streamparse Documentation

Release 1.1.0

Parsely

June 23, 2015

Contents

streamparse lets you run Python code against real-time streams of data. Integrates with Apache Storm.

CHAPTER 1

Quickstart

1.1 Dependencies

1.1.1 Java and Clojure

To run local and remote computation clusters, streamparse relies upon a JVM technology called Apache Storm. The integration with this technology is lightweight, and for the most part, you don't need to think about it.

However, to get the library running, you'll need

1. JDK 7+, which you can install with apt-get, homebrew, or an installler; and
2. lein, which you can install from the project's page or github

Confirm that you have `lein` installed by running:

```
> lein version
```

You should get output similar to this:

```
Leiningen 2.3.4 on Java 1.7.0_55 Java HotSpot(TM) 64-Bit Server VM
```

If `lein` isn't installed, follow these directions.

Once that's all set, you install streamparse using `pip`:

```
> pip install streamparse
```

1.2 Your First Project

When working with streamparse, your first step is to create a project using the command-line tool, `sparse`:

```
> sparse quickstart wordcount

Creating your wordcount streamparse project...
    create    wordcount
    create    wordcount/.gitignore
    create    wordcount/config.json
    create    wordcount/fabfile.py
    create    wordcount/project.clj
    create    wordcount/README.md
    create    wordcount/src
```

```
    create    wordcount/src/bolts/
    create    wordcount/src/bolts/__init__.py
    create    wordcount/src/bolts/wordcount.py
    create    wordcount/src/spouts/
    create    wordcount/src/spouts/__init__.py
    create    wordcount/src/spouts/words.py
    create    wordcount/tasks.py
    create    wordcount/topologies
    create    wordcount/topologies/wordcount.clj
    create    wordcount/virtualenvs
    create    wordcount/virtualenvs/wordcount.txt
Done.

Try running your topology locally with:

    cd wordcount
    sparse run
```

The quickstart project provides a basic wordcount topology example which you can examine and modify. You can inspect the other commands that `sparse` provides by running:

```
> sparse -h
```

1.3 Project Structure

streamparse projects expect to have the following directory layout:

File/Folder	Contents
config.json	Configuration information for all of your topologies.
fabfile.py	Optional custom fabric tasks.
project.clj	leiningen project file, can be used to add external JVM dependencies.
src/	Python source files (bolts/spouts/etc.) for topologies.
tasks.py	Optional custom invoke tasks.
topologies/	Contains topology definitions written using the Clojure DSL for Storm.
virtualenvs/	Contains pip requirements files in order to install dependencies on remote Storm servers.

1.4 Defining Topologies

Storm's services are Thrift-based and although it is possible to define a topology in pure Python using Thrift, it introduces a host of additional dependencies which are less than trivial to setup for local development. In addition, it turns out that using Clojure to define topologies, still feels fairly Pythonic, so the authors of streamparse decided this was a good compromise.

Let's have a look at the definition file created by using the `sparse quickstart` command.

```
(ns wordcount
  (:use     [streamparse.specs])
  (:gen-class))

(defn wordcount [options]
   [
    ;; spout configuration
    {"word-spout" (python-spout-spec
          options
```

```
           "spouts.words.WordSpout"
           ["word"]
           )
    }
    ;; bolt configuration
    {"count-bolt" (python-bolt-spec
           options
           {"word-spout" :shuffle}
           "bolts.wordcount.WordCounter"
           ["word" "count"]
           :p 2
           )
    }
  ]
)
```

The first block of code we encounter effectively states "import the Clojure DSL functions for Storm":

```
(ns wordcount
  (:use     [backtype.storm.clojure])
  (:gen-class))
```

The next block of code actually defines the topology and stores it into a function named "wordcount".

```
(defn wordcount [options]
   [
    ;; spout configuration
    {"word-spout" (python-spout-spec
           options
           "spouts.words.WordSpout"
           ["word"]
           )
    }
    ;; bolt configuration
    {"count-bolt" (python-bolt-spec
           options
           {"word-spout" :shuffle}
           "bolts.wordcount.WordCounter"
           ["word" "count"]
           :p 2
           )
    }
  ]
)
```

It turns out, the name of the name of the function doesn't matter much, we've used `wordcount` above, but it could just as easily be `bananas`. What is important, is that **the function must return an array with only two dictionaries and take one argument**.

The first dictionary holds a named mapping of all the spouts that exist in the topology, the second holds a named mapping of all the bolts. The `options` argument contains a mapping of topology settings.

An additional benefit of defining topologies in Clojure is that we're able to mix and match the types of spouts and bolts. In most cases, you may want to use a pure Python topology, but you could easily use JVM-based spouts and bolts or even spouts and bolts written in other languages like Ruby, Go, etc.

Since you'll most often define spouts and bolts in Python however, we'll look at two important functions provided by streamparse: `python-spout-spec` and `python-bolt-spec`.

When creating a Python-based spout, we provide a name for the spout and a definition of that spout via

`python-spout-spec`:

```
{"sentence-spout-1" (python-spout-spec
                     ;; topology options passed in
                     options
                     ;; name of the python class to ``run``
                     "spouts.SentenceSpout"
                     ;; output specification, what named fields will this spout emit?
                     ["sentence"]
                     ;; configuration parameters, can specify multiple
                     :p 2)
 "sentence-spout-2" (shell-spout-spec
                     options
                     "spouts.OtherSentenceSpout"
                     ["sentence"])}
```

In the example above, we've defined two spouts in our topology: `sentence-spout-1` and `sentence-spout-2` and told Storm to run these components. `python-spout-spec` will use the `options` mapping to get the path to the python executable that Storm will use and streamparse will run the class provided. We've also let Storm know exactly what these spouts will be emitting, namely a single field called `sentence`.

You'll notice that in `sentence-spout-1`, we've passed an optional map of configuration parameters `:p 2`, we'll get back to this later.

Creating bolts is very similar and uses the `python-bolt-spec` function:

```
{"sentence-splitter" (python-bolt-spec
                      ;; topology options passed in
                      options
                      ;; inputs, where does this bolt recieve it's tuples from?
                      {"sentence-spout-1" :shuffle
                       "sentence-spout-2" :shuffle}
                      ;; class to run
                      "bolts.SentenceSplitter"
                      ;; output spec, what tuples does this bolt emit?
                      ["word"]
                      ;; configuration parameters
                      :p 2)
 "word-counter" (python-bolt-spec
                 options
                 ;; recieves tuples from "sentence-splitter", grouped by word
                 {"sentence-splitter" ["word"]}
                 "bolts.WordCounter"
                 ["word" "count"])
 "word-count-saver" (python-bolt-spec
                     ;; topology options passed in
                     options
                     {"word-counter" :shuffle}
                     "bolts.WordSaver"
                     ;; does not emit any fields
                     [])}
```

In the example above, we define 3 bolts by name `sentence-splitter`, `word-counter` and `word-count-saver`. Since bolts are generally supposed to process some input and optionally produce some output, we have to tell Storm where a bolts inputs come from and whether or not we'd like Storm to use any stream grouping on the tuples from the input source.

In the `sentence-splitter` bolt, you'll notice that we define two input sources for the bolt. It's completely fine to add multiple sources to any bolts.

In the `word-counter` bolt, we've told Storm that we'd like the stream of input tuples to be grouped by the named field `word`. Storm offers comprehensive options for stream groupings, but you will most commonly use a **shuffle** or **fields** grouping:

- **Shuffle grouping**: Tuples are randomly distributed across the bolt's tasks in a way such that each bolt is guaranteed to get an equal number of tuples.
- **Fields grouping**: The stream is partitioned by the fields specified in the grouping. For example, if the stream is grouped by the "user-id" field, tuples with the same "user-id" will always go to the same task, but tuples with different "user-id"'s may go to different tasks.

There are more options to configure with spouts and bolts, we'd encourage you to refer to Storm's Concepts for more information.

1.5 Spouts and Bolts

The general flow for creating new spouts and bolts using streamparse is to add them to your `src` folder and update the corresponding topology definition.

Let's create a spout that emits sentences until the end of time:

```
import itertools

from streamparse.spout import Spout


class SentenceSpout(Spout):

    def initialize(self, stormconf, context):
        self.sentences = [
            "She advised him to take a long holiday, so he immediately quit work and took a trip arou
            "I was very glad to get a present from her",
            "He will be here in half an hour",
            "She saw him eating a sandwich",
        ]
        self.sentences = itertools.cycle(self.sentences)

    def next_tuple(self):
        sentence = next(self.sentences)
        self.emit([sentence])

    def ack(self, tup_id):
        pass  # if a tuple is processed properly, do nothing

    def fail(self, tup_id):
        pass  # if a tuple fails to process, do nothing
```

The magic in the code above happens in the `initialize()` and `next_tuple()` functions. Once the spout enters the main run loop, streamparse will call your spout's `initialize()` method. After initialization is complete, streamparse will continually call the spout's `next_tuple()` method where you're expected to emit tuples that match whatever you've defined in your topology definition.

Now let's create a bolt that takes in sentences, and spits out words:

```
import re

from streamparse.bolt import Bolt
```

```
class SentenceSplitterBolt(Bolt):

    def process(self, tup):
        sentence = tup.values[0]  # extract the sentence
        sentence = re.sub(r"[,.;!\?]", "", sentence)  # get rid of punctuation
        words = [[word.strip()] for word in sentence.split(" ") if word.strip()]
        if not words:
            # no words to process in the sentence, fail the tuple
            self.fail(tup)
            return

        self.emit_many(words)
        # tuple acknowledgement is handled automatically
```

The bolt implementation is even simpler. We simply override the default `process()` method which streamparse calls when a tuple has been emitted by an incoming spout or bolt. You are welcome to do whatever processing you would like in this method and can further emit tuples or not depending on the purpose of your bolt.

In the `SentenceSplitterBolt` above, we have decided to use the `emit_many()` method instead of `emit()` which is a bit more efficient when sending a larger number of tuples to Storm.

If your `process()` method completes without raising an Exception, streamparse will automatically ensure any emits you have are anchored to the current tuple being processed and acknowledged after `process()` completes.

If an Exception is raised while `process()` is called, streamparse automatically fails the current tuple prior to killing the Python process.

1.5.1 Bolt Configuration Options

You can disable the automatic acknowleding, anchoring or failing of tuples by adding class variables set to false for: `auto_ack`, `auto_anchor` or `auto_fail`. All three options are documented in *streamparse.bolt.Bolt*.

Example:

```
from streamparse.bolt import Bolt

class MyBolt(Bolt):

    auto_ack = False
    auto_fail = False

    def process(self, tup):
        # do stuff...
        if error:
          self.fail(tup)  # perform failure manually
        self.ack(tup)  # perform acknowledgement manually
```

1.5.2 Failed Tuples

In the example above, we added the ability to fail a sentence tuple if it did not provide any words. What happens when we fail a tuple? Storm will send a "fail" message back to the spout where the tuple originated from (in this case `SentenceSpout`) and streamparse calls the spout's `fail()` method. It's then up to your spout implementation to decide what to do. A spout could retry a failed tuple, send an error message, or kill the topology.

1.6 Remote Deployment

When you are satisfied that your topology works well via testing with:

```
> sparse run -d
```

You can submit your topology to a remote Storm cluster using the command:

```
sparse submit [--environment <env>] [--name <topology>] [-dv]
```

Before submitting, you have to have at least one environment configured in your project's `config.json` file. Let's create a sample environment called "prod" in our `config.json` file:

```
{
    "library": "",
    "topology_specs": "topologies/",
    "virtualenv_specs": "virtualenvs/",
    "envs": {
        "prod": {
            "user": "storm",
            "nimbus": "storm1.my-cluster.com",
            "workers": [
                "storm1.my-cluster.com",
                "storm2.my-cluster.com",
                "storm3.my-cluster.com"
            ],
            "log": {
                "path": "/var/log/storm/streamparse",
                "max_bytes": 100000,
                "backup_count": 10,
                "level": "info"
            },
            "virtualenv_root": "/data/virtualenvs/"
        }
    }
}
```

We've now defined a `prod` environment that will use the user `storm` when deploying topologies. Before submitting the topology though, streamparse will automatically take care of instaling all the dependencies your topology requires. It does this by sshing into everyone of the nodes in the `workers` config variable and building a virtualenv using the the project's local `virtualenvs/<topology_name>.txt` requirements file.

This implies a few requirements about the user you specify per environment:

1. Must have ssh access to all servers in your Storm cluster
2. Must have write access to the `virtualenv_root` on all servers in your Storm cluster

streamparse also assumes that virtualenv is installed on all Storm servers.

Once an environment is configured, we could deploy our wordcount topology like so:

```
> sparse submit
```

Seeing as we have only one topology and environment, we don't need to specify these explicitly. streamparse will now:

1. Package up a JAR containing all your Python source files
2. Build a virtualenv on all your Storm workers (in parallel)
3. Submit the topology to the `nimbus` server

1.6.1 Logging

The Storm supervisor needs to have access to the `log.path` directory for logging to work (in the example above, `/var/log/storm/streamparse`). If you have properly configured the `log.path` option in your config, streamparse will automatically set up a log files on each Storm worker in this path using the following filename convention:

```
streamparse_<topology_name>_<component_name>_<task_id>_<process_id>.log
```

Where:

- `topology_name`: is the `topology.name` variable set in Storm
- `component_name`: is the name of the currently executing component as defined in your topology definition file (.clj file)
- `task_id`: is the task ID running this component in the topology
- `process_id`: is the process ID of the Python process

streamparse uses Python's `logging.handlers.RotatingFileHandler` and by default will only save 10 1 MB log files (10 MB in total), but this can be tuned with the `log.max_bytes` and `log.backup_count` variables.

The default logging level is set to `INFO`, but if you can tune this with the `log.level` setting which can be one of critical, error, warning, info or debug. **Note** that if you perform `sparse run` or `sparse submit` with the `--debug` set, this will override your `log.level` setting and set the log level to debug.

When running your topology locally via `sparse run`, your log path will be automatically set to `/path/to/your/streamparse/project/logs`.

CHAPTER 2

API

2.1 Tuples

class streamparse.ipc.**Tuple**(*id*, *component*, *stream*, *task*, *values*)
Storm's primitive data type passed around via streams.

Variables

- **id** – the ID of the tuple.
- **component** – component that the tuple was generated from.
- **stream** – the stream that the tuple was emitted into.
- **task** – the task the tuple was generated from.
- **values** – the payload of the tuple where data is stored.

You should never have to instantiate an instance of a *streamparse.ipc.Tuple* yourself as streamparse handles this for you prior to, for example, a *streamparse.bolt.Bolt*'s `process()` method being called.

None of the emit methods for bolts or spouts require that you pass a *streamparse.ipc.Tuple* instance.

2.2 Spouts

Spouts are data sources for topologies, they can read from any data source and emit tuples into streams.

class streamparse.spout.**Spout**
Base class for all streamparse spouts.

For more information on spouts, consult Storm's Concepts documentation.

ack(*tup_id*)
Called when a bolt acknowledges a tuple in the topology.

Parameters **tup_id** (*str*) – the ID of the tuple that has been fully acknowledged in the topology.

emit(*tup*, *tup_id=None*, *stream=None*, *direct_task=None*, *need_task_ids=None*)
Emit a spout tuple message.

Parameters

- **tup** (*list*) – the tuple to send to Storm. Should contain only JSON-serializable data.
- **tup_id** (*str*) – the ID for the tuple. Leave this blank for an unreliable emit.

- **stream** (*str*) – ID of the stream this tuple should be emitted to. Leave empty to emit to the default stream.
- **direct_task** (*int*) – the task to send the tuple to if performing a direct emit.
- **need_task_ids** (*bool*) – indicate whether or not you'd like the task IDs the tuple was emitted (default: `True`).

Returns a `list` of task IDs that the tuple was sent to. Note that when specifying direct_task, this will be equal to `[direct_task]`. If you specify `need_task_ids=False`, this function will return `None`.

emit_many(*tuples*, *stream=None*, *anchors=None*, *direct_task=None*, *need_task_ids=None*)
Emit multiple tuples.

Parameters

- **tuples** (*list*) – a `list` containing `list` s of tuple payload data to send to Storm. All tuples should contain only JSON-serializable data.
- **stream** (*str*) – the ID of the steram to emit these tuples to. Specify `None` to emit to default stream.
- **anchors** (*list*) – IDs the tuples (or *streamparse.ipc.Tuple* instances) which the emitted tuples should be anchored to. If `auto_anchor` is set to `True` and you have not specified `anchors`, `anchors` will be set to the incoming/most recent tuple ID(s).
- **direct_task** (*int*) – indicates the task to send the tuple to.
- **need_task_ids** (*bool*) – indicate whether or not you'd like the task IDs the tuple was emitted (default: `True`).

fail(*tup_id*)
Called when a tuple fails in the topology

A Spout can choose to emit the tuple again or ignore the fail. The default is to ignore.

Parameters **tup_id** (*str*) – the ID of the tuple that has failed in the topology either due to a bolt calling `fail()` or a tuple timing out.

initialize(*storm_conf*, *context*)
Called immediately after the initial handshake with Storm and before the main run loop. A good place to initialize connections to data sources.

Parameters

- **storm_conf** (*dict*) – the Storm configuration for this Bolt. This is the configuration provided to the topology, merged in with cluster configuration on the worker node.
- **context** (*dict*) – information about the component's place within the topology such as: task IDs, inputs, outputs etc.

log(*message*, *level=None*)
Log a message to Storm optionally providing a logging level.

Parameters

- **message** (*str*) – the log message to send to Storm.
- **level** (*str*) – the logging level that Storm should use when writing the `message`. Can be one of: trace, debug, info, warn, or error (default: `info`).

next_tuple()
Implement this function to emit tuples as necessary.

This function should not block, or Storm will think the spout is dead. Instead, let it return and streamparse will send a noop to storm, which lets it know the spout is functioning.

2.3 Bolts

class `streamparse.bolt.`**Bolt**
Bases: `streamparse.base.Component`

The base class for all streamparse bolts.

For more information on bolts, consult Storm's Concepts documentation.

Example:

```
from streamparse.bolt import Bolt

class SentenceSplitterBolt(Bolt):

    def process(self, tup):
        sentence = tup.values[0]
        for word in sentence.split(" "):
            self.emit([word])
```

ack(*tup*)
Indicate that processing of a tuple has succeeded.

Parameters **tup** (*str or Tuple*) – the tuple to acknowledge.

auto_ack = True
A `bool` indicating whether or not the bolt should automatically acknowledge tuples after `process()` is called. Default is `True`.

auto_anchor = True
A `bool` indicating whether or not the bolt should automatically anchor emits to the incoming tuple ID. Tuple anchoring is how Storm provides reliability, you can read more about tuple anchoring in Storm's docs. Default is `True`.

auto_fail = True
A `bool` indicating whether or not the bolt should automatically fail tuples when an exception occurs when the `process()` method is called. Default is `True`.

emit(*tup*, *stream=None*, *anchors=None*, *direct_task=None*, *need_task_ids=None*)
Emit a new tuple to a stream.

Parameters

- **tup** (*list*) – the Tuple payload to send to Storm, should contain only JSON-serializable data.
- **stream** (*str*) – the ID of the stream to emit this tuple to. Specify `None` to emit to default stream.
- **anchors** (*list*) – IDs the tuples (or *`streamparse.ipc.Tuple`* instances) which the emitted tuples should be anchored to. If `auto_anchor` is set to `True` and you have not specified `anchors`, `anchors` will be set to the incoming/most recent tuple ID(s).
- **direct_task** (*int*) – the task to send the tuple to.
- **need_task_ids** (*bool*) – indicate whether or not you'd like the task IDs the tuple was emitted (default: `True`).

Returns a `list` of task IDs that the tuple was sent to. Note that when specifying direct_task, this will be equal to `[direct_task]`. If you specify `need_task_ids=False`, this function will return `None`.

emit_many(*tuples*, *stream=None*, *anchors=None*, *direct_task=None*, *need_task_ids=None*)
Emit multiple tuples.

Parameters

- **tuples** (*list*) – a `list` containing `list` s of tuple payload data to send to Storm. All tuples should contain only JSON-serializable data.
- **stream** (*str*) – the ID of the steram to emit these tuples to. Specify `None` to emit to default stream.
- **anchors** (*list*) – IDs the tuples (or *streamparse.ipc.Tuple* instances) which the emitted tuples should be anchored to. If `auto_anchor` is set to `True` and you have not specified `anchors`, `anchors` will be set to the incoming/most recent tuple ID(s).
- **direct_task** (*int*) – indicates the task to send the tuple to.
- **need_task_ids** (*bool*) – indicate whether or not you'd like the task IDs the tuple was emitted (default: `True`).

fail(*tup*)
Indicate that processing of a tuple has failed.

Parameters **tup** (*str or Tuple*) – the tuple to fail (`id` if `str`).

initialize(*storm_conf*, *context*)
Called immediately after the initial handshake with Storm and before the main run loop. A good place to initialize connections to data sources.

Parameters

- **storm_conf** (*dict*) – the Storm configuration for this Bolt. This is the configuration provided to the topology, merged in with cluster configuration on the worker node.
- **context** (*dict*) – information about the component's place within the topology such as: task IDs, inputs, outputs etc.

log(*message*, *level=None*)
Log a message to Storm optionally providing a logging level.

Parameters

- **message** (*str*) – the log message to send to Storm.
- **level** (*str*) – the logging level that Storm should use when writing the `message`. Can be one of: trace, debug, info, warn, or error (default: `info`).

process(*tup*)
Process a single tuple *streamparse.ipc.Tuple* of input

This should be overridden by subclasses. *streamparse.ipc.Tuple* objects contain metadata about which component, stream and task it came from. The actual values of the tuple can be accessed by calling `tup.values`.

Parameters **tup** (streamparse.ipc.Tuple) – the tuple to be processed.

class `streamparse.bolt.`**BatchingBolt**
Bases: *streamparse.bolt.Bolt*

A bolt which batches tuples for processing.

Batching tuples is unexpectedly complex to do correctly. The main problem is that all bolts are single-threaded. The difficult comes when the topology is shutting down because Storm stops feeding the bolt tuples. If the bolt is blocked waiting on stdin, then it can't process any waiting tuples, or even ack ones that were asynchronously written to a data store.

This bolt helps with that grouping tuples based on a time interval and then processing them on a worker thread.

To use this class, you must implement `process_batch`. `group_key` can be optionally implemented so that tuples are grouped before `process_batch` is even called.

Example:

```
from streamparse.bolt import BatchingBolt

class WordCounterBolt(BatchingBolt):

    secs_between_batches = 5

    def group_key(self, tup):
        word = tup.values[0]
        return word  # collect batches of words

    def process_batch(self, key, tups):
        # emit the count of words we had per 5s batch
        self.emit([key, len(tups)])
```

auto_ack = True
A `bool` indicating whether or not the bolt should automatically acknowledge tuples after `process_batch()` is called. Default is `True`.

auto_anchor = True
A `bool` indicating whether or not the bolt should automatically anchor emits to the incoming tuple ID. Tuple anchoring is how Storm provides reliability, you can read more about tuple anchoring in Storm's docs. Default is `True`.

auto_fail = True
A `bool` indicating whether or not the bolt should automatically fail tuples when an exception occurs when the `process_batch()` method is called. Default is `True`.

group_key(*tup*)
Return the group key used to group tuples within a batch.

By default, returns None, which put all tuples in a single batch, effectively just time-based batching. Override this create multiple batches based on a key.

Parameters **tup** (Tuple) – the tuple used to extract a group key

Returns Any `hashable` value.

process_batch(*key*, *tups*)
Process a batch of tuples. Should be overridden by subclasses.

Parameters

- **key** (*hashable*) – the group key for the list of batches.
- **tups** (*list*) – a *list* of *streamparse.ipc.Tuple* s for the group.

secs_between_batches = 2
The time (in seconds) between calls to `process_batch()`. Note that if there are no tuples in any batch, the BatchingBolt will continue to sleep. Note: Can be fractional to specify greater precision (e.g. 2.5).

CHAPTER 3

Frequently Asked Questions (FAQ)

3.1 General Questions

3.1.1 Is streamparse compatible with Python 3?

Yes, streamparse is fully compatible with Python 3 starting with version 3.3 which we use in our unit tests.

3.1.2 How can I contribute to streamparse?

Thanks for your interest in contributing to streamparse. We think you'll find the core maintainers great to work with and will help you along the way when contributing pull requests.

If you already know what you'd like to add to streamparse then by all means, feel free to submit a pull request and we'll review it.

If you're unsure about how to contribute, check out our open issues and find one that looks interesting to you (we particularly need help on all issues marked with the "help wanted" label).

If you're not sure how to start or have some questions, shoot us an email in the streamparse user group and we'll give you a hand.

From there, get to work on your fix and submit a pull request when ready which we'll review.

3.1.3 How do I trigger some code before or after I submit my topology?

After you create a streamparse project using `sparse quickstart`, you'll have both a `tasks.py` in that directory as well as `fabric.py`. In either of these files, you can specify two functions: `pre_submit` and `post_submit` which are expected to accept three arguments:

- `topology_name`: the name of the topology being submitted
- `env_name`: the name of the environment where the topology is being submitted (e.g. `"prod"`)
- `env_config`: the relevant config portion from the `config.json` file for the environment you are submitting the topology to

Here is a sample `tasks.py` file that sends a message to IRC after a topology is successfully submitted to prod.

```
# my_project/tasks.py
from __future__ import absolute_import, print_function, unicode_literals

from invoke import task, run
from streamparse.ext.invoke import *


def post_submit(topo_name, env_name, env_config):
    if env_name == "prod":
        write_to_irc("Deployed {} to {}".format(topo_name, env_name))
```

3.2 Errors While Running streamparse

3.2.1 I received an "InvalidClassException" while submitting my topology, what do I do?

If the Storm version dependency you specify in your `project.clj` file is different from the version of Storm running on your cluster, you'll get an error in Storm's logs that looks something like the following when you submit your topology:

```
2014-07-07 02:30:27 b.s.d.executor [INFO] Finished loading executor __acker:[2 2]
2014-07-07 02:30:27 b.s.d.executor [INFO] Preparing bolt __acker:(2)
2014-07-07 02:30:27 b.s.d.executor [INFO] Prepared bolt __acker:(2)
2014-07-07 02:30:27 b.s.d.executor [INFO] Loading executor count-bolt:[4 4]
2014-07-07 02:30:27 b.s.d.worker [ERROR] Error on initialization of server mk-worker
java.lang.RuntimeException: java.io.InvalidClassException: backtype.storm.task.ShellBolt; local class
    at backtype.storm.utils.Utils.deserialize(Utils.java:93) ~[storm-core-0.9.2-incubating.jar:0.9.2-
    at backtype.storm.utils.Utils.getSetComponentObject(Utils.java:235) ~[storm-core-0.9.2-incubating
    at backtype.storm.daemon.task$get_task_object.invoke(task.clj:73) ~[storm-core-0.9.2-incubating.j
    at backtype.storm.daemon.task$mk_task_data$fn__3061.invoke(task.clj:180) ~[storm-core-0.9.2-incub
    at backtype.storm.util$assoc_apply_self.invoke(util.clj:816) ~[storm-core-0.9.2-incubating.jar:0.
    at backtype.storm.daemon.task$mk_task_data.invoke(task.clj:173) ~[storm-core-0.9.2-incubating.jar
    at backtype.storm.daemon.task$mk_task.invoke(task.clj:184) ~[storm-core-0.9.2-incubating.jar:0.9.
    at backtype.storm.daemon.executor$mk_executor$fn__5510.invoke(executor.clj:321) ~[storm-core-0.9.
    at clojure.core$map$fn__4207.invoke(core.clj:2485) ~[clojure-1.5.1.jar:na]
```

Check to ensure the version of Storm in your `project.clj` file matches the Storm version running on your cluster, then try resubmitting your topology.

```
(defproject my-project "0.0.1-SNAPSHOT"
  :source-paths ["topologies"]
  :resource-paths ["_resources"]
  :target-path "_build"
  :min-lein-version "2.0.0"
  :jvm-opts ["-client"]
  :dependencies  [[org.apache.storm/storm-core "0.9.1-incubating"] ;; this should match your Storm c
                  [com.parsely/streamparse "0.0.3-SNAPSHOT"]]
  :jar-exclusions     [#"log4j\.properties" #"backtype" #"trident" #"META-INF" #"meta-inf" #"\.yaml"]
  :uberjar-exclusions [#"log4j\.properties" #"backtype" #"trident" #"META-INF" #"meta-inf" #"\.yaml"]
)
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search