O-RAN Developer's Guide to CI Resources and Processes at the LF

- Continuous Integration Network Resources
- Development Procedures and Policies
    - Always create a branch
    - Write good git commit messages
    - Working with Docker images
    - Using Gerrit
        - Install git-review Command
        - Prohibited Content
        - Quickstart Guides
            - Quickstart: Create and submit a change for review
            - Quickstart: Submit a change to a branch other than "master"
            - Quickstart: Revise your open gerrit review
            - Quickstart: Revise any open gerrit review
            - Quickstart: Resolve a conflicted review
            - Quickstart: Squash commits after you forgot the "--amend" flag
        - Reviewing and merging changes
        - Create a Git branch or tag
    - Triggering Jenkins jobs from Gerrit
    - Staging Binary Artifacts
        - Releasing a Java/maven artifact
        - Releasing a Docker artifact
        - Releasing a Python package
        - Releasing a PackageCloud DEB/RPM package
- Configure Project for Sonar Analysis
    - Configure CXX/CMake Project for Code Coverage
    - Configure Golang/Go Project for Code Coverage
    - Configure Java/Maven Project for Code Coverage
    - Configure Python/Tox Project for Code Coverage
- Configure Project for Nexus IQ (CLM) Analysis
    - Configure Java/Maven Project for Nexus IQ (CLM)
    - Configure Python/Tox Project for Nexus IQ (CLM)
- Setting up development environment
    - Making Java code checked by Sonar
    - Setting up Eclipse for Sonar
- Jenkins Job Configuration
    - What Jobs are Required?
    - Writing JJB Templates
        - Choosing a build node
    - Testing JJB Templates
        - Let Jenkins create jobs
        - Create jobs directly
        - Test a job in the sandbox
- Jenkins Build Minion Labels and Images
    - Image Templates
    - Image Builder Jobs

This is a guide for software developers and testers about the continuous integration (CI) resources and processes used by the project and supported by the Linux Foundation.

Continuous Integration Network Resources

Most of these resources authenticate users via a Linux Foundation identity.

URL	Description
https://identity.linuxfoundation.org	The Linux Foundation identity management web site.
https://gerrit.o-ran-sc.org	The Gerrit code review server hosts the Git repositories and supports the review and merge workflow. The review process basically requires all users to follow something like a git pull-request process, but restricts the publication (push) of private branches.
https://jenkins.o-ran-sc.org	Jenkins is the continuous-integration server aka the build server. All users can see the contents of the Jenkins; no users can directly modify the configuration nor start a job in this Jenkins. Jobs are configured by Jenkins Job Builder (JJB) files in the ci-management gerrit repository.
https://jenkins.o-ran-sc.org/sandbox	The Jenkins sandbox is a place for testing Jenkins job configurations. Users must request login credentials on the sandbox, it does not use the LF identity or single sign-on feature. After receiving access, users can create jobs, reconfigure jobs, trigger builds etc. New JJB templates should be tested in the sandbox before submitting for review and use on the main Jenkins server.
https://jira.o-ran-sc.org	Jira is the web site that supports agile development with epics, user stories, and especially issues (bug reports).

https://nexus.o-ran-sc.org	Not heavily used by this project because Java is not a key technology, this Nexus 2 repository holds Maven (jar and pom) artifacts produced by builds. Snapshot and staging builds are all deployed to this repository. Release artifacts are created by promoting artifacts manually from the staging repository after suitable approval. Publicly accessible to users without credentials. All users should be able to access and browse artifacts through this URL.
https://nexus3.o-ran-sc.org/	This Nexus 3 server stores project Docker images and also mirrors external registries. Supports the following: • Public mirror registry: nexus3.o-ran-sc.org:10001. This acts as a mirror of the public registry at docker.io. • Release registry: nexus3.o-ran-sc.org:10002. This is a VIEW offering access to the public registry mirror AND the project releases (docker images in the release registry). Projects may request promotion of their docker images to this registry by using the self-release process. • Snapshot registry: nexus3.o-ran-sc.org:10003. Not used in ORAN, but in other projects, the snapshot registry contains docker images produced by certain Jenkins jobs. • Staging registry: nexus3.o-ran-sc.org:10004. This contains release candidate docker images produced by merge jobs. These registries are open for anonymous read access. The Jenkins server has credentials to push images to the snapshot and staging registries, and to promote images to the release registry. Manual push of images is not supported.
https://packagecloud.io	This site hosts binary artifacts such as Debian (.deb) packages, Red Hat Package Manager (.rpm) packages and more in these repositories: • o-ran-sc/master • o-ran-sc/staging • o-ran-sc/release Also see Binary Repositories at PackageCloud.io
https://sonarcloud.io	Sonar tools analyze Java and Python code for quality issues and code coverage achieved by unit tests (JUnit, tox) and publish results to this cloud-based server.

Development Procedures and Policies

This section guides the developer in submitting code, reviewing code, tracking the status of builds and requesting creation of release versions. These recommended practices come from the Linux Foundation.

Always create a branch

When working with a git repository cloned from Gerrit you can create as many local branches as you like. None of the branches will be pushed to the remote Gerrit server, the branches remain forever private. Creating branches for each task will allow you to work on multiple independent tasks in parallel, let you recover gracefully from various situations, and generally save aggravation and time. Also see these instructions on tagging and branching for releases:

Tagging and Branching Steps Process

Write good git commit messages

The Linux Foundation strongly recommends (and eventually may enforce) Gerrit commit message content. A git commit message must meet these requirements:

1. The first line should have a short change summary, up to 50 characters
2. The second line must be blank
3. The body of the commit message should have a detailed change description, wrap lines at 72 characters max
4. The line before the footer must be blank
5. The footer (last block of lines following a blank line) must consist of these lines:
    a. Issue-ID: line with a valid Jira issue number, composed and inserted manually by the committer
    b. Change-Id: line, which is automatically generated and inserted by git review
    c. Signed-off-by line, which is automatically generated and inserted by git commit

An example is shown below:

```
Null check for ClientResponse

NullPointerException might be thrown as cres is nullable here

Issue-Id: PROJECT-999
Change-Id: I14dc792fb67198ebcbabfe80d90c48389af6cc91
Signed-off-by: First Last <fl1234567890@my-company.com>
```

Please note that the Jira system's pattern matcher that finds issue IDs in Gerrit reviews is extremely limited. The "Issue-Id" line must be EXACTLY as shown above! The keyword must be written in mixed case as shown, then a colon, then a space, then the appropriate project name in ALL CAPS, then a hyphen, then the number.

Working with Docker images

The docker image name is set by the Jenkins job, as configured by the JJB. The method for choosing the docker image tag is also configured in the Jenkins job, one of the following:

- "latest": This constant string is always used
- "git-describe": The output of the "git describe" command is used. This requires that the repository has at least one **git** tag.
- "yaml-file": The file container-tag.yaml in the project repository is queried for the entry with key "tag"

For the non-constant options, the development team must manage the tag or the version string in the YAML file.

The Jenkins verify job creates a docker image to test the creation process, then discards the image at completion.

The Jenkins merge job creates a docker image and pushes the image to the staging registry.Jenkins job sets the PUSH registry to STAGING.

Creation of a final release image requires opening a ticket with LF release-engineering team. They sign the image found in the staging registry and move it manually to the release registry.

Using Gerrit

The project uses Gerrit to automate the process of reviewing all code changes before they are committed to the Git repository. For a tutorial on git you might start here:

> https://git-scm.com/doc

Also see this guide published by the LF Release Engineering team about using Gerrit:

> https://docs.releng.linuxfoundation.org/en/latest/gerrit.html

Install git-review Command

The command-line tool "git review" is the most reliable and can be used on any platform. As of this writing, version 1.28.0 or later is required; version 1.26.0 will not work. Windows users should install "Git Bash" to gain support for command-line git. The most common way of installing is to use pip:

```
pip install git-review
```

Prohibited Content

Gerrit is designed to host text files – source code. It enforces a size threshold on every commit, the default limit is 5MB. Further the Linux Foundation prohibits committing binary files such as compiled executables, jar files, zip archives and so on. An exception is made for binary picture (image) files in GIF, JPG and PNG formats, but the size limit must still be honored.

Quickstart Guides

The quickstart guides below describe the command-line procedures for performing common tasks on Gerrit.

Quickstart: Create and submit a change for review

```
git checkout -b my-local-branch
# work work work
git commit -as
    # Write a commit message including a line "Issue-ID:" with Jira ticket nr
git review master
```

Quickstart: Submit a change to a branch other than "master"

```
git review other-branch
```

Quickstart: Revise your open gerrit review

```
git checkout my-local-branch
# revise revise revise
git commit -a --amend
    # Revise your message but do NOT alter the Change-Id: line
git review master
```

Quickstart: Revise any open gerrit review

```
# Look up the change number shown top-left in Gerrit web, here "999".
git review -d 999
# revise revise revise
git commit -a --amend
    # Check your message -- do NOT alter the Change-Id: line
git review --no-rebase master
```

Quickstart: Resolve a conflicted review

```
git checkout master
git pull origin master
# Look up the change number in Gerrit, it's shown top-left, here 999
git review -d 999
git rebase master
# fix conflicts
git add .
git rebase --continue
git review master
```

Quickstart: Squash commits after you forgot the "--amend" flag

```
# count number of commits to be squashed, the 2 below is for
two
git reset --soft HEAD~2
git commit
```

Reviewing and merging changes

Submitted reviews with changes should appear in the Gerrit web interface. The Jenkins job builder will publish test results, giving "Verified +1" if the test succeeds and -1 if the test fails. A contributor or committer can review it with a -1/0/+1. A committer then must approve it with a +2 rating and merge the code to the master branch.

blocked URL

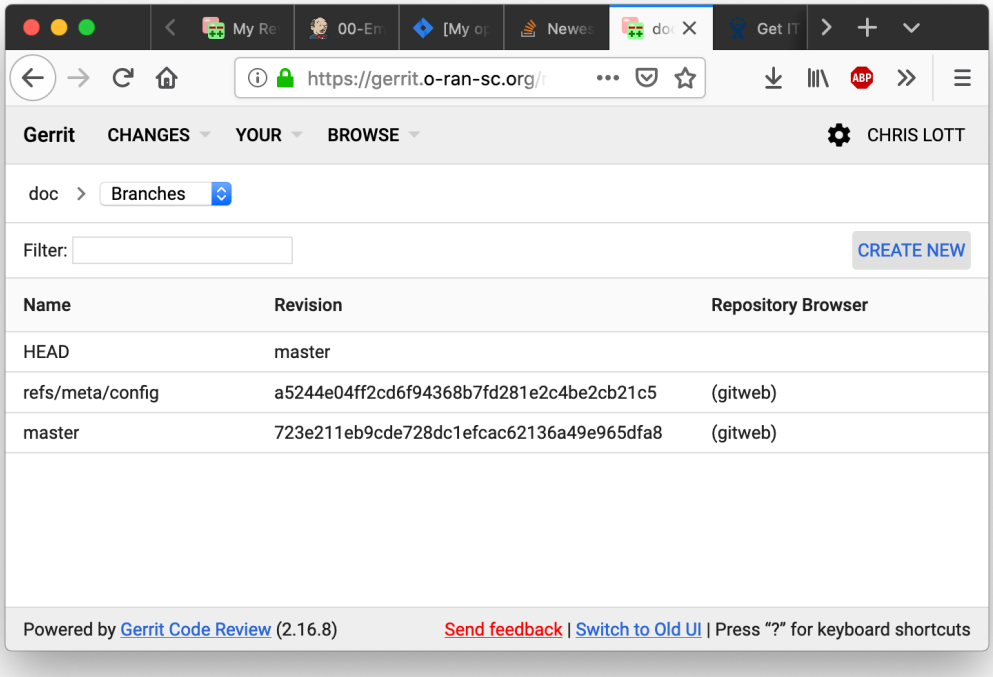
The committer may take any of several actions, such as clicking on the "Reply" button, adding reviewers, adding a review comment, and moving the flags to +2 and +1

blocked URL

Once a committer/contributor approves it, the code can be merged to the master branch.

Create a Git branch or tag

A project committer can create a branch at any time using the Gerrit web site. Similarly a project committer can add a tag to the repository using the Gerrit web site. To reach the administration screen for a repository, start with the "Browse" menu at the top, pick Repositories, find the repository in the list and click on its link. To create a branch, pick "Branches" on the left side, then click the Create New button, as shown in the screen shot below. Similarly to create a tag, from the administration screen for the repository pick "Tags" on the left side, then click the Create New button (screen not shown).



Triggering Jenkins jobs from Gerrit

Most Jenkins jobs can be launched by posting a comment in the Gerrit system on a review. The exact string that is required depends on the job type. Also the comment must not contain anything else than the keyword. E.g., check the last "recheck" comment in this review and how it initiated a recheck job run.

- Post "**recheck**" to re-run a verify job. This may be necessary due to intermittent network failures at Jenkins, because a dependency has become available, etc.
- Post "**remerge**" to re-run a merge job. This may be necessary due to intermittent network failures at Jenkins, to recreate a version of an artifact, etc.
- Post "**stage-release**" to run a staging job.
- Post "**run-sonar**" to run the Sonar analysis job (if applicable).
- Post "**jjb-deploy** <pattern>" on a ci-management change to create jobs matching the pattern (for example "myproject-*") to the Jenkins sandbox.

Staging Binary Artifacts

In the LF environment, "to stage" means to create an artifact in an ephemeral staging area, a Nexus registry or repository where artifacts are deleted after two weeks. This practice originated with Java-Maven jobs that follow the convention of using a version suffix string "-SNAPSHOT". The Java-Maven **merge** jobs create jar files and push them to a "snapshot" repository. To stage a Java-Maven artifact, a process that also strips the "-SNAPSHOT" suffix, post a comment "stage-release" on the a Gerrit change set and the build will start on the appropriate branch.

Non-Java-Maven merge jobs such as Docker merge jobs do not use the "-SNAPSHOT" version convention. These merge jobs generally create and push artifacts directly to a staging area. So for non-Java-Maven artifacts there is no staging workflow.

For Docker images you may check if a particular image exists in the staging repo using this link: https://nexus3.o-ran-sc.org/#browse/browse:docker.staging (make sure *not* to log in. Otherwise the link does not work). Go to correct image in the tree (note that the latest o-ran-sc components are below the top-level folder "o-ran-sc" and the same component under the same name directly on top level is some outdated version not relevant anymore). and then open the subtree "tags", e.g., o-ran-sc ric-plt-submgr tags ...). There should be at least one version under the tag subtree (e.g. this link for the near-RT RIC subscription manager image: https://nexus3.o-ran-sc.org/#browse/browse:docker.staging:v2%2Fo-ran-sc%2Fric-plt-submgr%2Ftags). If not, then there's no staging image in the staging repo for this component. Alternatively use this curl command: "curl -X GET https://nexus3.o-ran-sc.org:10004/v2/o-ran-sc/ric-plt-submgr/tags/list" (replace the part in bold with the correct component obtained via "curl -X GET https://nexus3.o-ran-sc.org:10004/v2/_catalog" (not sure if this shows components that do not currently have a tag)).

If there's no tag version in the staging repository, we will need to re-run a "merge job" as per "remerge" in the previous section "Triggering Jenkins jobs from Gerrit". This was done e.g. in this review https://gerrit.o-ran-sc.org/r/c/ric-plt/submgr/+/4526 for the subscription manager.

Releasing Binary Artifacts

In the LF environment, "to release" means to promote an artifact from an ephemeral staging area to a permanent release area. For example, move a Docker image from a Nexus staging registry where images are deleted after two weeks to a Nexus release registry where images are kept "forever." This self-service process is implemented by Jenkins jobs that react to files committed and merged via Gerrit. Guarding the process with the Gerrit merge task means only committers can release artifacts.

Once testing against the staging repo version has been completed (see above ⚠️) and the project has determined that the artifact in the staged repository is ready for release, the project team can use the new-for-2019 self-release process. For details on the JJB templates please see https://docs.releng.linuxfoundation.org/projects/global-jjb/en/latest/jjb/lf-release-jobs.html

Releasing a Java/maven artifact

1. Find the Jenkins stage job that created the release candidate.
    a. Go to Jenkins and select the tab for the product to release.
    b. Find the link for the "<product>-maven-stage-master" job and click it.
    c. From the list of jobs, find the number for the job that created the artifact to release, the date of the run can be of help here.
    d. Put the number at the end of the "log_dir" value seen in the example below.
2. Alternative way to find Jenkins stage job.
    a. Look among its output logs for the file with the name: staging-repo.txt.gz, it will have a URL like this:
    b. https://logs.acumos.org/production/vex-yul-acumos-jenkins-1/common-dataservice-maven-dl-stage-master/4/staging-repo.txt.gz
3. Create a new/update existing release yaml file in the directory "releases/" at the repo root.
    a. The file name should be anything, but most projects use a pattern like "release-maven.yaml". An example of the content appears below.
    b. The file content has the project name, the version to release, and the log directory you found above, altho in abbreviated form.
4. Create a new change set with just the new file, commit to git locally and submit the change set to Gerrit.
5. After the verify job succeeds, a project committer should merge the change set. This will tag the repository with the version string AND release the artifact.

Example release yaml file content:

```
---
distribution_type: maven
version: 1.0.0
project: example-project
log_dir: example-project-maven-stage-master/17/
```

After the release merge job runs to completion, the jar files should appear in the Nexus2 release repository.

Releasing a Docker artifact

For a Docker image the release yaml file must list the desired release tag and the existing container tags. Example release yaml file content:

```
---

distribution_type: container
container_release_tag: 1.0.0
container_pull_registry: nexus.o-ran-sc.org:
10003
container_push_registry: nexus.o-ran-sc.org:
10002
project: test
ref: b95b07641ead78b5082484aa8a82c900f79c9706
containers:
    - name: test-backend
      version: 1.0.0-20190806T184921Z
    - name: test-frontend
      version: 1.0.0-20190806T184921Z
```

After the release merge job runs to completion, the container images should appear in the Nexus3 release registry.

Releasing a Python package

For a Python package the release yaml file must list the log directory, python version and more. Example release yaml file content:

```
---
distribution_type: pypi
log_dir: ric-plt-lib-rmr-python-pypi-merge-master/1
pypi_project: rmr
python_version: '3.7'
version: 1.0.0
project: myproject
```

If you use a valid decimal value anywhere (like 3.7 above), put it in single quotes so it can be parsed as a string, not a number.

After the release merge job runs to completion, the packages should appear in the https://pypi.org index.

Releasing a PackageCloud DEB/RPM package

2020-Dec-14: There is a process that involves the keyword "stage-release" (see section "Triggering Jenkins jobs from Gerrit" above) to publish packages to packagecloud. This works with two merges. First one with updates to ci/package-tag.yaml and ci/control. Merge that change and after merging add a comment with only the keyword "stage-release". After this create a second review for an updated "releases/*.yaml" with correct version, log_dir and ref with commit hash. Submit that change for merging as well. This will do the actual moving of the package from the staging directory in packagecloud to the release directory. A bit of information on this is also available in section "PackageCloud Release Files" from this LF guide: link.

OLD notes: The self-release process for PackageCloud is in active development as of December 2019. Until it is ready, write a ticket at https://jira.linuxfoundation.org/servicedesk

Configure Project for Sonar Analysis

The SonarQube system analyzes source code for problems and reports the results, including test code-coverage statistics, to https://www.sonarcloud.io./organizations/o-ran-sc/projects. The analyses are usually run weekly by a Jenkins job. Analyzing and reporting static source-code features requires almost no configuration, basically just naming the directory with source code. However reporting coverage requires each project's build and test steps to generate code-coverage statistics, which requires automated unit tests that can be run by Jenkins plus additional configuration.

Every sonar analysis job consists of these steps:

1. compile source (except for interpreted languages like python, of course)
2. run tests to generate coverage statistics
3. analyze source code with sonar scanner
4. gather coverage stats with sonar scanner
5. publish code analysis and test stats to SonarCloud.io

All these steps run directly on the Jenkins build minion, with the Sonar steps usually implemented by a Jenkins plug-in. Projects that use a Dockerfile to create a Docker image should factor their build so that the build and test steps can be called by a docker-based build (i.e., from within the Dockerfile) **and** by a non-docker-based build process. In practice this usually means creating a shell script with all the necessary steps. The tricky part is installing all prerequisites, because a Docker base build image is invariably different from the Jenkins build minion image.

This section focuses on configuration to generate coverage data suitable for consumption by the Sonar scanner. See the section later in this document on Jenkins job configuration for details about that.

Configure CXX/CMake Project for Code Coverage

CXX projects require cmake and a dedicated tool, the *Sonar build wrapper*, as documented here: https://docs.sonarqube.org/latest/analysis/languages/cfamily The build wrapper is used to invoke `make`. The wrapper then gathers data without further configuration. This tool is automatically installed and invoked in CMake-style Jenkins jobs, implemented by the `cmake-sonarqube.sh` script from LF's global-jjb. Actually no configuration changes are required in the project's CMakeLists.txt or other files, just use of the appropriate Jenkins job template.

Execution of the build via make should create the output file build-wrapper-dump.json, which can be consumed by the Sonar scanner.

Examples from the O-RAN-SC project RMR:

- Repository https://gerrit.o-ran-sc.org/r/admin/repos/ric-plt/lib/rmr
- Job configuration https://gerrit.o-ran-sc.org/r/gitweb?p=ci-management.git;a=blob;f=jjb/ric-plt-lib-rmr/ric-plt-lib-rmr.yaml
- Sonar report https://sonarcloud.io/dashboard?id=o-ran-sc_ric-plt-lib-rmr-c

More details about configuration, building for scanning, and specific file and/or directory exclusion in C/C++ repositories is provided on the Configure Sonar for C/C++ page.

Configure Golang/Go Project for Code Coverage

Go projects should use the go-acc tool (``go get -v github.com/ory/go-acc``) to run tests and generate statistics. This yields better results than standard features in golang versions 1.12 and 1.13. Here's an example:

```
go-acc $(go list ./... | grep -vE '(/tests|/enums)' )
```

Execution of the build via this command should create the output file coverage.txt, which can be consumed by the Sonar scanner. However modules are not supported yet by the Sonar Scanner. The problem and some workarounds are described here: https://jira.sonarsource.com/browse/SONARSLANG-450

Examples from the O-RAN-SC project Alarm:

- Repository https://gerrit.o-ran-sc.org/r/admin/repos/ric-plt/e2mgr
- Job configuration https://gerrit.o-ran-sc.org/r/gitweb?p=ci-management.git;a=blob;f=jjb/ric-plt-e2mgr/ric-plt-e2mgr.yaml;hb=refs/heads/master
- Sonar report: https://sonarcloud.io/dashboard?id=o-ran-sc_ric-plt-e2mgr

Configure Java/Maven Project for Code Coverage

Java projects require maven and should use the jacoco maven plugin in the POM file, which instruments their code and gathers code-coverage statistics during JUnit tests. Here's an example:

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.4</version>
  <executions>
    <execution>
      <id>default-prepare-agent</id>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>default-report</id>
      <phase>prepare-package</phase>
      <goals>
        <goal>report</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Execution of the build via maven should create the output file target/jacoco.exec, which can be consumed by the Sonar scanner.

Examples from the O-RAN-SC project Dashboard:

- Repository https://gerrit.o-ran-sc.org/r/admin/repos/portal/ric-dashboard
- Job configuration https://gerrit.o-ran-sc.org/r/gitweb?p=ci-management.git;a=blob;f=jjb/portal-ric-dashboard/portal-ric-dashboard.yaml
- Sonar report https://sonarcloud.io/dashboard?id=o-ran-sc_portal-ric-dashboard

Configure Python/Tox Project for Code Coverage

Python projects require tox and should extend the tox.ini file that runs tests as follows.

First, add required packages within the `[testenv]` block:

```
deps=
  pytest
  coverage
  pytest-cov
```

Second, add the following commands within the `[testenv]` block:

```
commands =
  pytest --cov dir-name --cov-report xml --cov-report term-missing --cov-report html --cov-fail-under=70
  coverage xml -i
```

Execution of the build via tox (really just tests) should create the output file coverage.xml, which can be consumed by the Sonar scanner.

Examples from the O-RAN-SC project A1:

- Repository https://gerrit.o-ran-sc.org/r/admin/repos/ric-plt/a1
- Job configuration https://gerrit.o-ran-sc.org/r/gitweb?p=ci-management.git;a=blob;f=jjb/ric-plt-a1/ric-plt-a1.yaml
- Sonar report https://sonarcloud.io/dashboard?id=o-ran-sc_ric-plt-a1

Configure Project for Nexus IQ (CLM) Analysis

The Nexus IQ system supports component lifecycle management (CLM), which mostly means analyzing third-party libraries used by the project and reporting any issues with those dependencies such as known security vulnerabilities. The results are published at https://nexus-iq.wl.linuxfoundation.org/assets/index.html.

Configure Java/Maven Project for Nexus IQ (CLM)

No special project configuration is required.

Ensure the jenkins job template 'gerrit-maven-clm' is configured to define the required job. The job runs weekly, or on demand in response to posted comment "run-clm".

Configure Python/Tox Project for Nexus IQ (CLM)

The Python project must be configured to report its package dependencies for analysis by the Nexus IQ scanner. Add a new environment to the tox.ini file called "clm" with the following content:

```
[testenv:clm]
# use pip to report dependencies with versions
whitelist_externals = sh
commands = sh -c 'pip freeze > requirements.
txt'
```

Ensure the jenkins job template 'gerrit-tox-nexus-iq-clm' is configured to define the required job. The job runs weekly, or on demand in response to posted comment "run-clm".

Setting up development environment

Eclipse users can see Sonar results in an especially convenient way.

Making Java code checked by Sonar

To make Java code checked by Sonar locally an addition to the projects pom file is needed, see below.

```
<plugin>
    <groupId>org.sonarsource.scanner.maven</groupId>
    <artifactId>sonar-maven-plugin</artifactId>
    <version>${sonar-maven-plugin.version}</version>
</plugin>
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.4</version>
    <executions>
        <execution>
            <id>default-prepare-agent</id>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <id>default-report</id>
            <phase>prepare-package</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Then the Jenkins job needs to be updated, see the following commit for an example: https://gerrit.o-ran-sc.org/r/c/ci-management/+/2446.

Setting up Eclipse for Sonar

To be able to connect Eclipse to Sonar, the SonarLint plugin is needed. To install this plugin, follow the steps below:

1. In Eclipse, open the "Help Eclipse Marketplace...".
2. In the "Find:" box, type "sonarlint" and press "Go".
3. The latest version of SonarLint should show up at the top of the search results. Press the "Install" button.
4. Accept license and restart Eclipse after installation is finished.

When SonarLint is installed, it should be connected to the SonarCloud and bound to the project. To do this, follow the steps below:

1. In Eclipse, right click on the project and select "SonarLint Bind to SonarQube or SonarCloud...".
2. In the dialog, press the "New" button.
3. Make sure the "sonarcloud" radio button is selected and press "Next".
4. If you do not have a token generated in SonarCloud, press the "Generate token" button. Otherwise you can reuse yor token.
5. Follow the instructions in the web page you are redirected to to generate the token.
6. Paste the token in to the "Token:" box and press "Next".
7. In the "Organization:" box, type "o-ran-sc" and press "Next".
8. Press "Next".
9. Press "Finish".
10. Select "Window Show View Other..." and then "SonarLint bindings".
11. In the view, doubleclick the new binding.
12. In the dialog, Press "Add", select the project to bind, press "Ok", and press "Next".
13. Type your project's name. When it show up in the result list, select it and press "Finish".

Now Sonar issues should show up in your code.

Note! At the moment there is a bug that show up if Lombok is used in the code with a version below 1.18.12. If you have this problem, download Lombok version 1.18.12 or higher and repeat the installation procedure described here, https://howtodoinjava.com/automation/lombok-eclipse-installation-examples/

Jenkins Job Configuration

All jobs in the Jenkins server are generated from Jenkins Job Builder (JJB) templates. The templates are maintained in this project's ci-management git repository. The templates use features from the Linux Foundation Global JJB as well as features custom to this project.

Additional documentation resources:

Jenkins Job Builder: https://docs.openstack.org/infra/jenkins-job-builder/

LF Global JJB: https://docs.releng.linuxfoundation.org/projects/global-jjb/en/latest/

LF Ansible: https://docs.releng.linuxfoundation.org/en/latest/ansible.html

What Jobs are Required?

In general every repository requires these Jenkins jobs, one template for each:

- Verify: a change submitted to a gerrit repository should be verified by a Jenkins job, which for source generally means compiling code and running tests.
- Merge: a change merged to a gerrit repository usually publishes some redistributable (often binary) artifact. For example, a Docker merge job creates an image and pushes it to a Nexus3 registry.
- Sonar: most source-code projects are analyzed by Sonar to detect source-code issues and publish unit-test code-coverage statistics.
- Release: release jobs promote redistributable artifacts from a snapshot or staging (temporary) repository to a release (permanent) repository.
- Info: the project's INFO.yaml file controls the committers. On merge the contents are automatically pushed to the Linux Foundation LDAP server.

What about documentation (RST)? The verify and merge jobs are defined globally in the O-RAN-SC project. All changes to a projects docs/ subdirectory will be verified similarly to source code, and published to ReadTheDocs on merge.

Writing JJB Templates

Each gerrit repository usually requires several jobs (templates). When creating the templates, the usual convention is to group all CI materials like templates and scripts in a directory named for that repository. Each directory should have at a minimum one YAML file. For example, "ci-management/jjb /com-log/com-log.yaml". Most repositories have the following Jenkins items defined in a yaml file:

- Project view. This causes a tab to appear on the Jenkins server that groups all jobs for the repository together.
- Info file verifier. This checks changes to the repository's INFO.yaml file.
- Verify and merge jobs. These check submitted changes, and depend on the implementation language of the code in that repository.
- Sonar job to analyze static code and gather unit-test coverage statistics.
- Release job to promote an artifact from a staging repository to a release repository.

After creating or modifying a YAML file, submit the change to Gerrit where it will be verified and can be reviewed. Only the LF Release Engineering team can merge changes to the ci-management repository.

When creating new types of jobs follow these steps (see IT-25277 support case):

1. Adding a jobs yaml file
2. Making sure `mvn-settings` is set.
3. Creating a support request to create jenkins credentials (As pointed by `mvn-settings`)
4. Adding the set of settings files that point to jenkins credentials.

Choosing a build node

Every project has a set of predefined OpenStack virtual machine images that are available for use as build nodes. Choosing a build node determines the software image, number of CPU cores and amount of physical memory. Build node names, for example "centos7-builder-1c-1g", are maintained in the ci-management repository as files in this directory:

```
ci-management/jenkins-config/clouds/openstack/cattle/
```

Testing JJB Templates

Job templates should be tested by creating jobs in the Jenkins sandbox, then executing the jobs against a branch of the repository, the master branch or any change set (review) branch can be used. Jobs can be created in one of two ways:

Let Jenkins create jobs

Post a comment "jjb-deploy" on your change set submitted to the ci-management repo. This creates a request for the primary Jenkins server to create new jobs in the sandbox. Be patient if you choose this route. The comment takes this form:

```
jjb-deploy your-job-name*
```

The argument is a simple shell-style globbing pattern to limit the scope. This example should create all jobs that start with the prefix "your-job-name".

Create jobs directly

You can create jobs directly at the Sandbox from a personal computer. This allows rapid edit/test cycles. To do this:

- Request username and password at the Jenkins sandbox
- Generate and copy a new API token in the Jenkins sandbox user settings Configure tab
- Install the Python package jenkins-job-builder (version 3.2.0 as of this writing)
- Create a jenkins.ini configuration file with credentials, see below
- Test the templates locally: jenkins-jobs test -r jjb > /dev/null
- Create jobs: jenkins-jobs --conf jenkins.ini update -r jjb YOUR_JOB_NAME > /dev/null

Sample jenkins.ini file:

```
[job_builder]
ignore_cache=True
keep_descriptions=False
recursive=True
update=jobs

[jenkins]
query_plugins_info=False
url=https://jenkins.o-ran-sc.org/sandbox
user=your-sandbox-user-name
password=your-sandbox-api-token
```

Test a job in the sandbox

After pushing a job to the sandbox, either via the Jenkins `jjb-deploy` job or directly, you can run the job on the code in your repository, usually the master branch, to test the job. Follow this procedure:

- Go to https://jenkins.o-ran-sc.org/sandbox/ and click the **log in** link top right. You need special credentials for this, as discussed above.
- Find your job in the "All" view and click on it
- On the job page, find the link on the left "Build with Parameters" and click on it
- Check the parameters, then click the Build button.

You can also test a job on a Gerrit change set that has not yet been merged. Follow this procedure:

- In Gerrit, find your change set, click the icon with three dots at the top right, pick Download Patch
  - In the dialog-like thing that appears, copy the text that appears in the Checkout box (just click on the icon at the right). It looks like this:
    - git fetch "https://gerrit.o-ran-sc.org/r/your/repo" refs/changes/62/2962/1 && git checkout FETCH_HEAD
  - You need just this part: **refs/changes/62/2962/1**
- As described above, log in to the Jenkins sandbox, find your job, click Build with Parameters
- On the job parameters page, find these fields:
  - GERRIT_BRANCH
  - GERRIT_REFSPEC
- Paste the **refs/changes/62/2962/1** part you copied from Gerrit into BOTH fields
- Click the Build button

Jenkins Build Minion Labels and Images

Jenkins build minions are OpenStack virtual machines. The software, the number of cores, amount of virtual memory and amount of disk memory are determined by files in directory `ci-management/jenkins-config/clouds/openstack/cattle`. As of this writing the following build node labels are available for use as a `build-node` configuration parameter in a Jenkins job definition:

```
centos7-builder-1c-1g          ubuntu1804-builder-2c-2g.cfg
centos7-docker-2c-8g-200g.cfg  ubuntu1804-builder-4c-4g.cfg
centos7-docker-2c-8g.cfg       ubuntu1804-docker-4c-4g.cfg
```

Each config file contains an image name such as "ZZCI - CentOS 7 - builder - x86_64 - 20200317-165605.039". Each image can have a different set of software packages. It's fairly safe to assume that all have C compilers, Go compilers, Java virtual machines and Python interpreters. Discovering the exact software contents of a build node generally requires analyzing the Ansible roles and packer commands that are also maintained in the ci-management repository.

Image Templates

The images are created by a combination of ansible and packer jobs that are configured from the packer directory in the ci-management repository. The O-RAN-SC project uses two classes of minion provision templates: "builder" and "docker". The latter mostly adds the Docker daemon and other software to the former. These two templates are combined with the two choices of Linux distro, currently Centos and Ubuntu, to yield a minimum of four image combinations. To add software, start with the file `packer/provision/local-docker.yaml`.

You can discover the available image names by checking the image builder job results in Jenkins - next section.

Image Builder Jobs

The images are built automatically on a monthly basis by these Jenkins jobs:

> https://jenkins.o-ran-sc.org/view/ci-management/job/ci-management-packer-merge-centos-7-builder/
>
> https://jenkins.o-ran-sc.org/view/ci-management/job/ci-management-packer-merge-centos-7-docker/
>
> https://jenkins.o-ran-sc.org/view/ci-management/job/ci-management-packer-merge-ubuntu-18.04-builder/
>
> https://jenkins.o-ran-sc.org/view/ci-management/job/ci-management-packer-merge-ubuntu-18.04-docker/

These jobs are NOT triggered automatically if a change is made to a supporting file such as an Ansible role definition. Regular project members cannot launch these jobs manually, only the LF #RelEng team has the privilege. One way to avoid writing a ticket is to find a previously merged commit to a file in the packer area and post the the usual Jenkins comment "remerge" on it, but this is not foolproof.

Upon completion of a job the job page shows the image name that was built, a long string starting "ZZCI" and ending with a timestamp, see the screenshot below. Copy that string (very carefully!) into the appropriate config file, submit as a gerrit change set, and wait for merge. Then the new image will be available for use with the specified build node label.

