Version Control Your Jenkins Jobs with Jenkins Job Builder

Wayne Warren

wayne@puppetlabs.com

Abstract

Puppet Labs uses Jenkins to automate building and testing software. While we do derive benefit from this automation, there is a maintenance burden involved in keeping Jenkins configured correctly. The Jenkins Web UI can be tedious to use, even when managing a small number of jobs (Jenkins' unit of configurable work). The complexity of managing the system continues to increase as the number of projects, versions, and developers increases.

To address this burden, Puppet Labs has invested significant engineering time and effort into understanding, improving, and applying Jenkins Job Builder (JJB) to manage job configurations. JJB is a Python tool developed by the Openstack Infra developers whose features include: modular support for Jenkins plugins, support for parameterized job templates, and deletion / creation of managed jobs. Using it has helped mitigate configuration drift between jobs and pipelines, identify and decommission forgotten jobs, and establish a review process for Continuous Integration (CI) changes.

This paper explains how to use JJB to manage and version control Jenkins configuration. Readers should expect to walk away with an understanding of the following key concepts:

- Generic job/pipeline types (unit, packaging, component integration, system integration)
- Language and framework specific implementations of generic job/pipeline types
- Job template name guidelines for optimal reuse
- Configuration testing and deployment strategies

I will conclude with a brief preview of the future direction of Jenkins Job Builder development.

Biography

Wayne is a Quality Engineer dedicated to improving and streamlining Puppet Labs' approach to automating common test and build tasks related to the validation and release of Puppet Enterprise and its component projects. He has experience with embedded Linux kernels and file systems, developing functional test frameworks for embedded systems, and automating Jenkins configuration using JJB.

1 Introduction

Jenkins is a continuous integration (CI) tool that automates various build, test, and packaging steps. Because managing configuration for Jenkins jobs can be tedious, repetitive, and error-prone when using the Jenkins Web UI due to its click-heavy interface.

What might we look for in an alternative system for managing Jenkins job configuration? At a high level, an alternative must include changes that can be version controlled, tested out-of-band, and reviewed by teammates before being deployed. We should also be able to optimize configuration reuse between projects with similar CI needs as well as between Jenkins instances since we run multiple public and private Jenkins instances.

Jenkins Job Builder (JJB) is a set of Python scripts created and maintained by the OpenStack Infra developers. It takes an arbitrary number of YAML files that are processed according to Domain Specific Language (DSL) rules to produce a list of Jenkins job dictionaries. These dictionaries are used to generate XML in a modular fashion for the Jenkins job they represent. Modularity, in this context, refers to the way in which XML is generated on a per-plugin basis and makes it possible to add support for additional Jenkins plugins. Once all dictionaries have been converted to XML, JJB then uses Jenkins' HTTP API to POST these configurations to the specified Jenkins server to either create new jobs or update existing ones.

This paper aims to describe the JJB DSL, propose a review process, and introduce conventions for optimizing reuse of job configuration.

2 Background

2.1 Continuous Integration

Continuous Integration in software development refers to the practice of continuously merging feature code from developer branches into a designated “mainline” branch of a software project. A continuous integration service is one that provides automation to facilitate continuous integration by building and testing whenever a commit is merged into a software project. This automation ensures that builds and tests are run in a uniform environment at a consistent cadence that allows developers to discover regressions before software is shipped to users.

2.2 Jenkins

Jenkins is an open source, continuous integration service with a master/slave architecture. The “master” is used to manage configuration and queuing of jobs and “slaves” are used to execute the actionable content of those jobs. A “job” is a reusable, configurable unit of work that can be scheduled to run using different types of “triggers”. Jobs can be configured to perform various types of tasks including shell scripts, rake tasks, maven tasks, or many other build/test/integrate-oriented snippets of automatable behavior.

Jenkins provides various types of configuration to modify its behavior:

- **global**
  system configuration that manages the overall behavior of the Jenkins master
- **security**
  security-sensitive configuration for the Jenkins master
- **view**
  all configuration specific to a particular Jenkins view
- **job**
  all configuration specific to a particular job

Jenkins “core” is a daemon providing pluggable interfaces that can be used by auxiliary software (aka “Plugins”) to implement additional behavior for either the service (master and slaves) as a whole, or more commonly for individual jobs.

Jenkins is primarily designed to be configured through its web UI, which is a manual process requiring navigation of web forms. It also provides a command-line tool and an HTTP API. As of Jenkins 1.615, the HTTP API has documented support for configuration of **jobs** and **views**. Jenkins also has endpoints that accept JSON for **global** and **security** configuration that are used by its web UI, but there is no documented support for these endpoints to be used programmatically.

2.2.1 Problems with the Jenkins Web UI

Users typically configure Jenkins for the sake of modifying specific job behavior. This can include job timeout, log retention, Jenkins slave pinning, task execution, setting relationships between jobs, software configuration management (SCM) behavior, and many other types of behavior. These changes may need to be made for different reasons at different stages in an organization's growth or in a specific project's lifecycle. Because of this, the scope of the changes being made may not be limited to a single job—they may be a matter of policy or of differentiation between new and previous development cycles. It may be that an entire “pipeline” of jobs may need to be duplicated and modified to suit a slightly different need.

Whatever the reason for configuring Jenkins jobs, its Web UI is considered by many to be outdated and difficult to navigate. Problems we have found include:

- Difficulty navigating configuration web elements when attempting to investigate and modify a single job (let alone many jobs that constitute a pipeline of test, build, and packaging steps).
- Configuration drift between pipelines for projects with very similar build and testing requirements is inevitable since each job must be modified individually.
- Changes made by individual team members may lead to configuration or automation policy violations since the rest of the team does not have the opportunity to review and approve of changes before they are made.

2.2.2 Seeking Alternatives

So what might we look for in an alternative system for managing Jenkins job configuration? At a high level, there are a number of workflow and technical requirements that needed to be met to reduce the pain of interacting with Jenkins.

Workflow

- Must be able to version control our Jenkins configuration.
- Testing of configuration must be able to happen without affecting our production systems.
- Changes must be reviewed and approved by our team the same way that developers review code before merging.

Technical

- Must be able to optimize configuration reuse between projects with similar CI needs.
- Should be able to share configuration between Jenkins instances.

3 Jenkins Job Builder

After evaluating several alternatives side-by-side, comparing the pros and cons of each, we found that Jenkins Job Builder (JJB) met our requirements without departing from the workflows our developer and QA teams were accustomed to.

JJB is a command-line tool designed to simplify Jenkins interactions by enabling users to configure **job**s using human editable YAML. The sequence of events for updating jobs with JJB is roughly as follows:

1. Read all specified YAML files in as Python dictionaries.
2. Apply JJB DSL rules to the given dictionaries:
    1. Expand job templates
    2. Apply defaults to all jobs
    3. Interpolate variables specified at the project, job-group, and job-template levels.
3. For each resulting job dictionary:
    1. Interpolate variables and expand macros in-place
    2. Generate XML using JJB's Jenkins plugin modules
4. POST each XML string to the Jenkins HTTP API to either create a new job or update an existing job.

The key feature that makes this a viable alternative to the Jenkins Web UI is the use of YAML to store job configurations. The YAML can be placed under version control, allowing it to be reviewed using, for example, the Github Pull Request (PR) model of code review. The use of Jenkins' HTTP API to perform the configuration update means relatively few changes were necessary to our Jenkins deployments to implement this new approach to configuring jobs. It also means that configurations can be easily shared between Jenkins instances since deploying to a different instance only requires a slightly different configuration be passed to JJB. Finally, the hierarchical organization of jobs using JJB's “job-group” and “project” concepts, in combination with the “job-template” concept, means that configurations can be reused between jobs with similar CI requirements.

JJB does not currently support **global**, **security**, or **view** configurations. The following sections explore some of JJB's features in more depth.

3.1 Modular XML Generation

JJB's most prominent feature is its modular approach to XML generation in which support for each Jenkins Plugin is implemented as a function that generates the necessary XML snippet. Such XML snippets are inserted in the appropriate place in each job's XML configuration structure. Supporting additional plugins involves the following steps:

1. Add a function to the appropriate Python module in JJB's source; the module corresponds to the type of behavior (ie, “builder”, “publisher”, “trigger”) this plugin supports and the function corresponds to the plugin itself.
2. List that function in JJB's setup.cfg entry points. This enables the XML generating code to know about it at runtime.
3. Document, via doc strings on the function, the dictionary keys necessary to configure that particular plugin.

3.2 Plugin Version Detection

Complementing JJB's modular XML generation is its use of the Jenkins HTTP API to query the targeted Jenkins instance for a list of installed plugins and their versions. This allows JJB modules to conditionally generate XML structures based on the versions of plugins installed on the target Jenkins instance.

3.3 The JJB DSL

The JJB DSL provides the ability to reuse configuration snippets between jobs and pipelines. It provides CI configuration abstractions that allow projects and their jobs to be parameterized, hierarchically organized, and editable in any plaintext editor.

- **project**
  JJB “project” is the top-level object that instantiates all job templates and groups.
- **job-template**
  JJB “job-template” is a YAML dictionary (aka “hash” or “map” depending on the programming language) containing keys that map to Jenkins plugins and values that may include template strings.
- **job-group**
  JJB “job-group” is similar to a “project” except that it is reusable between different projects
- **defaults**
  JJB “defaults” is where default JJB variables can be set.
- <**macro**>
  Macros are top-level JJB elements which can be referenced within specific sub-elements of a job-template.

3.3.1 Templates

JJB utilizes the concept of “job-template” to define parameterized jobs that can be reified at run time with variables passed to them by either the “job-group” or “project” where they are used. The “job-group” and “project” concepts are DSL elements which provide the ability to group lists of jobs and job templates, and to override JJB variables used by job templates.

3.3.2 Macros

Macros are snippets of JJB code that get expanded in place where used in “job-template” and “job” objects. Like job-templates, these provide a DRY (Don't Repeat Yourself) element to job configuration, reducing the number of places where changes need to be made when modifying the behavior of many jobs at once (for example, setting build log retention policy).

3.3.3 Defaults

JJB defaults are a way of setting default values for variables that are interpolated into job templates at run time. Having a single place to set these values is important because it reduces the need to unnecessarily set default values on each job-template. Defaults have the lowest precedence when variable overrides occur between defaults, job-group, project, and job-template items in the JJB DSL.

3.4 Job Deletion

It is possible to use shell glob syntax to delete jobs managed by JJB from the command line. Combined with a thoughtful approach to naming job-templates this can be used to target pipelines specific to projects, branches of projects, or even categories of jobs.

3.5 Test Output

The JJB command line tool provides a flag that will cause it to run in “test” mode (as opposed to “update” mode). In test mode, JJB will either:

- emit all of its generated XML to stdout
- or given a directory name, create one file per job generated whose contents is the XML that would be used to update that job in update mode and whose name is the name of the job as it is referred to in Jenkins itself.

This test output mode is crucial to both development of JJB itself, where we use the output to ensure consistent output between one revision of JJB and the next for OpenStack's project configs, and for reporting changes between one revision and the next of an organization's Jenkins configs.

4 A Version-Controlled JJB Workflow

This section assumes some familiarity with general version control system (VCS) concepts such as “branches” and “commits”. Examples use the Git VCS, in particular the pull request model of review and code merge used by Github.

The basic idea is that since JJB enables jobs to be defined using YAML rather than XML, these files can be edited by humans and stored in a VCS repository in the same way that developers version control source code for software projects. The primary benefit of version controlling Jenkins Jobs in this way is that a team can then automatically synchronize live Jenkins jobs with their VCS copy--thus mitigating the potential for configuration drift between similar jobs edited asynchronously through the Jenkins web UI. If a team member wishes to make a permanent change to a particular set of jobs, they must follow the same process the team uses for making changes to source code for a given project.

4.1 Pull Requests (PR)

Pull Requests are a Github-specific development workflow item in which a user “submits a PR” containing their intended changes as a series of commits against the main line of development of a project. This PR workflow alerts other developers working on the same project that there is a change waiting to be merged and allows them to view comment on patches against various files in the PR. Github also provides an API by which external software may react against particular events on repositories of interest. For example, Jenkins can listen using this API for PRs against a repository and trigger CI jobs to run tests before changes are merged. This general PR workflow is exactly what Puppet Labs uses to allow our CI team to review changes before merging.

4.2 Testing

Puppet Labs stores its JJB configuration in a single monolithic git repository. Whenever someone makes a change to this repo, they submit a Github PR which triggers a Jenkins job that tests this PR and reports back to the PR with a SUCCESS/FAILURE message and a link back to the Jenkins job that ran. The only test that we currently use to validate our changes performs the following steps:

For each Jenkins instance managed by the config repo:

- Generate Jenkins job config XML from the master branch, output to a local directory using JJB’s “test” subcommand.
- Generate Jenkins job config XML from the PR branch, output to a local directory using JJB’s “test” subcommand.
- Use the “diff” command line tool to recursively compare the two output directories, save this information in a file named *<jenkins-instance-name>.diff*
- Archive as artifacts on the jenkins jobs all files that match the pattern “*.diff”.

These “diff” output files allow us to compare the differences in XML generated between revisions of the configuration repo as well as to detect when jobs are deleted or created.

4.3 Deployment

As with any version-controlled software, the end goal of JJB configuration changes is to be merged into the mainline branch of development. Once this happens in Puppet Labs' JJB configuration repository we have automated deploy jobs in Jenkins that are triggered to push changes out to all affected JJB-managed Jenkins instances ensuring that new jobs are created and existing jobs are updated to reflect the new state of the JJB config master branch.

In addition to this per-merge deploy, we also run JJB twice-daily to ensure a high degree of consistency between merges. This allows developers to make changes to jobs asynchronously during the day to troubleshoot or debug CI prior to proposing more permanent changes with additional PRs against the JJB configuration repo.

4.3 Benefits

So the theory seems sound, but how does it hold up in the real world? JJB's templating system and ability to group jobs together has led to an increase in individual CI Automation engineer productivity by allowing CI pipelines to be designed once and deployed many times. The initial design time for a pipeline tends to take a bit longer than if everything were done using the Jenkins Web UI, but once that pipeline template exists it is fairly trivial to deploy a copy of that template for new projects as long as those projects follow the conventions expected by the CI jobs.

5 Related Work

5.1 Job DSL & Workflow Plugins

The Job DSL Plugin makes available a new type of build step when configuring jobs. This build step allows the creation, configuration, and triggering of other jobs—all using the Groovy scripting language. This essentially enables the use of “meta” style jobs that manage the workflow of an entire “pipeline” of jobs. While this did meet some of our technical criteria—we could easily reproduce groups of jobs for different branches or projects—it didn't meet our workflow needs. Specifically, this plugin does not allow us to version control our Jenkins configuration or review changes made by teammates before those changes go into production.

The Workflow Plugin is similar to the DSL Plugin in that it provides a sort of “meta” facility of organizing jobs in a pipeline. It also uses Groovy to describe this organization. However, it suffers from the same drawbacks as the DSL Plugin, namely that does not allow version control or review of changes made by teammates before those changes go into production.

It's also worth mentioning that neither of these plugins takes us outside the Jenkins Web UI even if they do make interactions with the web ui more productive.

5.2 Templated XML Files

Prior to investigating existing alternatives to managing jobs purely through the Jenkins we had for some of our CI pipelines implemented a series of templated XML files that we could interpolate with variables and POST to the Jenkins HTTP API. This allowed us to relatively easily reproduce this same pipeline exactly for different projects. It also allowed job configuration to be kept in version control.

However, it lacked generalizable templates that could be reused for different purposes or within the context of different types of CI pipelines. It also moved the complexity of interacting with the Jenkins Web UI into the complexity of dealing with templated XML.

6 Conclusion

Jenkins is an automation service that improves the quality of continuous integration work flows for software projects by allowing them to automate testing, building, packaging, and in some cases deploying changes as they are merged into version control repositories. However, the Jenkins user experience and scalability to more than a few simple pipeline setups suffers from its unintuitive and tedious web interface.

Ideally, developers working on new software projects should not have to expend significant time or energy understanding the complex Jenkins machinery to set up CI when other projects have similar test/build/packaging needs—it should be a simple matter of naming the pipeline type, submitting a PR to

a configuration repo, and obtaining approval from teammates with more Jenkins knowledge. Additionally, when all that is necessary is to make a simple change to an existing pipeline, the behavior affected by that change should be represented by a well-documented set of parameters that can be changed without that developer having to understand all the workings under the hood.

Jenkins Job Builder allows us to achieve the necessary level of abstraction to consistently reproduce and maintain CI pipelines for specific project types. We also avoid the problems of configuration drift, unapproved changes, and the tedium of managing all our jobs with the Jenkins Web UI. Many of these benefits follow directly from being able to version control our configuration templates using the Git source control management tool.

7 Future Work

The JJB version 1.x and earlier library is only really intended to be used for the JJB command line tool. However, there is an effort within the Openstack Infra developer group to clean up the library with the intent of producing a programmatic API to allow JJB users to produce their own tools as well as to configure Jenkins directly using Python if they wish. This is the main development effort for JJB targeting the 2.x version series.