Jenkins

CONFIGURATION AS CODE

JENKINS JOB BUILDER

BY CHYRKOV OLEKSANDR

Contents

Contents 1

I Configuration As Code 4

1 Preface 5
1.1 Configuration as code . . . 5
1.1.1 Advantages of configuration as code . . . 5
1.2 Jenkins . . . 7
1.2.1 Plugins . . . 7
1.3 Jenkins Job Builder . . . 8
1.4 Who this book is for . . . 9
1.5 Book Convention . . . 10
1.5.1 Software versions . . . 10
1.5.2 Conventions Used in This Book . . . 10
1.6 Examples . . . 11
1.7 Summary . . . 12

II Jenkins Job Builder 13

2 Jenkins Job Builder 14
2.1 Installation . . . 14
2.1.1 Install . . . 14
2.1.2 Unit Tests . . . 15
2.1.3 Documentation . . . 16
2.2 Configuration . . . 17

2.2.1 Configuration File . . . 17
2.2.2 job_builder section . . . 18
2.2.3 jenkins section . . . 19
2.2.4 hipchat section . . . 20
2.2.5 stash section . . . 21
2.2.6 __future__ section . . . 22
2.2.7 Working Configuration . . . 23
2.3 Running JJB . . . 24
2.3.1 Running Jenkins Job Builder . . . 24
2.3.2 Test Mode . . . 24
2.3.3 Updating Jobs . . . 24
2.3.4 Passing Multiple Paths . . . 25
2.3.5 Recursive Searching of Paths . . . 26
2.3.6 Excluding Paths . . . 26
2.3.7 Deleting Jobs and Views . . . 27
2.3.8 Globbed Parameters . . . 28
2.3.9 Providing Plugins Info . . . 29
2.3.10 Investigating Help . . . 30
2.3.11 Command Reference . . . 30
2.4 Working With Jobs . . . 34
2.4.1 Modules . . . 34
2.4.2 First Job . . . 37
2.4.3 Job Template . . . 40
2.4.4 Job Group . . . 42
2.4.5 Views . . . 44
2.4.6 Item ID . . . 46
2.4.7 Macroses . . . 48
2.4.8 Custom Yaml Tags . . . 50
2.4.9 Variable Refference . . . 55
2.4.10 Folder . . . 57
2.4.11 Anchors and Aliases . . . 59
2.5 More Examples . . . 62
2.5.1 Freestyle Project . . . 62
2.5.2 Pipeline Project . . . 65
2.5.3 Multijob Project . . . 69
2.6 Organize Repository . . . 73
2.6.1 Intro . . . 73
2.6.2 Repo Layout . . . 74
2.6.3 Structured Jobs Example . . . 75
2.7 Tips and Trics . . . 83
2.7.1 Custom Modules . . . 83

2.7.2 Multiple Configs 94
2.7.3 Auth Token 95
2.8 Developing JJB 96
2.8.1 Developing JJB 96
2.9 Summary 98

Bibliography **99**

Part I

Configuration As Code

1

Preface

1.1 Configuration as code

Infrastructure as code is the approach to defining computing and network infrastructure through source code that can then be treated just like any software system. Such code can be kept in source control to allow auditability and ReproducibleBuilds, subject to testing practices, and the full discipline of ContinuousDelivery. As configuration is written as source code, you can use all best development practices to optimise it, such as: creating reusable definitions of plans, parameterisation, using loops to create lots of different entities like plans, jobs, or repositories.

1.1.1 Advantages of configuration as code

- **Automation and standardisation:** As configuration is written as source code, you can use all best development practices to optimise it, such as: creating reusable definitions of plans, parameterisation, using loops to create lots of different entities like plans, jobs, or repositories.
- **Versioning of changes:** You can store configuration code in a version control system, such as Git, to see who changed what and when in your environment. You can use tags to mark versions. You can use branches to isolate changes under construction and to work in parallel streams without affecting your production environment.
- **Traceability of changes:** If source code is versioned and properly managed (e.g. tagged), you can track which changes have been ap-

plied to your source code. Analysing code differences (e.g. via git diff) is quite simple and efficient.

- **Smooth promotion of changes from test to production:** It's a lot easier to promote changes using configuration as code. In the "UI world", you had to click through many UI pages, test that everything works and next tediously repeat the same steps on the production. With configuration as code you can simply deploy plans to a test environment, verify changes and then deploy to the production environment just by changing the target URL.

- **Coding assistance and validation:** Editing build plans in an IDE (such as Eclipse or IntelliJ IDEA) allows to you use IDE features such as: code autocompletion, parameter tool tips, pop-ups with JavaDoc, code refactoring, searching for usages of a given method/object and many more. You can also quickly perform offline validation by compiling and running the code.

1.2 Jenkins

Jenkins is an open source automation server written in Java. Jenkins helps to automate the non-human part of software development process, with continuous integration and facilitating technical aspects of continuous delivery. It is a server-based system that runs in servlet containers such as Apache Tomcat. It supports version control tools, including AccuRev, CVS, Subversion, Git, Mercurial, Perforce, ClearCase and RTC, and can execute Apache Ant, Apache Maven and sbt based projects as well as arbitrary shell scripts and Windows batch commands. Builds can be triggered by various means, for example by commit in a version control system, by scheduling via a cron-like mechanism and by requesting a specific build URL. It can also be triggered after the other builds in the queue have completed. Jenkins functionality can be extended with plugins.

1.2.1 Plugins

Plugins have been released for Jenkins that extend its use to projects written in languages other than Java. Plugins are available for integrating Jenkins with most version control systems and bug databases. Many build tools are supported via their respective plugins. Plugins can also change the way Jenkins looks or add new functionality. There are a set of plugins dedicated for the purpose of unit testing that generate test reports in various formats (for example JUnit bundled with Jenkins, MSTest, NUnit etc.) and automated testing which supports automated tests. Builds can generate test reports in various formats supported by plugins (JUnit support is currently bundled) and Jenkins can display the reports and generate trends and render them in the GUI.
Plugins make Jenkins extremely useful and flexible depends on variety of needs.

1.3 Jenkins Job Builder

Jenkins Job Builder is open source command line tool which allows to describe Jenkins jobs in YAML format.You could template all your jobs in YAML format and use those templates for Jenkins master configuration. Templates could be stored under version control for making changes and auditing easier. Development team can work together with jenkins jobs in programmatic manner. No more uncontrolled changes via Jenkins UI and dealing with inexplicit XML configs.

1.4 Who this book is for

This book is aimed at Jenkins engineers who are looking for clear approach how to manage, test and deploy Jenkins jobs as code. Much of the material focuses on practical recipes. Throughout the book, reader will describe his Jenkins jobs in suitable templates, deploy them to Jenkins master continuously, version his jobs and test them. Moreover, recipes from this book helps reader understand all possible flexibility of **Jenkins Job Builder** and organize his jobs effectively.

1.5 Book Convention

1.5.1 Software versions

- **Server machine:** CentOS Linux release 7.3.1611.
- **Jenkins:** Jenkins ver. 2.109.
- **Jenkins Job Builder:** Jenkins Job Builder version: 2.0.2.
- **Python:** Python 2.7.5.

1.5.2 Conventions Used in This Book

- **Bold text:** Using for keywords highlighting.
- **Red color:** Using for links.

1.6 Examples

All examples templates and scripts are collected in public Github repository. You can simply clone git repo with templates and use them during read this book. Cookbook Examples.

1.7 Summary

The purpose of this book is to give some practical examples of how to use tools for developing Jenkins server configurations as code.

As the configurations grow bigger, it's becoming increasingly difficult to support them. When working on configurations as a team, it's very important to select a tool that will be right for teamwork.

Configuration as code is easier to comprehend. This approach allows to implement versioning via configurations described as code, placed under version control systems. This way configuration as code can be tested programmatically.

As a matter of fact, you can implement continuous integration for continuous integration system. Versioning lets you revert changes in configurations to working revision at any time.

Configuration as code helps you take the DRY approach, as separate parts of code can be used simultaneously in multiple configurations.

Part II

Jenkins Job Builder

2

Jenkins Job Builder

2.1 Installation

2.1.1 Install

To install **Jenkins Job Builder** from source, run:

```
$ pip install jenkins-job-builder==2.0.2
```

Verify **jjb** version:

```
$ jenkins-jobs --version

Jenkins Job Builder version: 2.0.2
```

2.1.2 Unit Tests

Unit tests have been included and are in the tests folder. They are included as examples of documentation so to keep the examples up to date. Unit tests cover every module and can be executed for different python versions. Below a few commands how run different unit tests. Firstly, we need install **tox** utility for tests.

```
$ pip install tox
```

Python 2.7 unit tests:

```
$ tox -e py27
```

Python 3.4 unit tests:

```
$ tox -e py34
```

Python 3.5 unit tests:

```
$ tox -e py35
```

Python 3.6 unit tests:

```
$ tox -e py36
```

Test coverage:

```
$ tox -e cover
```

Python linter:

```
$ tox -e pep8
```

2.1.3 Documentation

Documentation is included in the doc folder. To generate docs locally execute the command:

```
$ tox -e docs
```

The generated documentation is then available under **doc/build/html/index.html**. As over time URLs change or become stale there is also a testenv available to verify any links added. To run locally execute the command:

```
$ tox -e docs-linkcheck
```

Note:

When behind a proxy it is necessary to use **TOX_TESTENV_PASSENV** to pass any proxy settings for this test to be able to check links are valid.

2.2 Configuration

2.2.1 Configuration File

After installation, you will need to create a configuration file. By default, **jenkins-jobs** looks for **~/.config/jenkins_jobs/jenkins_jobs.ini**, **<script directory>/jenkins_jobs.ini** or **/etc/jenkins_jobs/jenkins_jobs.ini** (in that order), but you may specify an alternative location when running **jenkins-jobs**. The file should have the following format:

jenkins_jobs.ini

```
[job_builder]
ignore_cache=True
keep_descriptions=False
include_path=.:scripts:~/git/
recursive=False
exclude=.*:manual:./development
allow_duplicates=False

[jenkins]
user=jenkins
password=1234567890abcdef1234567890abcdef
url=https://jenkins.example.com
query_plugins_info=False
##### This is deprecated, use job_builder section instead
#ignore_cache=True

[plugin "hipchat"]
authtoken=dummy

[plugin "stash"]
username=user
password=pass
```

2.2.2 job_builder section

- **ignore_cache:** (Optional) If set to True, **Jenkins Job Builder** won't use any cache.
- **keep_descriptions:** By default **jenkins-jobs** will overwrite the jobs descriptions even if no description has been defined explicitly. When this option is set to **True**, that behavior changes and it will only overwrite the description if you specified it in the yaml. **False** by default.
- **include_path:** (**Optional**) Can be set to a **":"** delimited list of paths, which **jenkins job builder** will search for any files specified by the custom application yaml tags **"include"**, **"include-raw"** and **"include-raw-escape"**.
- **recursive:** (**Optional**) If set to **True**, **jenkins job builder** will search for job definition files recursively.
- **exclude:** (**Optional**) If set to a list of values separated by **":"**, these paths will be excluded from the list of paths to be processed when searching recursively. Values containing no / will be matched against directory names at all levels, those starting with / will be considered absolute, while others containing a / somewhere other than the start of the value will be considered relative to the starting path.
- **allow_duplicates:** (**Optional**) By default **jenkins-jobs** will abort when a duplicate **macro**, **template**, **job-group** or **job name** is encountered as it cannot establish the correct one to use. When this option is set to **True**, only a warning is emitted.
- **allow_empty_variables:** (**Optional**) When expanding strings, by default **jenkins-jobs** will raise an exception if there's a key in the string, that has not been declared in the input YAML files. Setting this option to **True** will replace it with the empty string, allowing you to use those strings without having to define all the keys it might be using.
- **print_job_urls:** (**Optional**) If set to True it will print full jobs urls while updating jobs, so user can be sure which instance was updated. User may click the link to go directly to that job. False by default.

2.2.3 jenkins section

- **user:** This should be the name of a user previously defined in Jenkins. Appropriate user permissions must be set under the Jenkins security matrix: under the **Global** group of permissions, check **Read**, then under the Job group of permissions, check **Create**, **Delete**, **Configure** and finally **Read**.

- **password:** The API token for the user specified. You can get this through the Jenkins management interface under
  People ->username ->Configure and then click the
  Show API Token button.

- **url:** The base URL for your Jenkins installation.

- **timeout:** (**Optional** The connection timeout (in seconds) to the Jenkins server. By default this is set to the system configured socket timeout.

- **query_plugins_info:** Whether to query the Jenkins instance for plugin info. If no configuration files are found (either in the default paths or given through the command-line), **jenkins-jobs** will skip querying for plugin information. True by default.

2.2.4 hipchat section

- **send-as:** This is the **hipchat** user name that will be used when sending notifications.
- **authtoken:** The API token necessary to send messages to hipchat. This can be generated in the hipchat web interface by a user with administrative access for your organization. This authtoken is set for each job individually; the **JJB Hipchat Plugin** does not currently support setting different tokens for different projects, so the token you use will have to be scoped such that it can be used for any room your jobs might be configured to notify. For more information on this topic, please see the Hipchat API Documentation.

2.2.5 stash section

- **username:** This is the **stash** user name that will be used to connect to stash when using the **stash** publisher plugin and not defining it in the yaml part.
- **password:** This is the related password that will be used with the **stash** username when using the **stash** publisher plugin and not defining it in the yaml part.

2.2.6 __future__ section

This section is to control enabling of beta features or behaviour changes that deviate from previously released behaviour in ways that may require effort to convert existing **JJB** configs to adopt. This essentially will act as a method to share these new behaviours while under active development so they can be changed ahead of releases.

- **param_order_from_yaml:** Used to switch on using the order of the parameters are defined in yaml to control the order of corresponding XML elements being written out. This is intended as a global flag and can affect multiple modules.

2.2.7 Working Configuration

First of all, let's install Jenkins server on our Linux machine for testing purpose.

```
$yum -y install java
$yum -y install https://pkg.jenkins.io/redhat/jenkins-2.109-1.1.noarch.rpm
$systemctl start jenkins
```

After several simple installation steps Jenkins server will be available on **127.0.0.1:8080** address. We'll use it for our test configurations. Just copy **jenkins job builder** config for our Jenkins master instance and you are ready for further work. Specify your actual username and password from Jenkins user. In my case it is jjb/jjb.

jenkins_jobs.ini

```
[job_builder]
ignore_cache=True

[jenkins]
user=jjb
password=jjb
url=http://localhost:8080
query_plugins_info=True
```

Note:

We have installed OpenJDK Java version 1.8 for testing purposes. For production needs is recommended to use Oracle Java.

2.3 Running JJB

2.3.1 Running Jenkins Job Builder

After it's installed and configured, you can invoke **Jenkins Job Builder** by running **jenkins-jobs**. You won't be able to do anything useful just yet without a configuration.

2.3.2 Test Mode

Once you have a configuration defined, you can run the job builder in test mode. If you want to run a simple test with just a single YAML job definition file and see the XML output on stdout:

```
1    $jenkins-jobs test /path/to/foo.yaml
```

You can also pass **JJB** a directory containing multiple job definition files:

```
1    $jenkins-jobs test /path/to/defs -o /path/to/output
```

Which will write XML files to the output directory for all of the jobs defined in the defs directory.

If you run:

```
1    $jenkins-jobs test /path/to/defs -o /path/to/output --config-xml
```

the output directory will contain config.xml files similar to the internal storage format of Jenkins. This might allow you to more easily compare the output to an existing Jenkins installation.

2.3.3 Updating Jobs

When youre satisfied with the generated XML from the test, you can run:

```
1    $jenkins-jobs update /path/to/defs
```

Which will upload the job and view definitions to Jenkins if needed. **Jenkins Job Builder** maintains, for each host, a cache of previously configured jobs and views, so that you can run that command as often as you like, and it will only update the jobs configurations in Jenkins if the defined definitions has changed since the last time it was run.

Note:

If you modify a job directly in Jenkins, **jenkins-jobs** will not know about it and will not update it.
To update a specific list of jobs/views, simply pass the job/view names as additional arguments after the job definition path. To update Foo1 and Foo2 run:

```
1    $jenkins−jobs update /path/to/defs Foo1 Foo2
```

You can also enable the parallel execution of the program passing the workers option with a value of 0, 2, or higher. Use 0 to run as many workers as cores in the host that runs it, and 2 or higher to specify the number of workers to use:

```
1    $jenkins−jobs update −−workers 0 /path/to/defs
```

2.3.4 Passing Multiple Paths

It is possible to pass multiple paths to **JJB** using colons as a path separator on *nix systems and semi-colons on Windows systems.

For example:

```
1    $jenkins−jobs test \
2      /path/to/global:/path/to/instance:/path/to/instance/project
```

This helps when structuring directory layouts as you may selectively include directories in different ways to suit different needs. If you maintain multiple Jenkins instances suited to various needs you may want to share

configuration between those instances (global). Furthermore, there may be various ways you would like to structure jobs within a given instance.

2.3.5 Recursive Searching of Paths

In addition to passing multiple paths to **JJB** it is also possible to enable recursive searching to process all yaml files in the tree beneath each path.

For example:

```
For a tree:
  /path/
    to/
      defs/
        ci_jobs/
        release_jobs/
      globals/
        macros/
        templates/

$jenkins-jobs update -r /path/to/defs:/path/to/globals
```

JJB will search **defs/ci_jobs**, **defs/release_jobs**, **globals/macros** and **globals/templates** in addition to the defs and globals trees.

2.3.6 Excluding Paths

To allow a complex tree of jobs where some jobs are managed differently without needing to explicitly provide each path, the recursive path processing supports excluding paths based on absolute paths, relative paths and patterns.

For example:

```
For a tree:
  /path/
    to/
      defs/
        ci_jobs/
          manual/
```

```
7                release_jobs/
8                  manual/
9                qa_jobs/
10             globals/
11               macros/
12               templates/
13               special/
14
15     $jenkins-jobs update -r -x man*:./qa_jobs \
16       -x /path/to/defs/globals/special \
17       /path/to/defs:/path/to/globals
```

JJB will search the given paths, ignoring the directories **qa_jobs**, **ci_jobs/manual**, **release_jobs/manual**, and **globals/special** when building the list of yaml files to be processed. Absolute paths are denoted by starting from the root, relative by containing the path separator, and patterns by having neither. Patterns use simple shell globing to match directories.

2.3.7 Deleting Jobs and Views

Jenkins Job Builder supports deleting jobs and views from Jenkins. To delete a specific job:

```
1    $jenkins-jobs delete Foo1
```

To delete a list of jobs or views, simply pass them as additional arguments after the command:

```
1    $jenkins-jobs delete Foo1 Foo2
```

To delete only views or only jobs, simply add the argument **views-only** or **jobs-only** after the command:

```
1    $jenkins-jobs delete --views-only Foo1
2    $jenkins-jobs delete --jobs-only Foo1
```

The **update** command includes a **delete-old** option to remove obsolete

jobs:

```
$jenkins-jobs update --delete-old /path/to/defs
```

Obsolete jobs are jobs once managed by **JJB** (as distinguished by a special comment that JJB appends to their description), that were not generated in this **JJB** run. There is also a command to delete all jobs and/or views.

WARNING: Use with caution.

To delete all jobs and views:

```
$jenkins-jobs delete-all
```

To delete all jobs:

```
$jenkins-jobs delete-all --jobs-only
```

To delete all views:

```
$jenkins-jobs delete-all --views-only
```

2.3.8 Globbed Parameters

Jenkins Job Builder supports globbed parameters to identify jobs from a set of definition files. This feature only supports **JJB** managed jobs. To update jobs/views that only have 'foo' in their name:

```
$jenkins-jobs update ./myjobs \*foo\*
```

To delete jobs/views that only have 'foo' in their name:

```
$jenkins-jobs delete --path ./myjobs \*foo\*
```