rdc-etl Documentation

Release 1.0.0a6

Romain Dorgueil

Apr 16, 2017

Contents

Extract Transform Load (ETL) toolkit for python.

Looking for an ETL for python 3.5+? Discover Bonobo.

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

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__
```

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.

CHAPTER 2

Kickstart

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

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
```

Overview of concepts

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.

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.

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.

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

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 method ``add()``
>>> # 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()
```

API

class `rdc.etl.harness.base.`**`IHarness`**

ETL harness interface.

The harness is basically the executable stuff that will actually run a job.

class `rdc.etl.job.`**`Job`**`(`*debug=False*, *profile=False*`)`

`add_chain``(`**transforms*, ***kwargs*`)`

Main helper method to add chains of transforms to this harness. You can plug the whole chain from and to other transforms by specifying *input* and *output* parameters.

The transforms provided should not be bound yet.

```
>>> h = ThreadedHarness()
>>> t1, t2, t3 = Transform(), Transform(), Transform()
>>> h.add_chain(t1, t2, t3)
<rdc.etl.harness.threaded.ThreadedHarness object at 0x...>
```

`get_threads``()`

Returns attached threads.

`get_transforms``()`

Returns attached transorms.

`__call__``()`

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`.

class `rdc.etl.transform.`**`Transform`**`(`*transform=None*, *input_channels=None*, *output_channels=None*`)`

Base class and decorator for transformations.

`transform``(`*hash*, *channel=0*`)`

Core transformation method that will be called for each input data row.

`INPUT_CHANNELS`

List of input channel names.

`OUTPUT_CHANNELS`

List of output channel names

Example:

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

>>> print list(my_transform(
...         H(('foo', 'bar'), ('bar', 'alpha')),
...         H(('foo', 'baz'), ('bar', 'omega')),
...     ))
[H{'foo': 'BAR', 'bar': 'alpha'}, H{'foo': 'BAZ', 'bar': 'omega'}]
```

Builtin transformations reference

Extracts

Extracts are transformations that generate output lines from something that is not one of the input channel. As it will yield all data for each input row, the input given is usually only one empty line.

Extract (base class and decorator)

class `rdc.etl.transform.extract.`**`Extract`**(*extract=None*)

Base class for extract transforms.

`extract`

Generator, iterable or iterable-typed callable that is used as the data source. Often used as a shortcut to make fast prototypes of ETL processes from a dictionary, before going further with real data sources.

Each iterator value should be something Hash.copy() can take as an argument.

Example using a dict:

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

>>> data = ({'foo': 'bar'}, {'foo': 'baz'}, )
>>> my_extract = Extract(extract=data)

>>> list(my_extract({}))
[H{'foo': 'bar'}, H{'foo': 'baz'}]
```

Example using a callable:

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

>>> @Extract
... def my_extract():
...     return (
...         {'bar': 'baz'},
...         {'bar': 'boo'},
...     )

>>> list(my_extract({}))
[H{'bar': 'baz'}, H{'bar': 'boo'}]
```

Example using a generator:

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

>>> @Extract
... def my_extract():
...     yield {'bar': 'baz'}
...     yield {'bar': 'boo'}

>>> print list(my_extract({}))
[H{'bar': 'baz'}, H{'bar': 'boo'}]
```

Note: Whenever you can, prefer the generator approach so you're not blocking anything while computing remaining elements.

DatabaseExtract

class `rdc.etl.extra.db.extract.`**`DatabaseExtract`**(*engine*, *query=None*, *limit=None*)

Extract data from a database using some raw SQL and yield one output line per query result.

`engine`
The sqlalchemy engine to use for extraction.

`query`
The database query that will be used to extract data from database. Should not contain OFFSET/LIMIT, nor ";".

`pack_size`
The number of records to retrieve at a time (will be used to add OFFSET/LIMIT clauses to SQL).

FileExtract

class `rdc.etl.transform.extract.file.`**`FileExtract`**(*uri=None*, *output_field=None*)

Extract data from a file into a field.

`uri`
The path for source file. Can be either an absolute/relative filesystem path or an HTTP/HTTPS resource.

`output_field`
The field that will contain file content. Use the topic (_) field by default.

Loads

Load transformations are the opposite of extracts. It take data from input and loads it into an external "thing" (database, filesystem, webservice, ...).

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

DatabaseLoad

class `rdc.etl.extra.db.load.`**`DatabaseLoad`**(*engine=None*, *table_name=None*, *fetch_columns=None*, *discriminant=None*, *created_at_field=None*, *updated_at_field=None*, *insert_only_fields=None*, *allowed_operations=None*)

TODO doc this !!! test this !!!!

Maps

Maps are transforms that will yield rows depending on the value of one input field. In association with `FileExtract` for example, it can parse the file content format and yield rows that have an added knowledge.

By default, maps use the topic (_) field for input

Map (base class and decorator)

class `rdc.etl.transform.map.`**`Map`**(*map=None*, *field=None*)

Base class for mappers.

`map`

Map logic callable. Takes the hash's field value and yields iterable data.

`field`

The input field.

Example:

```
>>> from rdc.etl.transform.map import Map
>>> from rdc.etl.transform.util import clean

>>> @Map
... def my_map(s_in):
...     for l in s_in.split('\n'):
...         yield {'f%d' % i: v for i, v in enumerate(l.split(':'))}

>>> map(clean, my_map({'_': 'a:b:c\nb:c:d\nc:d:e'}))
[H{'f0': 'a', 'f1': 'b', 'f2': 'c'}, H{'f0': 'b', 'f1': 'c', 'f2': 'd'}, H{'f0':
↪'c', 'f1': 'd', 'f2': 'e'}]
```

CsvMap

class `rdc.etl.transform.map.csv.`**`CsvMap`**(*field=None*, *delimiter=None*, *quotechar=None*, *headers=None*, *skip=None*)

Reads a CSV and yield the values, line-by-line.

`delimiter`

The CSV delimiter.

`quotechar`

The CSV quote character.

`headers`

The list of column names, if the CSV does not contain it as its first line.

`skip`

The amount of lines to skip before it actually yield output.

XmlMap

class `rdc.etl.transform.map.xml.`**`XmlMap`**(*map_item=None*, *xpath=None*, *field=None*)

Reads a XML and yield values for each root children.

> **Warning:** This does not work, don't use (or fix before :p).

Definitions:

XML Item: In the context of an XmlMap, we define an XML Item as being either a children of the XML root if no xpath has been provided, or one item returned by the XPath provided.

map_item
Will be called for each input XML Item, and should return a dictionary of values for this item.

field
The input field (defined in parent).

xpath
XPath used to select items before running them through item_map().

Filters

Filters remove some lines from the flux.

class `rdc.etl.transform.filter.`**`Filter`**(*filter=None*)
Filter out hashes from the stream depending on the `filter` callable return value, when called with the current hash as parameter.

Can be used as a decorator on a filter callable.

filter
A callable used to filter the hashes. If return value is True for a given hash, then the hash will be yield to output. Otherwise, it will be burnt.

Example:

```
>>> from rdc.etl.transform.filter import Filter
>>> from rdc.etl.hash import Hash

>>> @Filter
... def my_filter(hash, channel):
...     return hash['keepme'] == True

>>> list(my_filter(
...         (('foo', 'bar'), ('keepme', True), ),
...         (('foo', 'baz'), ('keepme', False), ),
...     ))
[H{'foo': 'bar', 'keepme': True}]
```

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

class `rdc.etl.transform.join.`**`Join`**(*join=None, is_outer=False, default_outer_join_data=None*)
Join some key => value pairs, that can depend on the source hash.

This element can change the stream length, either positively (joining >1 item data) or negatively (joining <1 item data)

join(*hash, channel=0*)
Abtract method that must be implemented in concrete subclasses, to return the data that should be joined with the given row.

It should be iterable, or equivalent to False in a test.

If the result is iterable and its length is superior to 0, the result of this transform will be a cartesian product between this method result and the original input row.

If the result is false or iterable but 0-length, the result of this transform will depend on the join type, determined by the is_outer attribute.

- If is_outer == True, the transform output will be a simple union between the input row and the result of self.get_default_outer_join_data()
- If is_outer == False, this row will be sinked, and will not generate any output from this transform.

Default join type is inner, to preserve backward compatibility.

Example:

```
>>> from rdc.etl.transform.join import Join
>>> from rdc.etl.transform.util import clean

>>> @Join
... def my_join(hash, channel=STDIN):
...     return ({'a':1}, {'b':2}, )

>>> map(clean, my_join({'foo': 'bar'}, {'foo': 'baz'}, ))
[H{'foo': 'bar', 'a': 1}, H{'foo': 'bar', 'b': 2}, H{'foo': 'baz', 'a': 1}, H{'foo
↪': 'baz', 'b': 2}]
```

Utilities

Helper and utility transformations.

Log

class `rdc.etl.transform.util.`**`Log`**(*field_filter=None*, *condition=None*, *clean=None*)

Identity transform that adds a console output side effect, to watch what is going through Queues at some point of an ETL process.

Stop

class `rdc.etl.transform.util.`**`Stop`**(*transform=None*, *input_channels=None*, *output_channels=None*)

Sinker transform that stops anything through the pipes.

Override

class `rdc.etl.transform.util.`**`Override`**(*override_data=None*)

Simple transform that will overwrite some values with constant values provided in a Hash.

Clean

class `rdc.etl.transform.util.`**`Clean`**(*transform=None*, *input_channels=None*, *output_channels=None*)

Remove all fields with keys starting by _

SimpleTransform

class `rdc.etl.extra.simple.`**`SimpleTransform`**`(`**filters*`)`

SimpleTransform is an attempt to make a trivial transformation easy to build, using fluid APIs and a lot of easy shortcuts to apply filters to some fields.

The API is not stable and this will probably go into an "extra" module later.

Example:

```
>>> t = SimpleTransform()
```

Apply "upper" method on "name" field, and store it back in "name" field.

```
>>> t.add('name').filter('upper')
<rdc.etl.extra.simple._SimpleItemTransformationDescriptor object at ...>
```

Apply the lambda to "description" field content, and store it into the "full_description" field.

```
>>> t.add('full_description', 'description').filter(lambda v: 'Description: ' + v)
<rdc.etl.extra.simple._SimpleItemTransformationDescriptor object at ...>
```

Remove the previously defined "useless" descriptor. This does not remove the "useless" fields into transformed hashes, it is only usefull to override some parent stuff.

```
>>> t.useless = 'foo'
>>> t.delete('useless')
```

Mark the "notanymore" field for deletion upon transform. Output hashes will not anymore contain this field./

```
>>> t.remove('notanymore')
```

Add a field (output hashes will contain this field, all with the same "foo bar" value).

```
>>> t.test_field = 'foo bar'
```

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

Input / output design

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.

class rdc.etl.transform.**ITransform**

transform(*hash*[, *channel=STDIN*])

All input rows that comes to one of this transform's input channels will be passed to this method. If you only have one input channel, you can safely ignore the channel value, although you'll need it in method prototype.

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'})
```

Input and output

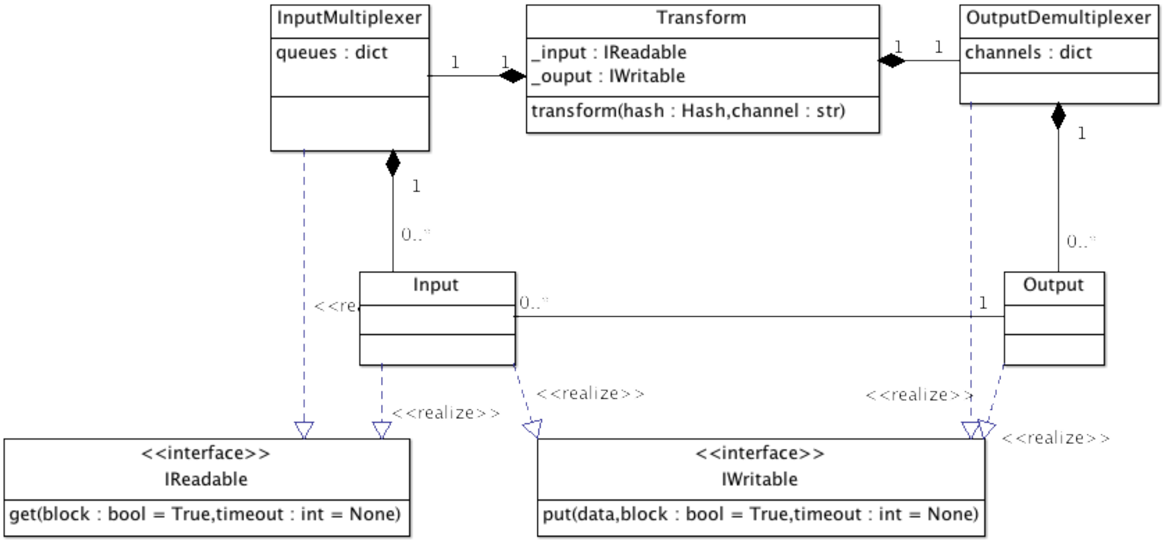
All transforms are expected to have the following attributes:

- _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).

class rdc.etl.io.**InputMultiplexer**(*channels*)

class rdc.etl.io.**OutputDemultiplexer**(*channels*)



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

Database extracts, loads and joins are implemented in the rdc.etl.extra.db package. It's considered as an "addon", because no work has been made yet on "connection management" in the core package.

You need sqlalchemy, below is an example.

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

import datetime
import sqlalchemy

from rdc.etl.extra.db import DatabaseExtract, DatabaseLoad
from rdc.etl.extra.util import TransformBuilder
from rdc.etl.job import Job
from rdc.etl.status.console import ConsoleStatus
from rdc.etl.transform import Transform


DB_CONFIG = {
    'user': 'root',
    'pass': '',
    'name': 'my_database',
    'host': 'localhost',
}
TABLE_NAME = 'products'


# Create SQLAlchemy engine
db_engine = sqlalchemy.create_engine('mysql://{user}:{pass}@{host}/{name}'.
↪format(**DB_CONFIG))


# Extract : use a SQL query
t1 = DatabaseExtract(
    db_engine,
    '''
```

```
        SELECT *
        FROM {table_name} t
        WHERE MOD(t.id, 100) > 98
    '''.format(table_name=TABLE_NAME)
)


# Transform : Update a timestamp
@TransformBuilder(Transform)
def UpdateChangeTimestamp(hash, channel):
    hash['updated_at'] = datetime.datetime.now()
    yield hash


t2 = UpdateChangeTimestamp()


# Load : same table as input (by choice)
t3 = DatabaseLoad(
    db_engine,
    TABLE_NAME,
    discriminant=('id', ),  # This is default behavior, but the selection criteria can be based on any field
    # combination as long as a select on those keys returns only ONE result line.
    updated_at_field=None,  # Avoid default updated_at behavior as we reimplemented it manually.
)

# Job creation
job = Job(profile=True)
job.add_chain(t1, t2, t3)
job.status.append(ConsoleStatus())

if __name__ == '__main__':
    job()
```

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())
```

ConsoleStatus

class `rdc.etl.status.console.`**`ConsoleStatus`**(*prefix=’‘*)

Outputs status information to the connected stdout. Can be a TTY, with or without support for colors/cursor movements, or a non tty (pipe, file, ...). The features are adapted to terminal capabilities.

`prefix`

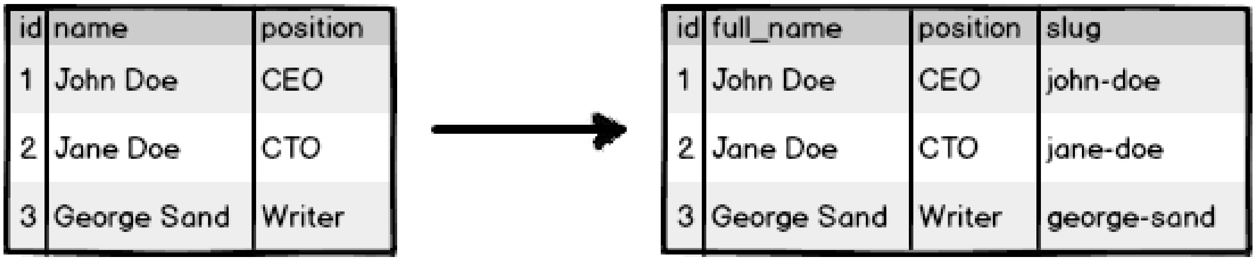
String prefix of output lines.

CHAPTER 8

Cookbook

Recipe: Simple data processing

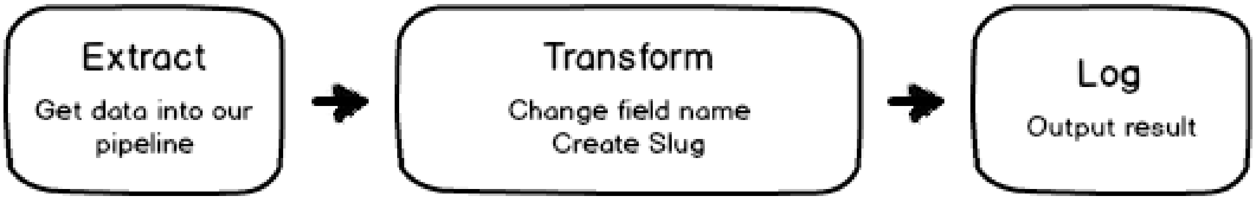
What we want to achieve



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

Pipeline structure



Code

```python
# -*- 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 will ignore
# 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()
```

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»
............................................................
```

Pitfalls

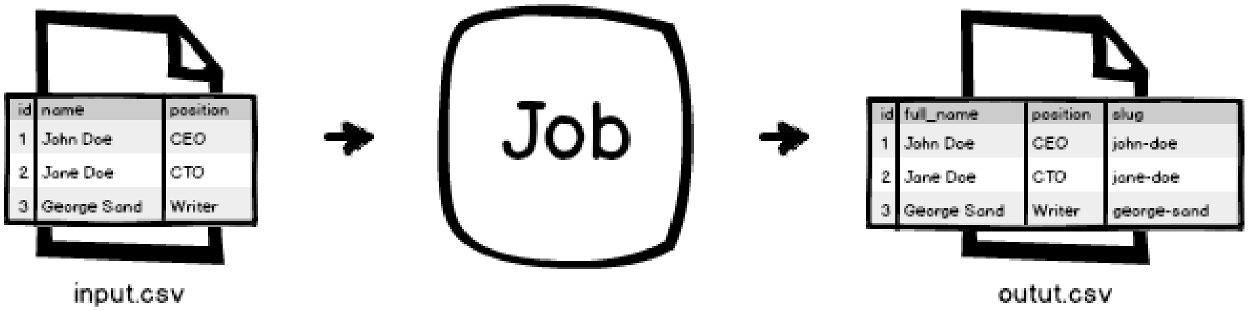
This job is pretty useless, because it reads hardcoded values and write the result to your current terminal. You may want to read:

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

Recipe: Read and write from/to CSV files

What we want to achieve

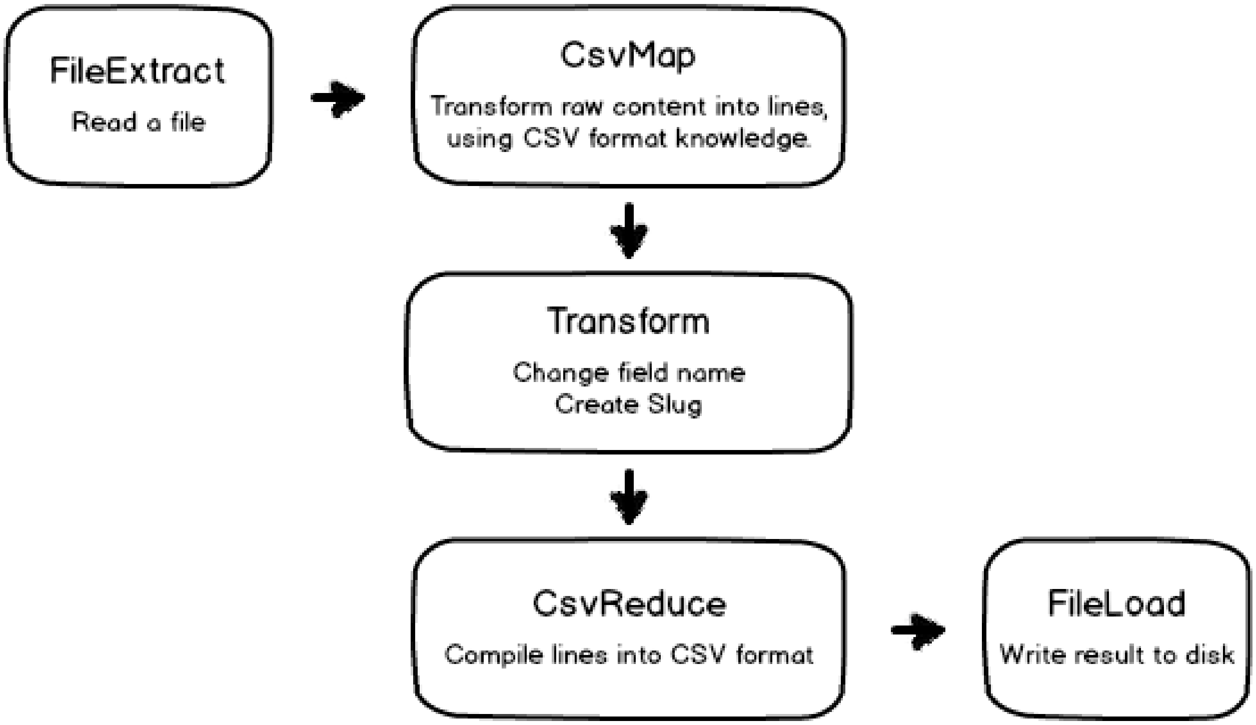
We want to write the exact transformation that we wrote in *Recipe: Simple data processing*, except that we will read data from an input CSV file, and write the result to an output CSV file.



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

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.

Roadmap

General

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

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)

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 ...)

Milestone 1.2

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

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

Python Module Index

Index