Jobbergate

Release 1.4.1

Mar 19, 2020

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.

CHAPTER 1

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
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

Appform

Abstraction layer for questions. Each classe represents different question types, and QuestionBase

class jobbergate.appform.**BooleanList**(*variablename*, *message*, *default=None*, *whentrue=None*, *whenfalse=None*)

Bases: *jobbergate.appform.QuestionBase*

Gives the use a boolean question, and depending on answer it shows *whentrue* or *whenfalse* questions. *whentrue* and *whenfalse* are lists with questions. Could contain multiple levels of BooleanLists.

Parameters

- **variablename** – The variable name to set
- **message** – Message to show
- **default** – Default value
- **whentrue** – List of questions to show if user answers yes/true on this question
- **whentrue** – List of questions to show if user answers no/false on this question

class jobbergate.appform.**Checkbox**(*variablename*, *message*, *choices*, *default=None*)

Bases: *jobbergate.appform.QuestionBase*

Gives the user a list to choose multiple entries from.

Parameters

- **variablename** – The variable name to set
- **message** – Message to show
- **choices** – List with choices
- **default** – Default value(s)

class jobbergate.appform.**Confirm**(*variablename*, *message*, *default=None*)

Bases: *jobbergate.appform.QuestionBase*

Asks a question with an boolean answer (true/false).

Parameters

- **variablename** – The variable name to set
- **message** – Message to show
- **default** – Default value

class jobbergate.appform.**Const**(*variablename*, *default*)

Bases: *jobbergate.appform.QuestionBase*

Sets the variable to the *default* value. Doesn't show anything.

Parameters

- **variablename** – The variable name to set
- **message** – Message to show
- **default** – Value that variable is set to

class jobbergate.appform.**Directory**(*variablename*, *message*, *default=None*, *exists=None*)

Bases: *jobbergate.appform.QuestionBase*

Asks for a directory name. If *exists* is *True* it checks if path exists and is a directory.

Parameters

- **variablename** – The variable name to set
- **message** – Message to show
- **default** – Default value
- **exists** – Checks if given directory exists

class jobbergate.appform.**File**(*variablename*, *message*, *default=None*, *exists=None*)

Bases: *jobbergate.appform.QuestionBase*

Asks for a file name. If *exists* is *True* it checks if path exists and is a directory.

Parameters

- **variablename** – The variable name to set
- **message** – Message to show
- **default** – Default value
- **exists** – Checks if given file exists

class jobbergate.appform.**Integer**(*variablename*, *message*, *minval=None*, *maxval=None*, *default=None*)

Bases: *jobbergate.appform.QuestionBase*

Asks for an integer value. Could have min and/or max constrains.

Parameters

- **variablename** – The variable name to set
- **message** – Message to show
- **minval** – Minumum value
- **maxval** – Maximum value
- **default** – Default value

class jobbergate.appform.**List**(*variablename*, *message*, *choices*, *default=None*)

Bases: *jobbergate.appform.QuestionBase*

Gives the user a list to choose one from.

Parameters

- **variablename** – The variable name to set
- **message** – Message to show
- **choices** – List with choices
- **default** – Default value

class jobbergate.appform.**QuestionBase**(*variablename*, *message*, *default*)

Bases: object

Baseclass for questions. All questions have variablename, message and an optional default.

Parameters

- **variablename** – The variable name to set
- **message** – Message to show
- **default** – Default value

class jobbergate.appform.**Text**(*variablename*, *message*, *default=None*)

Bases: *jobbergate.appform.QuestionBase*

Asks for a text value.

Parameters

- **variablename** – The variable name to set
- **message** – Message to show
- **default** – Default value

jobbergate.appform.**workflow**(*func=None*, *, *name=None*)

A decorator for workflows. Adds an workflow question and all questions added in the decorated question is asked after selecting workflow.

Parameters **name** – (optional) Descriptional name that is shown when choosing workflow

Add a workflow named debug:

```
@workflow
def debug(data):
    return [appform.File("debugfile", "Name of debug file")]
```

Add a workflow with longer name:

```
@workflow(name="Secondary Eigen step")
def 2ndstep(data):
    return [appform.Text("eigendata", "Definition of eigendata")]
```

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):
```

(continues on next page)

(continued from previous page)

```
    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')]

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

Workflow

Work flow module that could add pre and post functions to workflows

`jobbergate.workflow.`**`logic`**(*func=None*, *, *name=None*, *prepost=None*)

A decorator that registers functions as either pre or post to workflows.

Parameters **`name`** – (optional) Descriptive name that is used when choosing workflow

Hooking a pre-function to eigen implicit by function name:

```
# Hooking pre function to `eigen` implicit by function name
@logic
def pre_eigen(data):
    print("Pre function to `eigen` questions")
```

Explicit hooking post function to *eigen* workflow:

```
@logic(name="eigen", prepost="post")
def myfunction(data):
    print("Post function to `eigen` questions")
```

Pre and post that are run before and after all questions, respectively:

```
@logic
def pre_(data):
    print("Pre function that runs before any question")

@logic()
def post_(data):
    print("Post function that is run after all questions")
```

1.2.2 internal

cli

Creates dynamic CLI's for all apps

`jobbergate.cli.app_factory()`

This is the workhorse of cli module.

Click needs to have the code as a callback so we need to create the _callback function, which in turn returns a wrapper for each application. The app factory loops throug all the applications in the configed directory, and creates callbacks for each of them. This is the real workhorse in cli.

`jobbergate.cli.ask_questions(fields, answerfile, use_defaults=False)`

Asks the questions from all the fields.

Parameters

- **fields** (*list[jobbergate.appform.QuestionBase]*) – List with questions
- **answerfile** (*dict*) – dict with prepopulated answers
- **use_defaults** (*bool*) – option to use default value instead of asking, when possible

Returns all answers

Return type dict

`jobbergate.cli.flatten(deeplist)`

Helper function to flatten lists and tuples.

Parameters **deeplist** (*list*) – list of varying dept

Returns flattened list

Return type list

`jobbergate.cli.parse_field(field, ignore=None)`

Parses the question field and returns a list of inquirer questions.

Parameters

- **field** (*jobbergate.appform.QuestionBase*) – The question to parse
- **ignore** – function to decide if the question should be ignored/hidden

Returns inquirer question

Return type inquirer.Question

`jobbergate.cli.parse_prefill(arguments)`

Parses `-p/--prefill` command line arguments.

Parameters **arguments** (*list[string]*) – all arguments given to -p/–prefill

Returns dict with all command line answers

Return type dict

lib

Reads jobbergateconfig and declares functions neede for both web and cli version.

`jobbergate.lib.fullpath_import(path, lib)`

Imports a file from absolute path.

Parameters

- **path** – full path to lib
- **lib** – lib to import

views

The web part of jobbergate.

`jobbergate.views.application(application_name)`
route for /app/<application_name>

Parameters **application_name** – Name of application

Renders base questions for <application_name> and lets users answer them.

`jobbergate.views.applications()`
route for /apps/

Lets users select from available applications

`jobbergate.views.form_generator(application_name, templates, workflow)`
Generates form from workflow function

Parameters

- **application_name** (*string*) – Name of the application
- **templates** (*list[string]*) – List of availabe templates
- **workflow** – workflow function

Returns A populated QuestionaryForm

Return type FlaskForm

`jobbergate.views.home()`
route for /

Clears out session data and renders home.html template

`jobbergate.views.parse_field(form, field, render_kw=None)`
Parses the question field and populates a FlaskForm with the fields.

Parameters

- **form** (*FlaskForm*) – The form to populate
- **field** (`jobbergate.appform.QuestionBase`) – The question to parse
- **render_kw** – extra attributes for the field

Returns Form with all the fields

Return type FlaskForm

`jobbergate.views.renderworkflow(application_name, workflow)`
route for /workflow/<application_name>/<workflow>

Parameters

- **application_name** – application name
- **workflow** – workflow name

Renders <workflow> for <application_name> and lets user answer questions.

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 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

Index