Jobbergate

Release 1.4.3

Jimmy Hedman

May 03, 2023

CONTENTS:

Jobbergate is a questionnaire application that populates Jinja2 templates with given answers.

In its simplest form you only need a *views.py* that defines *mainflow* and a template file (called *templates/job_template.j2*) which gets populated with your answers. To support advanced workflows you could define multiple levels of questions, change to other templates, run functions before and after subworkflows, have follow up questions to boolean questions and so on.

To install, just do:

```
pip install jobbergate
```

Configure jobbergate.yaml to point to your directory where you have all applications. Set `JOBBERGATE_PATH` environment to point to where your jobbergate.yaml resides.

Jobbergate is a Flask application but could be run both as a web application and as a cli application.

To run as web application, just do:

```
flask run
```

To run as cli application, you can find out which applications it has in its configuration directory with:

```
flask --help
```

If you have an application called *simple* you run it with:

```
flask simple outputfile.sh
```

This will populate the simple application template with the answers you give in the following interactive session, and create `outputfile.sh`.

If you want the output file to be run in bash automatically, you may explicitly give the command in your implemented application. For example, if you define a function in your application's `controller.py` such as:

```
@workflow.logic
def post_generic(data):
    retval = {"cmd_command":f"cat {data['filename']}"}
    return retval
```

the application will run:

```
cat outputfile.sh
```

which shows the content of the output file. This feature can be suppressed by using the '–no-cmd' flag:

```
flask simple outputfile.sh --no-cmd
```

CHAPTER

ONE

WORKFLOW

1.1 Simple workflow

A simple workflow is implemented with the function *mainflow* defined in *views.py* and a template defined in *templates/job_template.j2*:

```
+-- views.py
+-+ templates/
  + job_template.j2
```

views.py:

```
from jobbergate import appform

def mainflow(data):
    return [appform.Text("jobname", "What is the jobname?", default="simulation")]
```

job_template.j2:

```
#!/bin/bash
#SBATCH -j {{ data.jobname }}
sleep 30
```

1.2 Workflow with implicit workflows

A workflow with implicit workflows is built by defining *mainflow* and functions decorated with *appform.workflow*:

```
+-- views.py
+-+ templates/
  + job_template.j2
```

views.py:

```
from jobbergate import appform

def mainflow(data):
    return [appform.Text("jobname", "What is the jobname?", default="simulation")]

@appform.workflow
```

(continues on next page)

(continued from previous page)

```
def debug(data):
    return [appform.Confirm("debug", "Add debug info?")]

@appform.workflow
def gpu(data):
    return [appform.Integer("gpus", "Number of gpus?", default=1, maxval=10)]
```

job_template.j2:

```
#!/bin/bash
#SBATCH -j {{ data.jobname }}


{% if data.gpus %}
NUMBER_OF_GPUS={{ data.gpus }}
{% else %}
NUMBER_OF_GPUS=0
{% endif %}

{% if data.debug %}
/application/debug_prepare
{% endif %}

/application/run_application -gpus $NUMBER_OF_GPUS
```

1.2.1 API

Controller

Controller is for running code before and after workflows run.

All `pre_`/`post_`-functions takes a dict as an argument that is populated with all cumulated info from earlier `pre_`/`post_`, all previous questions and configuration file.

Should return a dict or `None`.

```
from datetime import datetime
from jobbergate import workflow

@workflow.logic
def pre_(data):
   # adds current datetime to data
   return {'datetime': str(datetime.now())}
```

Templates

Views

Simple view (with no workflow selection)

Views is built functions returning lists of questions. `mainflow` is the only expected function, others are all optional.

Functions that jobbergate calls gets all know data as inparameter as *data*.

Simplest *view.py*:

```
from jobbergate import appform

def mainflow(data):
    return [appform.Text('jobbname', 'What is the jobbname', default='MyJob')]
```

View with decorator workflow

Views can have a workflow "split" that gives the user an option to select a diferent path.

'view.py' with workflow defined with decorator. This give the user the question to select between debug and precision workflow. debug gives the boolean question "Add extra debug flags" and precsision gives an integer question regarding "Steps per mm".

```
from jobbergate import appform

def mainflow(data):
    return [appform.Text('jobbname', 'What is the jobbname', default='MyJob')]

@appform.workflow
def debug(data):
    return [appform.Confirm('debugoptions', 'Add extra debug flags')]

@appform.workflow
def precision(data):
    return [appform.Integer('precision', 'Steps per mm', minval=1, maxval=100)]
```

View with `nextworkflow` question

A view can have workflow selected by a question with the variable `nextworkflow`. This should be a List to give the user a list to select from. This should not have any function decorated with `@appform.workflow`.

```
from jobbergate import appform

def mainflow(data):
    return [appform.Text('jobbname', 'What is the jobbname'),
            appform.List('nextworkflow', ['precision', 'debug'])]

def debug(data):
    return [appform.Confirm('debugoptions', 'Add extra debug flags')]
```

(continues on next page)

(continued from previous page)

```
def precision(data):
    return [appform.Integer('precision', 'Steps per mm', minval=1, maxval=100)]
```

1.2.2 internal

1.2.3 Configuration

Configuration could be done in `config.py` as objects and selected via environment variable `APP_SETTINGS`. This could be done to have differente setting for developement, test, production etc. This file is part of the installation and should seldom be changed.

Configuration could also be done in `jobbergate.yaml`, which overrides configuration done in `config.py`. It only overrides the same variables, so if you have different variables in the files they are all going to be set.

The environment variable `JOBBERGATE_PATH` points to the directory where `jobbergate.yaml` resides, and could therefor point to a project or user configuration.

Flask configuration

To start flask in debug mode, set `FLASK_DEBUG` to `true`.

LDAP

Jobbergate uses `flask-ldap3-login` to be able to authenticate via LDAP and Active Directory. Configuration options is described at flask-ldap3-login.

The configuration could reside in both `config.py` and in `jobbergate.yaml`.

A configuration for Active Directory could look like this:

```
class ProductionConfig(BaseConfig):
    """Production configuration."""

    BCRYPT_LOG_ROUNDS = 13
    SQLALCHEMY_DATABASE_URI = os.environ.get(
        "DATABASE_URL", "sqlite:///{0}".format(os.path.join(basedir, "prod.db"))
    )
    WTF_CSRF_ENABLED = True
    LDAP_SEARCH_FOR_GROUPS = False
    LDAP_USE_SSL = True
    LDAP_PORT = 636
    LDAP_HOST = "ad.server.examlpe.com"
    LDAP_USER_DN = "OU=Users"
    LDAP_BASE_DN = "dc=ad,dc=server,dc=example,dc=com"
    LDAP_USER_LOGIN_ATTR = "cn"
    LDAP_USER_RDN_ATTR = "cn"
```

Jobbergate configuration

`jobbergate.yaml` has one section called `apps:` that has `path:` pointing to the directory containing all the applications.

`jobbergate.yaml` is also passed in the data structure flowing through the application as `data["jobbergateconfig"]`.

Instead of using `jobbergate.yaml`, `JOBBERGATE_PATH` can also be defined as a module name in an implemented application, for example, in its `__init__.py` file, declare such as `os.environ["JOBBERGATE_PATH"] = "myapp"`. After module myapp having been installed, Jobbergate can read in `myapp` as `JOBBERGATE_PATH`.

Application specific

You could have an application specific configuration file called `config.yaml` that is added to the data structure flowing through the application.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search