Workflow Documentation

Release 1.2.0

Roman Chyla

October 23, 2014

Contents

CHAPTER 1

About

Workflow engine is a Finite State Machine with memory. It is used to execute set of methods in a specified order.

Here is a simple example of a workflow configuration:

```
[
  check_token_is_wanted, # (run always)
  [                      # (run conditionally)
     check_token_numeric,
     translate_numeric,
     next_token          # (stop processing, continue with next token)
     ],
  [                      # (run conditionally)
     check_token_proper_name,
     translate_proper_name,
     next_token          # (stop processing, continue with next token)
     ],
  normalize_token,       # (only for "normal" tokens)
  translate_token,
]
```

CHAPTER 2

Details

In the above simple configuration example, you can probably guess what the processing pipeline does with tokens - the whole task is made of four steps and the whole configuration is just stored as a Python list. Every task is implemented as a function that takes two objects:

- currently processed object
- workflow engine instance

Example:

```
def next_token(obj, eng):
    eng.ContinueNextToken()
```

There are NO explicit states, conditions, transitions - the job of the engine is simply to run the tasks one after another. It is the responsibility of the task to tell the engine what is going to happen next; whether to continue, stop, jump back, jump forward and few other options.

This is actually a *feature*, useful when there are a lot of possible exceptions and transition states to implement for NLP processing, as well as useful to make the workflow engine simple and fast – but it also has disadvantages, as the workflow engine will not warn in case of errors.

The workflow module comes with many patterns that can be directly used in the definition of the pipeline, such as IF, IF_NOT, PARALLEL_SPLIT and others.

The individual tasks then can influence the whole pipeline, available ‘’commands” are:

```
eng.stopProcessing  # stops the current workflow
eng.haltProcessing  # halts the workflow (can be used for nested wf engines)
eng.continueNextToken  # can be called many levels deep, jumps up to next token
eng.jumpTokenForward  # will skip the next object and continue with the next one
eng.jumpTokenBack  # will return back, start processing again
eng.jumpCallForward  # in one loop [call, call...] jumps x steps forward
eng.jumpCallBack  # in one loop [call, call...] jumps x steps forward
eng.breakFromThisLoop  # break from this loop, but do not stop processing
```

Consider this example of a task:

```
def if_else(call):
    def inner_call(obj, eng):
       if call(obj, eng):      #if True, continue processing
          eng.jumpForward(1)
       else:                   #else, skip the next step
          eng.jumpForward(2)
    return inner_call
```

We can then write *workflow definition* like:

```
if_else(stage_submission),
[
    [if_else(fulltext_available),  #this will be run only when fulltext is uploaded during form subm
     [extract_metadata, populate_empty_fields],
     [#do nothing ]],
    [if_else(check_for_duplicates),
     [stop_processing],
     [synchronize_fields, replace_values]],
    check_mandatory_fields,]
    ],
    [
    check_mandatory_fields,        # this will run only for 'review' stage
    check_preferred_values,
    save_record
]
```

CHAPTER 3

Tasks

Tasks are simple python functions, we can enforce rules (not done yet!) in a pythonic way using pydoc conventions, consider this:

```
def check_duplicate(obj, eng):
    """
    This task checks if the uploaded fulltext is a duplicate
         @type obj: InspireGeneralForm
         @precondition: obj.paths[]
                 list, list of paths to uploaded files
         @postcondition: obj.fulltext[]
                 list containing txt for the extracted document
                         obj.duplicateids[]
                 list of inspire ids records that contain the duplicate of this document
         @raise: stopProcessing on error
         @return: True if duplicate found

    """
    ...
```

So using the python docs, we can instruct workflow engine what types of arguments are acceptable, what is the expected outcome and what happens after the task finished. And let's say, there will be a testing framework which will run the workflow pipeline with fake arguments and will test all sorts of conditions. So, the configuration is not cluttered with states and transitions that are possible, developers can focus on implementation of the individual tasks, and site admins should have a good understanding what the task is supposed to do – the description of the task will be displayed through the web GUI.

Here are some examples of workflow patterns (images are from http://www.yawlfoundation.org) and their implementation in Python. This gives an idea that workflow engine remains very simple and by supplying special functions, we can implement different patterns.

3.1 Example: Parallel split

This pattern is called Parallel split (as tasks B,C,D are all started in parallel after task A). It could be implemented like this:

```
def PARALLEL_SPLIT(*args):
    """
    Tasks A,B,C,D... are all started in parallel
    @attention: tasks A,B,C,D... are not addressable, you can't
        you can't use jumping to them (they are invisible to
        the workflow engine). Though you can jump inside the
```

```
        branches
    @attention: tasks B,C,D... will be running on their own
        once you have started them, and we are not waiting for
        them to finish. Workflow will continue executing other
        tasks while B,C,D... might be still running.
    @attention: a new engine is spawned for each branch or code,
        all operations works as expected, but mind that the branches
        know about themselves, they don't see other tasks outside.
        They are passed the object, but not the old workflow
        engine object
    @postcondition: eng object will contain lock (to be used
        by threads)
    """

    def _parallel_split(obj, eng, calls):
        lock=thread.allocate_lock()
        i = 0
        eng.setVar('lock', lock)
        for func in calls:
            new_eng = duplicate_engine_instance(eng)
            new_eng.setWorkflow([lambda o,e: e.setVar('lock', lock), func])
            thread.start_new_thread(new_eng.process, ([obj], ))
            #new_eng.process([obj])
    return lambda o, e: _parallel_split(o, e, args)
```

And is used like this:

```
from workflow.patterns import PARALLEL_SPLIT
from my_module_x import task_a,task_b,task_c,task_d

[
 task_a,
 PARALLEL_SPLIT(task_b,task_c,task_d)
]
```

3.2 Example: Arbitrary cycles

This is just for amusement (and to see how complicated it looks in the configuration).

```
#!python
[
  ...        #here some conditional start
  task_a,
  task_b,
  task_c,
  if_else(some_test),
    [task_d, [if_else(some_test),
                lambda obj, eng: eng.jumpCallBack(-6),  #jump back to task_a
                some_other_task,
              ]]
    [some_other_task],
  ...
]
```

TODO

Jumping back and forward is obviously dangerous and tedious (depending on the actual configuration), we need a better solution.

3.3 Example: Synchronisation

After the execution of task B, task C, and task D, task E can be executed (I will present the threaded version, as the sequential version would be dead simple).

```
def SYNCHRONIZE(*args, **kwargs):
    """
    After the execution of task B, task C, and task D, task E can be executed.
    @var *args: args can be a mix of callables and list of callables
                the simplest situation comes when you pass a list of callables
                they will be simply executed in parallel.
                   But if you pass a list of callables (branch of callables)
                which is potentially a new workflow, we will first create a
                workflow engine with the workflows, and execute the branch in it
    @attention: you should never jump out of the synchronized branches
    """
    timeout = MAX_TIMEOUT
    if 'timeout' in kwargs:
        timeout = kwargs['timeout']

    if len(args) < 2:
        raise Exception('You must pass at least two callables')

    def _synchronize(obj, eng):
        queue = MyTimeoutQueue()
        #spawn a pool of threads, and pass them queue instance
        for i in range(len(args)-1):
            t = MySpecialThread(queue)
            t.setDaemon(True)
            t.start()

        for func in args[0:-1]:
            if isinstance(func, list) or isinstance(func, tuple):
                new_eng = duplicate_engine_instance(eng)
                new_eng.setWorkflow(func)
                queue.put(lambda: new_eng.process([obj]))
            else:
                queue.put(lambda: func(obj, eng))

        #wait on the queue until everything has been processed
        queue.join_with_timeout(timeout)

        #run the last func
        args[-1](obj, eng)
    _synchronize.__name__ = 'SYNCHRONIZE'
    return _synchronize
```

Configuration (i.e. what would admins write):

```
from workflow.patterns import SYNCHRONIZE
from my_module_x import task_a,task_b,task_c,task_d

[
```

```
    SYNCHRONIZE(task_b,task_c,task_d, task_a)
]
```

CHAPTER 4

API

This documentation is automatically generated from Workflow's source code. Workflow engine is a Finite State Machine with memory.

class workflow.engine.**GenericWorkflowEngine**(*processing_factory=None*, *callback_chooser=None*, *before_processing=None*, *after_processing=None*)

Workflow engine is a Finite State Machine with memory.

It is used to execute set of methods in a specified order.

example:

```
from merkur.workflows.parts import load_annie, load_seman
from newseman.general.workflow import patterns as p

workflow = [
    load_seman_components.workflow,
    p.IF(p.OBJ_GET(['path', 'text'], cond='any'), [
        p.TRY(g.get_annotations(), retry=1,
              onfailure=p.ERROR('Error in the annotation workflow'),
              verbose=True),
        p.IF(p.OBJ_GET('xml'),
        translate_document.workflow)
    ])
]
```

This workflow is then used as:

```
wfe = GenericWorkflowEngine()
wfe.setWorkflow(workflow)
wfe.process([{'foo': 'bar'}, {'foo': 'baz'}])
```

This workflow engine instance can be freezed and restarted, it remembers its internal state and will pick up processing after the last finished task.

```
import pickle
s = pickle.dumps(wfe)
```

However, when restarting the workflow, you must initialize the workflow tasks manually using their original definition

```
wfe = pickle.loads(s)
wfe.setWorkflow(workflow)
```

It is also not possible to serialize WFE when custom factory tasks were provided. If you attempt to serialize such a WFE instance, it will raise exception. If you want to serialize WFE including its factory hooks and workflow callbacks, use the `PhoenixWorkflowEngine` class instead.

addCallback(*key*, *func*, *before=None*, *after=None*, *relative_weight=None*)
Insert one callable to the stack of the callables.

addManyCallbacks(*key*, *list_or_tuple*)
Insert many callable to the stack of thec callables.

static after_processing(*objects*, *self*)
Standard post-processing callback, basic cleaning.

static before_processing(*objects*, *self*)
Standard pre-processing callback.

Save a pointer to the processed objects.

breakFromThisLoop()
Stop in the current loop but continues in those above.

static callback_chooser(*obj*, *self*)
Choose proper callback method.

There are possibly many workflows inside this workflow engine and they are meant for different types of objects, this method should choose and return the callbacks appropriate for the currently processed object.

Parameters

- **obj** – currently processed object
- **eng** – the workflow engine object

Returns set of callbacks to run

configure(***kwargs*)
Method to set attributes of the workflow engine.

Note: Use with extreme care (well, you can set up the attrs directly, I am not protecting them, but that is not nice). Used mainly if you want to change the engine's callbacks - if processing factory *before_processing*, *after_processing*.

Parameters **kwargs** – dictionary of values

continueNextToken()
Continue with the next token.

delVar(*key*)
Delete parameter from the internal storage.

execute_callback(*callback*, *obj*)
Execute the callback - override this method to implement logging.

getCallbacks(*key='*'*)
Return callbacks for the given workflow.

Parameters **key** – name of the workflow (default: '*') if you want to get all configured workflows pass None object as a key

Returns list of callbacks

getCurrObjId()
Return id of the currently processed object.

getCurrTaskId()
Return id of the currently processed task.

Note: The return value of this method is not thread-safe.

---

getObjects()
Return iterator for walking through the objects.

getVar(*key*, *default=None*)
Return named *obj* from internal stack. If not found, return *None*.

Parameters

- **key** – name of the object to return
- **default** – if not found, what to return instead (if this arg is present, the stack will be initialized with the same value)

Returns anything or None

haltProcessing()
Halt the workflow (stop also any parent *wfe*).

hasVar(*key*)
Return True if parameter of this name is stored.

jumpCallBack(*offset*)
Return *x* calls back in the current loop.

Note: Be careful with circular loop.

---

jumpCallForward(*offset*)
Jump to *x* th call in this loop.

jumpTokenBack(*offset*)
Return *x* tokens back - be careful with circular loops.

jumpTokenForward(*offset*)
Jump to *x* th token.

process(*objects*)
Start processing.

Parameters **objects** – either a list of object or instance of *TokenizedDocument*

Returns You never know what will be returned from the workflow. But many exceptions can be raised, so watch out for them, if there happened an exception, you can be sure something wrong happened (something that your workflow should handle and didn't). Workflow engine is not interfering into the processing chain, it is not catching exceptions for you.

static **processing_factory**(*objects*, *self*)
Default processing factory, will process objects in order.

As the WFE proceeds, it increments the internal counter, the first position is the number of the element. This pointer increases before the object is taken.

2nd pos is reserved for the array that points to the task position. The number there points to the task that is currently executed; when error happens, it will be there unchanged. The pointer is updated after the task finished running.

Parameters

- **objects** – list of objects (passed in by self.process())

- **cls** – engine object itself, because this method may be implemented by the standalone function, we pass the self also as a cls argument

removeAllCallbacks()
Remove all the tasks from the workflow engine instance.

removeCallbacks(*key*)
Remove callbacks for the given *key*.

replaceCallbacks(*key*, *funcs*)
Replace processing workflow with a new workflow.

reset()
Empty the stack memory.

restart(*obj*, *task*, *objects=None*)
Restart the workflow engine after it was deserialized.

run_callbacks(*callbacks*, *objects*, *obj*, *indent=0*)
Execute callbacks in the workflow.

Parameters

- **callbacks** – list of callables (may be deep nested)
- **objects** – list of processed objects
- **obj** – currently processed object
- **indent** – int, indendation level - the counter at the indent level is increases after the task has finished processing; on error it will point to the last executed task position. The position adjusting also happens after the task has finished.

setLogger(*logger*)
Set logger used by workflow engine.

Note: The logger instance must be pickable if the serialization should work.

---

setPosition(*obj_pos*, *task_pos*)
Set the internal pointers (of current state/obj).

Parameters

- **obj_pos** – (int) index of the currently processed object After invocation, the engine will grab the next obj from the list
- **task_pos** – (list) multidimensional one-element list that says at which level the task should restart. Example: 6th branch, 2nd task = [5, 1]

setVar(*key*, *what*)
Store the obj in the internal stack.

setWorkflow(*list_or_tuple*)
Set the (default) workflow which will be run on *process()* call.

Parameters **list_or_tuple** – workflow configuration

stopProcessing()
Break out, stop everything (in the current *wfe*).

class workflow.engine.**PhoenixWorkflowEngine**(**args*, ***kwargs*)
Implementation of serializable workflow engine.

Engine is able to be *serialized* and re-executed also with its workflow tasks - without knowing their original definition. This implementation depends on the picloud module - http://www.picloud.com/. The module must be installed in the standard location.

CHAPTER 5

Changes

Version 1.2.0 (released 2014-10-23):

- Fix interference with the logging level. (#22 #23)
- Test runner is using Pytest. (#21)
- Python 3 support. (#7)
- Code style follows PEP8 and PEP257. (#6 #14)
- Improved Sphinx documentation. (#5 #28)
- Simplification of licensing. (#27)
- Spelling mistake fixes. (#26)
- Testing with Tox support. (#4)
- Configuration for Travis-Cl testing service. (#3)
- Test coverage report. (#2)
- Unix style line terminators. (#10)

Version 1.0 (released 2011-07-07):

- Initial public release.
- Includes the code created by Roman Chyla, the core of the workflow engine together with some basic patterns.
- Raja Sripada <rsripada at cern ch> contributed improvements to the pickle&restart mechanism.

CHAPTER 6

Contributing

Bug reports, feature requests, and other contributions are welcome. If you find a demonstrable problem that is caused by the code of this library, please:

1. Search for already reported problems.
2. Check if the issue has been fixed or is still reproducible on the latest *master* branch.
3. Create an issue with **a test case**.

If you create a feature branch, you can run the tests to ensure everything is operating correctly:

```
$ ./run-tests.sh

...
Name                              Stmts   Miss  Cover   Missing
---------------------------------------------------------------
workflow/__init__                     2      0   100%
workflow/config                     231     92    60%   ...
workflow/engine                     321     93    71%   ...
workflow/patterns/__init__            5      0   100%
workflow/patterns/controlflow       159     66    58%   ...
workflow/patterns/utils             249    200    20%   ...
workflow/version                      2      0   100%
---------------------------------------------------------------
TOTAL                               969    451    53%

...

55 passed, 1 warnings in 3.10 seconds
```

CHAPTER 7

License

Workflow is free software; you can redistribute it and/or modify it under the terms of the Revised BSD License quoted below.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

CHAPTER 8

Authors

Workflow was originally developed by Roman Chyla. It is now being developed and maintained by the Invenio collaboration. You can contact us at info@invenio-software.org.

Contributors:

- Roman Chyla <roman.chyla@gmail.com>
- Raja Sripada <raja.sripada@cern.ch>
- Jiri Kuncar <jiri.kuncar@cern.ch>
- Tibor Simko <tibor.simko@cern.ch>
- Brett Anthoine <brett.anthoine@netplus.pro>

W

A

B

C

D

E

G

H

J

P

R

S

W