rdc-etl Documentation

Release 1.0.0a6

Romain Dorgueil

April 18, 2014

Extract Transform Load (ETL) toolkit for python.

DIY framework to create multithreaded python callables that can transform any stream(s) of key/value lists into any other stream(s).

Concepts are similar to heavy market tools like talend or pentaho, but unlike those, it's a lightweight framework and there is no wysiwyg editor provided.

CHAPTER 1

Install

1.1 Using PyPI

The project is currently marked as alpha. It's available on PyPI, but you need to specify a version spec for pip to find it:

```
$ pip install rdc.etl==1.0.0a3
```

You can also ask for the latest version:

```
$ pip install rdc.etl\>1.0.0a
```

You should be done. You can check in a python shell that it worked.

```
>>> from rdc.etl import __version__
>>> print __version__
```

1.2 Using git

You can also install `rdc.etl` from sources, using git. Depending on what you want to do, you can either use `master` branch which contains the latest stable code (aka what is published to PyPI), or the `dev` branch (aka the target of incoming cool features).

```
$ git clone https://github.com/rdcli/etl.git
$ cd etl
$ python setup.py develop
```

Note: Virtualenv usage is highly advised.

Kickstart

To get started, you should also read pragmatic examples in the *Cookbook*.

2.1 Create an empty project

If you want to bootstrap an ETL project on your computer, you can now do it using the provided PasteScript template.

```
pip install PasteScript
paster create -t etl_project MyProject
```

2.2 Overview of concepts

2.2.1 Extract

`Extract` is a flexible base class to write extract transformations. We use a generator here, real life would usually use databases, webservices, files ...

```
from rdc.etl.transform.extract import Extract

@Extract
def my_extract():
    yield {'foo': 'bar', 'bar': 'min'}
    yield {'foo': 'boo', 'bar': 'put'}
```

For more informations, see the extracts reference.

2.2.2 Transform

`Transform` is a flexible base class for all kind of transformations.

```
from rdc.etl.transform import Transform

@Transform
def my_transform(hash, channel):
    yield hash.update({
        'foo': hash['foo'].upper()
    })
```

For more informations, see the transformations reference.

2.2.3 Load

We'll use the screen as our load target ...

```
from rdc.etl.transform.util import Log

my_load = Log()
```

For more informations, see the loads reference.

Note: *Log* is not a "load" transformation stricto sensu (as it acts as an identity transformation, sending to the default output channel whatever comes in its default input channel), but we'll use it as such for demonstration purpose.

2.3 Run

Let's create a `Job`. It will be used to:

- Connect transformations
- Manage threads
- Monitor execution

```
from rdc.etl.job import Job

job = Job()
```

The `Job` has a `add_chain()` method that can be used to easily plug a list of ordered transformations together.

```
job.add_chain(my_extract, my_transform, my_load)
```

Our job is ready, you can run it.

```
job()
```

For more informations, see the jobs documentation.

CHAPTER 3

Jobs

3.1 Concept

The Scheduler and the Overseer

Jobs, (previsouly *harness*), are the glue that ties transformations together and let them interract.

```
>>> job = Job()
```

Jobs have a few purposes:

- **Manage the graph**. and their input/output channels and connections.

```
>>> # Add a transform. Each transform has its own thread. You should avoid using the lower level meth
>>> # unless you perfectly understand the underlying mechanisms.
>>> job.add_chain(t1, t2, t3)
```

- **Manage threads and work units**. Each transform is contained in a thread that will live from the job start to whatever means that the contained transform is now "dead". The job will dispatch work between those threads, and monitor their status.

```
>>> # Show thread status
>>> print '\n'.join(map(repr, h.get_threads()))
(1, - Extract-1 in=1 out=3)
(2, - SimpleTransform-2 in=3 out=3)
(3, - Log-3 in=3 out=3)
```

The format of the tuples shown is the following:

> (id, `state` `name` `statistics`)

`Id` is a simple numeric identifier that indexes the transform and associated thread. `State` is either "+" for "alive thread" or "-" for "finished/dead thread". `Name` is the thread name, most often built using the transform name and a thread id. `Statistics` is the number of lines that got read or written to input / output on this transform.

- **Manage execution**. Once configured, your ETL process will be runnable by calling the job instance.

```
>>> # Call the job == run the ETL process
>>> job()
```

3.2 API

CHAPTER 4

Transformations

Transformations are the basic bricks to build ETL processes. Basically, it gets lines from its `input` and sends transformed lines to its `output`.

You're highly encouraged to use the `rdc.etl.transform.Transform` class as a base for your custom transforms, as it defines the whole *I/O logic*. All transformations provided by the package are subclasses of `rdc.etl.transform.Transform`.

Builtin transformations reference

4.1 Extracts

4.1.1 Extract (base class and decorator)

4.1.2 DatabaseExtract

4.1.3 FileExtract

4.2 Loads

The code there is lacking quality and completion, even if it works.

4.2.1 DatabaseLoad

4.3 Maps

4.3.1 Map (base class and decorator)

4.3.2 CsvMap

4.3.3 XmlMap

4.4 Filters

Filters remove some lines from the flux.

4.5 Joins

Inner or outer join on data (similar to database joins/products)

Not to be mistaken for flow-based joins that work on I/O channels.

TODO

4.6 Utilities

Helper and utility transformations.

4.6.1 Log

4.6.2 Stop

4.6.3 Override

4.6.4 Clean

4.6.5 SimpleTransform

4.7 Flow-related

Flow related transformations are there to build jobs that will split data from one channel into more than one or the opposite, taking more than one input channel and “joining” data into one output channel.

TODO

Design notes

4.8 Input / output design

4.8.1 Basics

All you have to know as an ETL user, is that each transform may have 0..n input channels and 0..n output channels. Mostly because it was fun, we named the channel with representative *nix-file-descriptor-like names, but the similarity ends to the name.

The `input multiplexer` will group together whatever comes to one of the inputs channels and pass it to the transformation’s `transform()` method.

The transform method should be a generator, yielding output lines (with an optional output channel id):

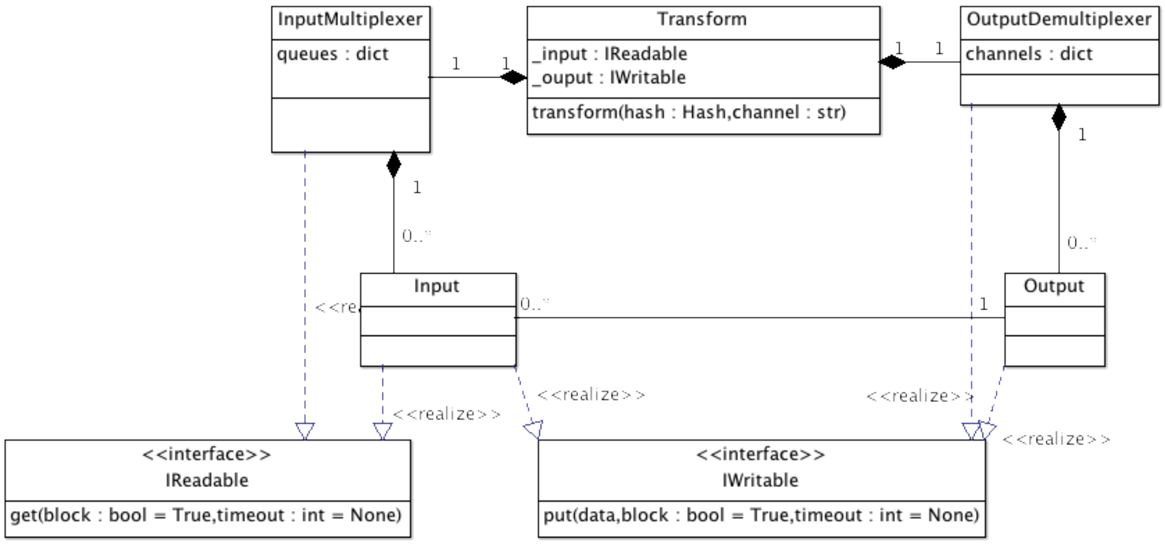
```
def transform(hash, channel=STDIN):
    yield hash.copy({'foo': 'bar'})
    yield hash.copy({'foo': 'baz'})
```

4.8.2 Input and output

All transforms are expected to have the following attributes:

- `_input`, which should implement `IReadable`
- `_output`, which should implement `IWritable`

When you're using `rdc.etl.transform.Transform`, the base class will create them for you as an `InputMultiplexer` and an `OutputDemultiplexer`, each one having a list of channels populated after reading the `INPUT_CHANNELS` and `OUTPUT_CHANNELS` transformation attributes. By default, transformations have one default `STDIN` input, one default `STDOUT` output and one alternate `STDERR` output. You can virtually have infinite input or outputs in your transformations (as though I have hard time imagining a use).



4.8.3 Example

Here is a simple transform that takes whatever comes to STDIN and put it on STDOUT and STDOUT2, and that puts everything that comes to STDIN2 and send it to STDERR.

```python
from rdc.etl.transform import Transform
from rdc.etl.io import STDIN, STDIN2, STDOUT, STDOUT2, STDERR

class MyTransform(Transform):
    INPUT_CHANNELS = (STDIN, STDIN2, )
    OUTPUT_CHANNELS = (STDOUT, STDOUT2, STDERR, )

    def transform(self, hash, channel=STDIN):
        if channel == STDIN:
            yield hash
            yield hash, STDOUT2
        elif channel == STDIN2:
            yield hash, STDERR
```

CHAPTER 5

Filesystem

Not really implemented, would like some abstraction for this.

You can use FileExtract to read a file into a field.

```
t = FileExtract('/tmp/filename', output_field='_content')
job.add_chain(t)
```

If you don't need to keep a lot of different things, you can use the default output_field (subject, context) that is _. It can be handy as transforms that only act on one field will read this one by default.

```
t1 = FileExtract('/tmp/file.csv')
t2 = CsvMap()
job.add_chain(t1, t2)
```

CHAPTER 6

Database

Not really implemented, would like some abstraction for this.

For now, use sqlalchemy engines.

CHAPTER 7

Statuses

Statuses are the tools to observe a process execution state. Not documented yet, but try the following before you run the job:

```
>>> from rdc.etl.status.console import ConsoleStatus
>>> job.status.append(ConsoleStatus())
```

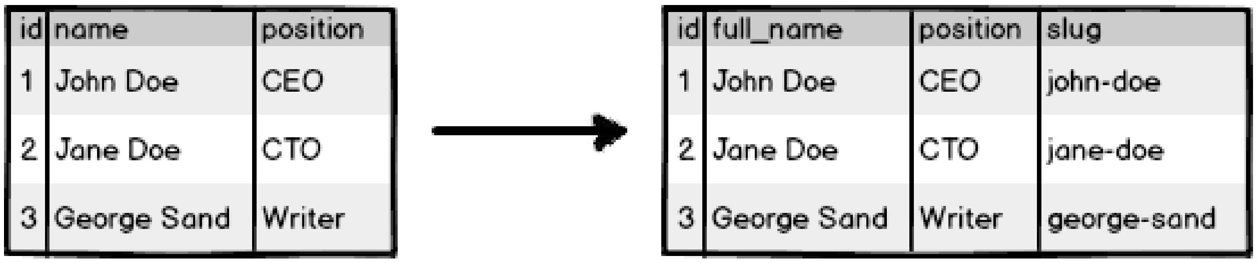
7.1 ConsoleStatus

CHAPTER 8

Cookbook

8.1 Recipe: Simple data processing

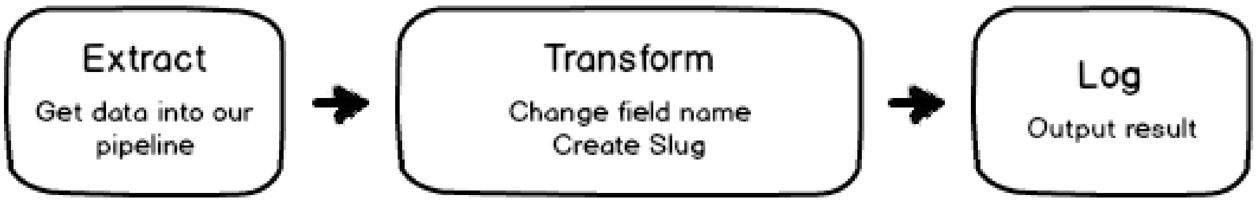
8.1.1 What we want to achieve



id	name	position
1	John Doe	CEO
2	Jane Doe	CTO
3	George Sand	Writer

id	full_name	position	slug
1	John Doe	CEO	john-doe
2	Jane Doe	CTO	jane-doe
3	George Sand	Writer	george-sand

8.1.2 Pipeline structure



8.1.3 Code

```
# -*- coding: utf-8 -*-

from rdc.common.util.text import slughifi
from rdc.etl.extra.util import TransformBuilder
from rdc.etl.hash import Hash
from rdc.etl.job import Job
from rdc.etl.transform.extract import Extract as _Extract
from rdc.etl.transform import Transform as _Transform
from rdc.etl.transform.util import Log


# Create our data extractor. Here, we use a simple generator to create it.
@TransformBuilder(_Extract)
def Extract():
    yield Hash((
        ('id', 1, ),
        ('name', 'John Doe', ),
        ('position', 'CEO', ),
    ))
    yield Hash((
        ('id', 2, ),
        ('name', 'Jane Doe', ),
        ('position', 'CTO', ),
    ))
    yield Hash((
        ('id', 3, ),
        ('name', 'George Sand', ),
        ('position', 'Writer', ),
    ))


# Transform our data
#
# A Transform created using a decorator is built from a function taking a hash and a channel id, we
# channel id here.
@TransformBuilder(_Transform)
def Transform(h, c):
    # Create slug applying a field transformation
    h['slug'] = slughifi(h['name'])

    # Rename 'name' field and call it 'full_name
    h.rename('name', 'full_name')

    # Send our modified hash to the default output channel/pipeline
    yield h


# Create the job
job = Job()
job.add_chain(Extract(), Transform(), Log())

# Run it
if __name__ == '__main__':
    job()
```

8.1.4 Output

```
$ python example/cookbook/01_simple.py

····{1}·····················································
  id:int → «1»
  position:str → «CEO»
  slug:str → «john-doe»
  full_name:str → «John Doe»
············································································

····{2}·····················································
  id:int → «2»
  position:str → «CTO»
  slug:str → «jane-doe»
  full_name:str → «Jane Doe»
············································································
```

```
····{3}·················································
  id:int → «3»
  position:str → «Writer»
  slug:str → «george-sand»
  full_name:str → «George Sand»
························································
```

8.1.5 Pitfalls

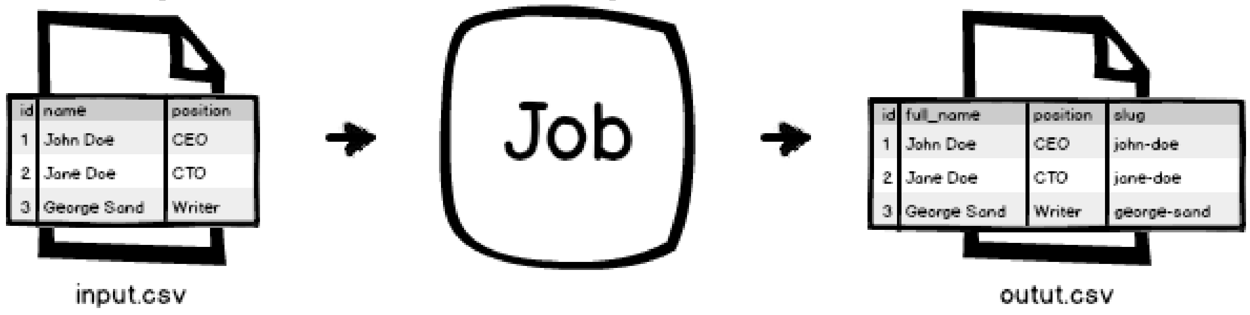
This job is pretty useless, because it reads hardcoded values and write the result to your current terminal. You may want to read:

- *Recipe: Read and write from/to CSV files*

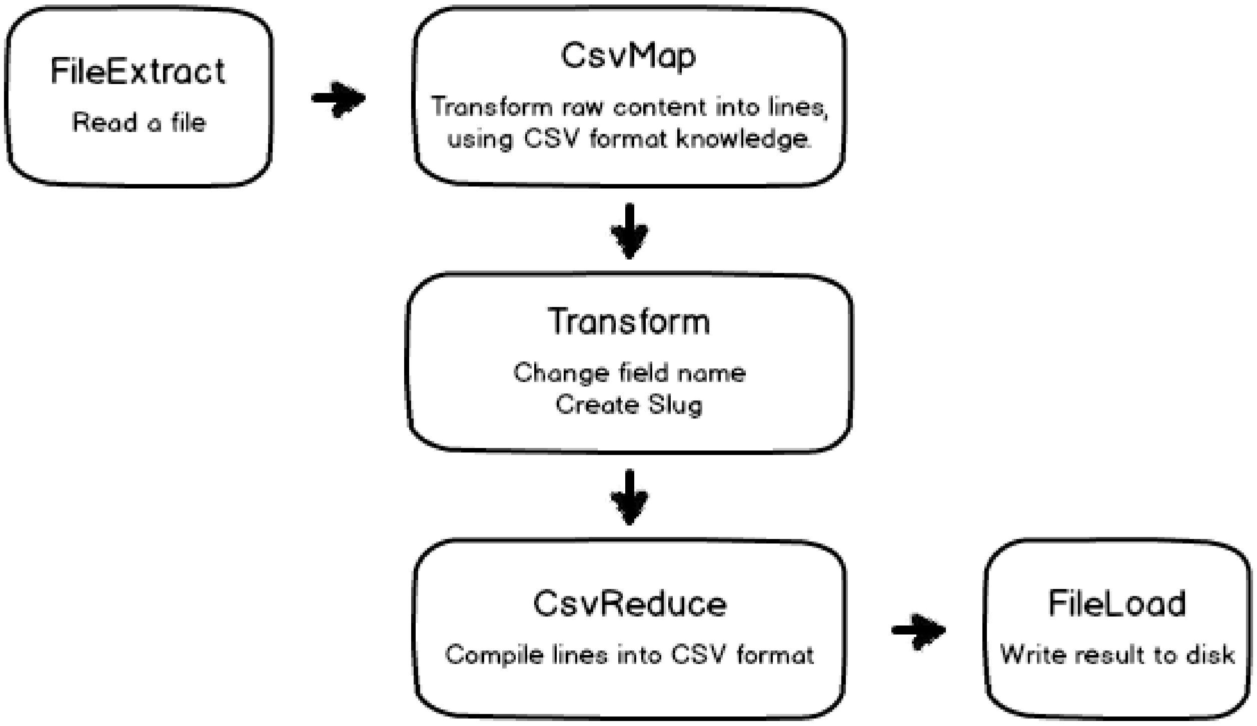
8.2 Recipe: Read and write from/to CSV files

8.2.1 What we want to achieve

We want to write the exact transformation that we wrote in *Recipe: Simple data processing*, except that we will read data from an input CSV file, and write the result to an output CSV file.



8.2.2 Pipeline structure



CHAPTER 9

Contributing

The code is available on github.

```
$ git clone https://github.com/rdcli/etl.git
```

The way to contribute is to fork the project in your own github account, and then make pull requests. If you don't want to use github, you can send pull requests by mail (`git format-patch` is your friend) to romain(at)rdc(dot)li.

It's probably a good idea to discuss ideas before starting to implement.

You're also *(more than)* very welcome to improve the documentation, or the unit tests.

The project roadmap is available below.

This package is used on live systems, and no backward incompatible feature will be implemented in 1.x after 1.0.0 has been released (at least, we'll try). See Semantic Versionning.

9.1 Roadmap

9.1.1 General

- Documentation, more documentation, better documentation
- Test coverage
- Examples
- "Job" tests

9.1.2 Milestone 1.0

IO channels management

- *(DONE)* Multiple input/output possible for each transformation, with default channels
- *(DONE)* "Converging stars" (V model), "diverging stars" (reverse V) and diamond should be possible
- See how we deal with cycles, I guess a "health check" pass is necessary to ensure that all paths have an end.

Error handling

- Exceptions are sent to stdout, destroying statuses
- There should be recoverable and fatal errors
- stderr should be a special output stream that handle exceptions, and all stdouts should be plugged into some handler.
- errors should appear in status
- React to Control-C (KeyboardInterrupt)

9.1.3 Milestone 1.1

Services/Connections/...

- what is a good name for this ?
- databases, webservices, filesystems, http, ...
- stats (r/w)

Display/status

- Better Log() (nice tables wanted)
- wsgi status ? (html) mail status ?
- Catchall for unplugged IO channels ? For example, all messages going to unplugged STDERR channels could be sent to a given transform, so we can act (email ...)

9.1.4 Milestone 1.2

- Whatever will be needed at this time, let's focus on first versions for now (ideas welcome).

9.1.5 Ideas

- *"daemon" jobs*. Live forever, whenever something triggers an input, it runs through the transformations. Use cases: live index update, PUT/POST webservice.

CHAPTER 10

Indices and tables

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