Defining Automated Jenkins Jobs for O-RAN SC Repos

O-RAN SC uses a Jenkins based platform for automating all of its build jobs. This service can be accessed at http://jenkins.o-ran-sc.org. However, this URL is only for job status viewing. Jenkins job definitions can not be created or modified on this web page. Instead jobs are add through a CI process: contributors edit files in the **ci-management** repo on gerrit.o-ran-sc.org; changes are submitted and merged; a Jenkins Job Builder (JJB) process is triggered that builds files in the ci-management repo into Jenkins Job definitions and pushes definitions onto this Jenkins server. From here on, the new jobs can be viewed from the http://jenkins.o-ran-sc.org web site, and triggered via Gerrit events, clock values, etc.

This document provides an overview of how to write JJB definition files into the ci-management repo. The repo contains a **global-job** submodule that has a large body of existing templates and scripts that O-RAN SC jobs can reference. Documentation from Linux Foundation for global-job can be found at: https://docs.releng.linuxfoundation.org/projects/global-jjb/en/latest/index.html.

- Directory Structure
- Docker based Building
  - Dockerfile
    - Building for Docker Container Image
    - Building for Linux Software Package
  - JJB Definition
    - Building for Docker Container Image
    - Building for Linux Software Package
- How to Add CI for My Repo

Directory Structure

Below is the directory structure of the ci-management repo, where the most interesting potion is the jjb directory where the per-repo job definitions reside. This is where we want to make changes for defining new or modifying existing Jenkins jobs. The global-jjb directory is initially empty when the repo is cloned. It is a submodule that can be populated by running: "git submodule update --init". Once initialized, this directory contains the global-jjb templates managed by the Linux Foundation Release Engineering team.

ci-management

```
.
|-- README.md
|-- docker
|   |-- README.md
|   |-- bldr-alpine3
|   |-- bldr-debian-python
|   |-- bldr-ubuntu16-c-go
|   `-- bldr-ubuntu18-c-go
|-- global-jjb
|-- jenkins-config
|   |-- clouds
|   |-- global-vars-production.sh
|   `-- global-vars-sandbox.sh
|-- jjb
|   |-- ci-management
|   |-- com-asn1
|   |-- com-golog
|   |-- com-log
|   |-- com-pylog
|   |-- common-views.yaml
|   |-- defaults.yaml
|   |-- doc
|   |-- global-jjb
|   |-- it-dep
|   |-- it-otf
|   |-- it-test
|   |-- oam
|   |-- oran-jjb
|   |-- portal-ric-dashboard
|   |-- pti-rtp
|   |-- read-the-docs
|   |-- ric-app-admin
|   |-- ric-app-mc
|   |-- ric-plt-a1
|   |-- ric-plt-appmgr
|   |-- ric-plt-dbaas
|   |-- ric-plt-e2
|   |-- ric-plt-e2mgr
|   |-- ric-plt-lib-rmr
|   |-- ric-plt-resource-status-manager
|   |-- ric-plt-rtmgr
|   |-- ric-plt-sdl
|   |-- ric-plt-sdlgo
|   |-- ric-plt-submgr
|   |-- ric-plt-tracelibcpp
|   |-- ric-plt-tracelibgo
|   |-- ric-plt-vespamgr
|   `-- shell
|-- packer
|   |-- common-packer
|   |-- provision
|   `-- templates
`-- tox.ini
```

Docker based Building

Many O-RAN SC repos have adopted the docker based building process. In this process, repo dev team provides a Dockerfile that completes all the steps of installing building artifacts, including installing additional tools, configuring the build, executing the building, and generating the result artifacts. The JJB build job will simply invoke docker build and if applicable publishes result artifact to relevant repository. This is the process we focus here in this document. For more "conventional" approach of defining JJB jobs, there are other documents such as this page for the ONAP project: https://wiki.onap.org/display/DW/Using+Standard+Jenkins+Job+%28JJB%29+Templates.

Compare to the traditional Linux Foundation JJB building process where the building is done on Linux Foundation standard build VMs, the docker based approach is very versatile, which is extremely important to the O-RAN SC because the programming technologies used by our repos are very diverse. To help with this building approach, a number of standard building base images are supported by the O-RAN SC Integration team. Each of these images provides the necessary tools for building a certain type of repos, for example for a specific programming language, and work flows..

Following the Docker based building approach, the CI is done at two places:

1. A docker file within the code repo which completes the build. This is under dev team's control.
2. a JJB definition in ci-management. This is under Linux Foundation engineer control.

The following subsections explain what needs to be done at each location using existing O-RAN-SC repos as examples.

Dockerfile

This is where the actual building happens. Depends on the type of artifact to be built, docker container image or Linux software package (rpm/deb), the process differs.

Building for Docker Container Image

Here we use the ric-plt/e2mgr repo as example because it builds into a docker container image for deployment.

First there needs to be a **container-tag.yaml** file to tell the building process what version tag to give to the result docker image. Note we use Semantic Versioning for tagging docker container images.

E2Manager/container-tag.yaml

```
# The Jenkins job requires a tag to build the Docker image.
# Global-JJB script assumes this file is in the repo root.
---
tag: 3.0.1
```

There must also be a **Dockerfile** that actually prescribes the building process. For ric-plt/e2mgr, it is a Dockerfile (see below) that actually builds two containers, the first one for building binaries (lines 1-26), and the second for run-time (lines 28-46) which copies only files needed for running the executable from the first container. This is a common practice for reducing runtime Docker container size. Only the second container is tagged with what the container-tag.yaml specifies and pushed to nexus3.o-ran-sc.org. Below is the line by line explanation for the Dockerfile.

Line 1: using an Integration team supported base building image for Go language for the first container.

Line 3-13: installing dependencies, configuring, and building binary executable.

Line 15-21: unit test.

Line 23: using a Ubuntu base image for the second container.

Line 25-29: install additional software.

Line 31-35: copy needed e2mgr runtime files from the first container.

Line 36-41: set up the second container for running the E2 Manager executable.

E2Manager/Dockerfile

```
FROM nexus3.o-ran-sc.org:10004/bldr-ubuntu16-c-go:2-u16.04-nng as ubuntu

WORKDIR /opt/E2Manager
COPY . .
ENV PATH=$PATH:/usr/local/go/bin:/usr/lib/go-1.12/bin
# Install RMr library and dev files
RUN wget --content-disposition  https://packagecloud.io/o-ran-sc/staging/packages/debian/stretch/rmr_1.10.0
_amd64.deb/download.deb
RUN dpkg -i rmr_1.10.0_amd64.deb
RUN wget --content-disposition https://packagecloud.io/o-ran-sc/staging/packages/debian/stretch/rmr-dev_1.10.0
_amd64.deb/download.deb
RUN dpkg -i rmr-dev_1.10.0_amd64.deb

RUN cd asn1codec && make
RUN go build app/main.go

# Execute UT
ENV LD_LIBRARY_PATH=/usr/local/lib


ENV GODEBUG=cgocheck=2,clobberfree=1,gcstoptheworld=2,allocfreetrace=0
ENV RIC_ID="bbbccc-abcd0e/20"
RUN go test ./...

FROM ubuntu:16.04

RUN apt-get update && apt-get install -y \
  net-tools \
  iputils-ping \
  curl \
  tcpdump

COPY --from=ubuntu /opt/E2Manager/router.txt /opt/E2Manager/router.txt
COPY --from=ubuntu /opt/E2Manager/main /opt/E2Manager/main
COPY --from=ubuntu /opt/E2Manager/resources /opt/E2Manager/resources
COPY --from=ubuntu /usr/local/lib/librmr_nng.so.1 /usr/local/lib/librmr_nng.so.1
COPY --from=ubuntu /usr/local/lib/libnng.so.1 /usr/local/lib/libnng.so.1
WORKDIR /opt/E2Manager
ENV LD_LIBRARY_PATH=/usr/local/lib \
    port=3800
ENV RMR_SEED_RT=router.txt
EXPOSE 3800
CMD ["sh", "-c", "./main  -port=$port"]
```

Building for Linux Software Package

Below is the Dockerfile for the ric-plt/lib/rmr repo, which builds into a deb and a rpm package, as an example for how to build Linux Software package in the docker based building process.

Here the Dockerfile is much simpler because it delegates most of the building work to a building script. The building and packaging all happen within the script. At the end, the resultant rpm/deb packages are placed under a specific location (**/export**). This kind of building must use the Integration team supported **buildpack-deps** base image, which has the necessary directories mounted at expected locations. This way after the docker build command completes, the JJB job can pick up and ship the built packages to packagecloud.io.

ci/Dockerfile

```
FROM buildpack-deps:stretch
RUN apt-get update && apt-get -q -y install cmake ksh alien
ADD . /tmp
WORKDIR /tmp

# build RMr, run unit tests, and generate packages and package lists
RUN ksh ci/ci_build.ksh

# Executing the container "as a binary" will cause the CI publish
# script to execute.  This will take the simple package list generated
# by the ci_build script and copy the list of packages to the target
# directory.  The target directory is /export by default, but can be
# overridden from the docker run command line. In either case, the
# assumption is that the target directory is mounted as a volume.
#
ENTRYPOINT [ "ci/publish.sh" ]
```

JJB Definition

Building for Docker Container Image

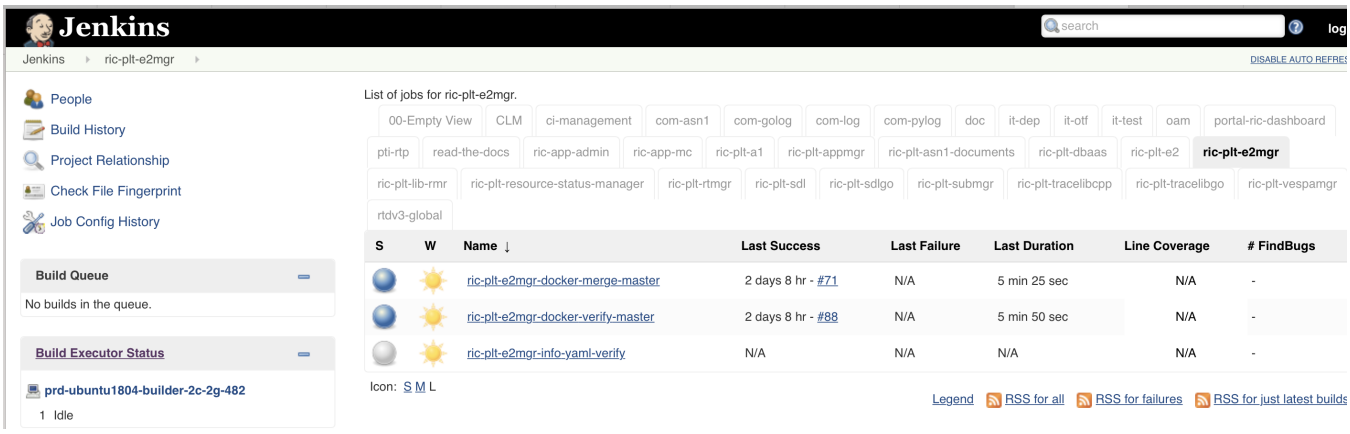
Again, the JJB definition file for the E2 manager is used as example for illustrating how to define docker image building jobs. The JJB definition for the E2 Manager can be found at the jjb/ric-plt-e2mgr directory of the ci-management repo.

job/ric-plt-e2mgr/ric-plt-e2mgr.yaml

```
---
....
- e2mgr_docker_common: &e2mgr_docker_common
    # values apply to all E2MGR docker projects
    name: e2mgr-docker-common
    # git repo
    project: ric-plt/e2mgr
    # jenkins job name prefix
    project-name: ric-plt-e2mgr
    # maven settings file has docker credentials
    mvn-settings: ric-plt-e2mgr-settings

- project:
    <<: *e2mgr_docker_common
    name: ric-plt-e2mgr
    # image name
    docker-name: '{name}'
    # Dockerfile is in a subdir
    docker-root: E2Manager
    # source of docker tag
    container-tag-method: yaml-file
    # use host network to clone from our gerrit
    docker-build-args: '--network=host'
    jobs:
      - '{project-name}-gerrit-docker-jobs'
    stream:
      - master:
          branch: master
....
```

Lines 24 and 25 declare the Jenkins Jobs by referencing job (or job group) definition templates in the global-job/jjb folder. In this example, the final jobs pushed on to the Jenkins server are with the following names: ric-plt-e2mgr-docker-merge-master and ric-plt-e2mgr-docker-verify-master.



Obviously, when copying such a block into a JJB definition file for a new repo, all the references to e2mgr need to be modified. In addition, line 19 is another important line. It specifies where the Dockerfile and container-tag.yaml files are provided. In this example, they are under the E2Manager subdirectory from the repo root. It must be customized to point where in the new repo the Dockerfile and container-tag.yaml files are stored.

Building for Linux Software Package

Again, the the JJB definition file for RMR library is used as example for illustrating how to define Linux software package building jobs. The JJB definition for the RMR library can be found under jjb/ric-plt-lib-rmr of the ci-management repo.

jjb/ric-plt-lib-rmr/ric-plt-lib-rmr.yaml

```
...
- rmr_common: &rmr_common
    name: rmr-common
    # git repo
    project: ric-plt/lib/rmr
    # jenkins job name prefix
    project-name: ric-plt-lib-rmr
    # maven settings file has credentials
    mvn-settings: ric-plt-lib-rmr-settings

# build and publish packages to PackageCloud.io
- project:
    <<: *rmr_common
    name: ric-plt-lib-rmr
    # image is not pushed, use trivial tag
    container-tag-method: latest
    # image name
    docker-name: '{name}'
    # use host network for network resources
    docker-build-args: '--network=host -f ci/Dockerfile'
    # exclude changes in the bindings subdirectory
    gerrit_trigger_file_paths:
      - compare-type: REG_EXP
        pattern: '^((?!src/bindings|\/COMMIT_MSG).)*$'
    jobs:
      - gerrit-docker-verify
      - oran-gerrit-docker-ci-pc-merge
    stream:
      - master:
          branch: master
...
```

The job definition is quite similar to the one building docker image. Here are some of the differences:

line 21: this line directly supplies the arguments to the docker build command, among which specifies which Dockerfile to use. With this approach the docker-root parameter is no longer needed.

line 22-24: this is a new trick, where we can tell Gerrit to only trigger this job if there are changes fitting certain pattern.

line 25-27: two different job templates are referenced here. In particular, the **oran-gerrit-docker-ci-pc-merge** job template has the steps for executing the docker build, and publishing result files placed in "/export" directory of the docker container by the building process to PackageCloud.

How to Add CI for My Repo

With an understanding of how CI is defined, new Jenkins jobs can be added by following what the examples above are doing:

1. Add Dockerfile and necessary scripts that complete the building of artifacts; test it locally, submit and merge the change;
2. Create a new yaml file to the ci-management repo at jjb/{{transformed_repo_name}}/{{transformed_repo_name}}.yaml, where the "transformation" is to replace '/' by '-', by copying from a similar repo; modify the file; submit the change to Gerrit for ci-management committers for review.
    a. There is a Jenkins SandBox that can be used for testing JJB definition before submitting to Gerrit for reviewing. Detailed documentation for how to use the sandbox can be found here: https://docs.releng.linuxfoundation.org/en/latest/jenkins-sandbox.html
3. Once the ci-management change is merged, find the newly defined jobs appearing at http://jenkins.o-ran-sc.org under the {{transformed_repo_name}} tab. You will not likely see the jobs actually started because they will only tart when one of the Gerrit or timestamp trigger is activated.