cloudbees UNIVERSITY

CloudBees Jenkins Platform: Certification Training

2 - Building an application with Jenkins

Table of Contents

2 - Building an application with Jenkins

Journey Description

IMPORTANT	This lab requires having covered the Lab 1 or at least mastering the technical stack

The goal of this journey is to bootstrap a Jenkins Open Source instance and use it to build and test the demo application covered in Lab 1.

It only covers the basic concepts, to ensure a complete understanding of both lab contents and certification goals.

We will cover:

- Starting Jenkins
- Basic administration of the Jenkins instance
- Installing plugins
- Creating jobs for our application
- Running builds on our jobs, experimenting with failures
- Changing build triggers
- Implementing a basic CD pipeline

Starting Jenkins

The goal of this exercise is start and access the Jenkins Open Source integrated instance.

The integrated Jenkins instance is running within a **Docker container** and is pre-configured for this exercise.

IMPORTANT	Jenkins 2.x introduced a Configuration Wizard. It has been disabled for this exercise, by setting the flag `jenkins.install.runSetupWizard` to `false` on the Jenkins JVM. Beware that this leaves Jenkins WITHOUT security enabled, which is dangerous. The lab of part 3 will cover this case. See this page for more informations: Jenkins Docs: Features controlled by system properties.

Accessing the Jenkins Open Source Instance

For "easy-bootstrap" reasons, the service has already been started.

- You can access it from the Lab instance **HomePage**, by clicking the "Jenkins" link.

TIP	Direct link to Jenkins: http://localhost:5000/jenkins

- You should see the Jenkins `2.19.4` homepage:

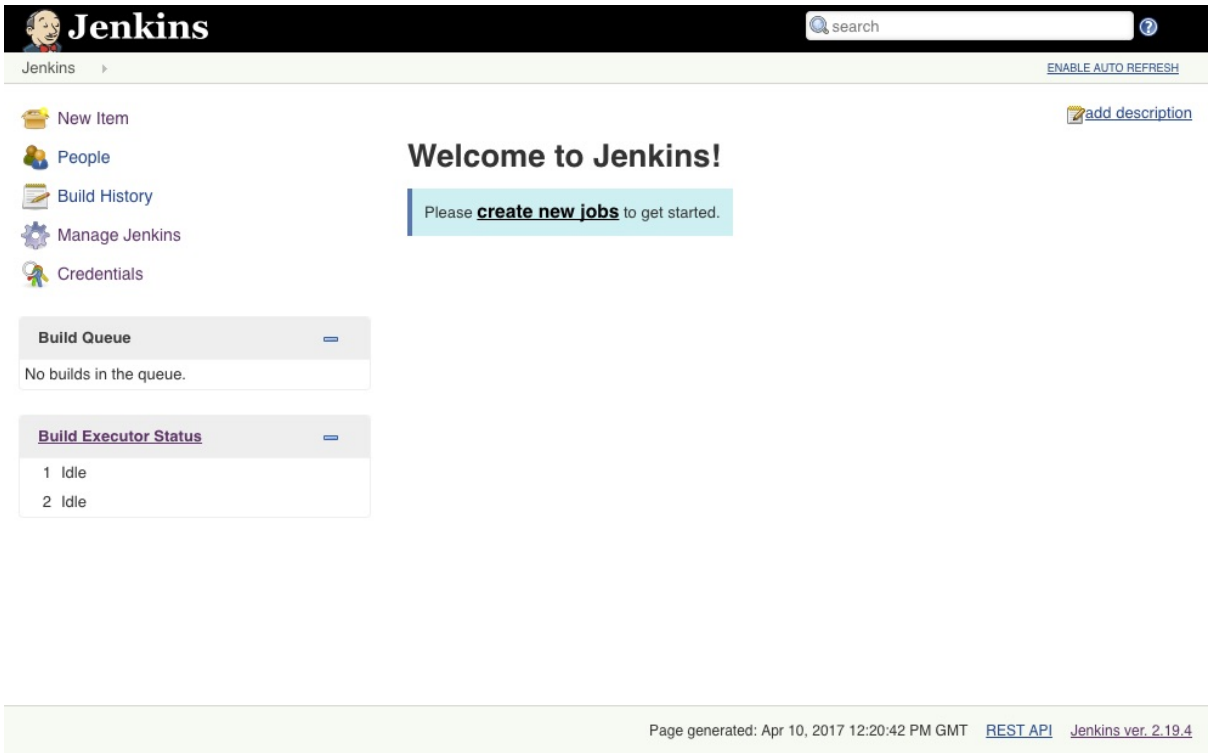

Figure 1. Empty Jenkins Homepage

TIP Please note that all Jenkins Open Source GUI-related actions will takes place from there

Managing the Jenkins instance

You can manage the Jenkins instance with the following set of docker commands from the Devbox service:

- Printing Jenkins logs:

```
cloudbees-devbox $ docker logs jenkins
```

TIP You can also enable the "follow mode" by passing the flag `-f` to the "docker logs" command: `docker logs -f jenkins`. Use `CTRL-C` to stop following logs.

- Accessing the Jenkins instance with an interactive shell:

```
cloudbees-devbox $ docker exec -ti jenkins /bin/bash # Spawn a bash shell on
Jenkins service
jenkins@fcf300221f78:/$ cd /home/jenkins # Browse to the JENKINS_HOME
bash-4.3 pwd
/home/jenkins
bash-4.3 ls -l
...
bash-4.3 ps aux | grep jenkins | grep java
...
bash-4.3 exit
cloudbees-devbox $
```

- Restarting Jenkins:

```
docker restart jenkins # Wait 2-3 minutes for Jenkins to start
```

IMPORTANT Ensure Jenkins restarted by going back to the Jenkins homepage

That's all for this exercise !

Managing Jenkins

The goal of this exercise is to configure our Jenkins instance for the next exercises

Start by browsing to the **Management Page** of the Jenkins instance:

- From Jenkins **Homepage**: click on **Manage Jenkins** on the left-menu

TIP Direct URL: http://localhost:5000/jenkins/manage

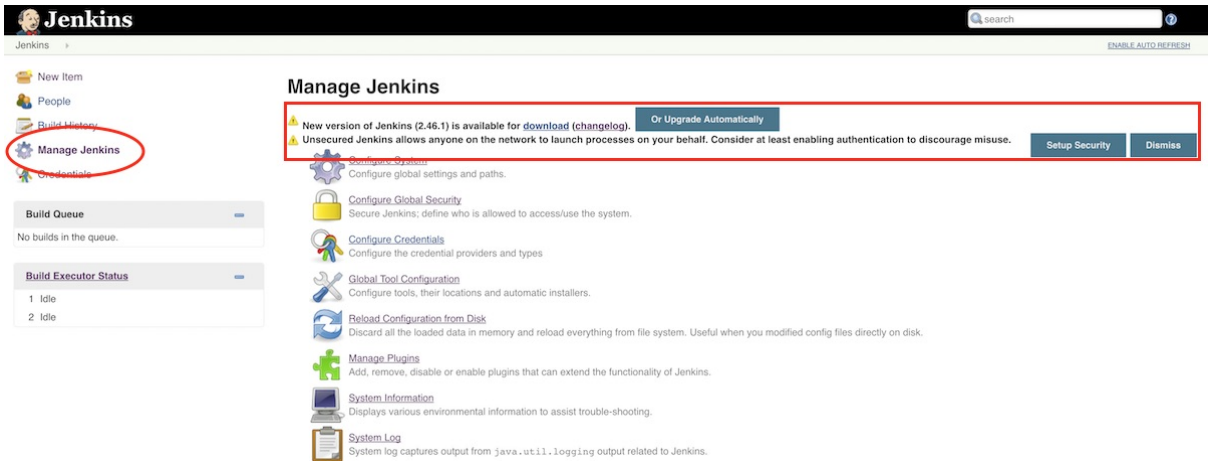

Figure 2. Jenkins Managing Page

- Check the "Administration" Alerts:
  - *New version of Jenkins is available:* We won't update since the Certification's Jenkins version is **fixed** to 2.19.4

- *Security disabled* alert: will be covered in Labs 3 exercises

TIP

Please note that security should enabled by default since Jenkins 2.x, by following the startup wizard.

It has been disabled for this exercise, as seen previously, so this Jenkins instance is **not** secured.

Jenkins Global Configuration

As an **Administrator**, it is time to set up some Jenkins system properties.

Start by clicking the **Configure System** links, to land to the main "Configuration Page".

Take time to review all the available options, help is provided with right button with the blue question mark.

TIP Direct URL: http://localhost:5000/jenkins/configure

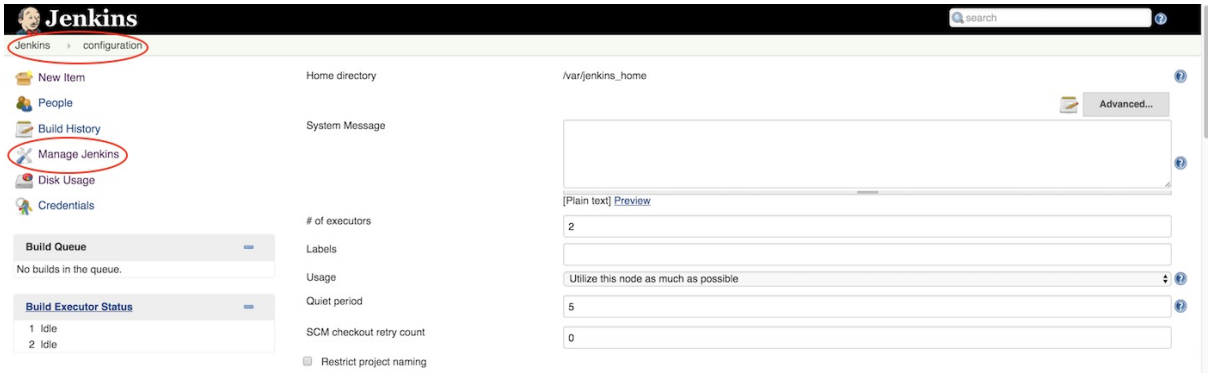

Figure 3. Jenkins Configuration Page

Global Environment Variables

As an *administrator*, you may need to provide environment variable to **ALL** users and objects in Jenkins.

- For running future **Docker** commands, we needs to set the `DOCKER_HOST` to our Docker Service:
  - In the **Global properties** section, tick the **Environment variables** checkbox
  - *Add* a new Key-Value pair with :
    - `DOCKER_HOST` as key
    - `tcp://docker-service` as value
- You can click the button **Apply**
  - It will save your configuration without closing the page

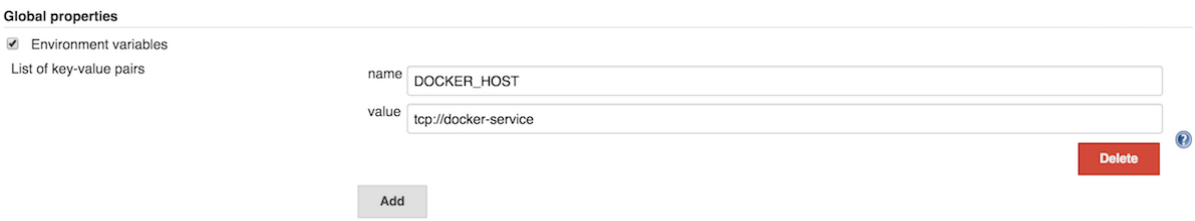

Figure 4. Environment Variable for Docker

> **TIP** Those variable can be overwritten locally on Jobs Configuration levels

Other Global Settings

There are other settings that should be reviewed:

- In the **Jenkins Location** section, check the ***Jenkins URL*** value: it should maps to your Jenkins instance url (http://localhost:5000/jenkins). This value is used by Jenkins to monitor Reverse-Proxy settings. It is the case in this Lab: we use "Nginx" in front of Jenkins.

> **TIP** If you see the message "Please set a valid host name, instead of localhost" when running the VirtualBox lab, don't take it in account. Jenkins is conservative about having a real domain name in production, but we are on a training.

- In the **Git Plugin** section, set the Global Config user.name to `jenkins`

> **TIP** This account is used by Jenkins when doing Git local Write Operations. A Maven release for example.

- Let's move to other administration panels:
  - Click the button **Save**, it will save the configuration and send you back to the **Jenkins Homepage**.

Global Tool Configuration

As an **Administrator**, we want to configure the "Tools" to let our users being able to build their applications.

Jenkins provides a dedicated administration section for managing those "Build Tools".

Start by browsing to **Manage Jenkins** → **Global Tool Configuration**.

> **TIP** Jenkins can automatically install Tools for you. For JDK, it will use Oracle/Sun JDK by default, but you can also provide your own scripts or archives for provisioning

- Configure a new **Global Tool** to let Jenkins manage the Maven Installations:
  - In the **Maven** section, click the button **Add Maven**

> **TIP** If you have already configured a Maven Installation, then you will have a button **Maven installations...** to click before manipulating the collections of "Maven Installations"

  - Configure this new "Maven Installation" with the following properties:
    - Name `maven3`
    - Uncheck the **Install automatically** option: Maven is already installed
    - The field **MAVEN_HOME** should appear now. Set it to this value: `/usr/share/java/maven-3`
- You can now safely click the button **Save** at the bottom of the page.

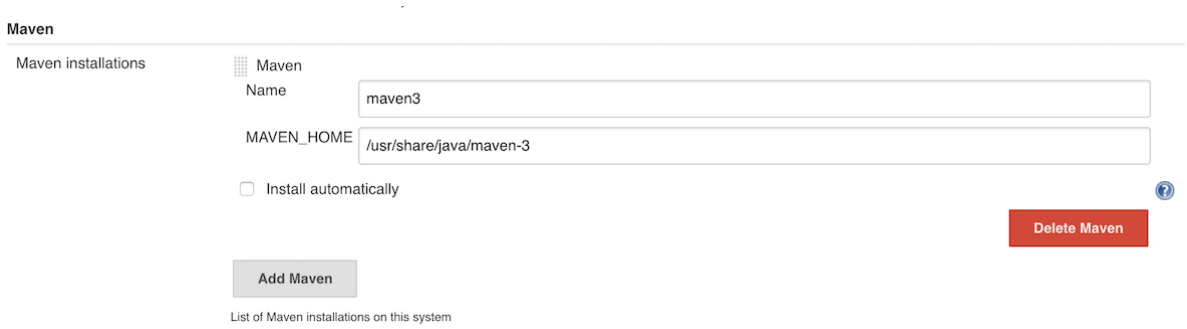

Figure 5. Tools Installation: Maven 3

Configuration using Apache Groovy

We are now going to add a new **Global Tool**: a Docker Installation, as we've done for Maven in the previous chapter.

Instead of using the previous GUI, we're going to use a programmatic way of doing it.

- Start by browsing to the Groovy Console:
  - From the **Manage Jenkins Page**, click on the **Script Console** link to access the Console page.

TIP	Direct URL to the console: http://localhost:5000/jenkins/script

- Run this Groovy code snippet, that will configure a new "DockerTool" installation, located in `/usr/` and named `docker-latest`:

```
a = Jenkins.instance.getExtensionList(
  org.jenkinsci.plugins.docker.commons.tools.DockerTool.DescriptorImpl.class
)[0];

a.installations=new
  org.jenkinsci.plugins.docker.commons.tools.DockerTool(
    "docker-latest",
    "/usr/",
    []
);

a.save();
```

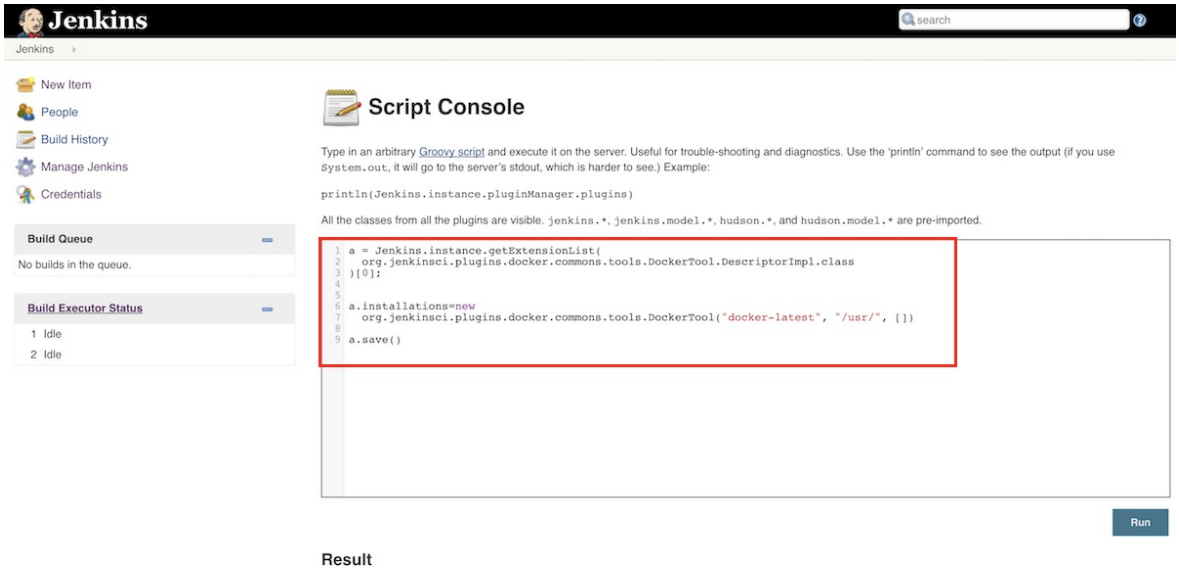

Figure 6. Installing Docker Tool with Script Console

- Return to the page **Manage Jenkins** → **Global Tool Configuration**, and click the button **Docker Installations...** to validate the result.

IMPORTANT

Running a script in GUI is not the best option always. You can execute this kind of script:

- With the Jenkins Command Line or with the REST API
- With the Jenkins Hook Script
  - Just copy Groovy scripts in `${JENKINS_HOME}/init.groovy.d` directory to have them run at Jenkins startup
  - The Jenkins official Docker image is already using this feature

Other Administration Utilities

We forgot to define the `2375` port to the `DOCKER_HOST` environment variable!

- We're going to correct this **without** Jenkins GUI by modifying the config.xml file directly. Using the Devbox Command Line:
  - Run the following `docker exec` command to reach the Jenkins machine and check the content of the file `/home/jenkins/config.xml`:

```
docker exec -ti jenkins cat /home/jenkins/config.xml
```

  - Run this command to "Search and Replace" the configuration value, inside the Jenkins machine:

```
docker exec -ti jenkins \
  sed -i \
  's#tcp://docker-service<#tcp://docker-service:2375<#g' \
  /home/jenkins/config.xml
```

- Check that your change has been written in the config.xml
- From the **Jenkins Manage Page**, select the **Reload Configuration from Disk** to have Jenkins reloading its configuration.
- Validate the change has been applied in the **Manage Jenkins** → **Configure System** page.

TIP	You can also stay on the command line and restart Jenkins with the docker restart command

Additional Exercises:

- Search for what the value of the `java.io.tmpdir` property of the Jenkins JVM is

TIP	Use the System Information Page: http://localhost:5000/jenkins/systemInfo

- Count how many `jenkins.util.Timer` Java Threads are currently running

TIP	Use the Thread Dump Console: http://localhost:5000/jenkins/threadDump

That's all for this section !

Installing necessary plugins

The goal of this exercise is to install 2 plugins using 2 different methods:

- Recommended: Using the Update Center
- Manual method

IMPORTANT	Please note that an Internet access (HTTP proxy supported) is **required**. If you do not have one, youcan safely skip the the **Update Center Method** section and will only be able to run the **Manual Method**.

Update Center Method

We are going to install a harmless plugin: the Beer plugin

TIP	This plugin will just add a page on Jenkins root to print some jokes about beer. It has no dependencies and is lightweight.

Start by going to the **Plugin Management Page**:

- From the **Manage Jenkins Page**, click on the **Manage Plugins** link

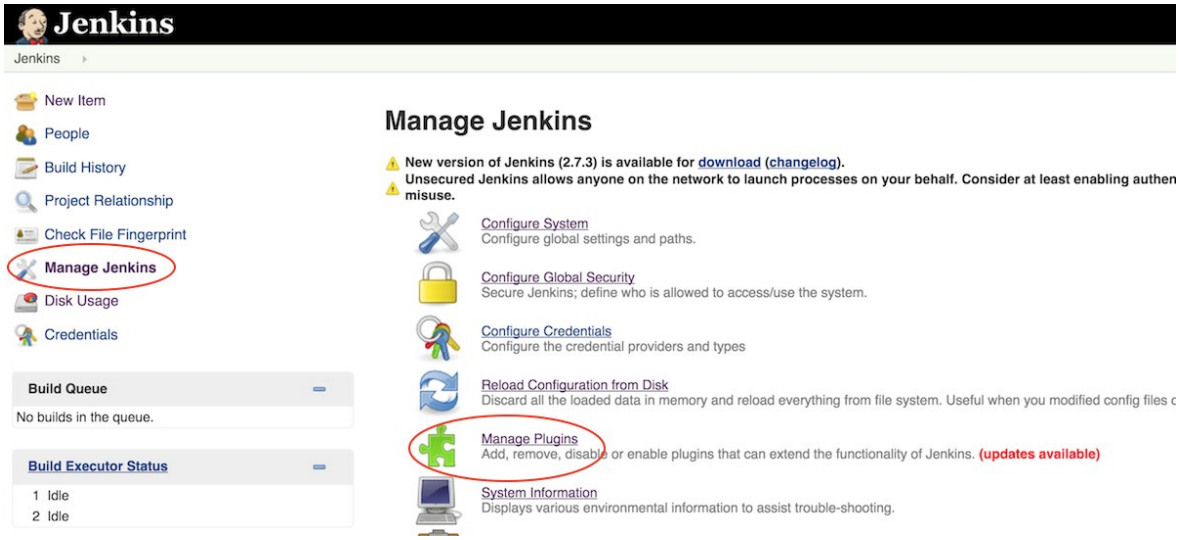

Figure 7. Accessing the Plugin Management

TIP

If you are running under an HTTP proxy, select the **Advanced** tab:

- There is an **HTTP Proxy Configuration** section where you can configure the HTTP proxy access
- Clicking the **Advanced** button will enable the **Validate Proxy** configuration fields before submitting the configuration

- Install the plugin with the "GUI Update Center":
  - Select the **Available** tab:
  - Use the *Search Filter* to search for `beer` keyword

IMPORTANT

The search in done on the fly: do **NOT** hit the "Return" Key, or you will be redirected you to the plugins installation's log page.

  - From there, select the beer plugin:

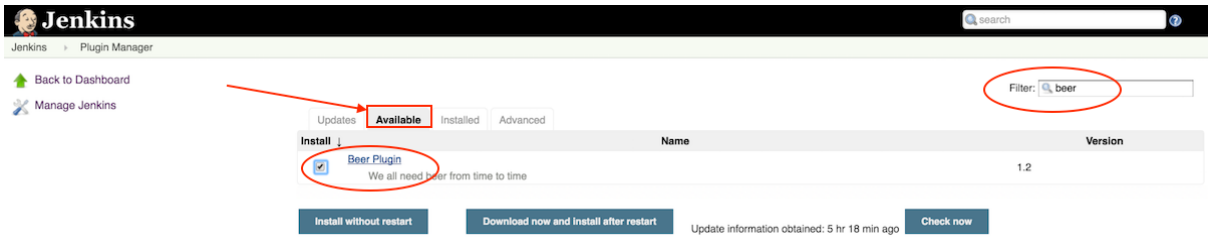

Figure 8. Searching for the Beer plugin

- Hit the **Install without restart** button, and you'll land on the "waiting installation" page:

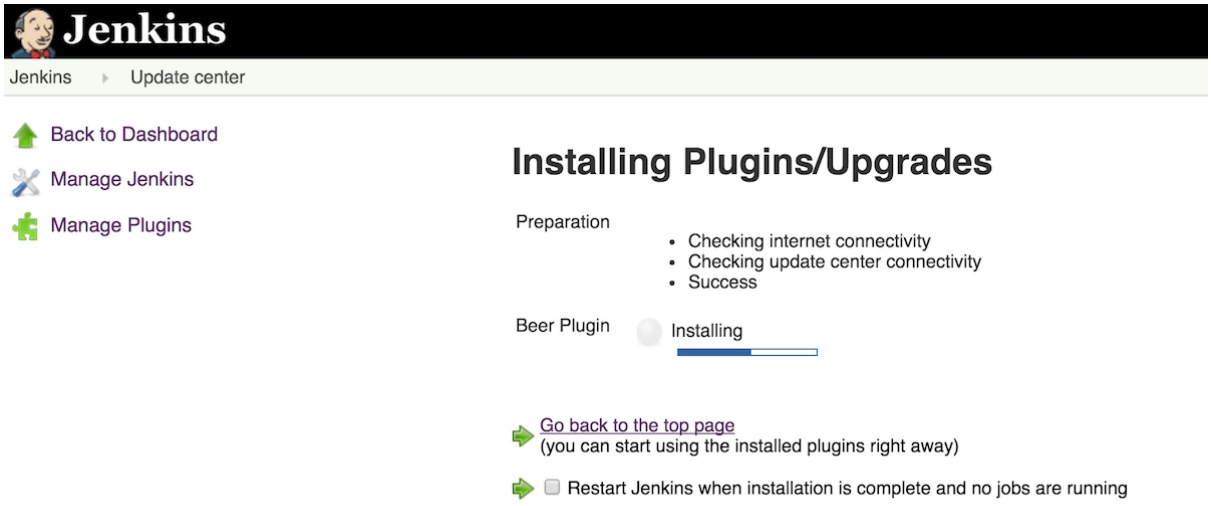

Figure 9. Waiting for the Plugin Installation

IMPORTANT	Please note that some plugins will require a Jenkins Restart and/or downloading dependencies (other plugins). Plan this carefully in production !
TIP	Note the checkbox "Restart Jenkins when no job running"

Once everything terminated, go back to the **Jenkins Homepage**. You should see a refreshing new Link in the left menu, to use when waiting for builds to complete:

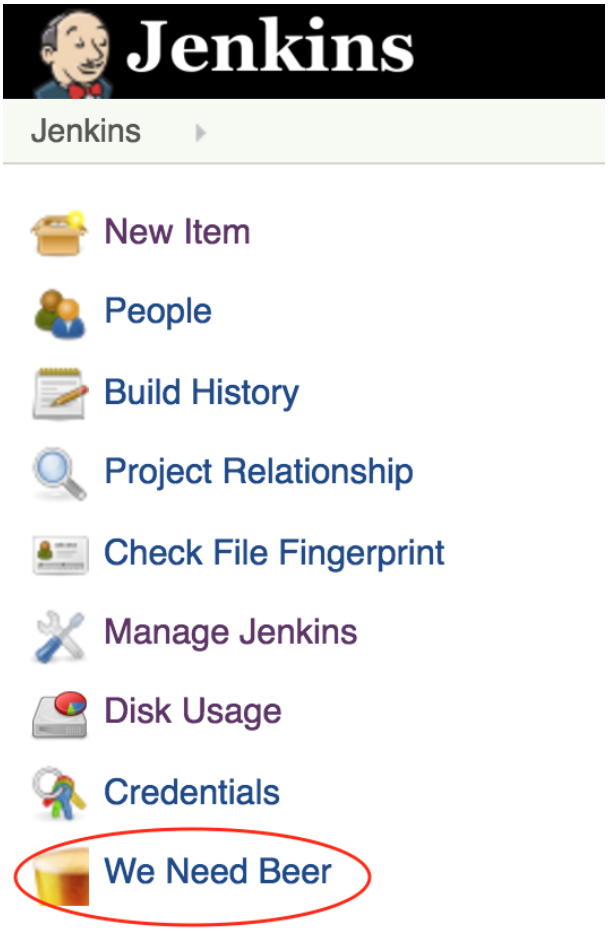

Figure 10. Beer Plugin is installed !

Manual Method

We are going to install **manually** another harmless plugin: the Chuck Norris plugin

TIP	This plugin provide "Chuck Norris Facts" (read: jokes) on the status build screens. It is also lightweight and has no dependencies.

Start by downloading the plugin, by opening the following link on your web browser: http://localhost:5000/chucknorris.hpi

TIP	By default, the download is done from the Lab VM to avoid Internet connectivity. However, if you have internet access, you can also download the plugin from the plugins.jenkins.io website: https://plugins.jenkins.io/chucknorris

- Going back to the **Manage Plugin** page, select the **Advanced** tab:
  - We're going to use the **Upload Plugin** section

TIP	This section requires you to have the `*.hpi` file of the plugin.

  - Select the previously downloaded plugin by using the file upload option
  - Hit the **Upload** button

- You'll then land again on the same **Waiting for installation** page:
  - Wait for the end of the plugin's installation as you did on the previous section
  - You should restart the Jenkins instance, by selecting the related checkbox

TIP	After a restart, you can check the installation state by browsing the **Installed** tab on the **Manage Plugin** tab. Use the plugin documentation to see how to use it (Clue: Post-Build Action)

Going further

- There are also other methods to install plugins:
  - Jenkins Official Docker Image is proposing to build Jenkins Docker Image with your own plugins: https://github.com/jenkinsci/docker#preinstalling-plugins
  - You can package your own Jenkins and plugins by using Maven WAR Overlay
  - You can copy plugins directly in `${JENKINS_HOME}/plugins` folder, and restart Jenkins instance
- But *outside* the **Update Center** method, the dependency management is **YOUR** responsibility. You have to manage your upgrade policy carefully, in order to have **Stability** across versions!
- Jenkins Docker Image provides a shell command to fetch the exhaustive list of plugins (+ versions) of a given Jenkins instance: https://github.com/jenkinsci/docker#preinstalling-plugins

That's all for this exercise !

Creating jobs for our application

The Goal of this exercise is to create and configure 2 Jobs for our application:

- First job will take care of:
  - Maven build lifecycle (compile, test, verify and install)
  - Building the Docker image
  - and launching a "Docker Smoke Test"
  - It will also archive JAR artifacts
- The 2nd job will be used to launch and teardown Docker containers of our application
  - This will be a "Staging Deployment" task
  - It will require the name of the docker image as parameter
  - Will be triggered manually

From the Jenkins homepage:

"Builder" Job

Create a new job with the following settings:

- **Name:** `demoapp-build`

- **Type:** Freestyle project
- **Source Code Management** section:
  - Select **Git**
  - **Repository URL:** Copy/Paste the HTTP repository URL of the demo application, from the Gogs Git Server GUI.

TIP	Repository URL should be: http://localhost:5000/gitserver/butler/demoapp

  - **Repository Browser:** Select `gogs` from the dropdown box to let Jenkins know the link with the Gogs GUI, and type http://localhost:5000/gitserver/butler/demoapp as **URL**

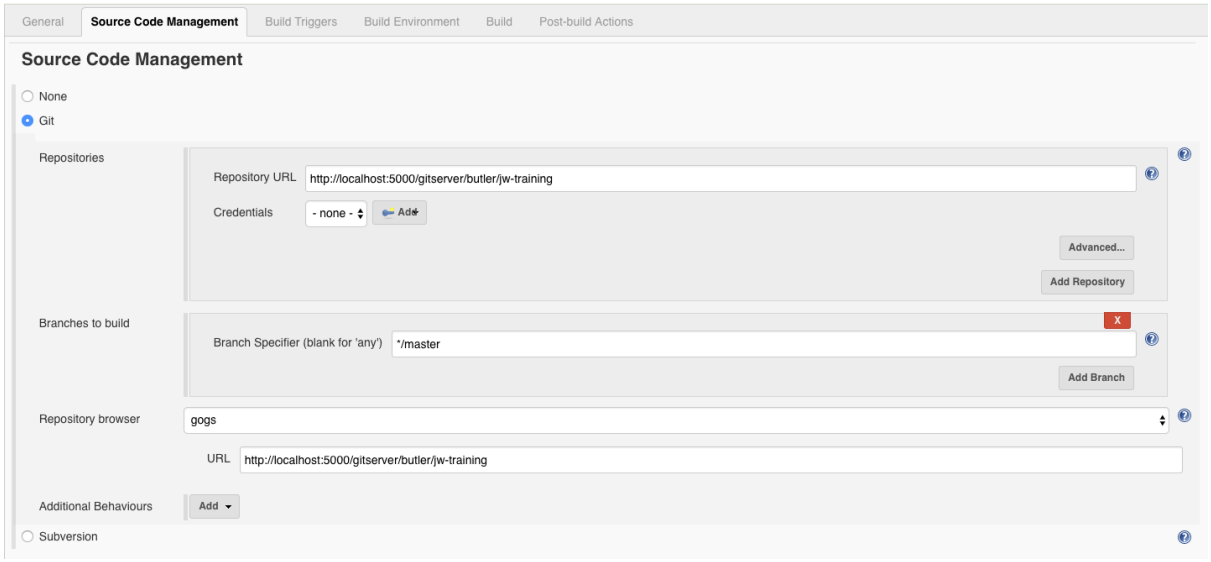

Figure 11. Builder Job Configuration: SCM

- **Build Triggers:**
  - We want to check **each** minute if there is a change in the SCM
  - Untick **ALL** checkboxes
  - Select **Poll SCM**, with this pattern:

```
* * * * *
```

IMPORTANT	As the warning message says, this configuration is not recommended. It is related to the heavy resource consumption it will generate. We will change later in the Lab.

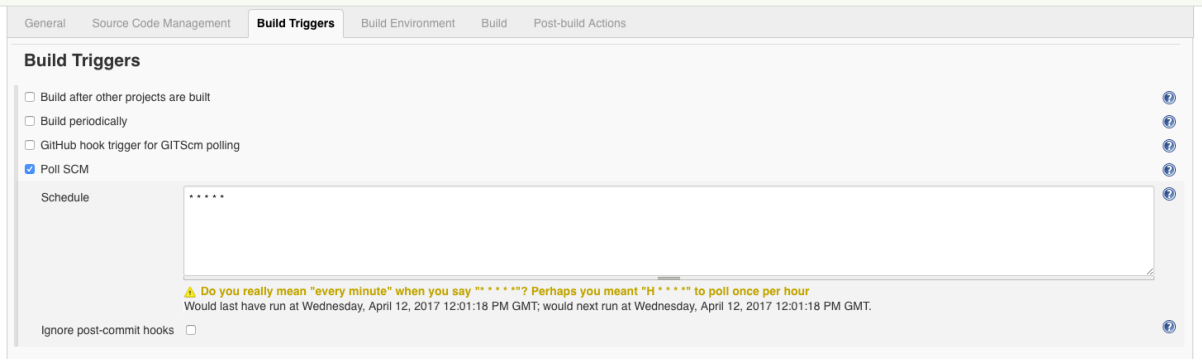

Figure 12. Builder Job Configuration: Triggers

- **Build** section:
  - First, add a new build step of type **Invoke top-level Maven targets**
    - We want this step to run a "full" Maven build lifecycle.
    - Set the **Maven Version** to the `maven3` Maven Tool we configured previously
    - Set the **Goals** field to:

```
clean install -fn -B
```

> **TIP** The `-fn` flag tells Maven not to fail if some tests fails. The build will continue forwards, and we will take care of this with the Post Build Steps.

  - Then, add an **Execute Shell** build step, after the maven one:
    - We want this step to build the Docker image and run a ***Smoke Test*** with this newly built image, regardless of tests results.
    - Set the content with the following shell script:

```
# Use the Git Commit and set the Docker Image
DOCKER_IMG_BASENAME="demo-app"
DOCKER_IMG_FULLNAME="demo-app:${GIT_COMMIT}"

# Build the Docker image
docker build -t "${DOCKER_IMG_FULLNAME}" ./
docker tag "${DOCKER_IMG_FULLNAME}" "${DOCKER_IMG_BASENAME}:latest"

# Simple Smoke test: Start and stop the Docker image
CID="$(docker run -d -P ${DOCKER_IMG_FULLNAME})"
docker kill "${CID}"
docker rm -v "${CID}"
```

> **TIP** Note the usage of the variable `${GIT_COMMIT}` which is provided by the Git Plugin.
>
> It is used for unique tagging of docker image. We also add another docker tag with "latest".

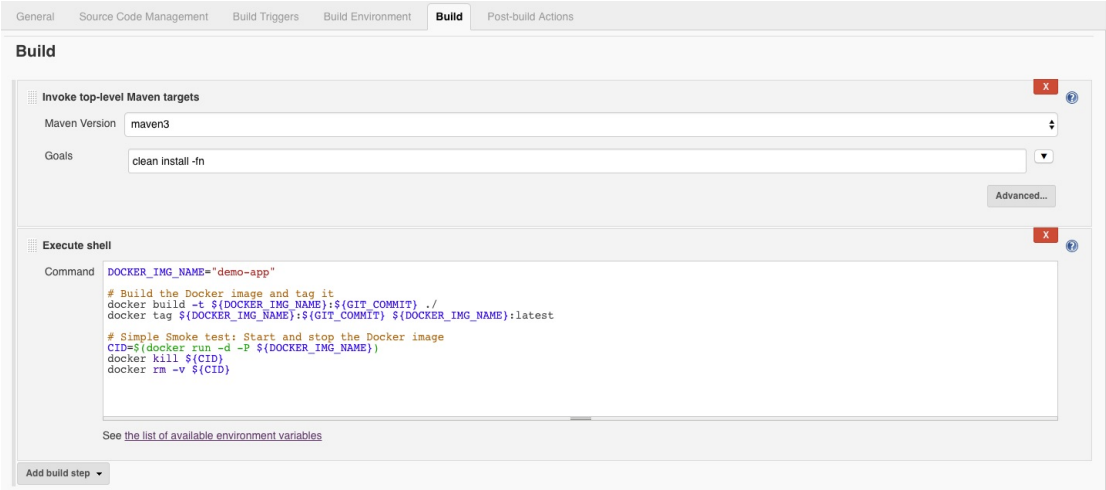

Figure 13. Builder Job Config: Build Steps

- Now, add a *Post-build Action* of kind **Publish JUnit test result report**
  - This action will run after the steps and will publish the Maven tests outputs (all kinds: unit, Integration, etc.). It will also set the build status to unstable or failed based on tests results.
  - Set the **Test report XMLs** field to:

```
**/surefire-reports/*.xml, **/failsafe-reports/*.xml
```

TIP

- The Surefire reports are related to the unit tests.
- The Failsafe reports are related to the integration tests.

- Finally, add another *Post-build Action* of kind **Archive the artifacts**
  - Set the **File to archive** field to:

```
target/**/*.jar
```

TIP

This pattern will archive all ".jar" files recursively in the `target` directory from the workspace. Note that the files must be specified with a relative path, not an absolute path.

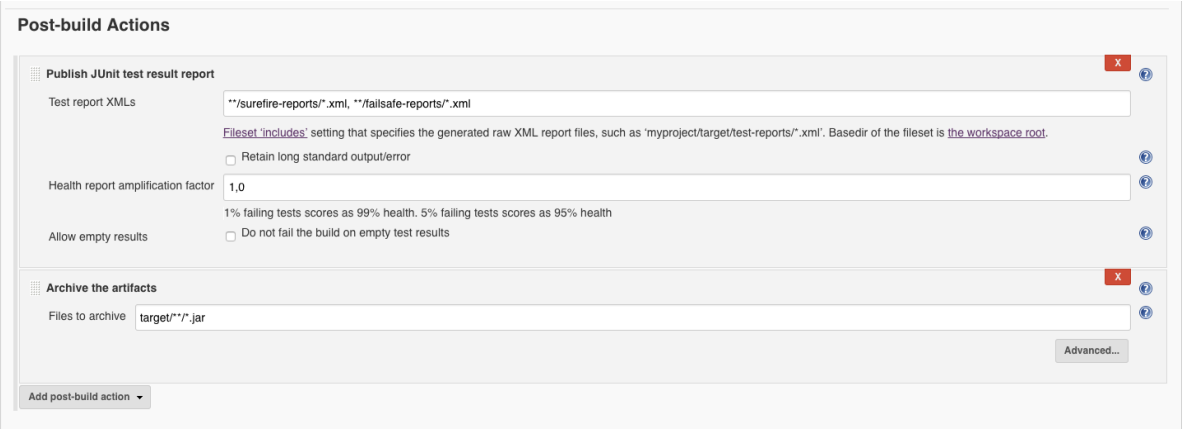

Figure 14. Builder Job Config: Post Build Actions

- You can now safely **Save** this configuration: you will redirected to the newly created job page

Deployer" Job

- Go back to the **Jenkins Homepage** and create a new job:
  - **Name:** `demoapp-staging-deployer`
  - **Type:** Freestyle Project
  - **Parameterized:**
    - We want to pass the "Docker Image Name" as parameter
    - Tick the **This build is parameterized** checkbox
    - Add a new **String Parameter:**
      - **Name:** `DOCKER_IMAGE`
      - **Default Value:** Let it empty
      - **Description:** Describe it as: `The Docker Image to deploy to staging`

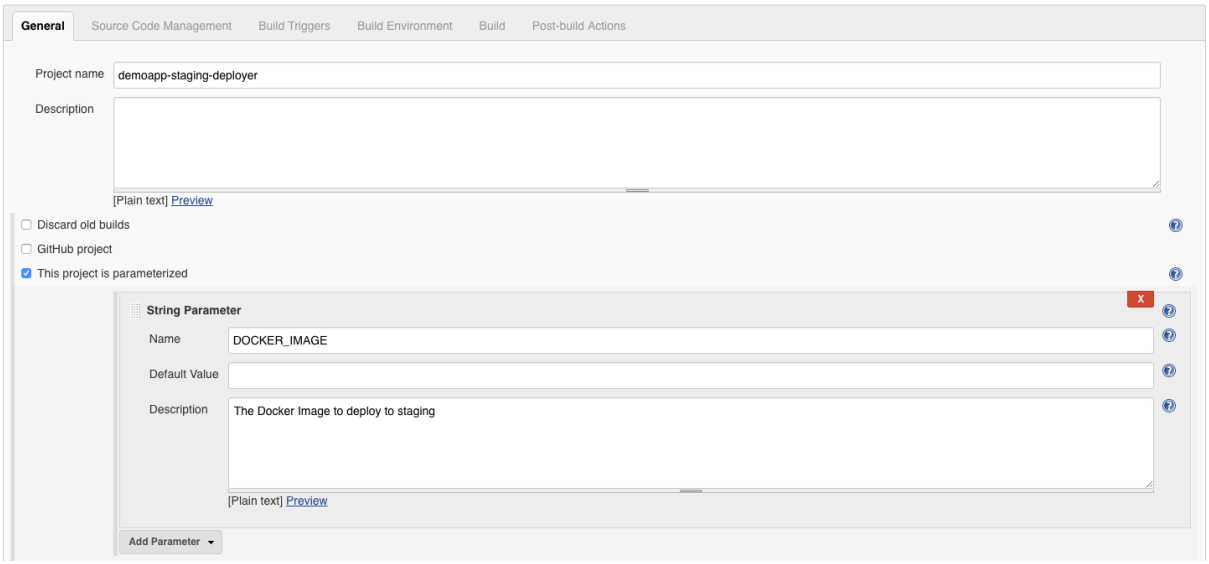

Figure 15. Example of Parameter

- **Source Code Management** section:
  - We do not want to use any source code.section
  - Let to the default value **None**
- **Build Triggers:** section:
  - We want to manually launch this job
  - Ensure **ALL** checkboxes are unticked
- **Build** section:
  - Add an **Execute shell** build step, with the content below:

```
STAGING_CONTAINER_NAME=staging-demo-app

# Try to clean already running container
docker stop ${STAGING_CONTAINER_NAME} \
  && docker rm -v ${STAGING_CONTAINER_NAME} \
  || echo "Nothing to Clean here"

# Launch the new container on the new image
docker run --name=${STAGING_CONTAINER_NAME} -d \
  -p 20000:8080 ${DOCKER_IMAGE}
```

Figure 16. Deployer Job Config: Steps and Actions

Trying our Jobs

- Manually launch the `demoapp-build` Job, using the button "Build Now"
  - **Expected status:** Unstable (Yellow)
- Browse the Output log to find:
  - Where the error is located (for next exercise)
  - The name of the Built Docker image: Post Build Action will have been executed even if build is unstable.

TIP	You can also use the Devbox Command Line if you are experienced with Docker

- Manually launch the `demoapp-staging-deployer` job, using the found "Docker Image Name" as parameter

TIP	The found value should be: `demo-app:latest`

- **Expected status:** Success (Blue)
- Try it by checking the staging environment on http://localhost:20000

Summary

- We now have a **Builder** job which build and test our application
- We have a **Deployer Job** that deploys the staging environment

IMPORTANT	Staging environment is available, but application is not fully reliable since Integration Tests did not pass.

That's all for this exercise !

Running builds, experimenting with failures

The goal of this exercise is to debug our builds and create successful builds then explore the functionality of the Jobs and Builds pages.

Correcting Build

As a **Developer**, when the build is not in a "Success" state, the main focus is to find the reasons for the failure and correct it.

Our main job, `demoapp-build` is currently in "Unstable" state: it means that it builds, but some tests fails.

- Start by browsing to the the Job Page: TIP: Direct URL: http://localhost:5000/jenkins/job/demoapp-build/
- Click on the **Latest Test Result** link, reporting "1 failure"

Direct URL to the latest test report: http://localhost:5000/jenkins/job/demoapp-build/lastCompletedBuild/testReport/

- Once in the **Test Result** page, unfold the Error Details by clicking the **+** icon, in order to see the Cause of the only Test failing:

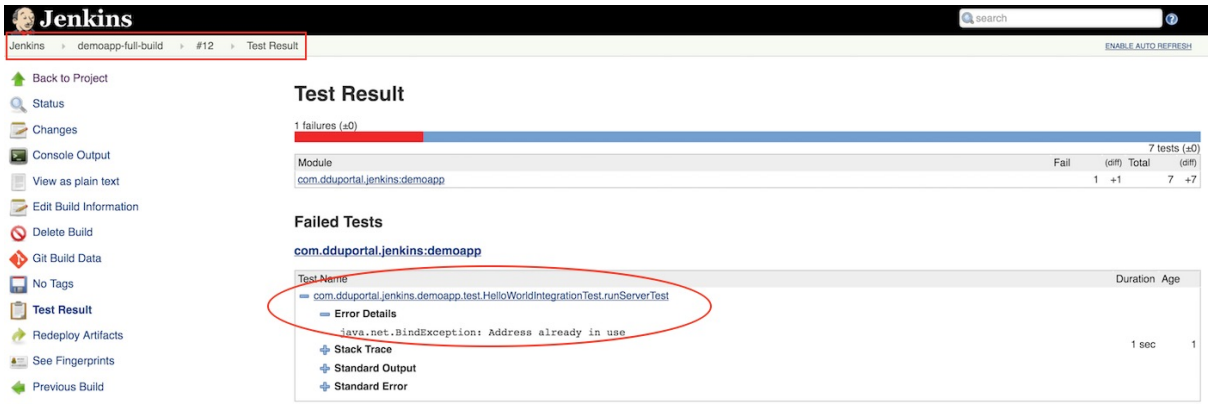

Figure 17. Viewing the Test Errors

- Error thrown:

```
java.net.BindException: Address in use
```

- By default, the application is started for the Integration Tests on the `8080` port
  - But... Jenkins is already listening on the `8080` port
  - Verify this using the Devbox:

```
# Spawn a bash shell on Jenkins machine
cloudbees-devbox $ docker exec -ti jenkins /bin/bash

# Run a command on the port 8080 of the local (127.0.0.1) interface
bash-4.3$ curl -I http://127.0.0.1:8080
HTTP/1.1 404 Not Found
Cache-Control: must-revalidate,no-cache,no-store
Content-Type: text/html;charset=ISO-8859-1
Content-Length: 1267
Server: Jetty(winstone-2.8)
# Jetty Server is responding to HTTP protocol: port is already used
```

- To correct the failing build, we need to configure the integration tests to run on another (free) port: the port **8082**.
  - Log in the WebIDE as butler user
  - Edit the following file (which manages the integration configuration):

```
src/test/resources/helloworld-integrationtest.yaml
```

IMPORTANT	Do **NOT** edit the file `helloworld.yml` located at the root of the repository. It only manages the application runtime, **not** the integration tests.

  - DropWizard Configuration Reference
  - Adapt content to the bits below.

IMPORTANT	**DO NOT** use the 'tab' character to modify the file.

```yaml
template: Hello, %s!
defaultName: IntegrationTest
server:
  rootPath: /api/
  applicationConnectors:
  - type: http
    port: 8082
```

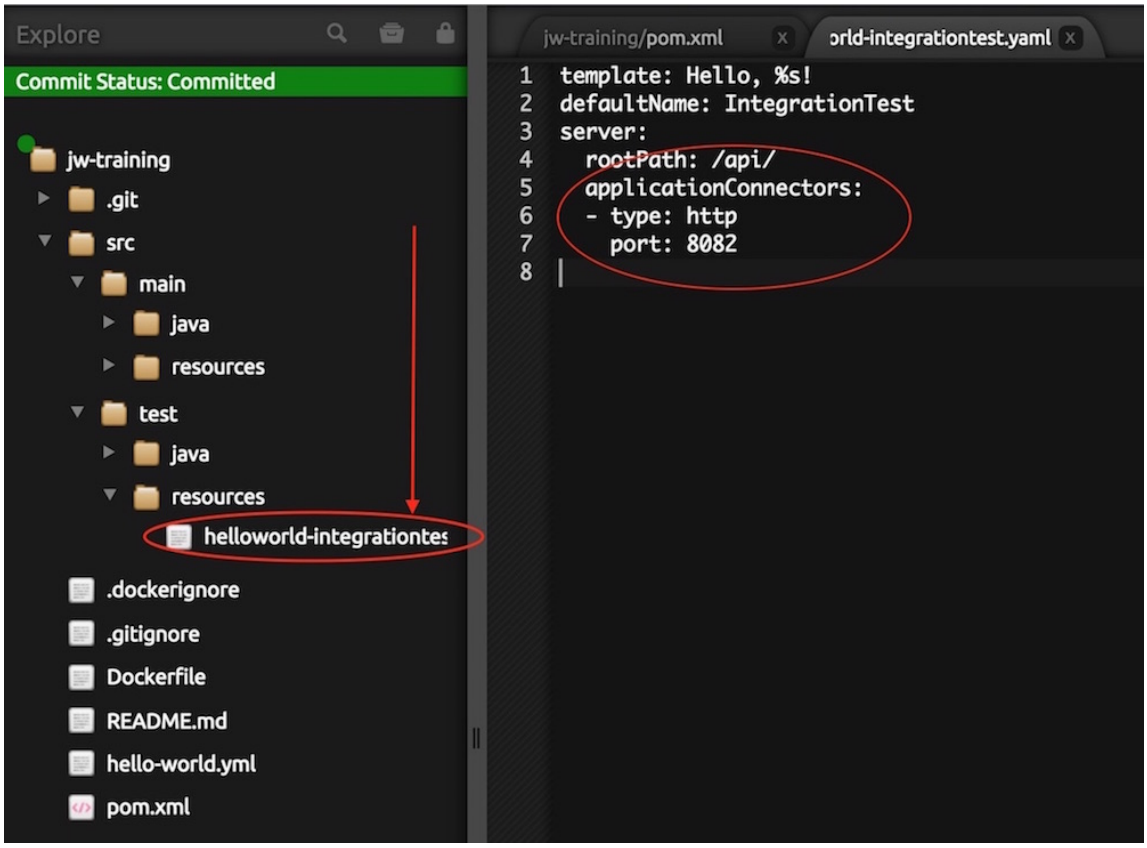

Figure 18. Resulting Corrected YAML

- Now, time to **Commit** and **Push**.
- Since we configured the **Builder** job to Poll the SCM every minute, a build will start in the next minute. This build should be successful.

IMPORTANT	Port allocation is a shared resource, and has to be managed as such. There are a lot of patterns for accomplishing this, like using fungible Jenkins Agent If you cannot use this, you may use Port Allocator Plugin that will allow Jenkins to manage the TCP allocation port, providing the value in an environment variable

Browsing Jobs & Build Pages

These exercises are here to let you re-discover some useful information on the builds and job pages

- Try to reach the GitServer's code diff you've just committed, only using **Jenkins** links

TIP	Use the **Recent Changes** link from the Project page, or the **Changes** section from a given build page

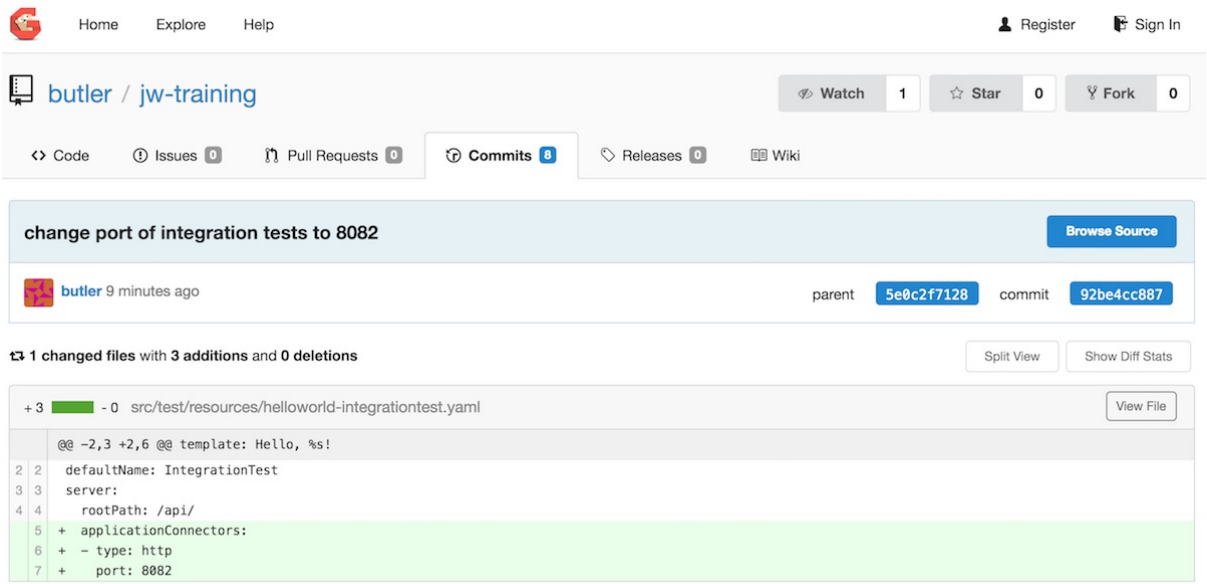

Figure 19. Diff page to reach

- More help:

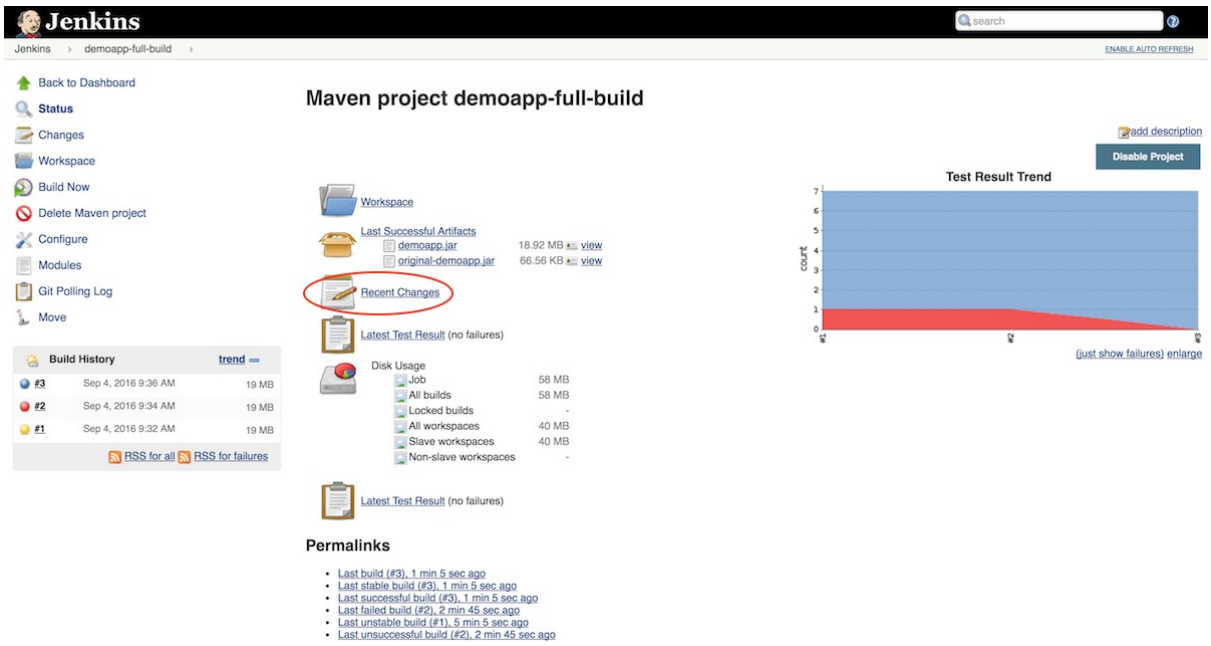

Figure 20. Job Main Page

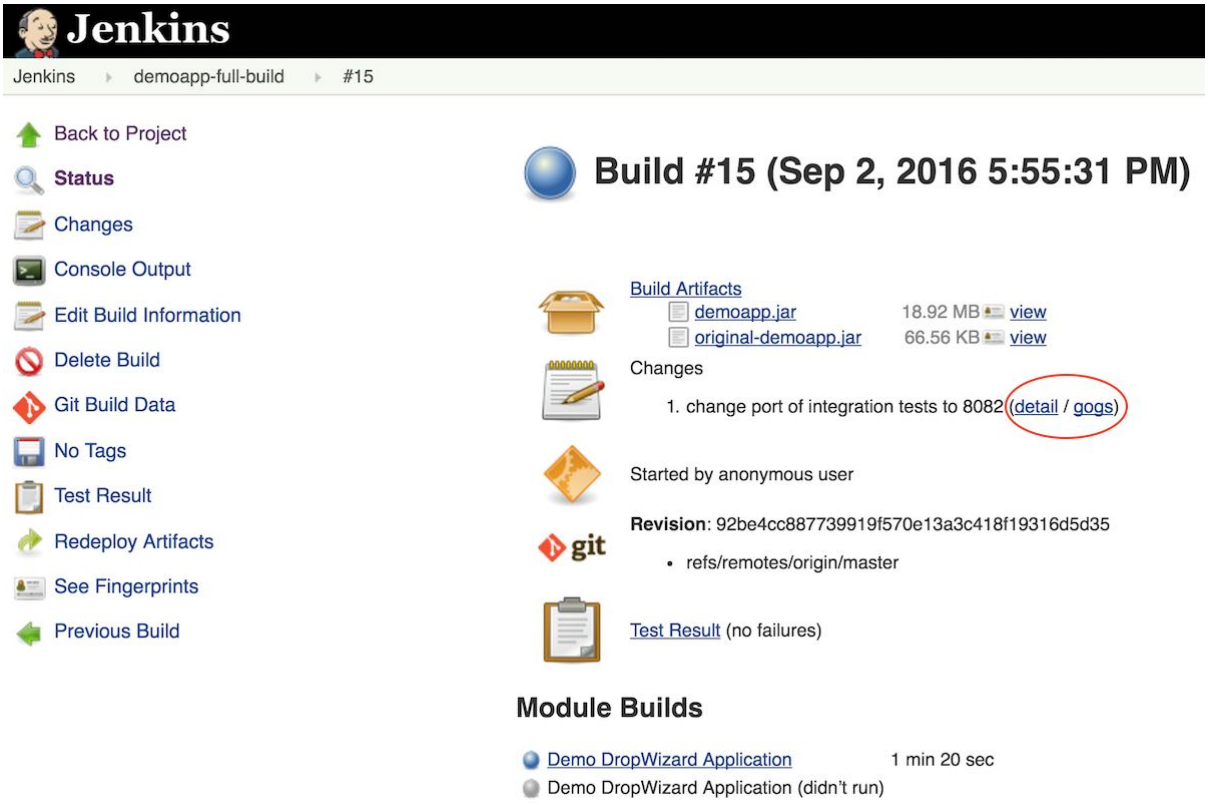

Figure 21. A Build Main Page

That's all for this exercise!

Triggering Jobs

The goal of this exercise is to experiment a bit with Job Triggers.

Triggering another job

Jenkins implements a trigger system for this:

- Given we have:
  - An **upstream** project *JobA*
  - And a **downstream** project *JobB* depending on *JobA*
- When a build of *JobA* terminates, it will kick-of a build of *JobB*

We want to deploy in `staging` as soon as Integration tests have passed. We don't want to manually launch the `automated` deployment.

IMPORTANT	The Jenkins built-in triggering won't be enough here: staging deployment require a parameter, which is the Docker's Image Name to deploy. The Parameterized Trigger Plugin is required to pass parameter to a downstream job, and already installed in your Jenkins instance.

- Navigate to the `demoapp-build` 's configuration

> **TIP** Direct link: http://localhost:5000/jenkins/job/demoapp-build/configure

- Add another **Post Build Action**:
  - *After* the Artifacts Archiving (drag it if needed)
  - Kind: **Trigger parameterized build on other projects**:
  - **Project to build:** Provide the **Deployer** build name: `demoapp-staging-deployer`
  - **Trigger only when build is:** Stable
  - **Add Parameters:**
    - Type: **Predefined parameters**
    - Value:

```
DOCKER_IMAGE=demo-app:${GIT_COMMIT}
```

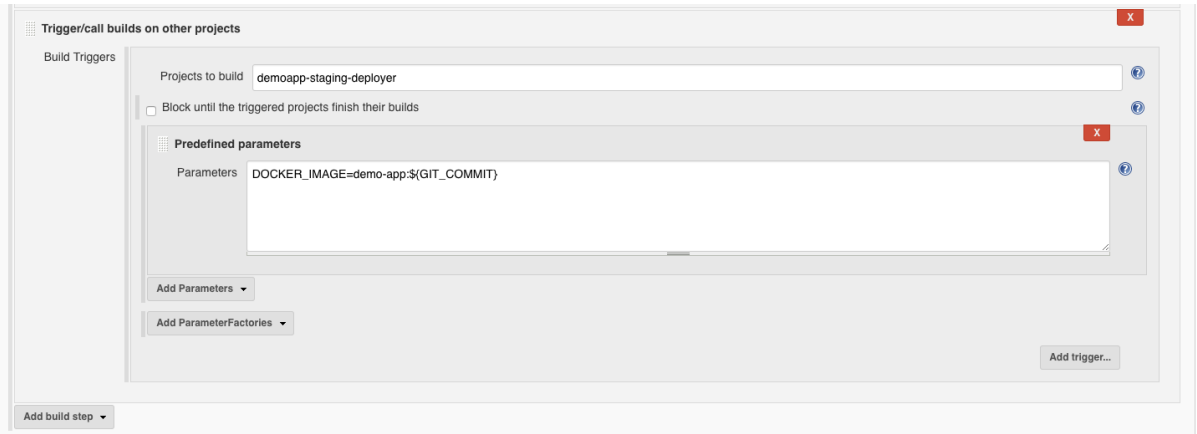

Figure 22. Triggering Deployer Job on Success

- Launch *manually* a new build on `demoapp-build`:
  - The job `demoapp-staging-deployer` will be **automatically** built.
- Browse the latest `demoapp-staging-deployer` to validate that the build cause was due to the **upstream** project:

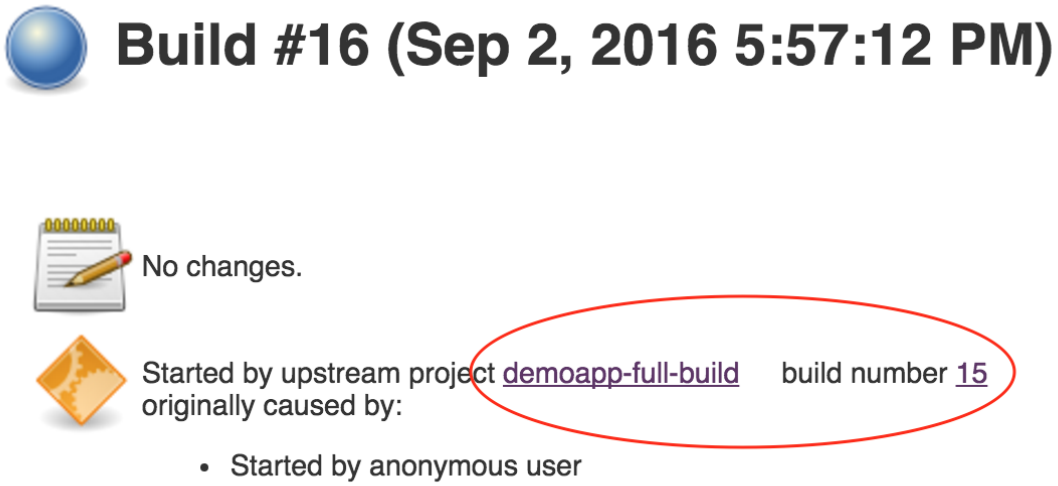

Figure 23. Build cause page to reach

- You can even check which values were passed to your build, by using the left-menu link **Parameters**:

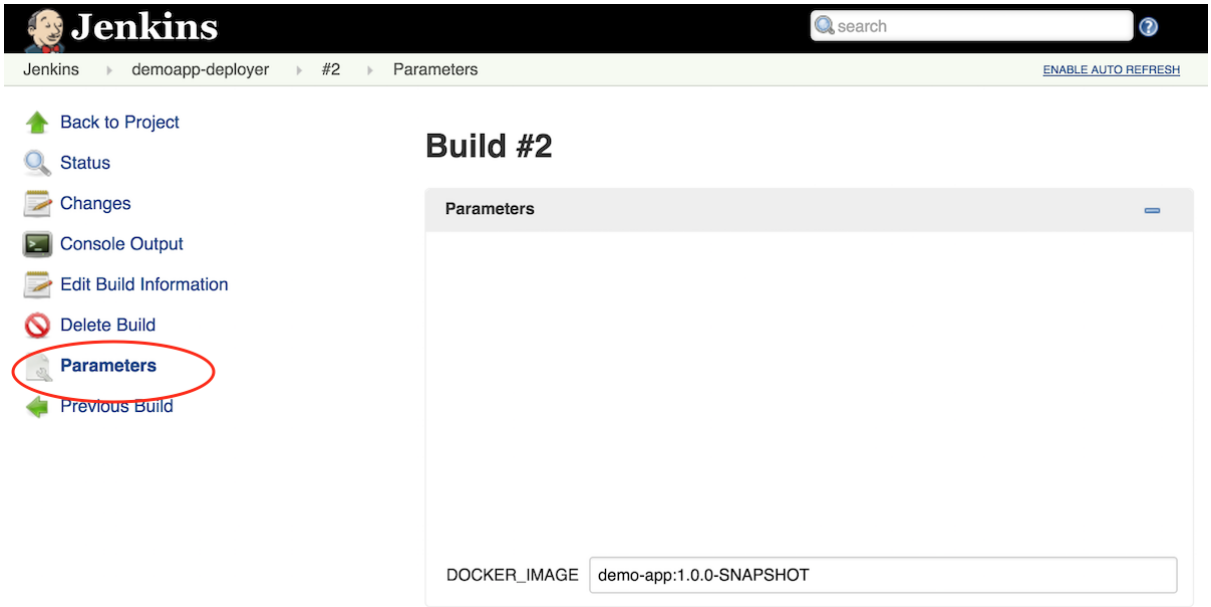

Figure 24. Parameters passed to this build

Polling must die: Triggering with Hooks

- Polling SCM or building regularly is not a good practice:
  - **Admin view:** Waste of resources (unnecessary requests)
  - **Developer view:** Time *between* a change is committed/pushed and the build kick off is too much.
- **Improvement:** we are going to trigger build using a **Hook**:
  - A *hook* is a request made by an external system when event occurs
  - Gogs (the Git Server) provides **WebHook**, which will make HTTP requests based on some event on the Gogs side.

IMPORTANT	This exercise requires having the Gogs Webhook Plugin installed. This is **already** done on your Jenkins instance

- Start by editing `demoapp-build` configuration:
  - Direct link: http://localhost:5000/jenkins/job/demoapp-build/configure
  - Ensure that all build triggers are **disabled** to stop the SCM polling and make your administrators happy people !

TIP	Now, the `demoapp-build` can only be built **manually**. We are sure to not interfere with the hooks settings.

- Log in the Gogs Git Server GUI, as butler
- Browse to the **Webhooks** section of the repository **Settings** page:

TIP	Direct link: http://localhost:5000/gitserver/butler/demoapp/settings/hooks

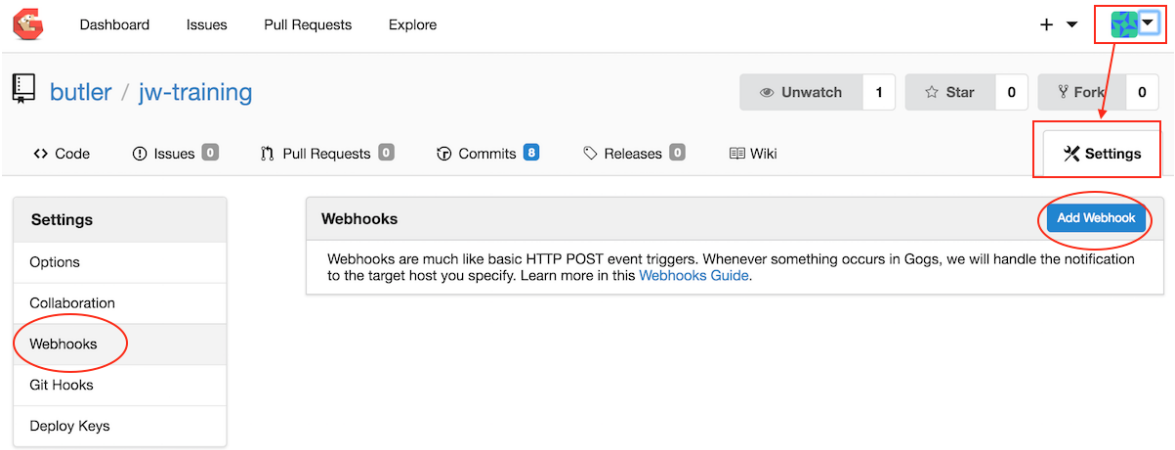

Figure 25. Viewing the WebHooks Settings

- Create a new Webhook by clicking the button **Add WebHook**, and select the type **Gogs**
- Configure the new Webhook with the following settings:
  - **Payload URL:** http://localhost:5000/jenkins/gogs-webhook/?job=demoapp-build
  - Let **Content-Type** and **Secret** to their default values
  - Select trigger **Just on push event**
  - Validate by clicking the Green Button:

TIP	The Payload URL pattern is specific to Gogs, following the plugin's documentation: `${JENKINS_URL}/gogs-webhook/?job=${JOB_NAME}`. Note that the `git-plugin` and other "hook" plugins may implements their own end-points

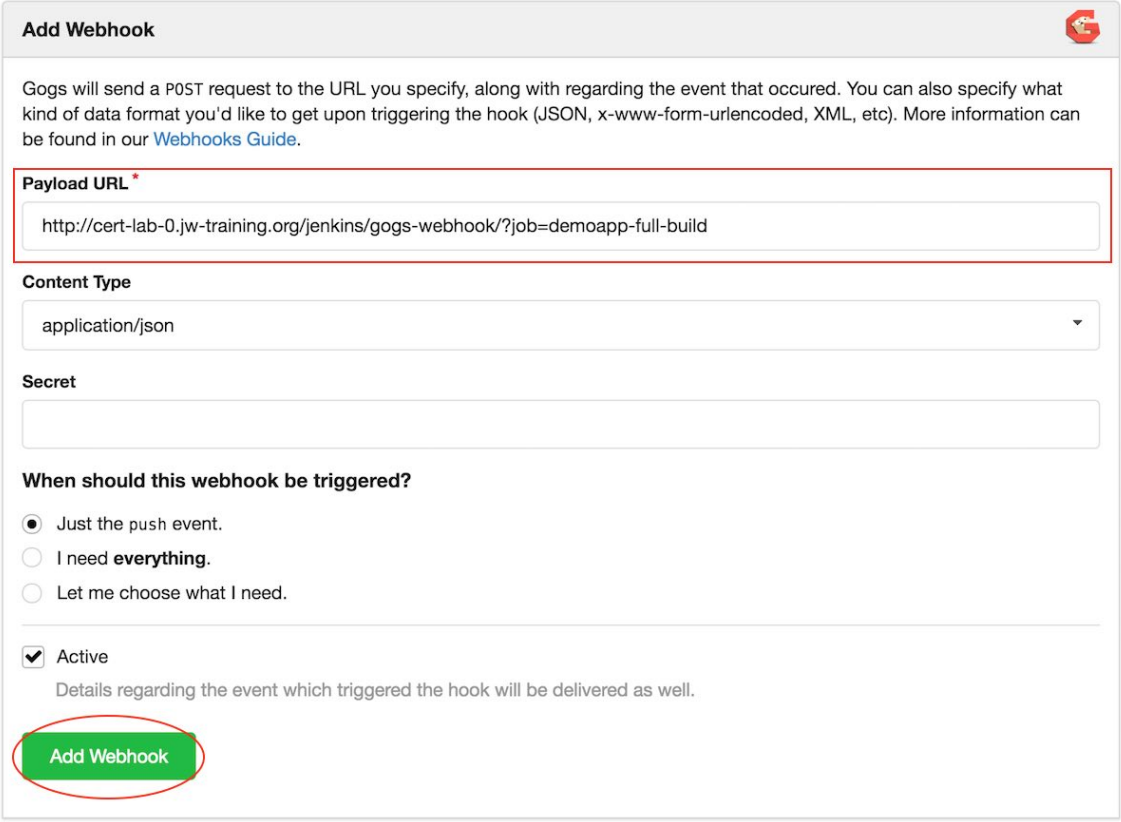

Figure 26. Creating a new WebHook

- Once the webhook is created, click its name to view its configuration:

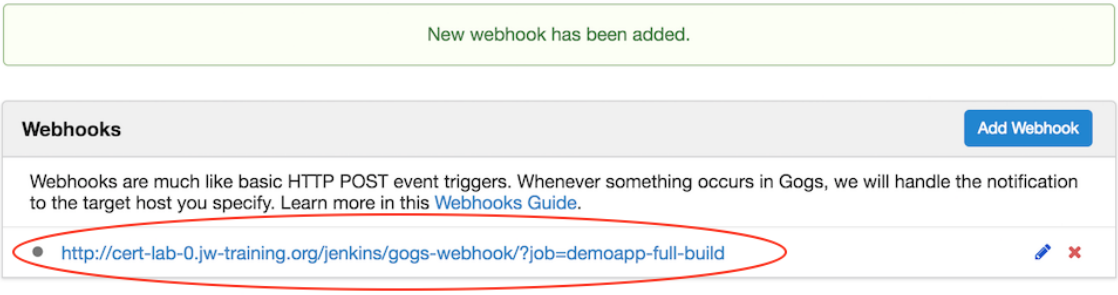

Figure 27. New WebHook created

- From there, you can validate by launching a **Test Delivery** Webhook that will kick-off a build on Jenkins

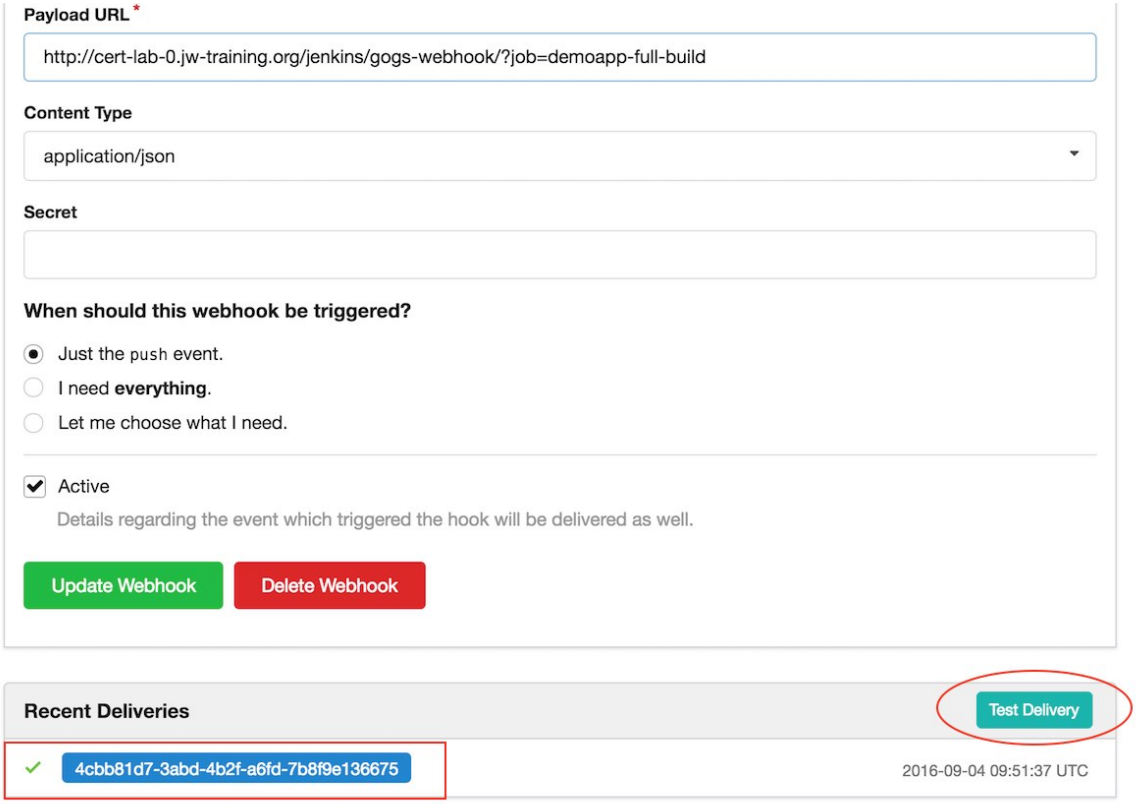

Figure 28. Testing the new WebHook

- Browse to the WebIDE
  - Make a new **harmless** change: add a new line in the `README.md`
  - Commit and push
- Another build should start **immediately**
- Once the latest build succeeds, browse to its **main page** and check the new meta-datas related to this "Hook" system:

TIP	The **Gogs-ID** is the UID of the hook, that can be found on Gogs side for future traceability !

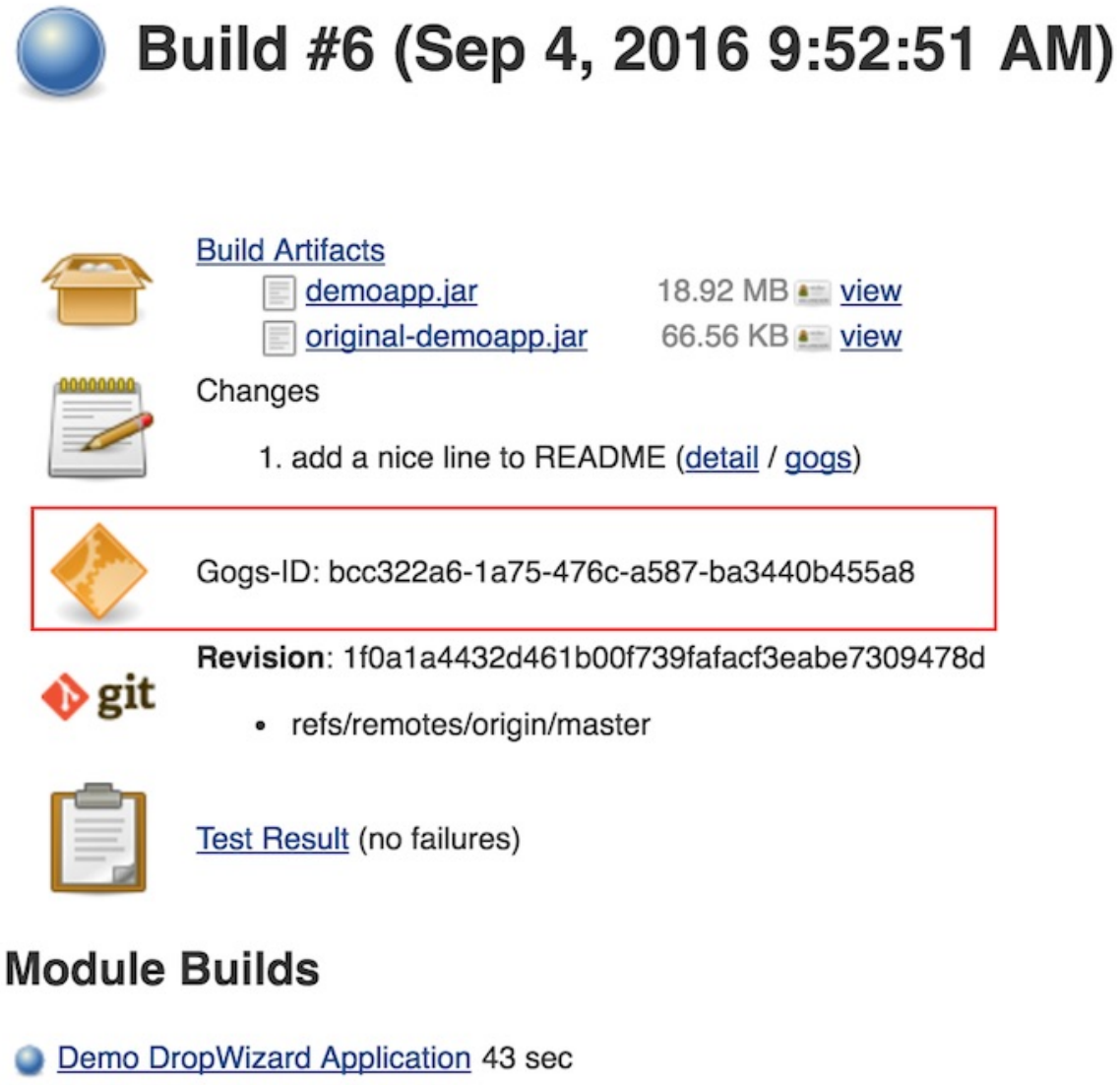

Figure 29. WebHook Triggered Build Page

That's all for this exercise !

Journey Summary

The labs covered:

- Starting and administrating Jenkins
- Installing plugins
- Creating, configuring and running jobs
  - Different kinds of jobs, with parameterization
- Triggering builds based on different events