Jenkins2 Pipelines Demo

Agenda / demo plan

- Intro
- D0 - Demo setup
- D1 - Pipeline 101
- D2 - Build result / status
- D3 - Manual step
- D4 - Parameterized build
- D5 - Pipeline as Code
- D6 - Parallel execution
- D7 - Blue Ocean
- D8 - Some useful commands (steps)
- D9 - What more do you need?
- D99 - All included Pipeline

Intro

niclas@yoga: ~/ whoami

Nyss fyllda: 51 år

Bor i: Björkhagen tillsammans med Fru och två grabbar (17 och 20 år)

Kallar mig själv gärna för: “Datakille”

Älskar: Teknik, tyngdlyftning, handboll och swimrun

Grejjar gärna med: Linux, Docker, Java, IoT, Raspberry PI, Z-Wave

Jobbar just nu i: Tools-Teamet inom Ärendehantering

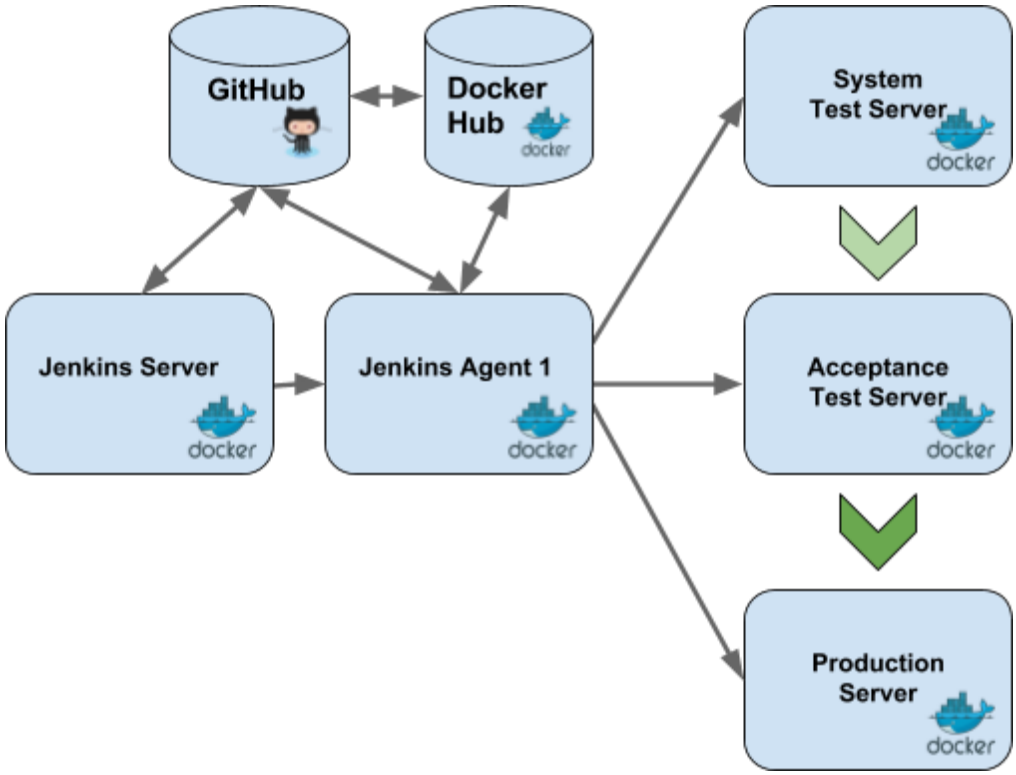
Demo Goal

- Become an “Amateur Plumber”
- Get acquainted with the most common Pipeline-features (steps)
- Learn to use “Pipeline as Code”

Prerequisites / Tools

- IntelliJ
- Groovy
- Docker
- Java

D0 - Demo Setup



We will use a few “pre-boiled” Docker containers
(for simplicity and to simulate real infrastructure)

- A Jenkins master that does minimal work (bootstrap, visualisation)
- The “hard work plumbing” will be done in Jenkins agents
  (build, test, deploy, etc, …)
  Create agents for different purposes (java, maven,... etc)..
  Increase number of agents for scalability
- Pipeline code in its own Git-repository (not together with source code)

Pre-boiled stuff

- Pre-configured Jenkins 2 server (pmdevel/jenkins2-server:welcome_blueocean (Docker Hub, pmdevel)
- 2 Pre-configured Jenkins 2 build agents (java and maven) DooD (Docker Hub, pmdevel)
- A Spring Boot Web App project to build and test in the demo (GitHub, pmdevel)
- A JUnit test project to use for testing of the Web app (GitHub, pmdevel)
- Pipeline demo project with pipeline.gdsl (pipeline code completion) (GitHub, pmdevel)

Jenkins2 configuration

- Vanilla Jenkins2 version 2.32.3 with recommended plugins
- Added plugin ‘Pipeline Utility Steps’
- Added plugins for ‘Blue Ocean’ view
- Restricted Master node (1 executor, label ‘master’, “only build jobs with matching label”)
- Added “Username with password” credential for communication with agents
- Added “Username with password” credential for GitHub (pmdevel)
- Added build-agent (Type SSH, use agent-credential)

Start Jenkins and container

```
docker run --name jenkins2-server -d -p 8080:8080
pmdevel/jenkins2-server:welcome_blueocean
```

Expose docker socket to “anyone”

chmod:a /var/run/docker.sock

```
sudo chmod o+rw /var/run/docker.sock
```

```
docker run --name jenkins2-maven-agent-1 -d -p 2221:22 -v
/var/run/docker.sock:/var/run/docker.sock -v $(which docker):/usr/bin/docker
pmdevel/jenkins2-agent:maven
```

Check that docker works inside container

```
ssh -p 2221 jenkins@localhost docker images
```

Login to Jenkis2
http://localhost:8080 (jenkins/jenkins)

Jenkins 2 - Look-Around

Jenkins - Configuration (Hantera Jenkins)

- Credentials
- Plugins (Blue Ocean, Pipeline Utility Steps)
- Nodes (Build Agent, number of workers, lables, etc)
- In-process Script Approval
- Blue Ocean

How do I work?

- Code in IntelliJ (Syntax check, Code Completion and Groovy DSL)
- Copy/Paste into Pipeline Script Text
- Run

Maybe try to checkout, Jenkins and run from IDE !?

- Access to underlying Groovy libraries

D1 - Pipeline 101

GitHub: *pmdevel/jenkins2-demo-pipelines/pipeline_101*

Simple pipeline with 4 steps.

- Build
- Junit Test
- Dockerize
- Deploy

It does exactly that...

Steps used:

stage	- Create stages
node	- Controll execution
git	- Checkout from GitHub
sh	- Execute stuff on agents underlaying os (mvn, docker, etc...)
echo	- Logging
stash / **unstash**	- Keep state between stages
readMavenPom	- Load Maven pom-file into DOM object for manipulation
readProperties	- Load propertis file into a property object for easy access
writeFile	- Create files on disk

See reference: https://jenkins.io/doc/pipeline/steps/

D2 - Build result / status

GitHub: *pmdevel/jenkins2-demo-pipelines/pipeline_101*

Demo on how to control currentBuild.result (status of pipeline, color in GUI)

currentBuild.result = **'SUCCESS'** (green)
currentBuild.result = **'UNSTABLE'** (yellow)
currentBuild.result = **'FAILURE'** (red)

Steps can set status

- junit sets status 'UNSTABLE' on failing JUnit tests
- sh sets status 'FAILURE' on exit codes > 0
  Exit code > 0 also throws Exception and aborts pipeline execution

Steps used:

sh - Execute stuff on agents underlaying os (mvn, docker, etc...)

D3 - Manual step

Demo on how to manually control execution of pipeline.
If you do not want to automatically deploy changes into your production environment, you can add a *manual step.*
Result from input is ‘continue processing’ or ‘abort’
A result object contains result from user interaction.

Be careful not to “lock” worker threads

Steps used:

timeout	- Wait for something for a specified time
input	- Wait for interactive input from user

D4 - Parameterized build

A pipeline can be parametrized so that you can start it with different input parameters.
For example a VCS url, deployment environment stuff, etc...

When creating pipeline, check box ‘This build is parameterized’
Add the parameters you want.

The parameters will be ready to use in your pipeline as any other variable.

Ex
echo(“Parameter FOOBAR=${FOOBAR}”)

OBS! OBS! OBS!
Boolean parameters must be converted to boolean if you want to do something like:
If (SOME_BOOLEAN_PARAMETER) {
  then do…
}

Ex
boolean someBooleanParameter = SOME_BOOLEAN_PARAMETER.toBoolean()

You also want to load the pipeline definition from Git. Maybe not trigger on every change...

D5 - Pipeline as Code

In the same way you automate the building, testing of your source code, you would like to be able to automate the creation of your Pipeline or for backup.
This is also a nice feature if you want to set up a new Jenkins Master

Your Pipeline can be stored as a configuration file (XML)
This can be accomplished via Jenkins CLI
http://localhost:8080/cli

Prerequisites
Set up Public Key authentication for jenkins user

- Go to: http://localhost:8080/user/jenkins/configure
- Paste your SSH public key into “SSH Public Keys”
- READY!!!

In project ‘jenkins2-demo-pipelines’ you can find ***jenkins-cli.jar***
Run command:

```
java -jar jenkins-cli.jar -s http://localhost:8080/ list-jobs
```

Get a predefined job

```
java -jar jenkins-cli.jar -s http://localhost:8080/ get-job
pipeline-101-with-parameters > pipeline-101-with-parameters.xml
```

Create a job from XML configuration

```
java -jar jenkins-cli.jar -s http://localhost:8080/ create-job
pipeline-101-with-parameters < pipeline-101-with-parameters.xm
```

You also want to load the pipeline definition (Jenkinsfile) from Git. Maybe not trigger on every change...

D6 - Parallel Execution

To speed up things like testing parallel execution can be used. Jenkins pipeline support this in the DSL.

Steps used:
parallel - Execute stuff in parallel

D7 - Blue Ocean

- A new “view” on your Pipelines…
- Can co-exist with existing Jenkins GUI
- Is supposed to focus on the development team
  (Easier to find information)
- Support view of parallel executions

GitHub Accesstoken: `fc37e009bb3bf37251d8631fdb6dd8e8443eef4f`

D8 - Some useful commands (steps)

emailext - Extended Email
Send mail on broken builds, failed unit tests
Can use HTML body

httpRequest - Perform HTTP request and return a response object
Send mail on broken builds, failed unit tests

jiraXxx - A bunch of JIRA commands
Access Projcts, Issues, add comments, etc

dockerXxx - A bunch of Docker commands
Build images, Run images, etc
I think it is a better approach to Dockerize the agents and “hide” this from the Pipeline
(I have tried both) Can also be done via sh “docker xxx…”

Kubernetes:Xxx - A bunch of Kubernetes commands
Build, deploy and run images (pod’s) in Kubernetes

readJSON - Read JSON from files in the workspace

readMavenPom - Read a maven project file

readJSON - Read JSON from files in the workspace

writeJSON - Write JSON to a file in the workspace

archive - Use stash instead
(Or artifact repository like Docker HUB, Nexus or Artifactory for binaries)

load - Evaluate a Groovy source file into the Pipeline script

D9 - What more do you need?

VCS - integration

To be able to track changes in your Continuous Delivery Pipeline it is “nice to have” Release Notes. They are usually derived from the check-in’s found in your Version Control System.

At Pensionsmyndigheten we control Deployment to production environments via JIRA workflow (It requires some integration)

Database - integration

Useful for storing build information between pipeline executions
(Jenkins <-> JIRA integration)

D99 - All included Pipeline

A pipeline with all the previous features

- Build and Junit test a Web App
- Deploy in two system test environments for parallel testing
- Fail if any test is unsuccessful
- Deploy in acceptance test environment
- Fail if any tests is unsuccessful
- If manual confirmation is required, wait for it
  Continue if confirmed / Fail otherwise
- Deploy in production environment

OBS! Pipeline execution will fail until the following In-script Process Approvals are approved:

- method hudson.model.Run getNumber
- method hudson.model.Run setDisplayName java.lang.String
- method org.apache.maven.model.Model getVersion
- method org.jenkinsci.plugins.workflow.support.steps.build.RunWrapper getRawBuild