Workflow Documentation

Release 2.0.1.dev20160623

Roman Chyla

Apr 28, 2017

Contents

CHAPTER 1

What is Workflow?

The Workflow library provides a method of running Finite State Machines with memory. It can be used to execute a set of methods, complete with conditions and patterns.

Workflow allows for a number of independent pieces of data to be processed by the same logic, while allowing for the entire process to be forwarded, backwarded, paused, inspected, re-executed, modified and stored.

CHAPTER 2

How do I use Workflow?

In the following sections we will take a look at Workflow's features by working on examples of increasing complexity. Please keep an eye out for comments in the code as they provide crucial information.

Basic workflow use

Basic use is comprised of the following steps:

1. Instantiate a **workflow engine**. For this example we will use the simplest provided one, but you may also extend it to add custom behaviour.

```
from workflow.engine import GenericWorkflowEngine
my_engine = GenericWorkflowEngine()
```

2. Create **tasks**. These are purpose-built function functions that the workflow engine can execute.

   The engine always passes **(current_token, current_engine)** as arguments to these functions, so they need to support them. Note the **add_data** function needs to be able to accept more arguments. For this we use a closure.

```
from functools import wraps

def print_data(obj, eng):
    """Print the data found in the token."""
    print obj.data

def add_data(number_to_add):
    """Add number_to_add to obj.data."""
    @wraps(add_data)
    def _add_data(obj, eng):
        obj.data += number_to_add
    return _add_data
```

2. Create a **workflow definition** (also known as **callbacks**). This is a (sometimes nested) list of **tasks** that we wish to run.

```
my_workflow_definition = [
    add_data(1),
    print_data
]
```

3. Define **tokens**. This is the data that we wish to feed the workflow. Since the data we will deal with in this example is immutable, we need to place it in **token wrappers** . Another reason you may wish to wrap your data is to be able to store *metadata* in the object.

```
class MyObject(object):
    def __init__(self, data):
        self.data = data

my_object0 = MyObject(0)
my_object1 = MyObject(1)
```

4. **Run** the engine on a list of such wrappers with our workflow definition.

   The engine passes the tokens that we give it one at a time through the workflow.

```
my_engine.callbacks.replace(my_workflow_definition)

my_engine.process([my_object0, my_object1])
# The engine prints: "1\n2"
my_object0 == 1
my_object1 == 2
```

5. **Bonus**! Once the engine has ran, it can be reused.

```
my_engine.process([my_object0, my_object1])
# The engine prints: "2\n3"
my_object0 == 2
my_object1 == 3
```

Loops and interrupts

Let's take a look at a slightly more advanced example. There are two things to note here:

- How control flow is done. We provide, among others, **IF_ELSE** and **FOR** statements. They are simple functions - therefore you can make your own if you wish to. We will see examples of this in the Details section.
- Control flow can reach outside the engine via exceptions. We will raise the **WorkflowHalt** exception to return the control to our code before the workflow has even finished and then even resume it.

In this example, we have a series of lists composed of 0 and 1 and we want to:

1. Add [0, 1] at the end of the list.
2. Repeat a until list >= [0, 1, 0, 1].
3. Add [1] when we are done.

Here are some example transformations that describe the above:

- [] –> [0, 1, 0, 1, 1]

- [0, 1] –> [0, 1, 0, 1, 1]
- [0, 1, 0, 1] –> [0, 1, 0, 1, 0, 1, 1]

Time for some code! Let’s start with the imports. Pay close attention as their arguments are explained briefly here.

```
from workflow.engine import GenericWorkflowEngine
from workflow.errors import HaltProcessing
from workflow.patterns.controlflow import (
FOR,          # Simple for-loop, a-la python. First argument is an iterable,
              # second defines where to save the current value, and the third
              # is the code that runs in the loop.

HALT,         # Halts the engine. This brings it to a state where it can be
              # inspected, resumed, restarted, or other.

IF_ELSE,      # Simple `if-else` statement that accepts 3 arguments.
              # (condition, tasks if true, tasks if false)

CMP,          # Simple function to support python comparisons directly from a
              # workflow engine.
)
```

Now to define some functions of our own.

Note that the first function leverages *eng.extra_data*. This is a simple dictionary that the *GenericWorkflowEngine* exposes and it acts as a shared storage that persists during the execution of the engine.

The two latter functions wrap engine functionality that’s already there, but add *print* statements for the example.

```
def append_from(key):
    """Append data from a given `key` of the engine's `extra_data`."""
    def _append_from(obj, eng):
        obj.append(eng.extra_data[key])
        print "new data:", obj
    return _append_from

def interrupt_workflow(obj, eng):
    """Raise the `HaltProcessing` exception.

    This is not handled by the engine and bubbles up to our code.
    """
    print "Raising HaltProcessing"
    eng.halt("interrupting this workflow.")

def restart_workflow(obj, eng):
    """Restart the engine with the current object, from the first task."""
    print "Restarting the engine"
    eng.restart('current', 'first')
```

We are now ready to create the workflow:

```
my_workflow = [
    FOR(range(2), "my_current_value", # For-loop, from 0 to 1, that sets
                                      # the current value to
                                      # `eng.extra_data["my_current_value"]`
        [
            append_from("my_current_value"),  # Gets the value set above
                                              # and appends it to our token
        ]
    ), # END FOR

    IF_ELSE(
        CMP((lambda o, e: o), [0, 1 ,0, 1], "<"),   # Condition:
                                                     # "if obj < [0,1,0,1]:"

        [ restart_workflow ],                      # Tasks to run if condition
                                                   # is True:
                                                   # "return back to the FOR"

        [                                          # Tasks to run if condition
                                                   # is False:

        append_from("my_current_value"),           # "append 1 (note we still
                                                   # have access to it)
        interrupt_workflow                         # and interrupt"
        ]
    ) # END IF_ELSE
]
```

Because our workflow interrupts itself, we will wrap the call to *process* and *restart*, in *try-except* statements.

```
# Create the engine as in the previous example
my_engine = GenericWorkflowEngine()
my_engine.callbacks.replace(my_workflow)

try:
    # Note how we don't need to keep a reference to our tokens - the engine
    # allows us to access them via `my_engine.objects` later.
    my_engine.process([[], [0,1], [0,1,0,1]])
except HaltProcessing:
    # Our engine was built to throw this exception every time an object is
    # completed. At this point we can inspect the object to decide what to
    # do next. In any case, we will ask it to move to the next object,
    # until it stops throwing the exception (which, in our case, means it
    # has finished with all objects).
    while True:
        try:
            # Restart the engine with the next object, starting from the
            # first task.
            my_engine.restart('next', 'first')
        except HaltProcessing:
            continue
        else:
            print "Done!", my_engine.objects
            break
```

Here is what the execution prints:

```
new data: [0]
new data: [0, 1]
Restarting the engine
new data: [0, 1, 0]
new data: [0, 1, 0, 1]
new data: [0, 1, 0, 1, 1]
Raising HaltProcessing
new data: [0, 1, 0]
new data: [0, 1, 0, 1]
new data: [0, 1, 0, 1, 1]
Raising HaltProcessing
new data: [0, 1, 0, 1, 0]
new data: [0, 1, 0, 1, 0, 1]
new data: [0, 1, 0, 1, 0, 1, 1]
Raising HaltProcessing
Done! [[0, 1, 0, 1, 1], [0, 1, 0, 1, 1], [0, 1, 0, 1, 0, 1, 1]]
```

Celery support

Celery is a widely used distributed task queue. The independent nature of workflows and their ability to be restarted and resumed makes it a good candidate for running in a task queue. Let's take a look at running a workflow inside celery.

Assuming workflow is already installed, let's also install celery:

```
$ pip install 'celery[redis]'
```

Onto the code next:

```python
from celery import Celery

# `app` is required by the celery worker.
app = Celery('workflow_sample', broker='redis://localhost:6379/0')

# Define a couple of basic tasks.
def add(obj, eng):
    obj["value"] += 2

def print_res(obj, eng):
    print obj.get("value")

# Create a workflow out of them.
flow = [add, print_res]

# Mark our execution process as a celery task with this decorator.
@app.task
def run_workflow(data):
    # Note that the imports that this function requires must be done inside
    # it since our code will not be running in the global context.
    from workflow.engine import GenericWorkflowEngine
    wfe = GenericWorkflowEngine()
    wfe.setWorkflow(flow)
    wfe.process(data)

# Code that runs when we call this script directly. This way we can start
# as many workflows as we wish and let celery handle how they are
# distributed and when they run.
if __name__ == "__main__":
    run_workflow.delay([{"value": 10}, {"value": 20}, {"value": 30}])
```

Time to bring celery up:

1. Save this file as */some/path/workflow_sample.py*
2. Bring up a worker in one terminal:

```
$ cd /some/path
$ celery -A workflow_sample worker --loglevel=info
```

3. Use another terminal to request *run_workflow* to be ran with the above arguments:

```
$ cd /some/path
$ python workflow_sample.py
```

You should see the worker working. Try running *python workflow_sample.py* again.

Storage-capable engine

The Workflow library comes with an alternative engine which is built to work with SQLAlchemy databases (*DbWorkflowEngine*). This means that one can store the state of the engine and objects for later use. This opens up new possibilities:

- A front-end can be attached to the engine with ease.
- Workflows can be stored for resume at a later time..
- ..or even shared between processing nodes.

In this example we will see a simple implementation of such a database-stored, resumable workflow.

We will reveal the problem that we will be solving much later. For now, we can start by creating a couple of SQLAlchemy schemas:

- One to attach to the workflow itself (a workflow will represent a student)
- and one where a single grade (a grade will be the grade of a single test)

Note that the *Workflow* model below can store an element pointer. This pointer (found at *engine.state.token_pos*) indicates the object that is currently being processed and saving it is crucial so that the engine can resume at a later time from that point.

```python
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String, create_engine, ForeignKey, Boolean
from sqlalchemy.orm import sessionmaker, relationship

# Create an engine and a session
engine = create_engine('sqlite://')
Base = declarative_base(bind=engine)
DBSession = sessionmaker(bind=engine)
session = DBSession()


class Workflow(Base):
    __tablename__ = 'workflow'
    id = Column(Integer, primary_key=True)
    state_token_pos = Column(Integer, default=-1)
    grades = relationship('Grade', backref='workflow',
                          cascade="all, delete, delete-orphan")

    def save(self, token_pos):
        """Save object to persistent storage."""
        self.state_token_pos = token_pos
        session.begin(subtransactions=True)
        try:
            session.add(self)
            session.commit()
        except Exception:
            session.rollback()
            raise


class Grade(Base):
    __tablename__ = 'grade'
    id = Column(Integer, primary_key=True)
    data = Column(Integer, nullable=False, default=0)
    user_id = Column(Integer, ForeignKey('workflow.id'))

    def __init__(self, grade):
        self.data = grade
        session.add(self)

Base.metadata.create_all(engine)
```

Next, we have to tell *DbWorkflowEngine* how and when to use our storage. To do that we need to know a bit about the engine's *processing_factory* property, which is expected to provide this structure of methods and properties:

- *before_processing*
- *after_processing*
- *before_object*
- *after_object*
- *action_mapper* (property)
  - *before_callbacks*
  - *after_callbacks*
  - *before_each_callback*
  - *after_each_callback*
- *transition_exception_mapper* (property)
  - *StopProcessing*
  - *HaltProcessing*
  - *ContinueNextToken*
  - *JumpToken*
  - *Exception*
  - ... (Can be extended by adding any method that has the name of an expected exception)

The *transition_exception_mapper* can look confusing at first. It contains not Exceptions, but methods that are called when exceptions with the same name are raised.

- Some exceptions are internal to the engine only and never bubble up. (eg *JumpToken*, *ContinueNextToken*)
- Others are partly handled internally and then bubbled up to the user to take action. (eg *Exception*)

Let's use the above to ask our engine to:

1. Save the first objects that it is given.
2. **Save to our database every time it finished processing an object and** when there is an expected failure.

For now, all we need to know is that in our example, *HaltProcessing* is an exception that we will intentionally raise and we want to save the engine when it occurs. Once again, follow the comments carefully to understand the code.

```python
from workflow.engine_db import DbWorkflowEngine
from workflow.errors import HaltProcessing
from workflow.engine import TransitionActions, ProcessingFactory

class MyDbWorkflowEngine(DbWorkflowEngine):

    def __init__(self, db_obj):
        """Load an old `token_pos` from the db into the engine."""

        # The reason we save token_pos _first_, is because calling `super`
        # will reset.
        token_pos = db_obj.state_token_pos

        self.db_obj = db_obj
        super(DbWorkflowEngine, self).__init__()

        # And now we inject it back into the engine's `state`.
        if token_pos is not None:
            self.state.token_pos = token_pos
        self.save()

    # For this example we are interested in saving `token_pos` as explained
    # previously, so we override `save` to do that.
    def save(self, token_pos=None):
        """Save the state of the workflow."""
        if token_pos is not None:
            self.state.token_pos = token_pos
        self.db_obj.save(self.state.token_pos)

    # We want our own processing factory, so we tell the engine that we
    # have subclassed it below.
    @staticproperty
    def processing_factory():
        """Provide a processing factory."""
        return MyProcessingFactory


class MyProcessingFactory(ProcessingFactory):
    """Processing factory for persistence requirements."""

    # We also have our own `transition_actions`
    @staticproperty
    def transition_exception_mapper():
        """Define our for handling transition exceptions."""
        return MyTransitionActions

    # Before any processing is done, we wish to save the `objects` (tokens)
    # that have been passed to the engine, if they aren't already stored.
    @staticmethod
    def before_processing(eng, objects):
        """Make sure the engine has a relationship with
        its objects."""
        if not eng.db_obj.grades:
            for obj in objects:
                eng.db_obj.grades.append(obj)

    # We wish to save on every successful completion of a token.
    @staticmethod
    def after_processing(eng, objects):
        """Save after we processed all the objects successfully."""
        eng.save()


class MyTransitionActions(TransitionActions):

    # But we also wish to save when `HaltProcessing` is raised, because this
    # is going to be an expected situation.
    @staticmethod
    def HaltProcessing(obj, eng, callbacks, e):
        """Save whenever HaltProcessing is raised, so
        that we don't lose the state."""
        eng.save()
        raise e
```

And now, for the problem that we want to solve itself. Imagine an fictional exam where a student has to take 6 tests in one day. The tests are processed in a specific order by a system. Whenever the system locates a failing grade, as punishment, the student is asked to take the failed test again the next day. Then the checking process is resumed until the next failing grade is located and the student must show up again the following day.

Assume that a student, Zack P. Hacker, has just finished taking all 6 tests. A workflow that does the following checking can now be implemented like so:

```
from workflow.patterns.controlflow import IF, HALT
from workflow.utils import staticproperty

my_workflow_instance = Workflow()
my_db_engine = MyDbWorkflowEngine(my_workflow_instance)

def grade_is_not_passing(obj, eng):
    print 'Testing grade #{0}, with data {1}'.format(obj.id, obj.data)
    return obj.data < 5

callbacks = [
    IF(grade_is_not_passing,
        [
            HALT()
        ]),
]

my_db_engine.callbacks.replace(callbacks)
try:
    my_db_engine.process([
        Grade(6), Grade(5), Grade(4),
        Grade(5), Grade(2), Grade(6)
    ])
except HaltProcessing:
    print 'The student has failed this test!'

# At this point, the engine has already saved its state in the database,
# regardless of the outcome.
```

The above script prints:

```
Testing grade #1, with data 6
Testing grade #2, with data 5
Testing grade #3, with data 4
The student has failed this test!
```

"Obviously this system is terrible and something must be done", thinks Zack who was just notified about his "4" and logs onto the system, armed with a small python script:

```
def amend_grade(obj, eng):
    print 'Amending this grade..'
    obj.data = 5

evil_callbacks = [
    IF(grade_is_not_passing,
        [
            amend_grade
        ]),
]

# Load yesterday's workflow and bring up an engine for it.
revived_workflow = session.query(Workflow).one()
my_db_engine = MyDbWorkflowEngine(revived_workflow)

print '\nWhat Zak sees:', [grade.data for grade in revived_workflow.grades]

# Let's fix that.
my_db_engine.callbacks.replace(evil_callbacks)
print 'Note how the engine resumes from the last failing test:'
my_db_engine.restart('current', 'first', objects=revived_workflow.grades)
```

These words are printed in Zack's terminal:

```
What Zak sees: [6, 5, 4, 5, 2, 6]
Note how the engine resumes from the last failing test:
Testing grade #3, with data 4
Amending this grade..
Testing grade #4, with data 5
Testing grade #5, with data 2
Amending this grade..
Testing grade #6, with data 6
```

When someone logs into the system to check how Zack did..

```
revived_workflow = session.query(Workflow).one()
print '\nWhat the professor sees:', [grade.data for grade in revived_workflow.grades]
```

Everything looks good:

```
What the professor sees: [6, 5, 5, 5, 5, 6]
```

The moral of this story is to keep off-site logs and back-ups. Also, workflows are complex but powerful.

CHAPTER 3

Signals support

Adding to the exception and override-based mechanisms, Workflow supports a few signals out of the box if the *blinker* package is installed. The following exceptions are triggered by the *GenericWorkflowEngine*.

Signal	Called by
workflow_started	ProcessingFactory.before_processing
workflow_finished	ProcessingFactory.after_processing
workflow_halted	TransitionActions.HaltProcessing

CHAPTER 4

Useful engine methods

Other than *eng.halt*, the GenericWorkflowEngine provides more convenience methods out of the box.

Sample call	Description
eng.stop()	stop the workflow
eng.halt("list exhausted")	halt the workflow
eng.continue_next_token()	continue from the next token
eng.jump_token(-2)	jump *offset* tokens
eng.jump_call(3)	jump *offset* steps of a loop
eng.break_current_loop()	break out of the current loop

By calling these, any **task** can influence the whole pipeline. You can read more about the methods our engines provide at the end of this document.

CHAPTER 5

Patterns

The workflow module also comes with many patterns that can be directly used in the definition of the pipeline, such as **PARALLEL_SPLIT**.

Consider this example of a task:

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

We can then write a **workflow definition** like this:

```
[
    if_else(stage_submission),
        [
            [
                if_else(fulltext_available),
                    [extract_metadata, populate_empty_fields],
                    []
            ],
            [
                if_else(check_for_duplicates),
                    [stop_processing],
                    [synchronize_fields, replace_values]
            ],
            check_mandatory_fields,
        ],
        [
            check_mandatory_fields,
            check_preferred_values,
            save_record
        ]
]
```

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 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()
        eng.store['lock'] = lock
        for func in calls:
            new_eng = eng.duplicate()
            new_eng.setWorkflow([lambda o, e: e.store.update({'lock': lock}), func])
            thread.start_new_thread(new_eng.process, ([obj], ))
    return lambda o, e: _parallel_split(o, e, args)
```

Subsequently, we can use PARALLEL_SPLIT 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)
]
```

Note that PARALLEL_SPLIT is already provided in *workflow.patterns.PARALLEL_SPLIT*.

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 6

GenericWorkflowEngine API

This documentation is automatically generated from Workflow's source code.

class `workflow.engine.`**`GenericWorkflowEngine`**

Workflow engine is a Finite State Machine with memory.

Used to execute set of methods in a specified order.

See *docs/index.rst* for extensive examples.

static **`abort`**`()`

Abort current workflow execution without saving object.

static **`abortProcessing`**`(`**args*, ***kwargs*`)`

Abort current workflow execution without saving object.

`break_current_loop``()`

Break out of the current callbacks loop.

`callback_chooser``(`*obj*`)`

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

Returns list of callbacks to run

Note: This method is part of the engine and not part of *Callbacks* to grant those who wish to have their own logic here access to all the attributes of the engine.

`continue_next_token``()`

Continue with the next token.

`current_object`

Return the currently active DbWorkflowObject.

`current_taskname`
Get name of current task/step in the workflow (if applicable).

`execute_callback` (*callback*, *obj*)
Execute a single callback.

Override this method to implement per-callback logging.

`halt` (*msg='‘*, *action=None*, *payload=None*)
Halt the workflow (stop also any parent *wfe*).

Halts the currently running workflow by raising HaltProcessing.

You can provide a message and the name of an action to be taken (from an action in actions registry).

Parameters

- **`msg`** (`str`) – message explaining the reason for halting.
- **`action`** (`str`) – name of valid action in actions registry.

Raises HaltProcessing

`has_completed`
Return whether the engine has completed its execution.

`init_logger` ()
Return the appropriate logger instance.

`jump_call` (*offset*)
Jump to *offset* calls (in this loop) away.

Parameters **`offset`** (`int`) – Number of steps to jump. May be positive or negative.

static **`jump_token`** (*offset*)
Jump to *offset* tokens away.

`process` (*objects*, *stop_on_error=True*, *stop_on_halt=True*, *initial_run=True*, *reset_state=True*)
Start processing *objects*.

Parameters

- **`objects`** – list of objects to be processed
- **`stop_on_error`** – whether to stop the workflow if HaltProcessing is raised
- **`stop_on_error`** – whether to stop the workflow if WorkflowError is raised
- **`initial_run`** – whether this is the first execution of this engine

Raises Any exception that is not handled by the *transitions_exception_mapper*.

`processing_factory`
alias of `ProcessingFactory`

`restart` (*obj*, *task*, *objects=None*, *stop_on_error=True*, *stop_on_halt=True*)
Restart the workflow engine at given object and task.

Will restart the workflow engine instance at given object and task relative to current state.

obj must be either:

- “prev”: previous object
- “current”: current object
- “next”: next object

- "first": first object

task must be either:

- "prev": previous task
- "current": current task
- "next": next task
- "first": first task

To continue with next object from the first task:

```
wfe.restart("next", "first")
```

Parameters

- **obj** (*str*) – the object which should be restarted
- **task** (*str*) – the task which should be restarted

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.

static skipToken(**args*, ***kwargs*)
Skip current workflow object without saving it.

static skip_token()
Skip current workflow object without saving it.

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

class workflow.engine.**MachineState**(*token_pos=None*, *callback_pos=None*)
Machine state storage.

Properties

token_pos

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

callback_pos

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.

callback_pos_reset()
Reset *callback_pos* to its default value.

reset()
Reset the state of the machine.

token_pos_reset()
Reset *token_pos* to its default value.

class workflow.engine.**Callbacks**
Callbacks storage and interface for workflow engines.

The reason for interfacing for a dict is mainly to prevent cases where the state and the callbacks would be out of sync (eg by accidentally adding a callback to the beginning of a callback list).

add(*func*, *key='*'*)
Insert one callable to the stack of the callables. :type key: str

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

classmethod cleanup_callables(*callbacks*)
Remove non-callables from the passed-in callbacks.

..note:: Tuples are flattened into normal members. Only lists are nested as expected.

clear(*key='*'*)
Remove tasks from the workflow engine instance, or all if no key.

clear_all()
Remove tasks from the workflow engine instance, or all if no key.

empty()
Is it empty?

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

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

Returns list of callbacks

replace(*funcs*, *key='*'*)
Replace processing workflow with a new workflow.

class workflow.engine.**_Signal**
Helper for storing signal callers.

workflow_error(*eng*, **args*, ***kwargs*)
Call the *workflow_error* signal if signals is installed.

workflow_finished(*eng*, **args*, ***kwargs*)
Call the *workflow_finished* signal if signals is installed.

workflow_halted(*eng*, **args*, ***kwargs*)
Call the *workflow_halted* signal if signals is installed.

workflow_started(*eng*, **args*, ***kwargs*)
Call the *workflow_started* signal if signals is installed.

CHAPTER 7

DbWorkflowEngine API

class `workflow.engine_db.`**`DbWorkflowEngine`**(*db_obj*, ***kwargs*)

GenericWorkflowEngine with DB persistence.

Adds a SQLAlchemy database model to save workflow states and workflow data.

Overrides key functions in GenericWorkflowEngine to implement logging and certain workarounds for storing data before/after task calls (This part will be revisited in the future).

`database_objects`
Return the objects associated with this workflow.

`final_objects`
Return the objects associated with this workflow.

`halted_objects`
Return the objects associated with this workflow.

`known_statuses`
alias of `WorkflowStatus`

`name`
Return the name.

`processing_factory`
alias of `DbProcessingFactory`

`running_objects`
Return the objects associated with this workflow.

`save`(*status=None*)
Save the workflow instance to database.

`status`
Return the status.

`uuid`
Return the status.

class `workflow.engine_db.`**`WorkflowStatus`**`(`*label*`)`
Define the known workflow statuses.

class `workflow.engine_db.`**`ObjectStatus`**`(`*label*`)`
Specify the known object statuses.

CHAPTER 8

Changes

Version 2.0.0 (released 2016-06-17):

Incompatible changes

- Drops *setVar()*, *getVar()*, *delVar()* and exposes the *engine.store* dictionary directly, with an added *setget()* method that acts as *getVar()*.
- Renames s/getCurrObjId/curr_obj_id/ and s/getCurrTaskId/curr_task_id/ which are now properties. Also renames s/getObjects/get_object/ which now no longer returns index.
- Removes PhoenixWorkflowEngine. To use its functionality, the new engine model's extensibility can be used.
- Moves *processing_factory* out of the *WorkflowEngine* and into its own class. The majority of its operations can now be overridden by means of subclassing *WorkflowEngine* and the new, complementing *ActionMapper* and *TransitionActions* classes and defining properties. This way *super* can be used safely while retaining the ability to *continue* or *break* out of the main loop.
- Moves exceptions to *errors.py*.
- Changes interface to use pythonic names and renames methods to use more consistent names.
- *WorkflowHalt* exception was merged into *HaltProcessing* and the *WorkflowMissingKey* exception has been dropped.
- Renames ObjectVersion to ObjectStatus (as imported from Invenio) and ObjectVersion.FINAL to ObjectVersion.COMPLETED.

New features

- Introduces *SkipToken* and *AbortProcessing* from *engine_db*.
- Adds support for signaling other processes about the actions taken by the engine, if blinker is installed.

- Moves callbacks to their own class to reduce complexity in the engine and allow extending.
- *GenericWorkflowEngine.process* now supports restarting the workflow (backported from Invenio)

Improved features

- Updates all *staticproperty* functions to *classproperty* to have access to class type and avoid issue with missing arguments to class methods.
- Re-raises exceptions in the engine so that they are propagated correctly to the user.
- Replaces *_i* with *MachineState*, which protects its contents and explains their function.
- Allows for overriding the logger with any python-style logger by defining the *init_logger* method so that projects can use their own.
- Splits the DbWorkflowEngine initializer into *with_name* and *from_uuid* for separation of concerns. The latter no longer implicitly creates a new object if the given uuid does not exist in the database. The uuid comparison with the log name is now reinforced.
- Updates tests requirements.

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 9

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 10

License

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

Copyright (C) 2011, 2012, 2014, 2015, 2016 CERN.

All rights reserved.

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.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

In applying this license, CERN does not waive the privileges and immunities granted to it by virtue of its status as an Intergovernmental Organization or submit itself to any jurisdiction.

CHAPTER 11

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@inveniosoftware.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>
- Dimitrios Semitsoglou-Tsiapos <dsemitso@cern.ch>
- Jan Aage Lavik <jan.age.lavik@cern.ch>

Index