Using Standard Jenkins Job (JJB) Templates

This page introduces how the ONAP toolchain is setup and particularly how the Jenkins instance(s) is/are managed

In order to version control the Jenkins configuration, the configuration of all Jenkins jobs is managed though a tool called Jenkins Job Builder (JJB).

The below pages give an overview of how this tool is used in ONAP and how the current jobs are setup

Preamble

Though the documentation is well written and provides all the needed information to understand how to write jobs, we wanted to give another approach on the Jenkins Job Builder usage. It's mainly lead by our discovery of the tool on our implementation case.

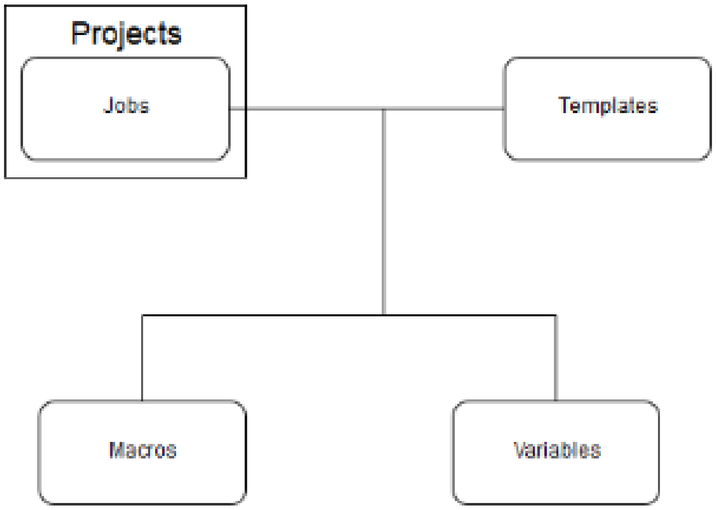
For most of our questions, the official documentation had the answer. Please consider reading it in case of trouble. If you think the information is interesting enough to share, we would be very pleased if you can make a contribution to this document.

For the sake of ease, you can find the official documentation here : http://docs.openstack.org/infra/jenkins-job-builder/

Overview

JJB, or Jenkins Job Builder, allows you to get your Jenkins job setup with simple human reable *yaml files*.
It allows you to have Jenkins configuration stored as code, versionnable, reusable, auditable.

As a picture is worth few lines, here are both in one:



Using Jenkins Job Builder, we describe projects that contains *jobs*. Those jobs are mainly based on *templates*.
And to describe both, you have access to user defined *macros* and *variables*.

The projects will then be mapped to grouping tabs on Jenkins, and jobs to the actual jenkins jobs. Note that the project can describe things that will be common to each jobs.

For the sake of example, the following guide will use the MSO project to picture how to introduce the jobs needed.

File hierarchy

The following file hierarchy of the repository (ci-management) is similar to the following :

```
jenkins-scripts
jjb
--- ci-management
 | --- ci-management-macros.yaml
 | --- ci-management.yaml
 | --- raw-include-packer-validate.sh
--- global-defaults.yaml
--- global-macros.yaml
--- global-templates-java.yaml
--- global-templates-python.yaml
--- include-raw-deploy-archives.sh
--- mso
  | --- mso-libs.yaml
  | --- mso.yaml

packer
scripts
```

The main and most interesting part is the *jjb* folder where the whole Jenkins Job Builder configuration lies.

The folders contains project specific information while files at the main folder containers templates, macros and default global values like their names states.

Projects

For Mso projects, we wanted to separate the main Mso app and the libs that should be compiled, tested and deployed. For that purpose, we created 2 yaml files in the *mso* folder : one for the main mso app, *mso.yaml*, and on for the mso-lib repository which contains the modified openstack library, *mso-libs. yaml*. If we had more libs, we would have integrated those in the mso-lib file.

File structure

```
---
- project:
    name: mso-libs
    project-name: 'mso-libs'
    jobs:
      - '{project-name}-{stream}-verify-java'
      - '{project-name}-{stream}-merge-java'
      - '{project-name}-{stream}-release-java-daily'

    project: 'mso/libs'
    stream:
      - 'master':
          branch: 'master'
    mvn-settings: 'mso-settings'
    # due to a strange macro / variable translation problem this needs to be
    # passed as a string block to properly get the properties correctly defined
    # in the job
    maven-deploy-properties: |
      deployAtEnd=true
    files: '**'
    archive-artifacts: ''
```

The most interesting part of this file is the *jobs* section. It lists all the jobs that should be created.

The 3 jobs are composed of the name of a template, which itself has 2 parameters : *project-name* and *stream*.

The values for those parameters are taken from the variable project-name and the stream list. This will result in 3 jobs in Jenkins. By adding another stream, then there would be 6 jobs deployed on Jenkins, 3 for the master stream, 3 for the added stream.

Another interesting part is the *project* section. Which corresponds to the *gerrit* project that is concerned by out jobs.

Official documentation

http://docs.openstack.org/infra/jenkins-job-builder/definition.html#project

Templates

```
job-template:
    name: '{project-name}-{stream}-verify-java'

    project-type: freestyle
    concurrent: true
    node: '{build-node}'

    properties:
      - ecomp-infra-properties:
          build-days-to-keep: '{build-days-to-keep}'

    parameters:
      - ecomp-infra-parameters:
          project: '{project}'
          branch: '{branch}'
          refspec: 'refs/heads/{branch}'
          artifacts: '{archive-artifacts}'

    scm:
      - gerrit-trigger-scm:
          refspec: '$GERRIT_REFSPEC'
          choosing-strategy: 'gerrit'

    wrappers:
      - ecomp-infra-wrappers:
          build-timeout: '{build-timeout}'

    triggers:
      - gerrit-trigger-patch-submitted:
          server: '{server-name}'
          project: '{project}'
          branch: '{branch}'
          files: '**'

    builders:
      - provide-maven-settings:
          global-settings-file: 'global-settings'
          settings-file: '{mvn-settings}'
      - maven-target:
          maven-version: 'mvn33'
          goals: 'clean install'
          settings: '{mvn-settings}'
          settings-type: cfp
          global-settings: 'global-settings'
          global-settings-type: cfp
```

The global templates can be found in the the *global-templates-java.yaml* file. Those are generic templates, that should meet most project needs.

Note: if your project needs more specific ones, you should be able to overwrite the template parameters in the projects. If it's not sufficient, please create your template in your project directory.

Current ones contain the following parts : - name - node - properties - parameters - scm - wrappers - triggers - builders

name

The name can contain *parameters* that will be expanded using the variables provided upstream (in the project). Example : `name: '{project-name}-{stream}-verify-java'`

If the project-name of the project using that template is 'mso' and its stream is 'master', then it will be expanded to 'mso-master-verify-java'

Note that the naming convention we would like to keep on to keep on the project is trigger related. Also, please append the language name so that it doesn't conflicts with other programming language projects.

node

The Jenkins minion that will be used to execute the job. See *build-node* variable in *global-defaults* for the default one. Other possibilities can be found at the end of this document.

properties

This sections should be used to set properties from jenkins job properties.

http://docs.openstack.org/infra/jenkins-job-builder/properties.html

parameters

This section provides allows you to setup build parameters for the Jenkins job.

http://docs.openstack.org/infra/jenkins-job-builder/parameters.html

scm

This section provides information about which repo should be fetched, and how. In the example case, the `gerrit-trigger-scm` is a macro that contains the *gerrit* definition to fetch a repo.

http://docs.openstack.org/infra/jenkins-job-builder/scm.html

wrappers

Wrappers are intended to change the behaviour of the build. See the official documentation in case you might need them.

http://docs.openstack.org/infra/jenkins-job-builder/wrappers.html

triggers

As there's no manual execution of the jobs (except on special request), the triggers describe how and when the build should be executed.

Most interesting ones are:

- time based (cron like) triggers
- gerrit event specific triggers
    - by pathc submmitted or merged
    - by comment posted

http://docs.openstack.org/infra/jenkins-job-builder/triggers.html

Comment posted triggers

Gerrit comments also trigger most builds. The current comments supported are:

"run-sonar" - to trigger the Sonar scan job
"run-clm" - to trigger the IQ scan
"reverify" - to trigger a verification on an unmerged change
"remerge" - to trigger a merge job, useful whenever the automated merge job fails
"please release" - to trigger the daily release job to post binaries in Nexus.

For more detail on how these triggers are configured, please refer to:

https://github.com/onap/ci-management

builders

The builders is the build configuration of the projects. It will be mapped to the jenkins *Build* section of the job configuration.

Examples :

- shell builder : provides a simple way to execute shell actions
- maven-target : to execute the maven top level goals
- config-file-provider:

Note that in the example template further above in this document, the `provide-maven-settings` is a macro that adds a `config-file-provider` builders with some default options.

http://docs.openstack.org/infra/jenkins-job-builder/builders.html

Official documentation

http://docs.openstack.org/infra/jenkins-job-builder/definition.html#job-template

Macros

In JJB, there options that can take quite a few parameters while those parameters remains set with the same value on multiple job or templates.

Macro are interesting in this case as you can provide a *name* and default options for a specific element.

Example of use are a **scm configuration** or a **specific builder** with a shell script to update a revision file.

Some global macros are available in the `global-macros.yaml`. We encourage you to wrote project specific macros in your project folder.

Variables

Only useful information here is that global variables used as default values are stored in `global-defaults.yaml`.

If you would need project specific variables, we encourage you to create them in your project folder.

Defaults

Defaults files are for storing information that should available for every job (For example, max number of builds to keep)

They are similar to templates, they also allow to reuse part of our configuration, but are less flexible, not allowing variables for example.

We use them to store basic information.

`global-defaults.yaml` contains default values that will automatically be applied to any Jenkins job unless we specifically say otherwise.

Release process

Direct access

The project technically has direct rights to create a release artifact and push it into the releases repository without going through staging. There's no project specific credentials to setup.

Staging

Normally staging is used for larger releases that involve needing to have fully integrated test across multiple different projects, we can, however utilize it for single projects.

To do a staging release we normally do a little more scripting around it as we configure the staging profiles to need to be explicitly selected.

Testing your changes

You can test your configuration on a local Jenkins or get access to the sandbox

Note that you will be able to preview it your changes, but except for the most basic ones, that will not run. The sandbox isn't configured to mimic the prod Jenkins

In order to get access to the sandbox, send a request email to HelpDesk specifying your linux foundation account

There are 2 actions you can take to test your changes :

- use the `jenkins-jobs test` command
- upload your jobs to the sandbox using

You will need to configure to which jenkins `jenkins-job` connect to and how with a configuration file

Beware that query_plugins_info=False must be present in the [Jenkins] section of your file

Install & configuration

First you need to install Jenkins job builder tool. This is done using the pip software :

```
$ pip install --user jenkins-job-builder
```

Then copy the configuration script provided at the root of the repository, `jenkins.init-example`, in `~/.config/jenkins_jobs/jenkins_jobs.ini`

And edit to update your personal information. Beware that the **password field should not contain your password**, but you **jenkins API key**.

Jenkins-jobs test

This command generates the jenkins configuration files based on the jenkins job builder files.

Usage is : `jenkins-jobs test <path_to_jjb_folder>`

Jenkins-jobs update

If you have setup the sandbox server in the configuration file, then you should be able to push your changes to the sandbox server.

Note that all those who have write access to the server might update the configuration at anytime. Please use this only when required.

Jobs cannot be executed on the server, it's only a configuration check.

Types of Jobs :

Verify job

A verify job is meant to be run at each review, it will build the project with the proposed changes and will report the results, this will also be reported in gerrit reviews. verify+1 means the build was a success and the change is safe to be merged

This job should not produce any artifacts is only meant to verify a proposed change (no deploy)

Note : in order to trigger that job again, a keyword can be used in gerrit review, anyone can just reply : *recheck* and this will trigger a new build

Merge job

A merge job is meant to be run after changes are merged to the target branch, it will build the project and produce SNAPSHOT artifacts, this job should be triggered automatically upon successful merge.

Note : in order to trigger that job again, a keyword can be used in gerrit review, anyone can just reply : *remerge* and this will trigger a new build

Release job

A release job is meant to produce releasable artifacts, these are currently setup to run daily. it will build the project and produce RELEASE artifacts. Note that these are pushed to staging, though the use of staging profiles.

Note : in order to trigger that job again, a keyword can be used in gerrit review, anyone can just reply : *please release* and this will trigger a new build

Maven Staging

In order to push to a staging repository instead of directly to the release repository, you need to add the nexus-maven-staging plugin.

This plugin will automagically (as they say) take over the deploy phase.

The configuration differs a bit if you use solely maven3 or mix it with maven2.

For maven3 it's quite simple, you have to add the plugin to you `<build>` section :

blocked URL

If you need a maven2 / maven3 mix or more information, refer to this page

The behavior when the plugin is added will be as follows :

- if you have a SNAPSHOT version in your pom : the deploy will push to SNAPSHOT repository
- if you have a RELEASE version in your pom : the deploy phase will push to the STAGING repository

Example

Here are the mso jobs :

```
---

- project:

    name: mso

    project-name: 'mso'

    jobs:

      - '{project-name}-{stream}-verify-java'

      - '{project-name}-{stream}-merge-java'

      - '{project-name}-{stream}-release-java-daily'


    project: 'mso'

    stream:

      - 'master':

          branch: 'master'

    mvn-settings: 'mso-settings'

    files: '**'

    archive-artifacts: ''

    build-node: ubuntu1604-basebuild-4c-4g
```

They point to templates located in `global-templates-java.yaml`. Parameters that are 'passed' or 'overridden' from the template are marked in **greer** **.** For example, we also pass archive-artifacts but it's set to blank.

So let's look at **`{project-name}-{stream}-verify-java`** :

```
---

- job-template:

    # Job template for Java verify jobs

    #

    # The purpose of this job template is to run "maven clean install" for projects using this template.

    #

    # Required Variables:

    #     branch:    git branch (eg. stable/lithium or master)


    name: '{project-name}-{stream}-verify-java'


    project-type: freestyle
```

```
concurrent: true
node: '{build-node}'

properties:
  - ecomp-infra-properties:
      build-days-to-keep: '{build-days-to-keep}'

parameters:
  - ecomp-infra-parameters:
      project: '{project}'
      branch: '{branch}'
      refspec: 'refs/heads/{branch}'
      artifacts: '{archive-artifacts}'

scm:
  - gerrit-trigger-scm:
      refspec: '$GERRIT_REFSPEC'
      choosing-strategy: 'gerrit'

wrappers:
  - ecomp-infra-wrappers:
      build-timeout: '{build-timeout}'

triggers:
  - gerrit-trigger-patch-submitted:
      server: '{server-name}'
      project: '{project}'
      branch: '{branch}'
      files: '**'

builders:
  - provide-maven-settings:
      global-settings-file: 'global-settings'
      settings-file: '{mvn-settings}'
  - maven-target:
      maven-version: 'mvn33'
      goals: 'clean install'
      settings: '{mvn-settings}'
      settings-type: cfp
      global-settings: 'global-settings'
```

```
          global-settings-type: cfp
```

In this templates, we have links from `global-macros.yaml`. They will be in **`blue`**

Docker Images

The following components were used for building MSO docker images.

Note that this is usable by components that are using maven to build the docker image as recommended in the docker strategy.

It may be that your components needs to add new macros and templates based on your project specific way to build the docker image

MSO project pom.xml

The pom.xml configuration use in MSO, is using a sepcific pom.xml that uses a docker maven plugin to generate the image and push it to the nexus repository

pom.xml

```
                        <plugin>
                                <groupId>io.fabric8</groupId>
                                <artifactId>docker-maven-plugin</artifactId>
                                <version>0.16.5</version>
                                <configuration>
                                        <verbose>true</verbose>
                                        <apiVersion>1.23</apiVersion>
                                        <images>
                                                <image>
                                                        <name>onap/wildfly:1.0</name>
                                                        <alias>mso-arquillian</alias>
                                                        <build>
                                                                <cleanup>try</cleanup>
                                                                <dockerFileDir>docker-files</dockerFileDir>
                                                                <dockerFile>docker-files/Dockerfile.wildfly-10<
/dockerFile>
                                                        </build>
                                                </image>
                                                <image>
                                                        <name>onap/mso:%l</name>
                                                        <alias>mso</alias>
                                                        <build>
                                                                <cleanup>try</cleanup>
                                                                <dockerFileDir>docker-files</dockerFileDir>
                                                                <dockerFile>docker-files/Dockerfile.mso-chef-
final</dockerFile>
                                                                <assembly>
                                                                        <basedir>/</basedir>
                                                                        <user>jboss:jboss:jboss</user>
                                                                        <basedir>/opt/jboss/wildfly/standalone
/deployments</basedir>
                                                                        <descriptor>../../../../deliveries/src
/main/assembly/war-pack/mso-wars.xml</descriptor>
                                                                </assembly>
                                                        </build>
                                                </image>
                                        </images>
                                </configuration>
                                <executions>
                                        <execution>
                                                <id>clean-images</id>
                                                <phase>pre-clean</phase>
                                                <goals>
                                                        <goal>remove</goal>
                                                </goals>
                                                <configuration>
                                                        <removeAll>true</removeAll>
                                                        <image>onap/mso-arquillian:%l,onap/mso:%l</image>
                                                </configuration>
                                        </execution>
                                        <execution>
                                                <id>generate-images</id>
                                                <phase>generate-sources</phase>
                                                <goals>
                                                        <goal>build</goal>
                                                </goals>
                                        </execution>
                                        <execution>
                                                <id>push-images</id>
                                                <phase>deploy</phase>
                                                <goals>
                                                        <goal>build</goal>
                                                        <goal>push</goal>
                                                </goals>
                                                <configuration>
                                                        <image>onap/mso-arquillian:%l,onap/mso:%l</image>
                                                </configuration>
                                        </execution>
                                </executions>
                        </plugin>
```

Jenkins Job Builder configuration

In the LF infrastructure, you need to define a specific job that will be responsible of building that specific pom.xml, see below the sample configuration for MSO

mso/mso.yaml

```
- project:
    name: mso
    project-name: 'mso'
    jobs:
      - ...
      - '{project-name}-{stream}-docker-java-daily':
          docker-pom: 'pom.xml'
          # use 'default' as value if you don't have any mvn profile
          mvn-profile: 'docker'
    project: 'mso'
    stream:
      - 'master':
          branch: 'master'
    mvn-settings: 'mso-settings'
    files: '**'
    archive-artifacts: ''
    build-node: ubuntu1604-basebuild-4c-4g
```

In the MSO project, we added a *{project-name}-{stream}-docker-java-daily* job. We need to provide the path to the pom building the docker under the *docke r-pom* parameter.

The build-node should have docker installed. Currently, Only the ubuntu1604-8c-8g nodes do have docker installed. The template *{project-name}-{stream}-docker-java-daily* doesn't use the build-node provided in the project but forces the usage of the ubuntu1604-8c-8g. So if you create your own template for docker, your are able to force the right image to be used only for that purpose.

the following job template is used along with the macro defining all the needed parameters for the job to execute (this essentially defines all the params in a freestyle jenkins job, pulling the repo, running maven on the docker pom as well as the goals needed and environment for the plugin to know where to push the images)

Please note the special docker-login builder, used to perform a docker login to the registries before pushing your image, it will use the script located in ci-management/jjb

global-templates-java.yaml

```
- job-template:
    name: '{project-name}-{stream}-docker-java-daily'
    project-type: freestyle
    build-node: centos7-basebuild-4c-4g
    properties:
      - ecomp-infra-properties:
          build-days-to-keep: '{build-days-to-keep}'
    parameters:
      - ecomp-infra-parameters:
          project: '{project}'
          branch: '{branch}'
          refspec: 'refs/heads/{branch}'
          artifacts: '{archive-artifacts}'
    scm:
      - gerrit-trigger-scm:
          refspec: ''
          choosing-strategy: 'default'
    wrappers:
      - ecomp-infra-wrappers:
          build-timeout: '{build-timeout}'
    triggers:
      # 11 AM
UTC

      - timed: 'H 12 * * *'
    builders:
      - provide-maven-settings:
          global-settings-file: 'global-settings'
          settings-file: '{mvn-settings}'
      - docker-login
      - maven-docker-push-daily:
          mvn-settings: '{mvn-settings}'
          pom: '{docker-pom}'
          mvn-profile: '{mvn-profile}'
```

global-macros.yaml

```
- builder:
    name: maven-docker-push-daily
    builders:
      - maven-target:
          maven-version: 'mvn33'
          pom: '{pom}'
          goals: 'clean deploy -B -P {mvn-profile}'
          settings: '{mvn-settings}'
          settings-type: cfp
          global-settings: 'global-settings'
          global-settings-type: cfp
          properties:
            - maven.test.skip=true
            - docker.pull.registry=nexus3.onap.org:10001
            - docker.push.registry=nexus3.onap.org:10003
```

Images can have multiple tags (versions) and the latest tag is a special tag that should be moved once a new build is available

Following our discussions with the LF, they propose to slightly adapt our naming convention to be able to store staging docker images

We have 2 repositories :

Snapshot repository –Staging repository

The LF proposal is to use the Snapshot repository for storing snapshot images. There is also a staging repository to store staging (release candidate) images.

In order to differentiate them, we should add a label to easily identify them.

These repositories would hold our daily builds (as detailed in the wiki) and we should have a common way for users to pull the latest version of the build, this is the purpose of the **latest** tag that you can move to the newest image.

The latest staging image is going to be used by instantiation to run E2E tests.

We should have a common naming scheme for all docker images like we have now : **onap/<image>**

We should all align on a common tag format that uniquely identify which project version this image relates to, we propose to also add a timestamp and a label (staging or snapshot), this would identify the image uniquely : **X.Y.Z-STAGING-TIMESTAMP or X.Y.Z-SNAPSHOT-TIMESTAMP**

We are also using the following LATEST tag to identify a component : **X.Y-STAGING-**latest

<u>Example :</u>

onap/appc:1.0.0-SNAPSHOT-20170220T115900

onap/appc:1.0.0-STAGING-20170220T115900

onap/appc:1.0-STAGING-latest (this would point to the latest build and should be set automatically by the job, either through the maven plugin or with the –t option of the docker command)

In terms of image builds :

- creating a SNAPSHOT image is optional, you may want to consider building a special snapshot image for component testing but so far we have not enforced this.
- creating a STAGING image is mandatory, all E2E testing running the containers should be using STAGING images.
    - The LATEST tag is best applied to the latest STAGING build

The intent of having a staging set of images is to be able to mimic the staging behavior of Nexus2 : Once a release is ready we need to be able to pickup the STAGING image that has successfully passed E2E testing and push it to the RELEASE Repository

Release repository –

this should hold our release version that we support, here we assume having a format like : **X.Y.Z** is enough

(the **latest** tag could also be used to point to our latest release version)

As recommended by the LF: a special Jenkins Job needs to be created to pull the approved version of a particular release STAGING image (one that has been built and tested through E2E) and re tag it with the appropriate RELEASE version and push it to the RELEASE docker repository

Beware that pushing to the RELEASE repository is only allowed once per version (no ability to redeploy)

See more information on the Release process

<u>Example :</u>

onap/appc:1.0.0

onap/appc:latest

Automating the generation of the proper label

To generate the right label automatically during your build you can use the following method (this is valid for a maven build but can easily be ported to a script build) add the following plugin into your POM.xml and use a variable to handle the tag passed to the docker plugin :

```
               <!-- If the maven profile "docker" is specified the parameter -Dmso.git.url=<MsoGitRepo> must be provided
                     i.e: mvn clean install -P docker -Dmso.git.url=https://gerrit.onap.org/r-->
                <mso.git.url>${env.GIT_NO_PROJECT}</mso.git.url>
+                <mso.project.version>${project.version}</mso.project.version>
        </properties>

        <build>
                <finalName>${project.artifactId}-${project.version}</finalName>
                <plugins>
+                     <plugin>
+                 <groupId>org.codehaus.groovy.maven</groupId>
+                 <artifactId>gmaven-plugin</artifactId>
+                 <executions>
+                   <execution>
+                     <phase>validate</phase>
+                     <goals>
+                       <goal>execute</goal>
+                     </goals>
+                     <configuration>
+                       <source>
+                         println project.properties['mso.project.version'];
+                         def versionArray;
+                         if ( project.properties['mso.project.version'] != null ) {
+                             versionArray = project.properties['mso.project.version'].split('\\.');
+                         }
+
+                         if ( project.properties['mso.project.version'].endsWith("-SNAPSHOT") ) {
+                             project.properties['project.docker.latesttag.version']=versionArray[0] + '.' + versionArray[1] + "-SNAPSHOT-latest";
+                         } else {
+                             project.properties['project.docker.latesttag.version']=versionArray[0] + '.' + versionArray[1] + "-STAGING-latest";
+                         }
+
+                         println 'New Tag for docker:' + project.properties['project.docker.latesttag.version'];
+                       </source>
+                     </configuration>
+                   </execution>
+               </executions>
+             </plugin>
                        <plugin>
                                <groupId>org.apache.maven.plugins</groupId>
                                <artifactId>maven-scm-plugin</artifactId>
@@ -111,6 +141,7 @@
                                                        <build>
                                                                <tags>
                                                                        <tag>${project.version}-STAGING-${maven.build.timestamp}</tag>
+                                                                        <tag>${project.docker.latesttag.version}</tag>
                                                                </tags>
                                                                <cleanup>try</cleanup>
                                                                <dockerFileDir>docker-files</dockerFileDir>
```

SONAR

Sonar instance is available at https://sonar.onap.org/

Sonar push is typically done on the daily release job and is included in templates such as '{project-name}-{stream}-release-version-java-daily'.

Here is the specific step that does the sonar scan in this profile, in case you need to add it to a new profile :

blocked URL

In your pom.xml, add this plugin configuration (for more information on plugin parameters, please refer to http://docs.sonarqube.org/):

```
<plugin>
<groupId>org.codehaus.mojo</groupId>
<artifactId>sonar-maven-plugin</artifactId>
<version>3.2</version>
</plugin>
```

JavaDoc

This explains how to configure your project to upload the javadoc to the Linux Foundation Nexus.

Step-by-step guide

Project configuration

1. To allow javadoc generation, add the maven-javadoc-plugin to your project pom.xml :

   ```
   <reporting>
     <plugins>
       <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-javadoc-plugin</artifactId>
         <version>2.10.4</version>
         <configuration>
           <failOnError>false</failOnError>
           <doclet>org.umlgraph.doclet.UmlGraphDoc</doclet>
           <docletArtifact>
             <groupId>org.umlgraph</groupId>
             <artifactId>umlgraph</artifactId>
             <version>5.6</version>
           </docletArtifact>
           <additionalparam>-views</additionalparam>
           <useStandardDocletOptions>true</useStandardDocletOptions>
         </configuration>
       </plugin>
     </plugins>
   </reporting>
   ```

2. Add the maven-site plugin :

   ```
   <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-site-plugin</artifactId>
     <version>3.6</version>
     <dependencies>
       <dependency>
         <groupId>org.apache.maven.wagon</groupId>
         <artifactId>wagon-webdav-jackrabbit</artifactId>
         <version>2.10</version>
       </dependency>
     </dependencies>
   </plugin>
   ```

3. Distribution management setup :

   ```
   <properties>
           ...
           <nexusproxy>https://nexus.onap.org</nexusproxy>
           <sitePath>/content/sites/site/org/onap/mso/${project.version}</sitePath>
           ...
   </properties>
   ```

   ```
   <distributionManagement>
       <site>
           <id>ecomp-site</id>
           <url>dav:${nexusproxy}${sitePath}</url>
       </site>
   </distributionManagement>
   ```

   Be sure to use ***ecomp-site*** as id for your site, so that it matches the server credentials provided by the Linux Foundation.

Jenkins job configuration

Add the following lines to your projects yaml definition :
project.yaml

```
- project:
     ...
     jobs:
      - '{project-name}-{stream}-stage-site-java':
          site-pom: 'pom.xml'
          trigger-job: '{project-name}-{stream}-release-version-java-daily'
     ...
```

The added job(s) will be triggered when *trigger-job* ends successfully. This allow to publish the staging version documentation only when the staging release build succeeds and avoid overwriting the current documentation. The template ***{project-name}-{stream}-stage-site-java*** actually invokes *mvn site:site* and *mvn site:stage-deploy* on your project using the pom.xml specified as *site-pom*.