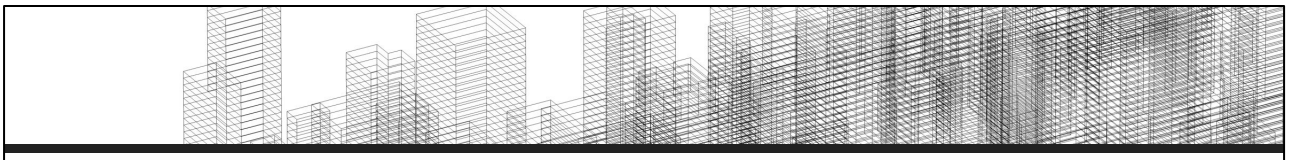


CI/CD for Fedora packaging with Zuul

Matthieu Huin, Fabien Boucher

Good afternoon everybody

We are Fabien Boucher and Matthieu Huin. Our day to day work is to maintain some Zuul deployments at Red Hat. Beside operations, we help people get started with Zuul but also we contribute code to the Zuul project.

In this talk, we will explain why we would like to propose a new CI/CD system to the Fedora community, especially for the packaging context, and what we have done so far.

What's a Zuul?

- Open Source CI/CD system well adapted to the Pull Request workflow
  - Supports Gerrit, Github, **Pagure**, *Gitlab*
  - **One pipeline per PR event**
  - **Cross repository testing**
  - Jobs dependencies with artifacts sharing
  - CI-as-Code
    - Jobs are **YAML definitions + Ansible** in Git repositories
    - Shared jobs libraries
    - Speculative testing on CI changes
  - On demand job execution environments
    - VMs (Openstack, AWS, …)
    - Containers (K8S, Openshift)

First let's introduce Zuul quickly. Zuul is a relatively new player in the CI/CD world, but is quickly gaining traction with its features:

Zuul is an Open Source CI/CD system that is adapted to code review workflows such as Pull Requests. It:

- Supports Gerrit, github, pagure, and soon gitlab.
- Implements event-driven pipelines: Zuul is listening to events of Code Review systems, so jobs can be configured to be triggered on specific events.
- Supports natively cross-repository testing, in other words dependencies between un-merged Pull-Requests
- Supports jobs inheritance, jobs dependencies and jobs chaining with artifacts sharing.
- Is CI-as-code
  - indeed, 99% of the CI configuration is stored as YAML in Git repositories. Jobs are Ansible playbooks so no agent is needed on test nodes, only SSH + Python. It also means access to the Ansible ecosystem and existing roles.
  - Changes on CI configuration are validated and gated by Zuul.
  - jobs can be easily shared and re-used as libraries, Zuul only needs the Git URL of the library.
- Manages the lifecycle of resources used by jobs: Zuul will spawn environments as and when needed, and destroy them to free resources up once a job concludes. Some of the supported environment providers are OpenStack, AWS, Kubernetes and OpenShift.

All these features make Zuul a good fit for handling the CI of RPM packages.

Zuul for Packaging

- Already implemented two workflows for RPM packaging based on Zuul
  - RDO
  - Software Factory
- Why not try for Fedora then?

And we can attest this from experience:

- We've actually already implemented a CI workflow for packaging based on Pull Requests and Zuul not once, but twice.
  - RDO, for packaging OpenStack on CentOS
  - Software Factory, for packaging CI tools like Zuul on CentOS

We thought this could be something the Fedora Community would like to try out, so we decided to set this workflow up for Fedora packagers to experiment with.

What are the gains ?

- A more familiar, Pull Request-based workflow for newcomers
- More automation, from creating a Pull Request to publishing the package
- Testing on demand
- Build and runtime dependencies handling in CI
- A more open CI system to which every packager can contribute
- Easier integration with third-party CIs (Flavors/Vendors)

This new CI workflow would bring many benefits to Fedora:

- It revolves around the Pull Request workflow, with which open source contributors are usually already familiar. This may help with making it easier for newcomers to start contributing to Fedora's packaging.
- It opens more possibilities for automation at every step of a Pull Request's life cycle.
- Even if they aren't following the PR workflow, experienced packagers can still use it to trigger testing on demand before a package is published.
- This system allows contributors to test dependencies together before they're published, which is a huge gain of time.
- The fact that most of the CI can be defined with YAML files, and jobs are Ansible playbooks, should also lower the entry cost to contribute, thus making CI everybody's business.
- The workflow also allows for easier integration with the CI validation of third parties like other Fedora flavors and vendors.

How do Zuul and Pagure interact?

- Pagure → Zuul
  - Zuul listens to events related to Pull Requests and git repositories on Pagure
    - opened, updated, flagged, tagged, commented, git ref updated, …
- Zuul → Pagure
  - Zuul acts on Pull Requests with job results
    - comment, flag, …

Let's get a bit more technical about the current implementation.

Zuul listens to Pagure's event hook system to trigger its pipelines, and makes calls to the web API of Pagure.

More in details:

- Zuul runs a web server that listens for webhook calls related to events happening on Pull Request and Git repositories, then it triggers configured jobs for the relevant pipeline, for the related events and repositories.
- And once jobs are done, Zuul reports jobs results and CI status via the Pagure API on the related Pull-Request.

A Zuul pipeline

```
- pipeline:
    name: check
    manager: independent
    require:
      src.fedoraproject.org:
        merged: False
    trigger:
      src.fedoraproject.org:
        - event: pg_pull_request
          action: comment
          comment: (?i)^\s*recheck\s*$
        - event: pg_pull_request
          action:
            - opened
            - changed
    start:
      src.fedoraproject.org:
        status: 'pending'
        status-url: "https://fedora.softwarefactory-project.io/zuul/status.html"
        comment: false
    success:
      src.fedoraproject.org:
        status: 'success'
    failure:
      src.fedoraproject.org:
        status: 'failure'
```

Here we can see a sample of a definition of a “check” pipeline.

The trigger part defines which events make the Pull-Request’s jobs executed in that pipeline. Here the pipeline is triggered for the following events: opened, changed and comment with the recheck term.
The require part defines the attributes a Pull-Request must own to enter the pipeline. Here the Pull Request must be open to get triggered.
The start/success and failure stanza tell Zuul about the action to perform according to the jobs’ results, such as to report comment and CI status.

Where is it hosted ?

CI configuration hosted on pagure.io

- fedora-zuul-jobs-config
- zuul-distro-jobs
- fedora-zuul-jobs

Zuul instance is hosted by the Software Factory project

- https://softwarefactory-project.io
- Resources shared with projects such as RDO, Distributed-CI, Packit, Software Factory, ...

Images managed by the Software Factory operators

In order to propose a ready to use system, we have implemented a flexible and customizable workflow around a suite of jobs. The jobs are hosted on pagure.io through three Git repositories:

- Fedora-zuul-jobs-config: this repository defines which jobs, in which pipeline must be triggered for a given repository. Also it defines some jobs that need a heightened security context from Zuul like those using secrets (like publishing).
- Zuul-distro-jobs: A generic library of Zuul jobs related to RPM packaging and publication. The library brings support for Koji, Copr, Mock, rpminspect, and others.
- Fedora-zuul-jobs: In this last repository, we find all other jobs specific to Fedora and usually inherited from the zuul-distro-jobs library

Computing resources are provided by softwarefactory-project.io, that is maintained by our team. Resources are shared, on a single Zuul deployment, with other projects such as RDO, Distributed-CI, Packit and the Software Factory project itself.

CI configuration

- Jobs
  - **rpm-scratch-build** : SRPM build from PR content, scratch build on Koji, retrieve artifacts (RPMs and build logs)
  - **rpm-build** : regular build on Koji
  - **rpm-linter** : runs rpmlint on RPMs artifacts
  - **rpm-rpminspect** : runs rpminspect on RPMs artifacts
  - **rpm-test** : runs the embedded (in distgit) test/tests.yml playbook

Now let's talk about the jobs we wrote to provide a flexible packager workflow:

- First, rpm-scratch-build, it builds a SRPM, then run a RPM scratch build on Koji and finally retrieve the artifacts from Koji.
- Secondly, rpm-build, it simply runs a regular build on Koji.
- Then, rpm-linter and rpm-rpminspect, are jobs that run respectively the rpmlint and the rpminspect commands.
- Finally, rpm-test, this is a job that runs the test playbook that is embedded in the distgit repository.

CI configuration

- Pipeline “recipes” (project-templates)
  - **build (**check pipeline)
    - run **rpm-scratch-build** job
    - Expose artifacts
  - **lint** (check pipeline)
    - Requires Build’s artifacts
    - Run the **rpm-linter** and **rpm-rpminspect** jobs
  - **test** (check pipeline)
    - Requires Build’s artifacts
    - run the **rpm-test** job
  - **publish** (gate, promote pipelines)
    - merge the Pull Request and run the **rpm-build** job

```
- project-template:
    name: build
    check:
      jobs:
        - rpm-scratch-build

- project-template:
    name: lint
    check:
      jobs:
        - rpm-linter:
            dependencies:
              - rpm-scratch-build
        - rpm-rpminspect:
            voting: False
            dependencies:
              - rpm-scratch-build
```

```
- project:
    name: rpms/python-gear
    templates:
      - build
      - lint
      - test
      - publish
```

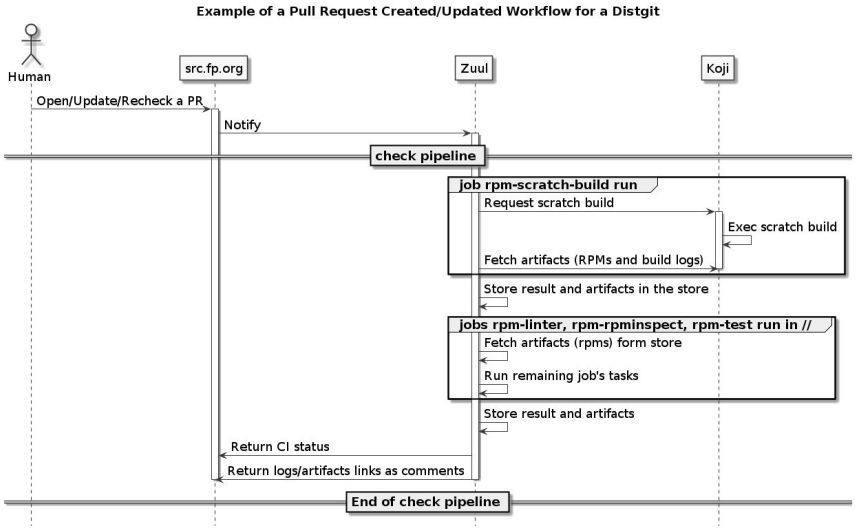
Each jobs is meant to run at a specific phase during the life cycle of a Pull Request, that’s why we wrote “project-templates” that can be seen as recipes that associate jobs to pipelines.

On the top right, you can see the definition of the build and lint project-templates and at the bottom you can see how we attach a list of template to a repository.

So we wrote the following recipes:

- First, the build template that runs, in the check pipeline, the Koji scratch build and expose the built RPMs to child jobs.
- Secondly, the lint template that runs, in the check pipeline, the linter jobs, rpmlint and rpminspect, by using as a context the RPMs exposed by the parent job. Note the usage of the “dependencies” keyword in the project-template definition. Here it means the rpm-scratch-build job is the parent job.
- Then, the test template that runs, in the check pipeline, the embedded functional tests, contained in the distgit, by using as a context the RPMs exposed by the parent job.
- Finally, the publish template, through the gate and promote pipelines, that merges the Pull Request and runs the rpm-build job.

Creating a Pull Request



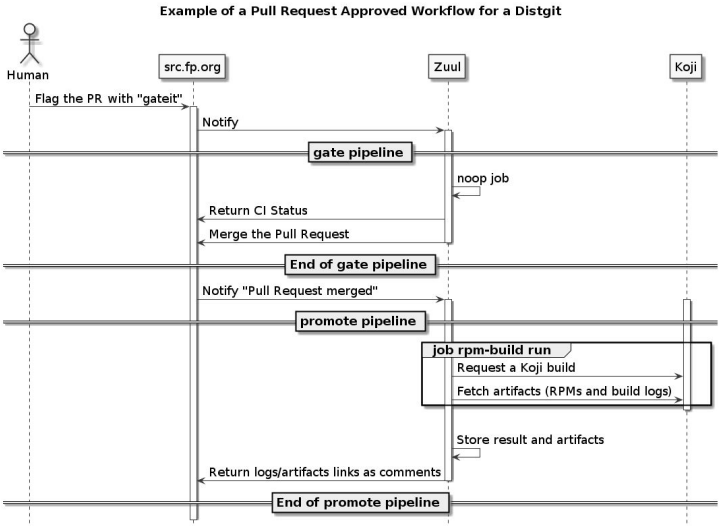
In this slide, let's see what's the workflow we have defined when a Pull Request is created, updated or "recheck".

1. First, a contributor creates a Pull-Request
2. Then, Zuul get notified of that event
3. Due to that, Zuul triggers the check pipeline and start the rpm-scratch-build job
4. The job runs a Koji scratch build and retrieve the artifacts from Koji
5. Once the job is done, Zuul stores the RPMs in a public storage

As rpm-linter, rpm-rpminspect and rpm-test jobs were defined with a dependency on rpm-scratch-build, then,

6. Zuul starts them, in parallel
7. Each job, fetch the rpms from the store and run their others tasks
8. Once each jobs are done, Zuul stores their artifacts in a public storage
9. And finally, for each jobs, Zuul returns a link to the job result page and sets a CI status (failure or success) on the Pull Request.

Merging a Pull Request



The second part of the workflow is around the Pull-Request approval and package publication.

Let's see how this behaves:

- To approve a Pull Request, a packager sets the Pull Request tag "gateit" and if that Pull Request's CI status was success then Zuul trigger the gate pipeline
- In the gate pipeline, Zuul runs a special job called "noop", this is a special job that do nothing and always succeed.
- As the gate pipeline's jobs succeeded, Zuul merges the Pull Request.

Due to the merge,

- Zuul receives the "Pull Request merged" event from Pagure then triggers the promote pipeline
- The rpm-build job get started and requests a rpm build on Koji
- Finally Zuul stores the result and returns the job status as a comment in the Pull-Request

In the case of the master branch, for rawhide, this is not shown on this diagram, but bodhi will get notified of the build and let the change land in the next Fedora Rawhide compose.
As you can see, this kind of workflow, could be an additional and easy way to contribute to Fedora.

Also, for packagers, that manage a large amount of packages, they could benefit, not especially from that workflow, but instead from the Pull Request as a place to run CI jobs and get early feedback.

Runtime and build dependencies

- Expliciting dependencies between two or more Pull Requests
  - Magic keyword in the Pull Request initial message: “Depends-On: <PR URI>
  - Dependencies can be set to any repository known to Zuul
- Zuul prepares the jobs’ resources according to the dependencies
  - Job workspace on the test node (repositories’ sources)
  - Artifacts from previously run jobs can also be included in the job’s inventory
- Jobs (roles and playbooks) are able to handle dependencies
  - RPMs
    - At build time: dependent RPMs, injected in the local mock root
      - “I need to bump the library X to bump the version of my package”
    - At testing time: add custom repository files in the test node’s /etc/yum.repos.d

Now time to talk more about dependencies.

Dependencies between Pull Requests can be defined thanks to the “Depends-On” keyword followed by the URL of a dependent Pull Request.
This keyword can appear multiple time in the Pull Request initial message to define multiple dependencies.
Also, and this is really nice in the packing context, dependency between Pull Requests can be defined across different repositories, and even better it works across different platforms such as Pagure and Github.

So given a dependency graph, Zuul prepares jobs workspaces with all dependent sources checkouted at the right version, ready to be used and validated. If dependencies expose artifacts, then they will be exposed to the job as well.

Thanks to this feature, we can write jobs able to handle dependencies easily. For instance, we can handle RPM build dependencies via a job that will inject dependent RPM in a mock root to make build dependencies available during the build. This is truly awesome, for instance, when you need to bump the version of a package that require the bump of dependent library to build.
Lastly, regarding the runtime dependency, the rpm-test job already handles dependencies by preparing a yum repository to make the dependent RPMs available during the job.

Demo

- Open a Pull Request to trigger “check” jobs
  - Quick tour of the Zuul web UI
- From the Pull Request Proposal to the approval
- Inspect CI results
- Depends-on usage
  - Runtime dependency
  - CI configuration change

Now let’s have a look to some concretes example:

- Open a PR: https://src.fedoraproject.org/fork/fbo/rpms/python-gear/diff/master..test-rpminspect
  - Show the status page
  - Show the projects page
  - Show the jobs page and click on the rpm-scratch-build, show we see infos about the jobs and where it is defined
  - Show the labels
  - Show jobs history
  - Show the live console
- Show previous run result page https://fedora.softwarefactory-project.io/zuul/build/3dba569f0a6c41b8ac178464f0a40a9e
  - Explain the build result page
- Show a PR page full workflow: https://src.fedoraproject.org/rpms/nodepool/pull-request/6
- Depends-on Runtime dep:
  - Go in the ARA report for the rpm-test job https://src.fedoraproject.org/rpms/python-gear/pull-request/10
    - Show the repo setup task
    - Show the inventory zuul.artifacts
- Depends-on CI configuration change

- Show the PR page https://src.fedoraproject.org/rpms/python-gear/pull-request/11
- Show the new job link to the artifact (job well executed)

Early adopters & feedback wanted!

- Read the wiki: https://fedoraproject.org/wiki/Zuul-based-ci
- Ask questions on the Fedora Mailing List
- Reach us on IRC:
  - Freenode: #zuul, #softwarefactory
- Talk to us at devconf, we have stickers!

Now what about the next steps.

- First we would like to hear from you, what do you think about this effort to bring Zuul as a CI for Fedora distgit Pull Request?
- Also we are looking for early adopters, packagers and CI engineers to experience the system and help to improve it

And from a technical point of view, here are some idea of improvement for the future, for instance:

- Regarding, the workflow, we can think of:
  - An auto-merge workflow where a packager do not need to approve a Pull Request, but instead let Zuul merge the Pull Request as soon as the CI jobs pass with success.
  - Also, we can think of a job for stable branches that automatically create updates on Bodhi
  - Finally, a really great improvement could be a smart rpm build job ables to select the right mock root type (local mock or koji scratch build) whether the Pull Request dependencies' contains or not a build requirement.
- And obviously we could add more validation jobs

Thank you!