WfEpy Documentation

Release 0.1.1

Filip Pobořil

Jan 15, 2020

Contents:

CHAPTER 1

API

class wfepy.**Workflow**

Workflow graph - collection of tasks.

Variables ***task*** – collection of tasks, dict with tasks name as key

check_graph()

Check workflow graph - if some task is missing, all task are marked properly as start, join or end points, . . .

Raises ***WorkflowError*** – when there are some problems with workflow graph

create_runner(**args*, ***kwargs*)

Create `Runner` from this workflow.

end_points

List of names of tasks that are marked as end points.

load_tasks(*module*)

Load tasks from module and add them to workflow graph. Can be also module name, then module will be get from *sys.module* by that name.

Raises ***WorkflowError*** – if name of loaded task is not unique

start_points

List of names of tasks that are marked as start points.

class wfepy.**Runner**(*workflow*, *context=None*)

Workflow execution engine.

Variables

- **workflow** – `Workflow`
- **context** – arbitrary user object, passed to all tasks
- **state** – state of execution

dump (*file_path*)
Dump runner to file. Stored dump contains `context` and `state` so runner execution can be restored and finished later.

finished
Workflow execution finished. True when reached end points and there is no task that should be executed.

load (*file_path*)
Load runner from file. See also `dump()`.

run ()
Execute tasks from workflow.

Some tasks might end in state in which they cannot be executed (waiting for external event or join point waiting for preceding tasks). If there is no task that can be executed run will stop executing and `finished` property will be `False`. In that case run should be called again (with some delay or runner can be dumped to file by `dump()` and executed later).

See `TaskState` for list of task states.

task_execute (*task*)
Execute `Task`.

transition_eval (*transition*)
Evauluate `Transition.cond`.

class wfepy.**Task** (*func*, *name=NOTHING*, *labels=NOTHING*)
Workflow task. Wraps function for use in workflow.

Wrapped function must accept context from `Runner` via only parameter and should return `True` or `False` whether task was completed and execution can continue with following tasks.

If wrapped function returned `False` execution will stop and task will be executed again in next run. This way can be implemented waiting, eg. for external event.

Variables

- **function** – wrapped function
- **name** – task name (by default function name)
- **labels** – task labels
- ***followed_by*** – connection to next tasks (set of `Transition`)
- **preceded_by** – names of preceding tasks, generated by `Workflow`
- **is_start_point** – task is start point of workflow
- **is_join_point** – task is join point of multiple tasks
- **is_end_point** – task is end point of workflow

has_labels (*labels*, *reducer=<built-in function any>*)
Check if task has labels.

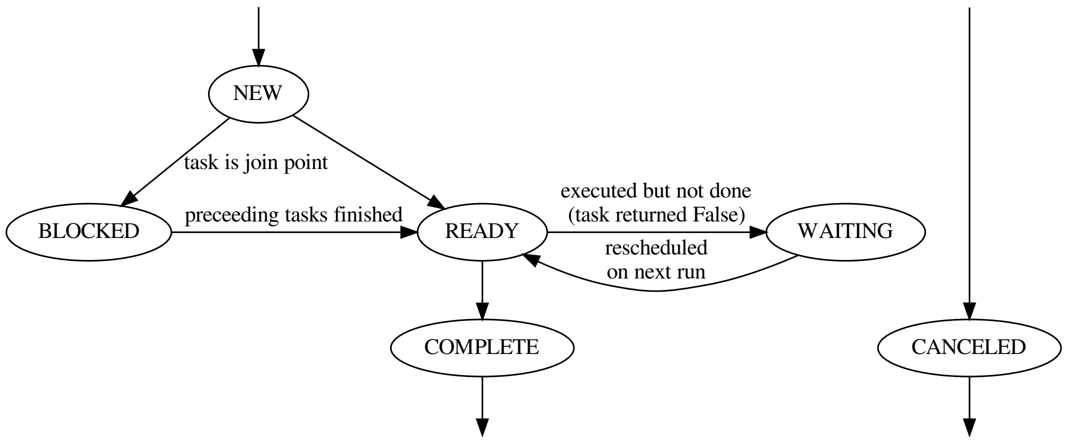
Reducer is used to reduce multi-labels check to single boolean value. *all* checks if task have all labels, *any* checks if task has at least one of labels.

class wfepy.**TaskState**
Enumeration of task states.

Variables

- **NEW** – task new in queue

- **WAITING** – task is waiting, function returned `False`
- **BLOCKED** – task is waiting for completion of preceding tasks
- **READY** – task is ready for execution
- **COMPLETE** – task was executed and will be expanded
- **CANCELED** – task was not executed because transition condition was not met



class wfepy.**Transition**(*dest*, *cond=None*)
Transition to following task.

Variables

- **dest** – name of following task
- **cond** – condition whether following task should be executed, function that will receive context from *Runner* and must return bool (allows to create conditional branching and looping in graph)

class wfepy.**WorkflowError**
Generic workflow error.

1.1 Decorators

class wfepy.**DecoratorStack**(*function*, *decorator_list=NOTHING*)
Utility to collect function decorators and execute them in reverse order at once.

classmethod add(*decorator*)
Create decorator function that will create *DecoratorStack* using *create()* and add decorator to list of decorators.

add_decorator(*decorator*)
Add decorator to stack.

apply_to(*func*)

Apply decorators to func and return new func created by chain of decorators.

Return value of each function is used as argument of next function and first function will receive `func` as argument.

classmethod create(*func*)

Create new *DecoratorStack* from function or other stack.

classmethod reduce(*decorator*)

Create decorator function that will create *DecoratorStack* using *create()*, add decorator to list of decorators and apply decorators from stack to decorated function.

wfepy.**task**(**args*, ***kwargs*)

Decorator to mark function as workflow task. See *Task* for arguments documentation.

wfepy.**followed_by**(**args*, ***kwargs*)

Add transition to next task. See *Transition* for arguments documentation.

wfepy.**start_point**()

Mark task as start point. See *Task*.

wfepy.**join_point**()

Mark task as join point. See *Task*.

wfepy.**end_point**()

Mark task as end point. See *Task*.

CHAPTER 2

How to contribute

Thanks for considering contributing to WfEpy.

2.1 Reporting issues

- Under which versions of WfEpy does this happen? Check if this issue is fixed in the repository.

2.2 Submitting patches

- Include tests if your patch is supposed to solve a bug, and explain clearly under which circumstances the bug happens. Make sure the test fails without your patch.
- Follow PEP8 and produce nice code. Code must pass flake8 check.
- For features: Before submitting MR with new feature consider creating RFE issue. We want to keep this library as simple as possible so new features should be reviewed, signed, etc.

2.2.1 Running the testsuite

You probably want to set up a virtualenv and then install all dendencies by running `pip install -e .`.

The minimal requirement for running the testsuite is `py.test` and `flake8` with `flake8-bugbear` extension. You can install it with:

```
pip install pytest flake8 flake8-bugbear
```

Then you can run the testsuite with:

```
pytest
flake8
```

For a more isolated test environment, you can also install `tox` instead of `pytest`. You can install it with:

```
pip install tox
```

The `tox` command will then run all tests including flake8 and other tests.

CHAPTER 3

Examples

3.1 Simple

Whole worfklow is build from tasks and connections between them.

Tasks are functions with `task()` decorator and connection between tasks is defined by `followed_by()` decorator. First argument of `followed_by()` decorator is name of next tasks, that should be executed when current task is finished.

Tasks names are intentionally strings so you don't need to care about imports or order of declarations in file. But that is not requirement, `followed_by()` also accept other tasks (function decorated with `task()`).

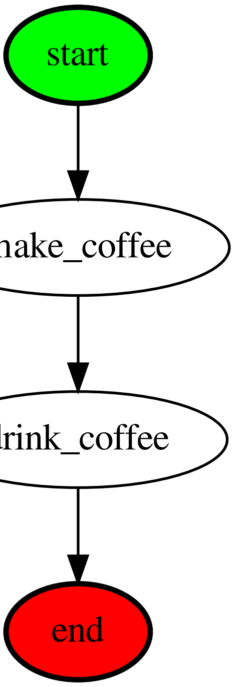
```
import wfepy as wf


@wf.task()
@wf.start_point()
@wf.followed_by('make_coffee')
def start(ctx):
    # All tasks must return True or False if they were finished or waiting for
    # some external event or something and must be executed again later.
    return True


@wf.task()
@wf.followed_by('drink_coffee')
def make_coffee(ctx):
    return True


@wf.task()
@wf.followed_by('end')
def drink_coffee(ctx):
    import random
    if not random.choice([True, False]):
```

(continues on next page)

(continued from previous page)

```
        # Still drinking. Returing False means this task was not completed and
        # must be executed again on next run.
        return False
    return True


@wf.task()
@wf.end_point()
def end(ctx):
    return True
```

Workflow can be converted to graph. Nice to have in documentation or for debugging purposes. Even this workflow is pretty simple, real-world workflow can be complex with lot of tasks declared across many files, with conditional branches, . . .



Finally, workflow can be executed. Example script that will execute workflow from example above.

```
import logging
import wfepy
import wfepy.utils

logging.basicConfig(level=logging.INFO)


# Import module with tasks.
import simple
```

(continues on next page)

(continued from previous page)

```


# Create new workflow.
wf = wfepy.Workflow()
# Load tasks from module and add them to workflow.
wf.load_tasks(simple)
# Check if graph is OK, all tasks are defined, decorated correctly, ...
wf.check_graph()

# Render graph.
wfepy.utils.render_graph(wf, 'basic.gv')

# Create runner for workflow.
runner = wf.create_runner()

# Execute workflow.
runner.run()

# Check if workflow finished, no tasks are waiting.
while not runner.finished:
    logging.info('Workflow is not finished, trying run it again...')
    runner.run()
```

Output from script

```
INFO:wfepy.workflow:Executing task start
INFO:wfepy.workflow:Task start is complete
INFO:wfepy.workflow:Executing task make_coffee
INFO:wfepy.workflow:Task make_coffee is complete
INFO:wfepy.workflow:Executing task drink_coffee
INFO:wfepy.workflow:Task drink_coffee is waiting

INFO:root:Workflow is not finished, trying run it again...
INFO:wfepy.workflow:Executing task drink_coffee
INFO:wfepy.workflow:Task drink_coffee is waiting

INFO:root:Workflow is not finished, trying run it again...
INFO:wfepy.workflow:Executing task drink_coffee
INFO:wfepy.workflow:Task drink_coffee is complete
INFO:wfepy.workflow:Executing task end
INFO:wfepy.workflow:Task end is complete
INFO:wfepy.workflow:Reached end point end
```

Task `drink_coffee` was waiting for something and no other tasks could be executed, so process stopped.

Waiting tasks are tasks that returned `False` while finished tasks must return `True`. This allow implement waiting for events, for example when user must add comment to Jira task before process can continue.

3.2 Branches

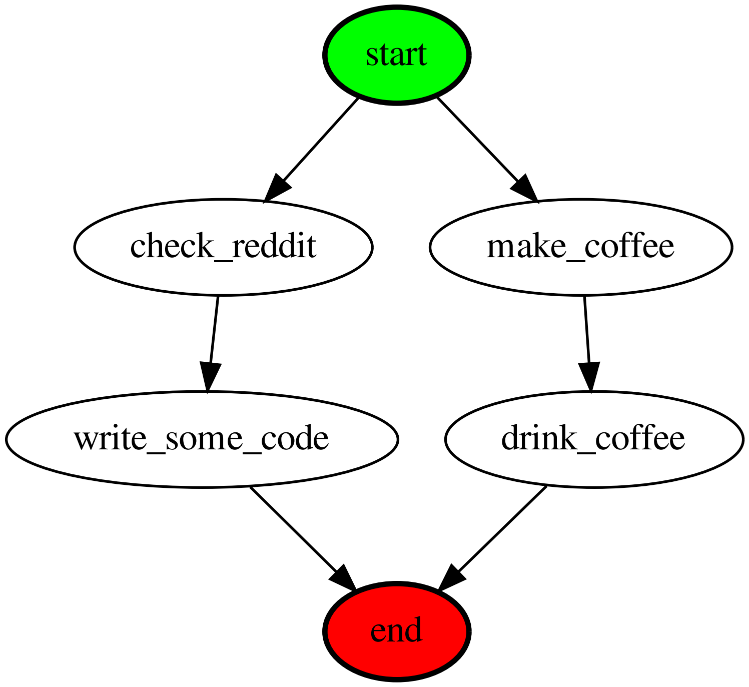
Task can be also followed by multiple tasks so process will be executing multiple task branches in parallel. Task are not executed in parallel by threads or processes but it still can be used to execute as much as possible tasks if task in one branch is waiting.

Looking at coffee drinking example, you can do some other things while waiting until coffee and while drinking.

```python
import random
import wfepy as wf


@wf.task()
@wf.start_point()
@wf.followed_by('make_coffee')
@wf.followed_by('check_reddit')
def start(ctx):
    return True


@wf.task()
@wf.followed_by('drink_coffee')
def make_coffee(ctx):
    return True


@wf.task()
@wf.followed_by('write_some_code')
def check_reddit(ctx):
    return True


@wf.task()
@wf.followed_by('end')
def write_some_code(ctx):
    return random.choice([True, False])


@wf.task()
@wf.followed_by('end')
def drink_coffee(ctx):
    return random.choice([True, False])


@wf.task()
@wf.join_point()
@wf.end_point()
def end(ctx):
    return True
```

Task `start` has multiple `followed_by` decorations so when this task finish, process will expand followed by list and start executing tasks from both branches. In the end of workflow branches are joined in `end` task. Join points must be explicitly marked by `join_point` decorator to avoid mistakes.

If you forgot to mark join point (or start point or end point) *`wfepy.Workflow.check_graph()`* will raise error and you should fix it.


start
check_reddit
make_coffee
write_some_code
drink_coffee
end

CHAPTER 4

WfEpy

WfEpy (Workflow Engine for Python) is Python library for creating workflows and automating processes. It is designed to be as simple as possible so developers can focus on tasks logic, not how to execute workflow, store state, etc.

Documentation: https://wfepy.readthedocs.io/en/stable/

4.1 Basic features

The library provides the following features:

- Workflow defined in code, via decorators
- Flat workflow structure
- Visualisation features (via graphviz)
- Partial execution model (workflow can be triggered multiple times until final completion)
- Allows long running tasks (can be weeks/months or more) without persistent processes
- No scheduler included, but can be triggered by cron
- Serialization / deserialization included
- Multiple start and end points are supported

The library adds some restrictions:

- Workflow functions must return boolean, where:
  - *True* means the task has completed and workflow can be advanced
  - *False* means the task is still waiting (e.g. for user input)
- Workflow functions carry around a *context* object (normally a dict)

4.2 Details

The workflow is defined via decorators attached to functions, such as:

```
@wfepy.task()
@wfepy.start_point()
@wfepy.followed_by('make_coffee')
def start(context):
    ...

@wfepy.task()
@wfepy.followed_by('drink_coffee')
def make_coffee(context):
    ...

@wfepy.task()
@wfepy.followed_by('end')
def drink_coffee(context):
    ...

@wfepy.task()
@wfepy.end_point()
def end(context):
    ...
```

A function can be followed by multiple functions:

```
@wfepy.task()
@wfepy.followed_by('add_sugar')
@wfepy.followed_by('add_milk')
def make_coffee(context):
    ...
```

A function can be conditionally followed by another function:

```
@wfepy.task()
# only make foam when we've been requested 'cappucino'
@wfepy.followed_by('make_foam', lambda context: context.data.get('cappucino'))
# always add milk
@wfepy.followed_by('add_milk')
def make_coffee(context):
    ...
```

4.3 Execution model

WfEpy does not provide any scheduler, but can be triggered by cron. It works on a partial-execution model, meaning it can be triggered multiple times.

The workflow is attempted on every execution, but will only end when at least one of the end points have been reached. If the workflow can't be ended during an execution, then the state (including user data and currently-waiting tasks) is exported/serialized for the next attempt.

```
import coffee_workflow

wf = wfepy.Workflow()
wf.load_tasks(coffee_workflow)

runner = wf.create_runner()
if restore_state:
    runner.load('state-file')

runner.run()

runner.dump('state-file')
```

This simple design provides many options on workflow execution and customization. Most workflow libraries out there require external dependencies like databases, message bus/queue systems etc. Our library requires no such things, just python and its package dependencies.

4.4 Installation

Install it using pip

```
pip3 install wfepy
```

or clone repository

```
git clone https://github.com/redhat-aqe/wfepy.git
cd wfepy
```

and install Python package including dependencies

```
python3 setup.py install
```

Index