Transmogrifier Documentation

Release 1.5

Martijn Pieters

Nov 05, 2017

Contents

CHAPTER 1

Transmogrifier

Transmogrifier provides support for building pipelines that turn one thing into another. Specifically, transmogrifier pipelines are used to convert and import legacy content into a Plone site. It provides the tools to construct pipelines from multiple sections, where each section processes the data flowing through the pipe.

A “transmogrifier pipeline” refers to a description of a set of pipe sections, slotted together in a set order. The stated goal is for these sections to transform data and ultimately add content to a Plone site based on this data. Sections deal with tasks such as sourcing the data (from textfiles, databases, etc.) and characterset conversion, through to determining portal type, location and workflow state.

Note that a transmogrifier pipeline can be used to process any number of things, and is not specific to Plone content import. However, it’s original intent is to provide a pluggable way to import legacy content.

CHAPTER 2

Credits

Development sponsored by Elkjøp Nordic AS

Design and development Martijn Pieters at Jarn

Project name A transmogrifier is fictional device used for transforming one object into another object. The term was coined by Bill Waterson of Calvin and Hobbes fame.

2.1 Content

2.1.1 Pipelines

To transmogrify, or import and convert non-content, you simply define a pipeline. Pipe sections, the equivalent of parts in a buildout, are slotted together into a processing pipe. To slot sections together, you define a configuration file, define named sections, and a main pipeline definition that names the sections in order (one section per line):

```
>>> exampleconfig = """
... [transmogrifier]
... pipeline =
...     section 1
...     section 2
...     section 3
...
... [section 1]
... blueprint = transmogrifier.tests.examplesource
... size = 5
...
... [section 2]
... blueprint = transmogrifier.tests.exampletransform
...
... [section 3]
... blueprint = transmogrifier.tests.exampleconstructor
... """
```

As you can see this is also very similar to how you construct WSGI pipelines using paster. The format of the configuration files is defined by the Python ConfigParser module, with extensions that we'll describe later. At minimum, at least the transmogrifier section with an empty pipeline is required:

```
>>> mimimalconfig = """
... [transmogrifier]
... pipeline =
... """
```

Transmogrifier can load these configuration files either by looking them up in a registry or by loading them from a python package.

You register transmogrifier configurations using the `blueprint`` and ``pipeline` directives in the http://namespaces.plone.org/transmogrifier namespace. `blueprint` together with a name and a component path; `pipeline` with a name, and optionally a title and description.

```
<configure
    xmlns="http://namespaces.zope.org/zope"
    xmlns:transmogrifier="http://namespaces.plone.org/transmogrifier"
    i18n_domain="transmogrifier">

<transmogrifier:blueprint
    name="transmogrifier.logger"
    component="transmogrifier.blueprints.logger.Logger"
    />

<transmogrifier:pipeline
    name="exampleconfig"
    title="Example pipeline configuration"
    description="This is an example pipeline configuration"
    configuration="example.cfg"
    />

</configure>
```

You can then tell transmogrifier to load the 'exampleconfig' configuration. To load configuration files directly from a python package, name the package and the configuration file separated by a colon, such as 'transmogrifier.tests:exampleconfig.cfg'.

Registering files with the transmogrifier registry allows other uses, such as listing available configurations in a user interface, together with the registered description. Loading files directly let's you build reusable libraries of configuration files more quickly though.

In this document we'll use the shorthand *pipeline* to register example configurations:

```
>>> registerConfiguration('transmogrifier.tests.exampleconfig',
...                       exampleconfig)
```

Pipeline sections

Each section in the pipeline is created by a blueprint. Blueprints are looked up as named utilities implementing the ISectionBlueprint interface. In the transmogrifier configuration file, you refer to blueprints by the name under which they are registered. Blueprints are factories; when called they produce an ISection pipe section. ISections in turn, are iterators implementing the iterator protocol.

Here is a simple blueprint, in the form of a class definition:

```
>>> from transmogrifier.interfaces import ISectionBlueprint
>>> from transmogrifier.blueprints import Blueprint
>>> from zope.interface import implementer
>>> from zope.interface import provider
...
>>> @provider(ISectionBlueprint)
... class ExampleTransform(Blueprint):
...     def __iter__(self):
...         for item in self.previous:
...             item['exampletransformname'] = str(self.name)
...             yield item
...
>>> from zope.component import provideUtility
>>> provideUtility(ExampleTransform,
...                name='transmogrifier.tests.exampletransform')
```

Note that we register this class as a named utility, and that instances of this class can be used as an iterator. When slotted together, items 'flow' through the pipeline by iterating over the last section, which in turn iterates over it's preceding section (`self.previous` in the example), and so on.

By iterating over the source, then yielding the items again, each section passes items on to the next section. During the iteration loop, sections can manipulate the items. Note that items are python dictionaries; sections simply operate on the keys they care about. In our example we add a new key, `exampletransformname`, which we set to the name of the section.

Sources

The items that flow through the pipe have to originate from somewhere though. This is where special sections, sources, come in. A source is simply a pipe section that inserts extra items into the pipeline. This is best illustrated with another example:

```
>>> @provider(ISectionBlueprint)
... class ExampleSource(Blueprint):
...     def __iter__(self):
...         size = int(self.options['size'])
...         for item in self.previous:
...             yield item
...
...         for i in range(size):
...             yield dict(id='item%02d' % i)
...
>>> provideUtility(ExampleSource,
...                name='transmogrifier.tests.examplesource')
```

In this example we use the `options` dictionary to read options from the section configuration, which in the example configuration we gave earlier has the option `size` defined as 5. Note that the configuration values are always strings, so we need to convert the size option to an integer here.

The source first iterates over the previous section and yields all items unchanged. Only when that loop is done, does the source produce new items and puts those into the pipeline. This order is important: when you slot multiple source sections together, you want items produced by earlier sections to be processed first too.

There is always a previous section, even for the first section defined in the pipeline. Transmogrifier passes in a empty iterator when it instantiates this first section, expecting such a first section to be a source that'll produce items for the pipeline to process.

Constructors

As stated before, transmogrifier is intended for importing content into a Plone site. However, transmogrifier itself only drives the pipeline, inserting an empty iterator and discarding whatever it pulls out of the last section.

In order to create content then, a constructor section is required. Like source sections, you should be able to use multiple constructors, so constructors should always start with yielding the items passed in from the previous section on to a possible next section.

So, a constructor section is an ISection that consumes items from the previous section, and affects the site based on items, usually by creating content objects based on these items, then yield the item for a next section. For example purposes, we simply pretty print the items instead:

```
>>> import pprint
...
>>> @provider(ISectionBlueprint)
... class ExampleConstructor(Blueprint):
...     def __iter__(self):
...         for item in self.previous:
...             pprint.pprint(sorted(item.items()))
...             yield item
...
>>> provideUtility(ExampleConstructor,
...                name='transmogrifier.tests.exampleconstructor')
```

With this last section blueprint example completed, we can load the example configuration we created earlier, and run our transmogrification:

```
>>> from transmogrifier import Transmogrifier
...
>>> transmogrifier = Transmogrifier({})
>>> transmogrifier('transmogrifier.tests.exampleconfig')
[('exampletransformname', 'section 2'), ('id', 'item00')]
[('exampletransformname', 'section 2'), ('id', 'item01')]
[('exampletransformname', 'section 2'), ('id', 'item02')]
[('exampletransformname', 'section 2'), ('id', 'item03')]
[('exampletransformname', 'section 2'), ('id', 'item04')]
```

Developing blueprints

As we could see from the ISectionBlueprint examples above, a blueprint gets called with several arguments: `transmogrifier`, `name`, `options` and `previous`.

We discussed `previous` before, it is a reference to the previous pipe section and must be looped over when the section itself is iterated. The `name` argument is simply the name of the section as given in the configuration file.

The `transmogrifier` argument is a reference to the transmogrifier itself, and it can be used to reach the context we are importing to through it's `context` attribute. The transmogrifier also acts as a dictionary, mapping from section names to a mapping of the options in each section.

Finally, as seen before, the `options` argument is a mapping of the current section options. It is the same mapping as can be had through `transmogrifier[name]`.

A short example shows each of these arguments in action:

```
>>> @provider(ISectionBlueprint)
... class TitleExampleSection(Blueprint):
```

```
...     def __iter__(self):
...         pipeline = self.transmogrifier['transmogrifier']['pipeline']
...         pipeline_size = len([s.strip() for s in pipeline.split('\n')
...                              if s.strip()])
...
...         size = self.options['pipeline-size'] = str(pipeline_size)
...         site_title = transmogrifier.context.Title()
...
...         for item in self.previous:
...             item['pipeline-size'] = size
...             item['title'] = '%s - %s' % (site_title, item['id'])
...             yield item
...
>>> provideUtility(TitleExampleSection,
...                name='transmogrifier.tests.titleexample')
...
>>> titlepipeline = """
... [transmogrifier]
... pipeline =
...     section1
...     titlesection
...     section3
...
... [section1]
... blueprint = transmogrifier.tests.examplesource
... size = 5
...
... [titlesection]
... blueprint = transmogrifier.tests.titleexample
...
... [section3]
... blueprint = transmogrifier.tests.exampleconstructor
... """
...
>>> registerConfiguration('transmogrifier.tests.titlepipeline',
...                       titlepipeline)
...
>>> class Site(object):
...     def Title(self):
...         return 'Test Site'
...
>>> site = Site()
>>> site.Title()
'Test Site'
>>> transmogrifier = Transmogrifier(site)
...
>>> transmogrifier('transmogrifier.tests.titlepipeline')
[('id', 'item00'),
 ('pipeline-size', '3'),
 ('title', 'Test Site - item00')]
[('id', 'item01'),
 ('pipeline-size', '3'),
 ('title', 'Test Site - item01')]
[('id', 'item02'),
 ('pipeline-size', '3'),
 ('title', 'Test Site - item02')]
[('id', 'item03'),
 ('pipeline-size', '3'),
 ('title', 'Test Site - item03')]
[('id', 'item04'),
 ('pipeline-size', '3'),
 ('title', 'Test Site - item04')]
```

Configuration file syntax

As mentioned earlier, the configuration files use the format defined by the Python ConfigParser module with extensions. The extensions are based on the zc.buildout extensions and are:

- option names are case sensitive
- option values can use a substitution syntax, described below, to refer to option values in specific sections.
- you can include other configuration files, see *Including other configurations*.

The ConfigParser syntax is very flexible. Section names can contain any characters other than newlines and right square braces ("]"). Option names can contain any characters (within the ASCII character set) other than newlines, colons, and equal signs, can not start with a space, and don't include trailing spaces.

It is a good idea to keep section and option names simple, sticking to alphanumeric characters, hyphens, and periods.

Variable substitution

Transmogrifier supports a string.Template-like syntax for variable substitution, using both the section and the option name joined by a colon:

```
>>> substitutionexample = """
... [transmogrifier]
... pipeline =
...     section1
...     section2
...     section3
...
... [definitions]
... item_count = 3
...
... [section1]
... blueprint = transmogrifier.tests.examplesource
... size = ${definitions:item_count}
...
... [section2]
... blueprint = transmogrifier.tests.exampletransform
...
... [section3]
... blueprint = transmogrifier.tests.exampleconstructor
... """
...
>>> registerConfiguration('transmogrifier.tests.substitutionexample',
...                    substitutionexample)
```

Here we created an extra section called definitions, and refer to the item_count option defined in that section to set the size of the section1 pipeline section, so we only get 3 items when we execute this pipeline:

```
>>> transmogrifier = Transmogrifier({})
>>> transmogrifier('transmogrifier.tests.substitutionexample')
```

```
[('exampletransformname', 'section2'), ('id', 'item00')]
[('exampletransformname', 'section2'), ('id', 'item01')]
[('exampletransformname', 'section2'), ('id', 'item02')]
```

Including other configurations

You can include other transmogrifier configurations with the `include` option in the transmogrifier section. This option takes a list of configuration ids, separated by whitespace. All sections and options from those configuration files will be included provided the options weren't already present. This works recursively; inclusions in the included configuration files are honoured too:

```
>>> inclusionexample = """
... [transmogrifier]
... include =
...     transmogrifier.tests.sources
...     transmogrifier.tests.base
...
... [section1]
... size = 3
... """
...
>>> registerConfiguration('transmogrifier.tests.inclusionexample',
...                   inclusionexample)
...
>>> sources = """
... [section1]
... blueprint = transmogrifier.tests.examplesource
... size = 10
... """
...
>>> registerConfiguration('transmogrifier.tests.sources',
...                   sources)
...
>>> base = """
... [transmogrifier]
... pipeline =
...     section1
...     section2
...     section3
... include = transmogrifier.tests.constructor
...
... [section2]
... blueprint = transmogrifier.tests.exampletransform
... """
...
>>> registerConfiguration('transmogrifier.tests.base',
...                   base)
...
>>> constructor = """
... [section3]
... blueprint = transmogrifier.tests.exampleconstructor
... """
...
>>> registerConfiguration('transmogrifier.tests.constructor',
...                   constructor)
...
```

```
>>> transmogrifier = Transmogrifier({})
>>> transmogrifier('transmogrifier.tests.inclusionexample')
[('exampletransformname', 'section2'), ('id', 'item00')]
[('exampletransformname', 'section2'), ('id', 'item01')]
[('exampletransformname', 'section2'), ('id', 'item02')]
```

Like zc.buildout configurations, we can also add or remove lines from included configuration options, by using the += and -= syntax:

```
>>> advancedinclusionexample = """
... [transmogrifier]
... include =
...     transmogrifier.tests.inclusionexample
... pipeline -=
...     section2
...     section3
... pipeline +=
...     section4
...     section3
...
... [section4]
... blueprint = transmogrifier.tests.titleexample
... """
...
>>> registerConfiguration('transmogrifier.tests.advancedinclusionexample',
...                       advancedinclusionexample)
...
>>> transmogrifier = Transmogrifier(site)
>>> transmogrifier('transmogrifier.tests.advancedinclusionexample')
[('id', 'item00'),
 ('pipeline-size', '3'),
 ('title', 'Test Site - item00')]
[('id', 'item01'),
 ('pipeline-size', '3'),
 ('title', 'Test Site - item01')]
[('id', 'item02'),
 ('pipeline-size', '3'),
 ('title', 'Test Site - item02')]
```

When calling transmogrifier, you can provide your own sections too: any extra keyword is interpreted as a section dictionary. Do make sure you use string values though:

```
>>> transmogrifier('transmogrifier.tests.inclusionexample',
...                section1=dict(size='1'))
[('exampletransformname', 'section2'), ('id', 'item00')]
```

Conventions

At its most basic level, transmogrifier pipelines are just iterators passing 'things' around. Transmogrifier doesn't expect anything more than being able to iterate over the pipeline and doesn't dictate what happens within that pipeline, what defines a 'thing' or what ultimately gets accomplished.

But as has been stated repeatedly, transmogrifier has been developed to facilitate importing legacy content, processing data in incremental steps until a final section constructs new content.

To reach this end, several conventions have been established that help the various pipeline sections work together.

Items are mappings

The first one is that the 'things' passed from section to section are mappings; i.e. they are or behave just like python dictionaries. Again, transmogrifier doesn't produce these by itself, source sections (see *Sources*) produce them by injecting them into the stream.

Keys are fields

Secondly, *all* keys in such mappings that do not start with an underscore will be used by constructor sections (see *Constructors*) to construct Plone content. So keys that do not start with an underscore are expected to map to Archetypes fields or Zope3 schema fields or whatever the constructor expects.

Paths are to the target object

Many sections either create objects (constructors) or operate on already-constructed or pre-existing objecs. Such sections should interpret paths as the complete path for the object. For constructors this means they'll need to split the path into a container path and an id in order for them to find the correct context for constructing the object.

Keys with a leading underscore are controllers

This leaves the keys that do start with a leading underscore to have special meaning to specific sections, allowing earlier pipeline sections to inject 'control statements' for later sections in the item mapping. To avoid name clashes, sections that do expect such controller keys should use prefixes based on the name under which their blueprint was registered, plus optionally the name of the pipe section. This allows for precise targeting of pipe sections when inserting such keys.

We'll illustrate this with an example. Let's say a source section loads news items from a database, but the database tables for such items hold filenames to point to binary image data. Rather than have this section load those filenames directly and add them to the item for image creation, a generic 'file loader' section is used to do this. Let's suppose that this file loader is registered as `acme.transmogrifier.fileloader`. This section then could be instructed to load files and store them in a named key by using 2 'controller' keys named `_acme.transmogrifier.fileloader_filename` and `_acme.transmogrifier.fileloader_targetkey`. If the source section were to create pipeline items with those keys, this later fileloader section would then automatically load the filenames and inject them into the items in the right location.

If you need 2 such loaders, you can target them each individually by including their section names; so to target just the `imageloader1` section you'd use the keys `_acme.transmogrifier.fileloader_imageloader1_filename` and `_acme.transmogrifier.fileloader_imageloader1_targetkey`. Sections that support such targeting should prefer such section specific keys over those only using the blueprint name.

The transmogrifier.utils module has a handy utility method called `defaultKeys` that'll generate these keys for you for easy matching:

```
>>> from transmogrifier import utils
>>> keys = utils.defaultKeys('acme.transmogrifier.fileloader',
...                          'imageloader1', 'filename')
>>> pprint.pprint(keys)
('_acme.transmogrifier.fileloader_imageloader1_filename',
 '_acme.transmogrifier.fileloader_filename',
 '_imageloader1_filename',
 '_filename')
```

```
>>> utils.Matcher(*keys)('_filename', '_imageloader1_filename')
('_imageloader1_filename', True)
```

Keep memory use to a minimum

The above example is a little contrived of course; you'd generally configure a file loader section with a key name to grab the filename from, and perhaps put the loader *after* the constructor section and load the image data straight into the already constructed content item instead. This lowers memory requirements as image data can go directly into the ZODB this way, and the content object can be deactivated after the binary data has been stored.

By operating on one item at a time, a transmogrifier pipeline can handle huge numbers of content without breaking memory limits; individual sections should also avoid using memory unnecessarily.

Previous sections go first

As mentioned in the *Sources* section, when inserting new items into the stream, generally previous pipe sections come first. This way someone constructing a pipeline knows what source section will be processed earlier (those slotted earlier in the pipeline) and can adjust expectations accordingly. This makes content construction more predictable when dealing with multiple sources.

An exception would be a Folder Source, which inserts additional Folder items into the pipeline to ensure that the required container for any given content item exists at construction time. Such a source would inject extra items as needed, not before or after the previous source section.

Iterators have 3 stages

Some tasks have to happen before the pipeline runs, or after all content has been created. In such cases it is handy to realise that iteration within a section consists of three stages: before iteration, iteration itself, and after iteration.

For example, a section creating references may have to wait for all content to be created before it can insert the references. In this case it could build a queue during iteration, and only when the previous pipe section has been exhausted and the last item has been yielded would the section reach into the portal and create all the references.

Sources following the *Previous sections go first* convention basically inject the new items in the after iteration stage.

Here's a piece of pseudo code to illustrate these 3 stages:

```
def __iter__(self):
    # Before iteration
    # You can do initialisation here

    for item in self.previous
        # Iteration itself
        # You could process the items, take notes, inject additional
        # items based on the current item in the pipe or manipulate portal
        # content created by previous items
        yield item

    # After iteration
    # The section still has control here and could inject additional
    # items, manipulate all portal content created by the pipeline,
    # or clean up after itself.
```

You can get quite creative with this. For example, the reference creator could get quite creative and defer creation of references until it knew the referenced object has been created too and periodically create these references. This would keep memory requirements smaller as not *all* references to create have to be remembered.

2.1.2 Indices and tables

- genindex
- modindex
- search