'll need to specify which nodes to listen to when running `rqworker`.

The available parameters:

parameter	default	effect
`queue`	`'default'`	specify a queue to route to.
`connection`	supplied connection	a different connection - be careful with this, as you'll need to specify the connection string on the worker
`timeout`	`None`	timeout (in seconds) of a task
`result_ttl`	`500`	TTL (in seconds) of results

Running the Graph

We just need to start the RQ worker:

```
rqworker default words
```

And enter the following on the command line to start something fun processing (if you'd like, the relevant code is in `examples/rq/kickoff.py` in the project directory, start it and get a prompt with `ipython -i kickoff.py`):

```
from app import router
import random
words = 'the rain in spain falls mainly on the plain'.split(' ')
router(document=' '.join(random.choice(words) for i in range(50)))
```

And you should see the rqworker window quickly scrolling by with updated totals. Run the command a couple more times, if you like, and you'll see the totals keep going up.

Performance

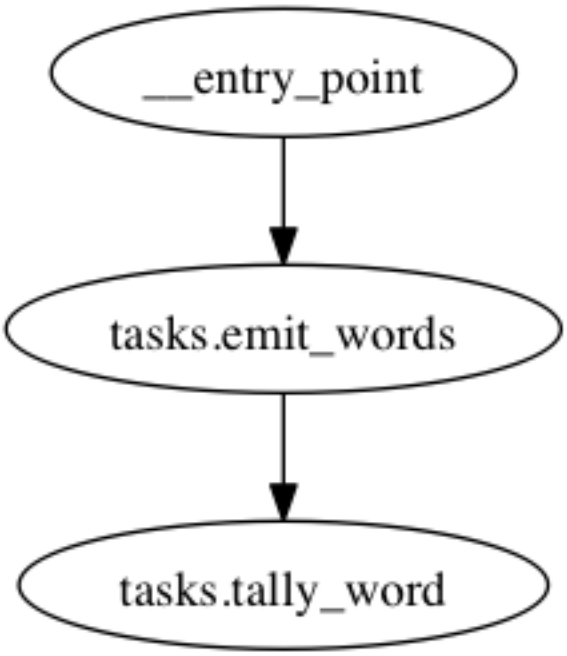
Because of the way RQ forks tasks, the graph is rebuilt for every task. To speed up this process, do it once on worker initialization. You can use this snippet (adapted from the RQ worker documentation)

```
#!/usr/bin/env python
import sys
import rq

# Preload libraries
from app import router
router.resolve_node_modules()

# Provide queue names to listen to as arguments to this script,
# similar to rqworker
with rq.Connection():
    qs = map(rq.Queue, sys.argv[1:]) or [rq.Queue()]

    w = rq.Worker(qs)
    w.work()
```

2.1.2 Using Celery to Distribute Processing

Emit makes it simple to use celery to distribute realtime processing across many worker nodes. To demonstrate this, we'll be scaling our quickstart example

We'll be making, in essence, this graph:



Setting up Celery

Create an `app.py` file for your celery initializaition code to live in:

```
1  'simple celery app'
2  from celery import Celery
3  from emit import CeleryRouter
4
5  import logging
6
7  app = Celery(
8      'celery_emit_example',
9      broker='redis://'
10 )
11 app.conf.update(
12     CELERY_IMPORTS=('tasks',)
13 )
14
15 router = CeleryRouter(celery_task=app.task, node_modules=['tasks'])
16
17 logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG)
```

Take note that `Router` is initialized using the default celery task in this case. This is probably the best way to do it, since per-task settings should belong in the task (possible in Emit's decorator), and app-level configuration should be on the `app` object (as on line 10).

Next we'll define (in `tasks.py`) a function to take a document and emit each word:

```
@router.node(('word',), entry_point=True)
def emit_words(msg):
    for word in msg.document.strip().split(' '):
        yield word
```

We don't have to give any special syntax to get these tasks to work with celery: since we specified it in the router, they just do.

However, if you want to give special celery attributes to a particular function, you can do that too:

```
@router.node(('word', 'count'), subscribe_to='tasks.emit_words', celery_task=app.task(rate_limit='5/s
def tally_word(msg):
    redis = Redis()
    return msg.word, redis.zincrby('celery_emit_example', msg.word, 1)
```

Obviously rate limiting to 5 per second in this case is a bit contrived, but you get the general idea: it's easy to configure tasks within the decorator by passing in the celery decorator.

The available parameters:

parameter	default	effect
`celery_task`	None	override the supplied celery task with a node-specific tas

Running the Graph

We'll need to boot up the celery daemon:

```
celery worker -A app.app -l INFO -E
```

And enter the following on the command line to start something fun processing (if you'd like, the relevant code is in `examples/celery/kickoff.py` in the project directory, start it and get a prompt with `ipython -i kickoff.py`):

```
from app import router
import random
words = 'the rain in spain falls mainly on the plain'.split(' ')
router(document=' '.join(random.choice(words) for i in range(50)))
```

You should get something like the following:

```
({'word': 'the'},
 {'word': 'spain'},
 {'word': 'in'},
 # ...
 {'word': 'falls'},
 {'word': 'falls'},
 {'word': 'mainly'})
```

And you should see the celery window quickly scrolling by with updated totals. Run the command a couple more times, if you like, and you'll see the totals keep going up.

2.1.3 Extending Router

To extend `emit.Router` (for example, to add a new dispatch backend) it's most helpful to override the following methods:

`__init__(self, your_args, *args, **kwargs)` This is the `__init__` pattern used by the current dispatch backends.

`dispatch(origin, destination, message)` Do dispatching. Typically passes along `origin` (as `_origin`) with the message.

`wrap_node(node, options)` Given a wrapped function (`node`), do additional processing on the function or node. Unhandled arguments to `Router.node` are passed as a dictionary as `options`.

Example

See the following example (the current `RQRouter` implementation):

```
class RQRouter(Router):
    'Router specifically for RQ routing'
    def __init__(self, redis_connection, *args, **kwargs):
        '''\
        Specific routing when using RQ

        :param redis_connection: a redis connection to send to all the tasks
                                 (can be overridden in :py:meth:`Router.node`.)
        :type redis_connection: :py:class:`redis.Redis`
        '''
        super(RQRouter, self).__init__(*args, **kwargs)
        self.redis_connection = redis_connection
        self.logger.debug('Initialized RQ Router')

    def dispatch(self, origin, destination, message):
        'dispatch through RQ'
        func = self.functions[destination]
        self.logger.debug('enqueueing %r', func)
        return func.delay(_origin=origin, **message)

    def wrap_node(self, node, options):
```

```
        '''
        we have the option to construct nodes here, so we can use different
        queues for nodes without having to have different queue objects.
        '''
        job_kwargs = {
            'queue': options.get('queue', 'default'),
            'connection': options.get('connection', self.redis_connection),
            'timeout': options.get('timeout', None),
            'result_ttl': options.get('result_ttl', 500),
        }

        return job(**job_kwargs)(node)
```

CHAPTER

THREE

USING EMIT IN OTHER LANGUAGES

You can use Emit in other languages through the multilang API.

3.1 Defining Tasks

We're going to define a node that takes a number and emits each integer in that range. Let's do it with Ruby! (why not?)

```
require "json"
message = JSON.parse(STDIN.read)

message["count"].times do |i|
  puts i.to_json
end
```

(the equivalent in Python is in `examples/multilang/test.py`)

The messages passed in and out are expected to be in JSON format. Output from the functions should be json strings separated by newlines.

3.2 Creating a Node

We'll be subclassing `emit.multilang.ShellNode` to tell emit how to execute our task:

```
@router.node(('n',))
class RubyShellNode(ShellNode):
    command = 'bundle exec ruby test.rb'
```

After that, you can call your node and subscribe as normal.

CHAPTER

FOUR

REGEX ROUTING

You can subscribe to arbitrary node's output by providing a regular expression. In this example, we'll use Redis' pubsub capabilities to notify an external receiver of all tasks passing through the graph.

The product of this example is in `examples/regex/graph.py`.

First, we'll create a function that yields multiple return values. In this case, we're going to naively parse a HTTP querystring.

```
@router.node(('key', 'value'), entry_point=True)
def parse_querystring(msg):
    'parse a querystring into keys and values'
    for part in msg.querystring.strip().lstrip('?').split('&'):
        key, value = part.split('=')
        yield key, value
```

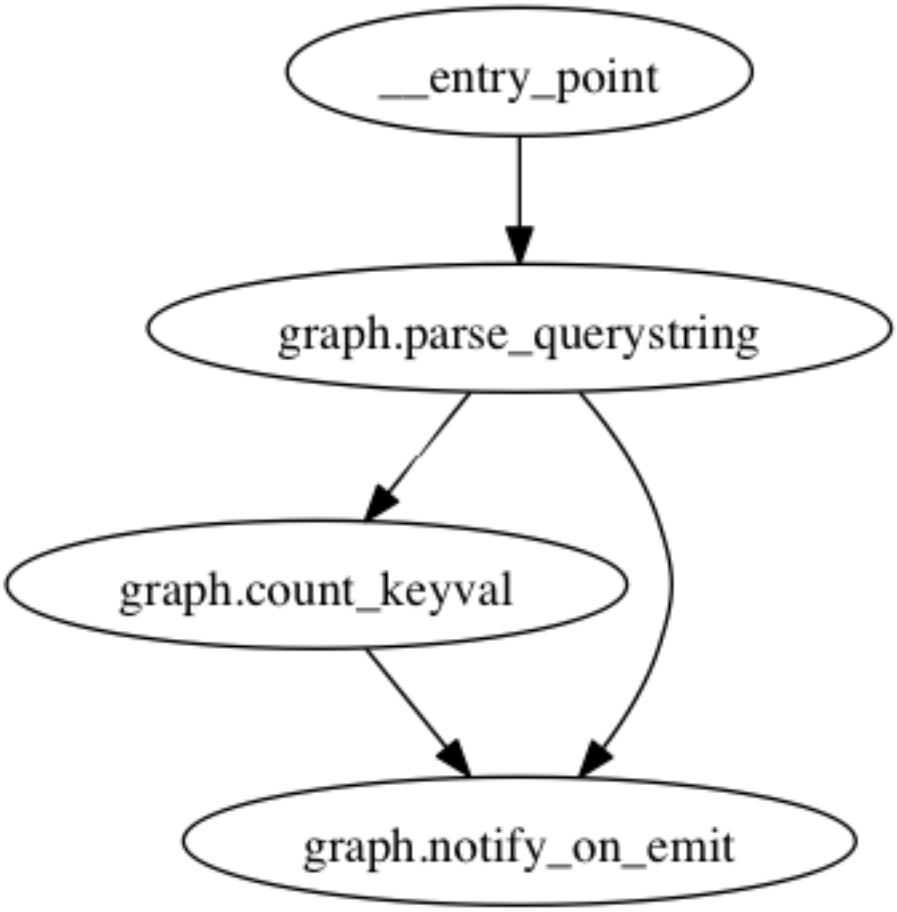
Now we're going to count keys and values:

```
@router.node(('key', 'value', 'count'), prefix('parse_querystring'))
def count_keyval(msg):
    count = redis.zincrby('querystring_count.%s' % msg.key, msg.value, 1)
    return msg.key, msg.value, count
```

Next, we'll make a function that publishes to Redis on every message:

```
@router.node(tuple(), '.+')
def notify_on_emit(msg):
    redis.publish('notify_on_emit.%s' % msg._origin, msg.as_json())
    return NoResult
```

Now, when you call `router(querystring='?a=1&b=2&c=3')`, `notify_on_emit` will publish seven messages: three with origin "graph.parse_querystring", three with origin "graph.count_keyval", and one with origin "__entry_point". The graph ends up looking like this:



You can also specify `ignores` in `Router.node`, which can cut a little fat out of an otherwise greedy regex.

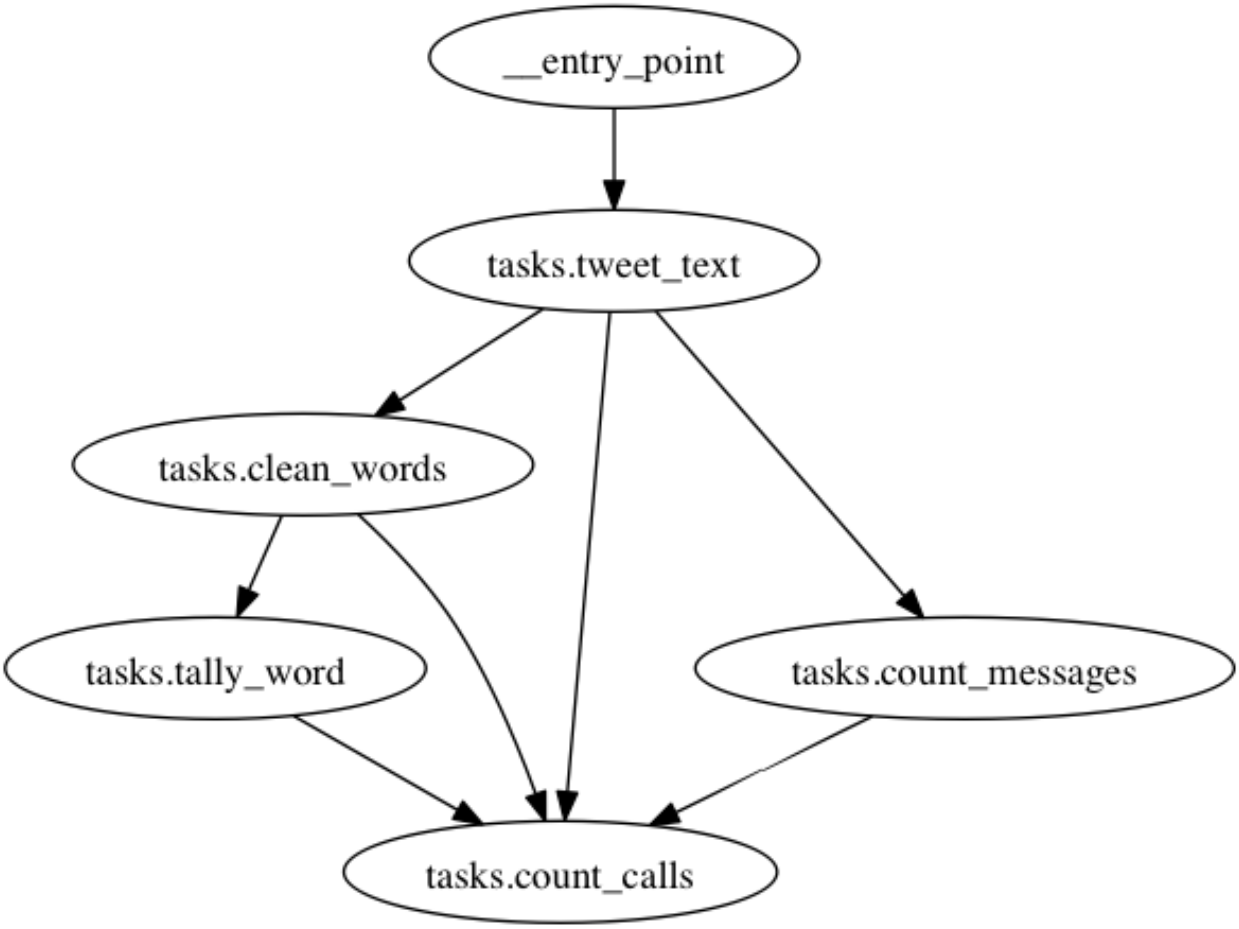
CHAPTER

FIVE

COMMAND LINE UTILITIES

5.1 emit_digraph - Generate Graph Images

Emit ships with a command-line program to inspect a graph: `emit_digraph`. Use it to generate graphs like this:



`emit_digraph` will output the code graphviz needs to properly generate the graph. (You'll need graphviz installed on your machine for this to render properly.) To use it, pass it the path of your router. (for example, `emit_digraph app.router` in the Celery example.) The output should look something like this:

```
digraph router {
"tasks.clean_words" -> "tasks.tally_words";
```

```
"tasks.clean_text" -> "tasks.clean_words";
"tasks.tweet_text" -> "tasks.count_messages";
"__entry_point" -> "tasks.tweet_text";
}
```

to make graphviz generate a PNG of the graph, pipe it into the following command:

```
emit_digraph app.router | dot -T png -o graph.png
```

CHAPTER

SIX

LOGGING

Emit is set up to handle logging using Python's standard logger. It currently uses the following levels:

- `DEBUG`: task registration and calls - very verbose
- `INFO`: route registration, receipts

So far there's been no need for anything above `INFO`, but that may change in the future.

6.1 Setting Up Logging

In some file (I recommend the file where the router is initialized, but your project may vary) insert the following lines:

```
import logging
logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG) # or INFO etc.
```

6.2 Setting Up Logging in Django

In your logging config, add a logger for "emit". Like so:

```
LOGGING = {
    # snip formatters, filters, handlers, etc
    'loggers': {
        # other loggers here
        'emit': {
            'handlers': ['console'],
            'level': 'INFO',
        }
    }
}
```

CHAPTER

SEVEN

TESTING

Testing your functions is pretty easy. Just call `Router.disable_routing`. Something like this:

```
from unittest import TestCase
from yourapp.app import router
from yourapp.tasks import do_task

class DoTaskTests(TestCase):
    def setUp(self):
        router.disable_routing()

    def test_blah(self):
        assert True
```

To re-enable routing, you'd call `Router.enable_routing`.

CHAPTER

EIGHT

API DOCUMENTATION

8.1 Router

class `emit.router.`**`Router`**(*message_class=None*, *node_modules=None*, *node_package=None*)
A router object. Holds routes and references to functions for dispatch

`__init__`(*message_class=None*, *node_modules=None*, *node_package=None*)
Create a new router object. All parameters are optional.

Parameters

- **message_class** (`emit.message.Message` or subclass) – wrapper class for messages passed to nodes
- **node_modules** (a list of `str`, or `None`.) – a list of modules that contain nodes
- **node_package** (`str`, or `None`.) – if any node_modules are relative, the path to base off of.

Exceptions None

Returns None

`__call__`(***kwargs*)
Route a message to all nodes marked as entry points.

Note: This function does not optionally accept a single argument (dictionary) as other points in this API do - it must be expanded to keyword arguments in this case.

`node`(*fields*, *subscribe_to=None*, *entry_point=False*, *ignore=None*, ***wrapper_options*)
Decorate a function to make it a node.

Parameters

- **fields** (ordered iterable of `str`) – fields that this function returns
- **subscribe_to** (`str` or iterable of `str`) – functions in the graph to subscribe to. These indicators can be regular expressions.
- **ignore** (`str` or iterable of `str`) – functions in the graph to ignore (also uses regular expressions.) Useful for ignoring specific functions in a broad regex.
- **entry_point** (`bool`) – Set to `True` to mark this as an entry point - that is, this function will be called when the router is called directly.

In addition to all of the above, you can define a `wrap_node` function on a subclass of Router, which will need to receive node and an options dictionary. Any extra options passed to node will be passed down to the options dictionary. See `emit.router.CeleryRouter.wrap_node` as an example.

Returns decorated and wrapped function, or decorator if called directly

Examples

Multiple fields:

```
@router.node(['quotient', 'remainder'])
def division_with_remainder(msg):
    return msg.numer / msg.denom, msg.numer % msg.denom
```

This function would end up returing a dictionary that looked something like:

```
{'quotient': 2, 'remainder': 1}
```

The next node in the graph would recieve a `emit.message.Message` with "quotient" and "remainder" fields.

Emitting multiple values:

```
@router.node(['word'])
def parse_document(msg):
    for word in msg.document.clean().split(' '):
        yield word
```

If the function returns a generator, Emit will gather the values together and make sure the generator exits cleanly before returning (but this may change in the future via a flag.) Therefore, the return value will look like this:

```
({'word': "I've"},
 {'word': 'got'},
 {'word': 'a'},
 {'word': 'lovely'},
 {'word': 'bunch'},
 {'word': 'of'},
 {'word': 'coconuts'})
```

Each message in the tuple will be passed on individually in the graph.

add_entry_point(*destination*)
Add an entry point

Parameters **destination** (*str*) – node to route to initially

disable_routing()
disable routing (usually for testing purposes)

dispatch(*origin*, *destination*, *message*)
dispatch a message to a named function

Parameters

- **destination** (`str`) – destination to dispatch to
- **message** (`emit.message.Message` or subclass) – message to dispatch

enable_routing()
enable routing (after calling `disable_routing`)

get_message_from_call(**args*, ***kwargs*)
Get message object from a call.

Raises `TypeError` (if the format is not what we expect)

This is where arguments to nodes are turned into Messages. Arguments are parsed in the following order:

- A single positional argument (a `dict`)
- No positional arguments and a number of keyword arguments

get_name(*func*)
Get the name to reference a function by

Parameters **func** (*callable*) – function to get the name of

regenerate_routes()
regenerate the routes after a new route is added

register(*name*, *func*, *fields*, *subscribe_to*, *entry_point*, *ignore*)
Register a named function in the graph

Parameters

- **name** (`str`) – name to register
- **func** (*callable*) – function to remember and call

`fields`, `subscribe_to` and `entry_point` are the same as in `Router.node()`.

register_ignore(*origins*, *destination*)
Add routes to the ignore dictionary

Parameters

- **origins** (`str` or iterable of `str`) – a number of origins to register
- **destination** (`str`) – where the origins should point to

Ignore dictionary takes the following form:

```
{'node_a': set(['node_b', 'node_c']),
 'node_b': set(['node_d'])}
```

register_route(*origins*, *destination*)
Add routes to the routing dictionary

Parameters

- **origins** (`str` or iterable of `str` or None) – a number of origins to register
- **destination** (`str`) – where the origins should point to

Routing dictionary takes the following form:

```
{'node_a': set(['node_b', 'node_c']),
 'node_b': set(['node_d'])}
```

resolve_node_modules()
import the modules specified in init

route(*origin*, *message*)
Using the routing dictionary, dispatch a message to all subscribers

Parameters

- **origin** (`str`) – name of the origin node
- **message** (`emit.message.Message` or subclass) – message to dispatch

wrap_as_node(*func*)
wrap a function as a node

wrap_result(*name*, *result*)
Wrap a result from a function with it's stated fields

Parameters

- **name** (`str`) – fields to look up
- **result** (*anything*) – return value from function. Will be converted to tuple.

Raises `ValueError` if name has no associated fields

Returns `dict`

class `emit.router.`**CeleryRouter**(*celery_task*, **args*, ***kwargs*)
Router specifically for Celery routing

__init__(*celery_task*, **args*, ***kwargs*)
Specifically route when celery is needed

Parameters **celery_task** (*A celery task decorator, in any form*) – celery task to apply to all nodes (can be overridden in `Router.node()`.)

dispatch(*origin*, *destination*, *message*)
enqueue a message with Celery

Parameters

- **destination** (`str`) – destination to dispatch to
- **message** (`emit.message.Message` or subclass) – message to dispatch

wrap_node(*node*, *options*)
celery registers tasks by decorating them, and so do we, so the user can pass a celery task and we'll wrap our code with theirs in a nice package celery can execute.

class `emit.router.`**RQRouter**(*redis_connection*, **args*, ***kwargs*)
Router specifically for RQ routing

__init__(*redis_connection*, **args*, ***kwargs*)
Specific routing when using RQ

Parameters **redis_connection** (`redis.Redis`) – a redis connection to send to all the tasks (can be overridden in `Router.node()`.)

dispatch(*origin*, *destination*, *message*)
dispatch through RQ

wrap_node(*node*, *options*)
we have the option to construct nodes here, so we can use different queues for nodes without having to have different queue objects.

8.2 Message

class `emit.message.`**Message**(**args*, ***kwargs*)
Convenient wrapper around a dictionary to provide attribute access

as_dict()
representation of this message as a dictionary

Returns dict

as_json()
representation of this message as a json object

Returns str

class emit.message.**NoResult**
single value to return from a node to stop further processing

8.3 Multilang

class emit.multilang.**ShellNode**
callable object to wrap communication to a node in another language

to use this, subclass ShellNode, providing "command". Decorate it however you feel like.

Messages will be passed in on lines in msgpack format. This class expects similar output: msgpack messages separated by a newline.

__call__(*msg*)
call the command specified, processing output

deserialize(*msg*)
deserialize output to a Python object

get_command()
get the command as a list

get_cwd()
get directory to change to before running the command

CHAPTER

NINE

GLOSSARY

Some of the terms in this project may be a little rough. I'm still considering what things should really be called here, but here's a primer:

- *Router*: an object (implemented in `emit.router.Router`) that keeps references to functions and their names and handles dispatch. It knows where everything is.
- *Node*: a function or callable class that receives messages, processes them in it's own way, and passes them on down the graph.
- *Subscription*: an edge in the graph - only flows one way.
- *Graph*: a directed graph, like graph theory. A collection of nodes connected by subscriptions.

CHAPTER

TEN

CHANGELOG

10.1 0.3.0

- Better documentation
- *RQ support*

10.2 0.2.0

- New argument for `node: ignores`. Pass it some regex to ignore items in otherwise broad subscriptions.
- Add support for Python 2.6

10.3 0.1.0

- Initial Release to PyPI

Supported Pythons:

- CPython 2.6
- CPython 2.7
- CPython 3.2
- PyPy 1.9

CHAPTER

ELEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

CHAPTER

TWELVE

LICENSE

PYTHON MODULE INDEX

e