Jenskipper Documentation

Release 0.0.1

Luper Rouch

Sep 27, 2018

Contents

Jenskipper is a tool to manage Jenkins from your VCS and the command line.

Similar tools already exist (jenkins-job-builder and job-dsl-plugin), but they are both based on domain-specific languages and try to abstract the jobs configurations. Jenskipper takes a more straightforward approach and works directly on Jenkins' XML, which has many advantages:

- Jenskipper can import your existing Jenkins jobs;
- all Jenkins plugins are supported in a consistent manner;
- the Jenkins GUI can still be used to edit or create new jobs, or learn how to configure new Jenkins plugins in Jenskipper;
- Jenskipper should be easier to learn.

The XML is extended by the powerful Jinja templating language, to permit factorization of build scripts and jobs.

Contents:

CHAPTER 1

Getting Started

In this section we will show you the basics to start working with Jenskipper.

1.1 Installation

Install jenskipper:

```
$ pip install jenskipper
```

1.2 Import your jobs

All commands are accessible via `jk`, type `jk --help` to display the integrated help. Start by importing your jobs:

```
$ jk import http://my.jenkins.server/ jenkins
```

This will import all jobs from `http://my.jenkins.server/` in the `jenkins` directory:

```
jenkins
|
+- jobs.yaml
|
+- contexts.yaml
|
+- pipelines.txt
|
`- templates
   |
   +- foo-tests.xml
   |
   `- bar-tests.xml
```

Now is probably a good time to add the imported repository to your favorite VCS.

Let's have a look at the files in the repository:

- `jobs.yaml` - the list of jobs that are managed by jenskipper; see *Jobs*;
- `contexts.yaml` - contexts for use in templates;
- `pipelines.txt` - a high-level view of how jobs are chained together; see *Defining jobs pipelines*;
- `templates/` - the jobs templates directory, containing the raw XML files pulled from the Jenkins server.

1.3 Start factorizing your jobs

Now you probably want to start factorizing your jobs configuration. The jobs are defined in the `jobs.yaml` file at the root of the repository, that should look like this:

```
foo-tests:
  template: foo-tests.xml

bar-tests:
  template: bar-tests.xml
```

Say you want to define a global email address where failure notifications must be sent. Open the `contexts.yaml` and define a new variable for the default context:

```
default:
  default_email: popov@company.com
```

This variable is then available in all templates through the Jinja templating language. Open `templates/foo-tests.xml` and look for the email notifications section:

```
<hudson.tasks.Mailer plugin="mailer@1.15">
    <recipients>popov@company.com</recipients>
    <dontNotifyEveryUnstableBuild>true</dontNotifyEveryUnstableBuild>
    <sendToIndividuals>false</sendToIndividuals>
</hudson.tasks.Mailer>
```

You can use the `default_email` variable by replacing `popov@company.com` with `{{ default_email }}`:

```
<hudson.tasks.Mailer plugin="mailer@1.15">
    <recipients>{{ default_email }}</recipients>
    <dontNotifyEveryUnstableBuild>true</dontNotifyEveryUnstableBuild>
    <sendToIndividuals>false</sendToIndividuals>
</hudson.tasks.Mailer>
```

If you want to use a different email address for a job, you can also override the context in `jobs.yaml`, for example:

```
foo-tests:
  template: foo-tests.xml
  context:
    default_email: bozo@company.com

bar-tests:
  template: bar-tests.xml
```

1.4 Push jobs to the server

To push your jobs to the server, you can use the `push` command. Note this will overwrite **all** the jobs on the servers, so make sure to give a heads up to your coworkers!

```
$ cd jenkins
$ jk push
```

You can also push only some jobs by specifying their names on the command line:

```
$ jk push bar-tests
```

If you want to preview changes before pushing them to the server, use the `diff` command:

```
$ jk diff bar-tests
```

Or to view the full rendered XML of a job:

```
$ jk show bar-tests
```

You can also trigger a build from the command line:

```
$ jk build bar-tests
```

You can even wait for the build to complete and display logs in case of error:

```
$ jk build bar-tests --block
```

1.5 Fetching new jobs from the server

If you want to pull new jobs from the server:

```
$ jk fetch-new
```

Note that you can't update existing jobs from the server. This is wanted, jenskipper operations are meant to be one way: after the initial import, Jenkins jobs are only updated from the jenskipper repository.

CHAPTER 2

Reference manual

2.1 Jobs

The `jobs.yaml` file at the root of the repository maps the files in the `templates/` directory to actual Jenkins jobs. It is a mapping with the job names as top level keys. Each entry is itself a mapping with the following keys:

template Required, the name of the template to render in the `templates/` directory.

default_context: Name of a context in `contexts.yaml` to pass to the template. If unspecified, defaults to `default`.

context Optional, a mapping containing extra context overriding `default_context`.

Extra jobs can be defined in the `extra-jobs.yaml` file. This can be useful in scenarios where jobs are generated by external tools.

2.2 Defining jobs pipelines

Sometimes you want to chain multiple jobs together. For example you might have a deploy job that must only be run after integration and unit tests succeeded.

In jenkins, you assemble pipelines by editing the jobs configurations, hooking jobs to each other one by one. This process is cumbersome, and it can be hard to visualize the whole pipeline.

Jenskipper solves this by separating the pipelines definitions from the jobs. Pipelines are defined in the `pipelines.txt` file. Here is an example chaining 3 jobs together:

```
unit-tests > integration-tests > deployment
```

By default the jobs chain is interrupted if one of the jobs fail. If you need to continue running jobs in the pipeline after a failure or an unstable result, use the `~>` and `?>` operators respectively:

```
unit-tests > integration-tests ~> deployment
```

Here the `deployment` job will be executed even if the `integration-tests` fails, but nothing will run if `unit-tests` fails.

A job may also trigger more than one job:

```
A > B > D
A > C
```

Or a job can be triggered by multiple jobs:

```
A > C
B > C
```

In this last example, `C` will be triggered if `A` *or* `B` finishes successfully.