jetstream Documentation

Release 1.0

Petri Savolainen

December 02, 2013

Contents

Contents:

CHAPTER 1

Overview

Jetstream provides a configurable data batch processing tool and a framework for data integration applications, ie. hooking incoming and outgoing data streams into any Python application and processing the data in various ways.

It does not enforce a particular event loop, http server or client library, SQL library, ORM or such. Those are all outside its scope. Instead, Jetstream can be extended by following kinds of data processing components:

- inputs for reading data from various sources
- inspectors for e.g. checking data conformity
- transformers for modifying data and/or creating stuff from it
- outputs for receiving data and possibly writing it somewhere else

Some common built-in components are included and it's easy to write more. Jetstream then provides facilities for running data processing pipelines composed of the components, by the configuration.

Both the components and the pipelines are configured using YAML. **No programming is required to process data using Jetstream**. On the other hand, Jetstream is easy to extend and/or incorporate into your own app.

- Free software: GPL3 licensed
- Documentation: http://jetstream.rtfd.org.

CHAPTER 2

Installation

At the command line:

```
$ easy_install jetstream
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv jetstream
$ pip install jetstream
```

Next, add a configuration file, by default named *config.yaml* and you can proceed to configuring Jetstream for use.

CHAPTER 3

Usage

There are two ways to use Jetstream: either to perform some tasks on your data using the ‘jet’ command-line tool provided by Jetstream, or as a part of your own application.

In either case, Jetstream must first be configured, using YAML. The configuration tells Jetstream what components to instantiate and how, and how to combine them into one or more pipes runnable by Jetstream.

3.1 Configuring Jetstream

The YAML configuration consists of two or more sections declaring the components to be used (at least an *Input* and *Output*) and a section declaring the *pipes*.

3.1.1 Configuring components

Each *component* is configured under a component type section which is either ‘inputs’, ‘introspectors’, ‘transformers’, or ‘outputs’. Under the type section, each component is listed as:

```
<component title>: &<component_id>
  description: <some description here>
  use: <a fully-qualified Python dotted name of the factory>
```

The description field is optional but recommended. Also note that the *component_id* can not include spaces.

An example configuration of an *Input component*:

```
inputs:
  my MySQL data source: &sqlsource
    description: an example MySQL data source
    use: mypackage.mymodule.get_my_sql_source
```

3.1.2 Configuring pipes

Each *pipe* is configured under ‘pipes’ section, and is of the form:

```
<pipe title>:
  - *<component_id>
  - *<component2_id>
```

Where there can be an arbitrary number of component id's listed.

An example configuration of a *pipe* with two components:

```
pipes:
   example pipe:
      - *sqlsource
      - *csvoutput
```

Any number of components can be freely arranged into a *pipe*, as long as it is started by an *Input*, and ends in an *Output* - although if there's no Output configured, Jetstream will add a standard Output that simply prints out the *data records*.

3.1.3 Full example

Here is the full configuration file from Jetstream tests:

```
inputs:
   dummy source: &source
      description: a dummy source
      use: tests.components.Input

inspectors:
   dummy validator: &validate
      description: a dummy validator
      use: tests.components.Validator

transformers:
   dummy mapper: &map
      description: a basic dummy mapper
      use: jetstream.util.FieldMapper
      map:
         - number: Numero
         - description: Selite
         - amount: Summa
   dummy constructor: &construct
      description: a simple object constructor
      use: jetstream.util.KlassConstructor

outputs:
   dummy subscriber: &subscribe
      description: a dummy subscriber
      use: tests.components.Subscriber

pipes:
   dummy pipe:
      - *source
      - *map
      - *validate
```

3.2 Using the Jetstream cli

Todo

write the cli & docs

3.3 Embedding Jetstream

To use Jetstream in your own project:

```
import jetstream
```

Todo

explain how to embed Jetstream

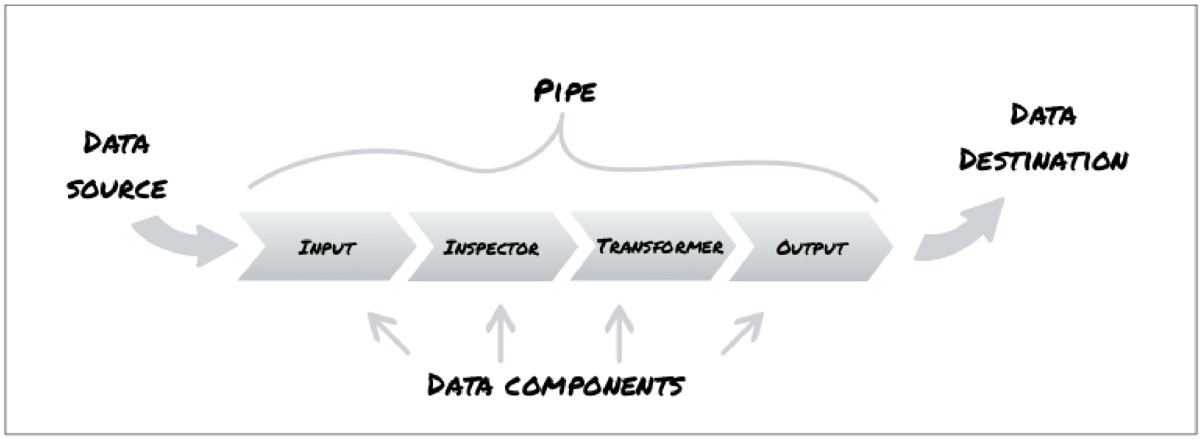
CHAPTER 4

Extending Jetstream

The primary means of extending Jetstream is by developing new data components, which is relatively easy. The architecture overview describes how *data components* fit in Jetstream & what their role is.

4.1 Architecture overview

This diagram illustrates how Jetstream provides for running a configurable sequence of data-processing steps, implemented as a '*pipe*' of *Input*, *Inspector*, *Transformer* and *Output* components.



Jetstream runs component pipes using what is called a *Streamer*.

4.2 Implementing components

If you're not sure what kind of component to develop, maybe it's useful to read on some thoughts on what different components can be used for, see `cases`.

To create a component, Jetstream needs to be able to call a component factory, passing it two parameters:

1. configuration settings (a mapping)

2. the data *stream* (an iterable)

It is the responsibility of the factory to return the component; a component is anything that implements the iterator protocol, producing the *data stream*. Such as for example a Python generator function.

In practice, it is convenient to implement the factory and the component using a single class that accepts the configuration settings and the stream as parameters to its constructor, implements an *__iter__* method that *yields* (making the method a generator), and implements *__call__* so that it returns *self*.

Which is precisely what these (abstract) base classes in the jetstream.base module provide.

They simply implement a passthru component that does nothing to the stream.

Note: For a minimal implementation, override the *__iter__* method of a base component with a generator function (anything that *yields*). The stream parameter (passed to the component at initialization) is accessible at *self._stream*.

Todo

invent type identification of generator function components so that they can by used as components, ie. without subclassing

4.2.1 Input components

In addition to implementing some data reading capability, Input components need to provide an indexer that returns a mapping of data to be indexed.

the *get_key* method that should return a (attributename, attributetype) tuple. Jetstream will use that information to index the data, associating it with the the Streamer run, which in turn logs information such as configuration, source and timestamp.

The component should be able to handle the data stream regardless of how many records it provides.

since the number of records that can be read may be limited in many ways; for example by the data source:

- data may contain a fixed number of records by nature
- there may be usage limits to data volume or time of day
- there is an error at data source

... or by user context:

- data may be limited by authentication > authorization
- only a subset of available data is requested by query terms
- user may want a smaller number of records than what is available

... or due to an error occurring at the data source or network:

- data stream may unexpectedly terminate, or may arrive only partially

It is of course also possible that data is available ad infinitum.

Todo

capability declarations related to stream reading abilities

4.2.2 Inspector components

4.2.3 Transformer components

4.2.4 Output components

and an Output is allowed to return an exhausted iterator.

4.3 Registering components

Components don't necessarily need to be registered; they can always be referred to using the *use* field of the YAML component configuration.

However, registering components makes it easier to use them.

Jetstream uses the standard setuptools entry points API for pluggable component registrations. Each entry point is expected to resolve to a component factory that is a callable accepting two parameters:

1. configuration settings (a mapping)
2. the data *streamer* (an iterable)

The entry points to register under are as follows, one for each component type:

- jetstream.input
- jetstream.inspector
- jetstream.transformer
- jetstrem.output

Here is an example entry point declaration to go into your package's *setup.py*:

```
entry_points = {
    'jetstream.input': [
        'NoSQLInput = jetstream.nosqlinput.component:get_component'
    ]
},
```

This would register the *get_component* function found in module *jetstream.nosqlinput.component* as the factory for an Input component called "NoSQLInput".

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given. You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Contribute new data components

Please consider contributing any data components your develop back to the Jetstream community.

In the naming of any contributed component package, please use form *jetstream.<title><type>* where *<type>* is the component type (input/ introspect/transform/output), and *<title>* describes the functionality. For example 'jetstream.podioinput' would be the name of an Input component that reads data from the Podio (http://www.podio.com) service via its HTTP API.

Also, please add "jetstream" as a package keyword.

5.1.2 Report Bugs

Report bugs at https://github.com/koodaamo/jetstream/issues.

If you are reporting a bug, please include:

- Your operating system name and version, Python version and Jetstream version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.3 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

5.1.4 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

5.1.5 Write Documentation

Jetstream could always use more documentation, whether as part of the official jetstream docs, in docstrings, or even on the web in blog posts, articles, and such.

Look through the GitHub issues for features. Anything tagged with "docs" is open to whoever wants to add the documentation.

5.1.6 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/koodaamo/jetstream/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *jetstream* for local development.

1. Fork the *jetstream* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/jetstream.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv jetstream
$ cd jetstream/
$ python setup.py develop
```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
  $ flake8 jetstream tests
        $ python setup.py test
  $ tox

To get flake8 and tox, just pip install them into your virtualenv.
```

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, please check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Add the new feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, and for PyPy. Check https://travis-ci.org/koodaamo/jetstream/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_jetstream
```

CHAPTER 6

Credits

6.1 Development Lead

- Petri Savolainen <petri.savolainen@koodaamo.fi>

6.2 Contributors

None yet. Why not be the first?

CHAPTER 7

History

7.1 0.1.0 (2013-11-5)

- First release on PyPI.

CHAPTER 8

Indices and tables

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