Testing Jenkins configuration changes
solidify your JCasC, Job DSL and Pipelines usage

Szymon Datko
szymon.datko@corp.ovh.com

Roman Dobosz
roman.dobosz@corp.ovh.com

OVH

1st May 2019 (Labour Day)
About us

Szymon Datko
- DevOps & local Bash wizard
- Open Source software lover
- makepkg, not war

Roman Dobosz
- Python expert
- retro/emul maniac
- emerge -vaNDu world
We already talked about Jenkins

https://www.youtube.com/watch?v=T7rD--ZOYRQ

⇑ click me! ⇑
Short recap: what is this mysterious Jenkins thing?

- One of the most popular automation servers.
- Powerful, Open Source, written in Java.
- Easy to start, configure, manage and use.
- Heavily extensible - plenty of plugins available.
- Widely used by the top IT companies!

There are three plug-ins that do come in handy for Jenkins configuration...

**Configuration as Code**

**Job DSL**

**Job Pipelines**

(Jenkinsfiles)

```groovy
pipeline {
  agent docker: 'maven:3.3.3'
  stages {
    stage('build') {
      steps {
        sh 'mvn --version'
        sh 'mvn install'
      }
    }
  }
}
```
How can we test the configuration?

Basically, we can verify two things:

- syntax,
- what it does.

~ in analogy to:

- unit tests,
- functional tests.
Workflow overview
Verifying Job Pipelines

Simplest to test configuration - parser is built-in in the Pipelines plugin.

Available via:

- Web User Interface,
  - provides hints within textarea field on Pipeline job editing page,

- HTTP API,
  - `${JENKINS_URL}/pipeline-model-converter/validate` endpoint.
  - always sends HTTP 200/OK status code (requires parsing of output),

- SSH CLI,
  - accessible via declarative-linter command,
  - requires configured user with ssh key and Overall/Read permissions,
  - returns nice exit status for shell.
Verifying Job Pipelines - example script (HTTP)

```bash
#!/bin/bash

SEARCH_DIR="${SEARCH_DIR:.-}"

JENKINS_URL="${JENKINS_URL:-https://my.jenkins.host.net}"

JENKINS_XPATH='concat(/crumbRequestField,":",/crumb)'

JENKINS_CRUMB="$(
   curl "${JENKINS_URL}/crumbIssuer/api/xml?xpath=${JENKINS_XPATH}"
)")

errors=()

while read -r jenkinsfile_path; do
   result=$(curl -X POST -H "${JENKINS_CRUMB}\n-F "jenkinsfile=<${jenkinsfile_path}\n"${JENKINS_URL}/pipeline-model-converter/validate")

   if [ "${result}" != 'Jenkinsfile successfully validated.' ]; then
      errors+=($'
"FILE: ${jenkinsfile_path}"$'
"${result}"$'
')
   fi

done < <(find "${SEARCH_DIR}" -iname '*.Jenkinsfile')

if [ ${#errors[@]} -gt 0 ]; then
   echo 'FAILURE Syntax errors encountered: THIS SHALL NOT BE MERGED!'
   echo "${errors[@]}
   exit 1
fi
```

Sz. Datko, R. Dobosz
Testing Jenkins configuration changes
1st May 2019
#!/bin/bash

SEARCH_DIR="$\{SEARCH_DIR:.-\}"

JENKINS_HOST="$\{JENKINS_HOST:-my.jenkins.host.net\}"

JENKINS_PORT="$\{JENKINS_PORT:-50000\}"

JENKINS_USER="$\{JENKINS_USER:-validator\}"

declare -i errors=0

while read -r jenkinsfile_path; do
    ssh "$\{JENKINS_USER\}@"$\{JENKINS_HOST\}" -p "$\{JENKINS_PORT\}" \ 
        declarative-linter < "$\{jenkinsfile_path\}"

        if [ $? -ne 0 ]; then
            errors++=1
        fi

done < <(find "$\{SEARCH_DIR\}" -iname '.*.Jenkinsfile')

if [ "$\{errors\}" -gt 0 ]; then
    echo 'FAILURE Syntax errors encountered: THIS SHALL NOT BE MERGED!'
    exit 1
fi

If the port number configured is random, one can find it the following way:
curl -Lv https://$\{JENKINS_HOST\}/login 2>&1 | grep -i 'x-ssh-endpoint'
No reliable syntax checker/linter on the market so far.

However:

- JCasC files are just pure YAML files,
- simple validation can detect obvious syntax errors:
  ```python
  python -c 'import yaml,sys; yaml.safe_load(sys.stdin)' < some-file.yaml
  ```

It is possible to go better:

- JCasC plugin provides the JSON Schema for configuration files under `${JENKINS_URL}/configuration-as-a-code/schema` endpoint,
- it’s entries are generated depending on the installed Jenkins plugins,
- it can be utilized for complex validation of JCasC configuration!
```python
#!/usr/bin/env python3

def validate(args):
    result = 0
    schema = get_schema(args.schema_url)

    for fname in args.instances:
        with open(fname) as fobj:
            try:
                jsonschema.validate(yaml.load(fobj), schema)
            except (jsonschema.exceptions.ValidationError,
                     jsonschema.exceptions.SchemaError) as err:
                print(err.message)
                result = 1

    return result

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-i', '--instances', nargs='+')
    parser.add_argument('-u', '--schema-url')
    sys.exit(validate(parser.parse_args()))
```
def get_schema(url):
    response = requests.get(url, verify=False)
    content = response.text

    # bad reference
    content = content.replace('"type" : "/definitions/class"', '"$ref" : "/definitions/"
    content = content.replace('"type" : "/def"', '"$ref" : "/def"

    # remove empty enums
    content = re.sub(r',s*"oneOf" : \[s*\]', '', content, flags=re.M)

    # fix bad names
    content = content.replace('[javaposse.jobdsl.dsl.GeneratedItems;
    javaposse.jobdsl.dsl.GeneratedItems']
    content = content.replace('/[Ljavaposse.jobdsl.dsl.GeneratedItems;"', '/javaposse.jobdsl.dsl.GeneratedItems"

    # fix bad references keys
    content = content.replace('"ref"', '"$ref"

    schema_dict = json.loads(content)
    return schema_dict
Verifying Job DSL

Job DSL is basically an extension of Groovy language; it might be verified in a programming-like ways.

- **syntax parsing:**
  - pass a script through the `GroovyShell().parse()`,
  - it may detect basic syntax errors on the script itself,

- **unit testing:**
  - it is possible to mock script’s parts and ensure expected calls are made,
  - this will give you an additional layer of confidence on jobs modification.
Verifying Job DSL - parser for DSL scripts

```java
import groovy.util.CliBuilder
import java.io.File

class Checker {
    static void main(String[] args) {
        def cli = new CliBuilder(
            usage: 'groovy parse.groovy [groovy-file, ...]'
        )
        def options = cli.parse(args)

        def return_code = 0
        for (String fname in options.arguments()) {
            File file = new File(options.arguments()[0])
            try {
                new GroovyShell().parse(file)
            } catch (Exception cfe) {
                System.out.println(cfe.getMessage())
                return_code = 1
            }
        }
        System.exit(return_code)
    }
}
```

Sz. Datko, R. Dobosz Testing Jenkins configuration changes 1st May 2019
Is that enough?

Words that seems legitimate does not always have a right meaning.

Taking example of Groovy language, the only way to perform the real validation is to run the code (i.e. as a functional/integration tests).

Image source: https://www.reddit.com/r/funny/comments/2ygq6b/its_is_my_life_jon_bovi/
Overview of deployment

1. Install Jenkins.
2. Install Jenkins plugins.
3. Add Jenkins configuration.
4. Skip setup wizard.
5. Adjust miscellaneous things.
6. Restart Jenkins.
7. Wait until Jenkins is ready.
8. Trigger seed job.
9. Verify seed job went fine.

Install Jenkins

You can install it manually, or just use a ready to go Ansible role:
https://github.com/geerlingguy/ansible-role-jenkins

```yaml
# tasks/install-jenkins.yml
---
- name: "Include basic vars"
  include_vars: "vars/jenkins_vars.yml"

- name: "Install curl"
  apt:
    name: 'curl'

- name: "Install Jenkins"
  include_role:
    name: geerlingguy.jenkins
```
Install Jenkins plugins

Simply download Jenkins plugins as hpi files and put in plugins/ directory.

```yaml
# tasks/install-jenkins-plugins.yml
---
- name: "Include plugin list"
  include_vars: "vars/jenkins_plugins.yml"

- name: "Install plugins using specified versions"
  get_url:
    url: "http://updates.jenkins.io/download/plugins/\{{ item.name }\}/{{ item.version }}/{{ item.name }}.hpi"
    dest: "/var/lib/jenkins/plugins/{{ item.name }}.hpi"
    owner: jenkins
    group: jenkins
    mode: 0644
    register: get_url_result
  until: get_url_result is succeeded
  retries: 3
  delay: 10
  with_items:
    - {{ jenkins_plugins }}

# vars/jenkins_plugins.yml
---
jenkinsPlugins:
  - { name: "ace-editor", version: "1.1" }
  - { name: "ansicolor", version: "0.6.2" }
  - { name: "git", version: "3.9.3" }
  - { name: "ssh-agent", version: "1.17" }
  ...
Add Jenkins configuration

# tasks/configure-jenkins.yaml

- name: "Create Jenkins Configuration-as-Code directory"
  file:
    path: /etc/jcasc
    state: directory
    owner: jenkins

- name: "Fill and copy template based JCasC files"
  template:
    src: "files/jcasc/{{ item }}.tpl"
    dest: "/etc/jcasc/{{ item }}"
    owner: jenkins
    mode: 0644
  with_items:
    - 'main.yaml'
    ...

- name: "Modify Jenkins service file"
  lineinfile:
    path: /etc/init.d/jenkins
    insertafter: "^DAEMON_ARGS=
    line: "DAEMON_ARGS="${DAEMON_ARGS}\n    --env=CASC_JENKINS_CONFIG=/etc/jcasc/"

We rely on JCasC plugin in this matter completely*

* at least as much as we can...
Skip setup wizard

Since our configuration is already handled by JCasC plugin, we can omit it.

```yaml
# tasks/skip-setup-wizard.yaml
---
- name: "Insert script to skip Jenkins setup wizard"
  copy:
    src: files/skip-setup-wizard.groovy
    dest: /var/lib/jenkins/init.groovy.d/skip-setup-wizard.groovy
    owner: jenkins
    group: jenkins
    mode: 0644

#!groovy

import jenkins.model.*
import hudson.util.*
import jenkins.install.*

def instance = Jenkins.getInstance()
instance.setInstallState(InstallState.INITIAL_SETUP_COMPLETED)
```
Adjust miscellaneous things

This is a right place for:

- installing additional tools, secrets, ssh keys and certificates,
- adjusting firewall configuration and setting web server as SSL proxy,
- configuring additional plugins that are not supported by JCasC yet,
  - e.g. locale plugin, Gerrit Trigger.

```yaml
# tasks/configure-plugins.yaml
- name: "Install plugins configuration"
  template:
    src: "files/config/{{ item }}.tpl"
    dest: "/var/lib/jenkins/{{ item }}"
    owner: jenkins
    group: jenkins
    mode: 0644
  with_items:
    - 'gerrit-trigger.xml'
    - 'locale.xml'
```

Now, once every configuration things are in place, we are ready to re-launch Jenkins with new settings applied.

With service module in Ansible it is simple just as that:

```
  # tasks/restart-jenkins.yaml

- name: "Restart Jenkins"
  service:
    name: jenkins
    state: restarted
```

Image source: https://me.me/i/friend-where-are-you-me-im-almost-there-7075143
Wait until Jenkins is ready

During start, Jenkins returns a 503 HTTP response (Service Unavailable). If it comes up, returning 200 HTTP code, it means JCasC went flawlessly.

```yaml
# tasks/wait-for-jenkins.yml
- name: "Wait for Jenkins to come up"
  uri:
    url: "http://127.0.0.1:{{ jenkins_http_port }}/"
    status_code: 200
  register: result
  until: result.status == 200
  retries: 50
  delay: 5
```

Please wait while Jenkins is getting ready to work ...

Your browser will reload automatically when Jenkins is ready.
Trigger seed job

- Achieved by sending POST request to the job’s endpoint,
- it may be required to authenticate and/or get CSRF token first,
- also an approval for seed job in Script Security plugin may be needed.

```yaml
# tasks/trigger-seed-job.yaml
---

- name: "Get CSRF crumb for requests"
  uri:
    url: "http://127.0.0.1:{{ jenkins_http_port }}/crumbIssuer/api/json"
    return_content: yes
    register: crumb

- name: "Trigger seed job"
  uri:
    url: "http://127.0.0.1:{{ jenkins_http_port }}/job/Create-jobs/build"
    method: POST
    status_code: 201
    headers:
      'Jenkins-Crumb': "{{ crumb.json.crumb }}"
```

```yaml
# jcasc/security.yml.tpl
---

security:
  globaljobdslsecurityconfiguration:
    useScriptSecurity: False
```

Sz. Datko, R. Dobosz
Testing Jenkins configuration changes
1st May 2019
Verify seed job went fine

- As a last step we need to ensure the triggered seed job went fine,
- if it did so, it means the Job DSL created jobs without issues,
- when we succeed here, all our deployment is fine!

```yaml
# tasks/verify-seed-job-success.yml
---
- name: "Verify seed job result"
  uri:
    url: "http://127.0.0.1:{{ jenkins_http_port }}/job/Create-jobs/lastBuild/api/json"
    return_content: yes
    register: output
  until: output.json.result != 'SUCCESS'
  retries: 50
  delay: 5
```

Image source: http://highstakesbassin.com/falcon-report/charades-highstakesbassin-com-style/attachment/can/
Is this enough?

Probably not. What we have achieved so far:

- JCasC configuration validation against its JSON schema,
- jobs definitions syntax (Job DSL) is parsed,
- jobs steps (Job Pipelines) syntax is checked,
- functional test for JCasC as Jenkins deployment job,
- jobs definitions (Job DSL) are also tested during deployment.

What is still missing:

- functional test for Jenkins–Gerrit integration,
- Job Pipelines correctness in isolated environment.
Conclusions

To sum up:

- testing things is important,
- valid configuration is as important as valid code,
- **Jenkins Configuration as Code:**
  - validate against JSON Schema,
- **Job DSL:**
  - use regular Groovy parser,
- **Job Pipelines:**
  - check with build-in parser,
- additional things need to be launched for completeness!
Thank you for your attention!

The slides are available: http://datko.pl/OS-Denver.pdf
Testing Jenkins configuration changes
solidify your JCasC, Job DSL and Pipelines usage

Szymon Datko
tsyzm.datko@corp.ovh.com

Roman Dobosz
roman.dobosz@corp.ovh.com

OVH

1st May 2019 (Labour Day)