flow Documentation

Release 0.5

Johan Egneblad

August 26, 2016

Contents

Flow is a business layer build upon Python One to use for building daemons, transfer plugins and batch jobs for Viz One.

Contents:

CHAPTER 1

Introduction

Flow is a framework for writing tools, daemons and transfer plugins for Viz One, Vizrt's Media Asset Management software. To communicate with Viz One, the Python SDK python-one is used. Please contact Vizrt for your copy and more information.

The idea behind Flow is to make it simple to integrate with Viz One on an enterprise level. The treats promoted by the framework are:

- Scalability (multi-threading and multi-server)
- Client statelessness
- Simplicity (one application does one thing)

To achieve this, all applications are limited to a certain structure:

```
Source -> Worker
```

1.1 The Source

Let's start with the *Source*. The task of the Source is to gather information in one way or another, do as little with it as possible, and hand it over to the *Worker*. The Source runs in a single thread, which is the reason for keeping the processing within it as simple as possible. Input to Sources can be anything, but usually *Stomp*, *Files* or *Standard Input*.

Examples of built-in sources are:

- Asset Entry Listener `flow.source.asset.AssetEntryListener` *Reacts on creations, updates and deletions of Asset Entries.*
- Unmanaged Files Listener `flow.source.location.UnmanagedFilesListener` *Reacts on creations, updates and deletions of Unmanaged Files.*
- Generic Stomp Listener `flow.source.stomp.GenericStompListener` *Subscribes to a manually given Stomp URL.*
- Interval `flow.source.interval.Interval` *Spawns a worker every nth second.*
- STDIN `flow.source.local.STDIN` *Reads from Standard Input and spawns a worker.*

You can also quite easily create your own Source class.

1.2 The Worker

The Flow process will at start-up create a thread-pool that is used to run worker threads. Workers are essentially classes with a `start` method and a `SOURCE` class specified. Optionally you can also have a `configure` method that will be run once on start-up.

The `start` method will take an object as argument. This is the interface between the *Source* and the *Worker*. The *Source* will call the `start` method with one object as argument as a new thread for each event it produces. If there are no free workers, the event will be held until there are, and then called. There is also a time out for execution time, meaning the waiting time cannot be infinitely long.

CHAPTER 2

Applications

Contents

2.1 Introduction

2.2 Built-in Applications

2.2.1 XML Import

class `xmlimport.`**`XmlImport`**(*instance_name=None*)

XML + Media Importer using the Unmanaged File API.

- If an XML file comes in, it will be read as an Import/Export payload and a Placeholder will be created (or updated if id is given and an asset with that id exists).
- If a non-XML file comes in, it will be considered “media”.
- If this “media” is mentioned by some read Import/Export payload, it will be imported to it’s Placeholder.
- If this “media” is not mentioned yet, it will be remembered (server-side, using the Client Config API).
- If an XML file comes in and mentiones a “media” that is “remembered”, it will be imported to the Placeholder directly.
- If an XML file comes in and references an Asset Entry that is no longer a Placeholder, the Metadata will be updated if changed.
- If a non-XML comes in, and there is an occupied Asset Entry pointing to it, the import will fail.

Example ini file for using this importer:

```
[Flow]
class = xmlimport.XmlImport

[Source]
location = xmlimport
skip empty files = no
```

Note that the “skip empty files” option has the following effect:

- `yes` means Tail mode
- `no` means No tail mode

```
[Xml]
format = default|custom
```

If you choose `default` here, the daemon will do a default Viz One XML Import based on the standard Import/Export format.

On the other hand, if you want a custom XML to be imported you can choose `custom` and it will be parsed and mapped according to the following rules specified by these sections in the INI:

```
[Namespaces]
short = http://long/name
# ...
```

Namespaces are optional. You will need to specify only the once used in the fields you want to parse.

```
[Field:NAME]
xpath = /path/to/value
type = string|integer|date|time|datetime|dictionary
format = formatstring for parsing dates
```

For each field you want to parse, create one of these. For string fields, you only need the `xpath`, since `type` defaults to `string`. The value will be stored under the name NAME for later use with the mapper. Fields of type `dictionary` will require a `source` argument, being an http link to the dictionary feed. Field of type `datetime` support a `default timezone` argument, which should be parsable by python; for instance `Europe/Stockholm` or `GMT`.

```
[Transform]
NAME = EXPR
compound_field = field1 + ':' + field2
```

The `Transform` section allows for simple data transformation. The left-hand side denotes the name to store under and the right-hand side should contain a valid python expression using the names from `Field` directives and previous `Transform` operations. They will be carried out in the order they are written.

```
[Vdf]
form = FORM
asset.title = FIELD1
asset.alternativeTitle = FIELD2
# ...
```

Last is the actual mapping taking place, where you can put the stored data into VDF fields. Remember to specify the form here. The current revision is always used, you should not try to specify a revision.

SOURCE

alias of `UnmanagedFilesListener`

2.2.2 XML Export

class `xmlexport.`**`XmlExportArdFTP`**`(`*instance_name=None*`)`

XML + Media Export Transfer Plugin using ArdFTP SITE commands.

class `xmlexport.`**`XmlExportFXP`**`(`*instance_name=None*`)`

XML + Media Export Transfer Plugin using FXP.

2.2.3 Metadata Mapper

class `metadatamapper.`**`MetadataMapper`**`(`*instance_name=None*`)`

Metadata-mapping daemon that reacts on Asset Entry updates.

Example ini file:

```
[Flow]
class = metadatamapper.MetadataMapper

[Source]
# AssetEntryListener has no configurable parameters

[Mappings]
asset.alternateTitle = metadata.get('asset.title') + ' alternative'
```

`SOURCE`

alias of `AssetEntryListener`

CHAPTER 3

Sources

Contents

3.1 Introduction

3.2 Built-in Sources

3.2.1 Generic Stomp Listener

class `flow.source.stomp.`**`GenericStompListener`**(*stomp_url*)

Source class that listens for events on a Stomp URL.

Options given under [Source]:

```
[Source]
stomp url = <full stomp url>
```

To use this with your flow daemon:

```
from flow import Flow
from flow.source import GenericStompListener

class MyFlow(Flow):
    SOURCE = GenericStompListener

    def start(self, message):
        # message is a vizone.net.message_queue.Message
        pass
```

3.2.2 Asset Entry Listener

class `flow.source.asset.`**`AssetEntryListener`**

Source class that listens for Asset Entry events. Make sure to have the Asset/Admin permission or it won't work.

Options given under [Source]:

```
[Source]
# None
```

To use this with your flow daemon:

```
from flow import Flow
from flow.source import AssetEntryListener

class MyFlow(Flow):
    SOURCE = AssetEntryListener

    def start(self, asset):
        # asset is a vizone.payload.asset.Item
        pass
```

3.2.3 Unmanaged Files Listener

class `flow.source.location.`**`UnmanagedFilesListener`**(*location=None*, *skip_empty_files='no'*)

Source class that listens for UnmanagedFile events for a given location.

Options given under [Source]:

```
[Source]
location = <handle of an import storage>
skip empty files = <yes/no>
```

To use this with your flow daemon:

```
from flow import Flow
from flow.source import UnmanagedFilesListener

class MyFlow(Flow):
    SOURCE = UnmanagedFilesListener

    def start(self, f):
        # f is a vizone.payload.media.UnmanagedFile
        pass
```

3.2.4 STDIN

class `flow.source.local.`**`STDIN`**(*payload_class=None*)

Source class that reads standard in, designed to be run once, and especially for Transfer Plugins.

Options given under [Source]:

```
[Source]
payload class = None|vizone.payload.transfer.PluginData|...
```

To use this with your flow daemon:

```
    from flow import Flow
    from flow.source.local import STDIN

    class MyFlow(Flow):
        SOURCE = STDIN

        def start(self, data):
            # data is a vizone.payload.transfer.PluginData if that format is
            pass
```

3.2.5 Interval

class `flow.source.interval.`**`Interval`**(*interval=60*, *window_start=None*, *window_end=None*)

Source class that triggers an event on interval.

Options given under [Source]:

```
    [Source]
    interval = 60 (seconds)
    window start = 01:00 [HH[:MM[:SS]]]
    window end = 03:00 [HH[:MM[:SS]]]
```

To use this with your flow daemon:

```
    from flow import Flow
    from flow.source import Interval

    class MyFlow(Flow):
        SOURCE = Interval

        def start(self, data):
            pass
```

Note that data is always `None` for this source.

3.3 Building a Web Interface

This is an example of a web interface built with Flow. The advantages of using Flow for this are:

- Easy to configure.
- The worker pool can be used to handle asyncronous jobs.

It's recommended to use bottle since it's part of the python-one package but any other framework can be used. This example features bottle however:

```
from flow import Flow, EventBased
from flow.needs import NeedsClient
from vizone.resource.asset import get_asset_by_id
from one_depends import bottle

class Web(EventBased, NeedsClient):
    _has_event_loop = True  # This prevents Flow from running a second
                            # event loop when bottle.run() exists.
    instance = None
    def __init__(self, interface='localhost', port=8080):
        self.server_interface = interface
        self.server_port = int(port)
        self.callback = None

        logging.log("Web Interface Settings", {
            'interface': self.server_interface,
            'port': self.server_port,
        }, 'pp')

        Web.instance = self  # We use this instance as a singleton

    def run(self):
        bottle.run(
            host=self.server_interface,
            port=self.server_port,
        )


class XmlServer(Flow):
    SOURCE = Web

    # this class could also implement the run(message) method and act
    # as a worker pool. The web method would then call
    # Web.interface.callback(message) to spawn a job. If a worker pool
    # is not needed, just leave this class like implemented here.


@bottle.get('/<asset_id>')
def get_asset(asset_id):
    """
    Serve XML for a given asset id.
    """
    try:
        asset = get_asset_by_id(asset_id, client=Web.instance.client)
    except HTTPClientError:
        return bottle.HttpError(status=404, body="There is no such asset.")

    bottle.response.status = 200
    bottle.response.set_header('Content-Type', asset.content_type)
    bottle.response.set_header('Content-Disposition', 'attachment; filename="%s.xml"' % asset.id)
    return output.generate()
```

And the corresponding INI file:

```
[Flow]
app name = xmlserver
class = xmlserver.XmlServer

[Source]
interface = localhost
port = 34566

[Viz One]
hostname = vizoneserver
username = user
password = password
use https = no
```

```
check certificates = no
```

CHAPTER 4

Utililities

Contents

4.1 Working with Assets

Tools for working with Asset Entries (Items)

4.1.1 Create or Update Asset

`flow.operation.`**`create_or_update_asset`**(*id=None*, *metadata=None*, *acl=None*, *mediaacl=None*, *tags=None*, *materialtype=None*, *category=None*, *rightscode=None*, *client=None*)

Creates or Updates an Item with a given `id`. Metadata updates will be retried three times if there are conflicts.

Parameters

- **`id`** (*`Optional[unicode]`*) – An asset id or "site identity"
- **`metadata`** (*`Optional[vizone.vdf.Payload]`*) – The metadata to update to. Can be a dict with a 'form' field too.

- **acl** (*Optional[vizone.vizone.payload.user_group.Acl]*) – Specify a ACL when creating the Asset
- **mediaacl** (*Optional[vizone.vizone.payload.user_group.Acl]*) – Specify a Media ACL when creating the Asset
- **tags** (*Optional[dict]*) – scheme => term dictionary for custom tags when creating the Asset
- **materialtype** (*Optional[unicode]*) – Set the Material Type to this when creating the Asset
- **category** (*Optional[unicode]*) – Set the Category to this when creating the Asset
- **rightscode** (*Optional[unicode]*) – Set the Rights Code to this when creating the Asset
- **client** (*Optional[vizone.client.Instance]*) – A Viz One client to use (None means the default)

Returns The updated or created Asset Entry

Return type vizone.payload.asset.Item

4.1.2 Modifying Group Permissions in an ACL

`flow.acl.`**`set_group_permissions`**(*acl*, *group_name*, *read=True*, *write=True*, *admin=True*)

Make sure a group is in the ACL and that is has the correct rights, If no rights are true, the group entry will be removed.

Parameters

- **vizone.payload.user_group.aclentry.Acl** (*acl*) – The ACL to operate on
- **str** (*group_name*) – The name of the group
- **bool** (*admin*) – Read right
- **bool** – Write right
- **bool** – Admin right

Returns True if altered, False otherwise

Return type bool

4.2 Working with Files

Tools useful for handling file import flows.

4.2.1 Import Unmanaged File

`flow.operation.`**`import_unmanaged_file`**(*asset*, *uri_list*, *client=None*)

Start an Unmanaged File import to a given Asset Entry

Parameters

- **asset** (*vizone.payload.asset.Item*) – The Asset Entry to import to

- **uri_list** (*vizone.urilist.UriList*) – A URI List containing the link to the media
- **client** (*Optional[vizone.client.Instance]*) – A Viz One client to use (None means the default)

Returns True if successful, False on error

Return type bool

4.2.2 Delete Unmanaged File

`flow.operation.`**`delete_unmanaged_file`**`(`*unmanaged_file*, *client=None*`)`

Delete an Unmanaged File from Viz One

Parameters

- **unmanaged_file** (*vizone.payload.media.UnmanagedFile*) – The Unmanaged File to delete
- **client** (*Optional[vizone.client.Instance]*) – A Viz One client to use (None means the default)

4.3 Building a Transfer Plugin

You can use Flow to build a Transfer Plugin quite easily.

4.3.1 Extending the TransferPlugin class

class `flow.transfer.`**`TransferPlugin`**`(`*instance_name=None*`)`

Transfer Plugin base class.

TransferPlugin child classes automatically inherits Flow and NeedsClient.

Very basic example (for more advanced ones, look at the XmlExport built-ins):

```
from flow.transfer import TransferPlugin
from vizone import logging

class MyTransferPlugin(TransferPlugin):
    def start(self, plugin_data):
        self.use(plugin_data)

        self.update_progress(0)

        logging.info(u'Asset is %s', self.asset.title)
        logging.info(u'Source URL is %s', self.source)
        logging.info(u'Destination URL is %s', self.destination)

        self.update_progress(100)

        # or why not:

        self.fail("There were errors!")
```

SOURCE

alias of `STDIN`

fail(*message*)
Failed the Transfer Step and exits the program with status 0. This allows for a cleaner reporting in Viz One.

Parameters **unicode|str** (`message`) – An error message that will show up in Viz One.

start(*data*, ***kwargs*)
You'll have to implement this method yourself.

update_progress(*progress*)
Update the progress of the associated Transfer Step.

Parameters **int** (`progress`) – As a percentage from 0 to 100, e.g. 67

use(*data*, *require_asset=True*, *require_source=True*, *require_destination=True*)
Extract and verify the plugin data that comes from the Transfer Subsystem. We should have gotten two FTP addresses, plugin settings in the form of a VDF payload containing username and password, as well as a transfer step and request to control the workflow and report back progress.

You can use the `require_asset`, `require_source` and `require_metadata` flags to control what to require in terms of content in the plugin data.

4.3.2 Setting up the Repository

Creating a Transfer Plugin package is easiest to do with `pluginmgr` on a Viz One server. It will help you to create the files you need. Before you start, create a new folder and add a file named `plugin` to it, with these contents:

```
type: runnable
package: myplugin
depends: python-one
methods: myplugin
mode: filecopy
title: My Plugin
author: Vizrt
version: 1.0
source-scheme: ftp
destination-scheme: ftp
destination-conflicts: none
killprocess: false
partial: false
partial-by-frame: false
```

You can now basically use `pluginmgr make` interactively until you get it right.

```
mkdir myplugin
cd myplugin
vim plugin
pluginmgr make

# vdf: vdf model at ~/myplugin/etc/xfer-plugin-myplugin.vdf
# required but missing at /opt/ardome/bin/pluginmgr line 847.
```

So, you need a VDF. This is for holding the settings of your plugin in a way that is editable in Viz One's Administration console. You can use `pluginmgr vdf` to create one:

```
pluginmgr vdf plugin.user:Username:user ...
    plugin.password:Password:user ...
    > etc/xfer-plugin-myplugin.vdf
```

Now try `pluginmgr make` again:

```
pluginmgr make

# Could not find the specified bin/myplugin in source tree
```

So there is no plugin executable to run. For flow this will be a script, named `bin/myplugin` (where `myplugin` would be what you specified as `method` above) create a folder `bin` and put a file `myplugin` with this in it:

```
#!/bin/bash

/opt/python-one/bin/wrap_python -m flow ...
    /opt/ardome/apps/myplugin/myplugin.ini -g
```

Note that the path might need to be adjusted later, but this is a decent convention. Now try `pluginmgr make` again:

```
pluginmgr make

# INFO:
# INFO: Edit files as necessary. Then run the following command to ...
#     build the apa-package:
# INFO:
# INFO:   $ apa dist xfer-plugin-myplugin/1.0
# INFO:
# INFO: To install:
# INFO:   # scamp install -i [version] xfer-plugin-myplugin--1.0.apa
# INFO:   # scamp apply
# INFO:   $ ardemctl restart xfer
# INFO:
```

Now `pluginmgr` is happy with the setup. Included here are also the instructions on how to build an APA package out of this. It's not time for this quite yet though, so just make a note of the command for later use. We should create the actual app as well:

```
mkdir -p apps/myplugin
vim apps/myplugin/myplugin.py
vim apps/myplugin/myplugin.ini
```

Good starting point for the `myplugin.py` file:

```python
from flow.transfer import TransferPlugin
from vizone import logging

class MyPlugin(TransferPlugin):
    def start(self, plugin_data):
        self.use(plugin_data)

        self.update_progress(0)

        logging.info(u'Asset is %s', self.asset.title)
        logging.info(u'Source URL is %s', self.source)
        logging.info(u'Destination URL is %s', self.destination)

        self.update_progress(100)
```

And for the `myplugin.ini` file:

```ini
[Flow]
app name = myplugin
class = myplugin.MyPlugin
```

```
[Source]
payload class = vizone.payload.transfer.PluginData

[Viz One]
use https = no
```

Now these files needs to be part of the APA package. To achieve this, edit the file `build/xfer-plugin-myplugin/FILES` (the first line is new):

```
apps/myplugin/* -> apps/myplugin

@ chmod 755
bin/myplugin -> xferplugin/bin/myplugin
@ nochmod

@ nochmod
@ chmod 644
etc/xfer-plugin-myplugin.xml -> xferplugin/etc/xfer-plugin-myplugin.xml

@ nochmod
@ chmod 644
etc/xfer-plugin-myplugin.vdf -> xferplugin/etc/xfer-plugin-myplugin.vdf

@ nochmod
```

After changing this, **do not run ‘‘pluginmgr make‘‘ again**, as this will overwrite the files in `build/`. You can actually delete the `plugin` file now and edit `etc/xfer-plugin-myplugin.xml` if you want to change any plugin settings.

4.3.3 Creating and Installing a Package

To build an APA package of your plugin, you can use the command given by `pluginmgr make` previously. Remember: don’t run it again now!

```
apa dist xfer-plugin-myplugin/1.0
```

Note: The command `apa` might need to be called with explicit path, being `/opt/scamp/bin/apa`.

This command should not give any output, but there should be an `.apa` file in your working directory. Install this with scamp and restart the transfer daemons:

```
sudo /opt/scamp/bin/scamp install xfer-plugin-myplugin--1.0.apa
sudo /opt/scamp/bin/scamp apply
ardemctl restart xfer
```

4.3.4 Setting up a Rewrite Rule so the Plugin Gets Used

That the plugin exists does not mean it’s automatically used. This example shows how to set up an export storage and use it to export to it. First create the export storage:

```
storagemgr add stg plugin-export description="Plugin Export"
storagemgr join export plugin-export
mkdir /home/ardome/plugin-export
sudo chown armedia:ardome /home/ardome/plugin-export
sudo ln -s /home/ardome/plugin-export /ardome/media/exp/plugin-export
storagemgr add mountpoint [INSERT SERVER HERE] plugin-export ...
    /ardome/media/exp/plugin-export
```

To make every transfer to this new export destination use the plugin you must create a rewrite rule. Run `confmgr edit transfer.rewrite`, and add a new one:

```
1:
  criteria:
    destination-storage:
      - plugin-export
  apply:
    destination-step-method: myplugin
```

4.4 Miscellaneous

Various tools for making life easier.

4.4.1 Storing Data on the Server

class `flow.store.`**`Store`**(*appname*, *client*)

A Store is a centralized store based on the Client Config API. Note that:

- The Client Config API operates per user.
- The Client Config API operates per application.
- Data can be any JSON serializable object, for instance nested python dicts, lists, strings, ints, floats and booleans.

Usage example:

```
from flow import Flow
from flow.needs import NeedsStore, NeedsClient
from flow.source import UnmanagedFilesListener

class MyFlow(Flow, NeedsStore):
    """
    MyFlow uses a Store.
    """
    source = UnmanagedFilesListener

    def start(self, f, info=None, log_id=-1):

        # If your Flow class inherits NeedsStore, it will be equipped with
        # an ``self.store`` attribute which can be used to get, update and
        # delete stored data on the server.
        stored_info = self.store.get('key')
        if stored_info is None:
            # there was nothing there

        self.store.put('key', {
            'any': 'json',
            'serializable': ['structure', 'goes', 'here']
        })
```

```
            # You can also delete
            self.store.delete('key')
```

delete(*key*)
Delete the stored `value` under a `key` for the object's application.

Parameters **key** (*unicode*) – The key to delete

get(*key*)
Get the value of a certain key for the object's application.

Parameters **key** (*unicode*) – The key used for storage

Returns The storede value or `None`

Return type dict

put(*key*, *value*)
Put a `value` under a `key` for the object's application.

Parameters

- **key** (*unicode*) – The key used for storage
- **value** (*dict*) – The data to store, as a dict

Returns The storede value is returned

Return type dict

4.4.2 Parsing Data Fields with the MultiParser

class `flow.data.`**MultiParser**(*type='string'*, *format=None*, *default_timezone='UTC'*, *source=None*)
A value parser that converts various data formats into Python and Viz One entities, including:

- `unicode` (default)
- `int`
- `float`
- `iso8601` (timestamp)
- `date` (date with configurable format)
- `time` (time with configurable format)
- `datetime` (date + time with configurable format)
- `dictionary` (dictionary term with given source reference)

The intended use is for putting the resulting object as value into a VDF Payload.

Example:

```
from flow.data import MultiParser

mp = MultiParser(type='date', format='%d/%m')
result = mp.convert('4/12')
```

For a more thourough example using the MultiParser together with configuration, please check out *xmlimport.XmlImport*.

For more information about date and time parsing syntax, please refer to https://docs.python.org/2/library/datetime.html#strftime-and-strptime-behavior

Parameters

- **type** (*str*) – string|integer|float|iso|date|time|datetime|dictionary
- **format** (*str*) – format string for parseing date, time and datetime
- **default_timezone** (*str*) – default time zone only used fore datetime
- **source** (*str*) – url do dictionary, should be an Atom-based feed

convert (*raw_value*, *client*)
Perform conversion configured when contructing the object.

Parameters

- **raw_value** (*unicode*) – The raw string to parse
- **client** (*vizone.client.Instance*) – The HTTP client to use when looking up dictionary terms.

Returns object

4.4.3 Locking Based on a Key

class flow.lock.**Locked** (*key*)
Thread-safe lock by key string.

Usage:

```
from flow.lock import Locked

with Locked('mystring'):
    # .. do stuff
```

Only one process per key can be run simultaneously, other attempt will be held until the lock is released.

4.4.4 Retrying on Conflict

flow.operation.**retry_on_conflict** (*max_retries=3*)
Decorator that can wrap a function or method and retry it upon a conflict exception.

Example:

```
@retry_on_conflict(max_retries=3)
def my_function(self, entry, conflict=False):

    # If there was a conflict, the entry needs to be refetched
    if conflict:
        entry.parse(self.client.GET(entry.self_link))

    # Do the operations
    self.client.PUT(entry.edit_link, entry)
```

Note that:

- Any argument will be reused as it, with changes.
- You can raise a Retry exception to retry for other reasons than a 409 Conflict.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search