[] | *./images/devonfw.png*

devonfw shop floor dev-SNAPSHOT

The devonfw community

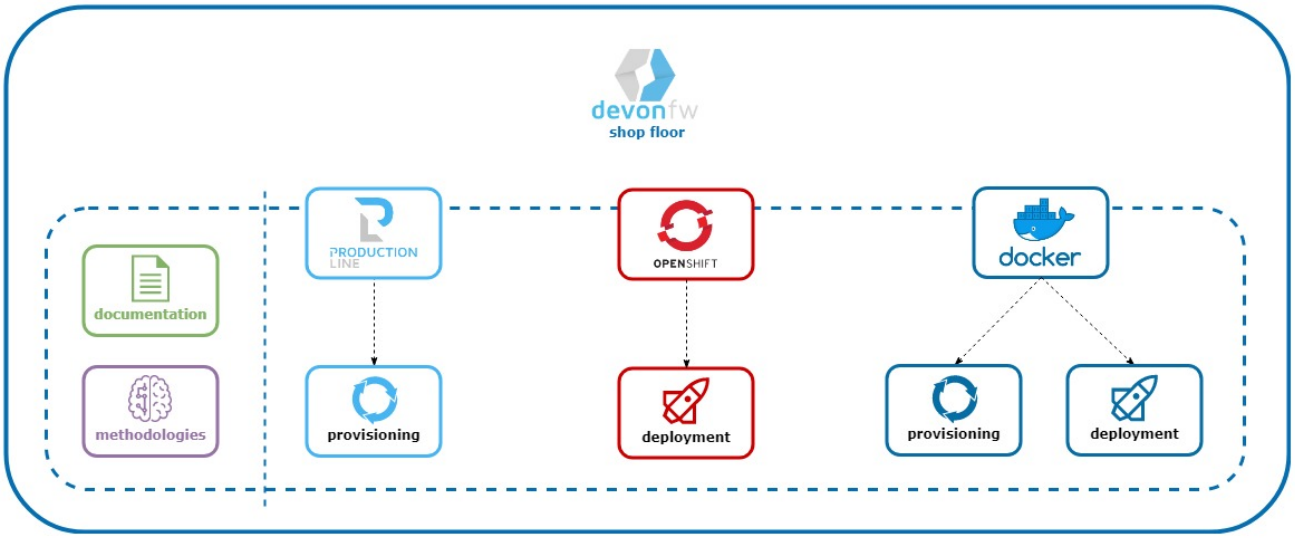
Version , 2019-07-03_11.44.56

Table of Contents

What is devonfw shop floor? . . . 1
1. How to use it . . . 1
1.1. Prerequisites - Provisioning environment . . . 2
1.2. Step 1 - Configuration and services integration . . . 2
1.3. Step 2 - Create the project . . . 2
1.3.1. Create and integrate git repository . . . 2
1.3.2. start new devonfw project . . . 2
1.3.3. cicd configuration . . . 2
1.4. Step 3 - Deployment . . . 3
2. Provisioning environments . . . 3
2.1. Production Line provisioning environment . . . 3
2.1.1. How to obtain your Production Line . . . 3
2.2. dsf4docker provisioning environment . . . 3
2.2.1. Architecture overview . . . 3
2.2.2. Prerequisite . . . 4
2.2.3. How to use it . . . 4
2.2.4. A little history . . . 5
3. Configuration and services integration . . . 5
3.1. Nexus Configuration . . . 5
3.1.1. Prerequisites . . . 5
3.1.2. Jenkins integration . . . 6
3.2. SonarQube Configuration . . . 8
3.2.1. Generate user token . . . 8
3.2.2. Webhook . . . 8
3.2.3. Jenkins integration . . . 8
4. Create project . . . 9
4.1. Create and integrate git repository . . . 9
4.2. start new devonfw project . . . 9
4.3. cicd configuration . . . 9
4.3.1. Manual configuration . . . 9
5. Deployment . . . 9
6. Annexes . . . 9
6.1. Custom Services . . . 9
6.1.1. BitBucket . . . 10
6.2. Mirabaud CICD Environment Setup . . . 21
6.2.1. 1. Install Docker and Docker Compose in RHEL 6.5 . . . 21
6.2.2. 2. Directories structure . . . 22
6.2.3. 3. CICD Services with Docker . . . 23

6.2.4. 4. CICD Services with Docker Compose . . . 25
6.2.5. 5. Service Integration . . . 28
6.2.6. Jenkins - GitLab integration . . . 29
6.2.7. Jenkins - Nexus integration . . . 36
6.2.8. Jenkins - SonarQube integration . . . 41
6.3. OKD *(OpenShift Origin)* . . . 49
6.3.1. What is OKD . . . 49
6.3.2. Install OKD *(Openshift Origin)* . . . 50
6.3.3. How to use Oc Cluster Wrapper . . . 51
6.3.4. devonfw Openshift Origin Initial Setup . . . 51
6.3.5. s2i devonfw . . . 52
6.3.6. devonfw templates . . . 53
6.3.7. Customize Openshift Origin for devonfw . . . 55

What is devonfw shop floor?



devonfw shop floor is a set of documentation, tools and methodologies used to configure the provisioning, development and uat environments used in your projects. devonfw shop floor allows the administrators of those environments to apply CI/CD operations and enables automated application deployment.

devonfw shop floor is mainly oriented to configure the provisioning environment provided by Production Line and deploy applications on an OpenShift cluster. In the cases where Production Line or OpenShift cluster are not available, there will be alternatives to achieve similar goals.

The **devonfw shop floor 4 OpenShift** is a solution based on the experience of priming devonfw for OpenShift by RedHat.

Let's start.

1. How to use it

This is the documentation about shop floor and its different tools. Here you are going to learn how to create new projects, so that they can include continuous integration and continuous delivery processes, and be deployed automatically in different environments.

1.1. Prerequisites - Provisioning environment

To start working you need to have some services running in your provisioning environment, such as Jenkins (automation server), GitLab (git repository), SonarQube (program analysis), Nexus (software repository) or similar.

To host those services we recommend to have a Production Line instance but you can use other platforms. Here is the list for the different options:

- Production Line.
- dsf4docker.

1.2. Step 1 - Configuration and services integration

The first step is configure your services and integrate them with jenkins. Here you have an example about how to manually configure the next services:

- Nexus.
- SonarQube.

1.3. Step 2 - Create the project

1.3.1. Create and integrate git repository

The second is create or git repository and integrate it with Jenkins.

Here you can find a manual guide about how it:

- GitLab new project.

1.3.2. start new devonfw project

It is time to create your devonfw project:

- visit our devon4ng guide.
- visit our devon4j guide.

1.3.3. cicd configuration

Now you need to add cicd files in your project.

Manual configuration

Jenkinsfile

Here you can find all that you need to know to do your Jenkinsfile.

1.4. Step 3 - Deployment

- dsf4openshift.

2. Provisioning environments

2.1. Production Line provisioning environment

The Production Line Project is a set of server-side collaboration tools for Capgemini engagements. It has been developed for supporting project engagements with individual tools like issue tracking, continuous integration, continuous deployment, documentation, binary storage and much more!

For aditional information use the official documentation.

2.1.1. How to obtain your Production Line

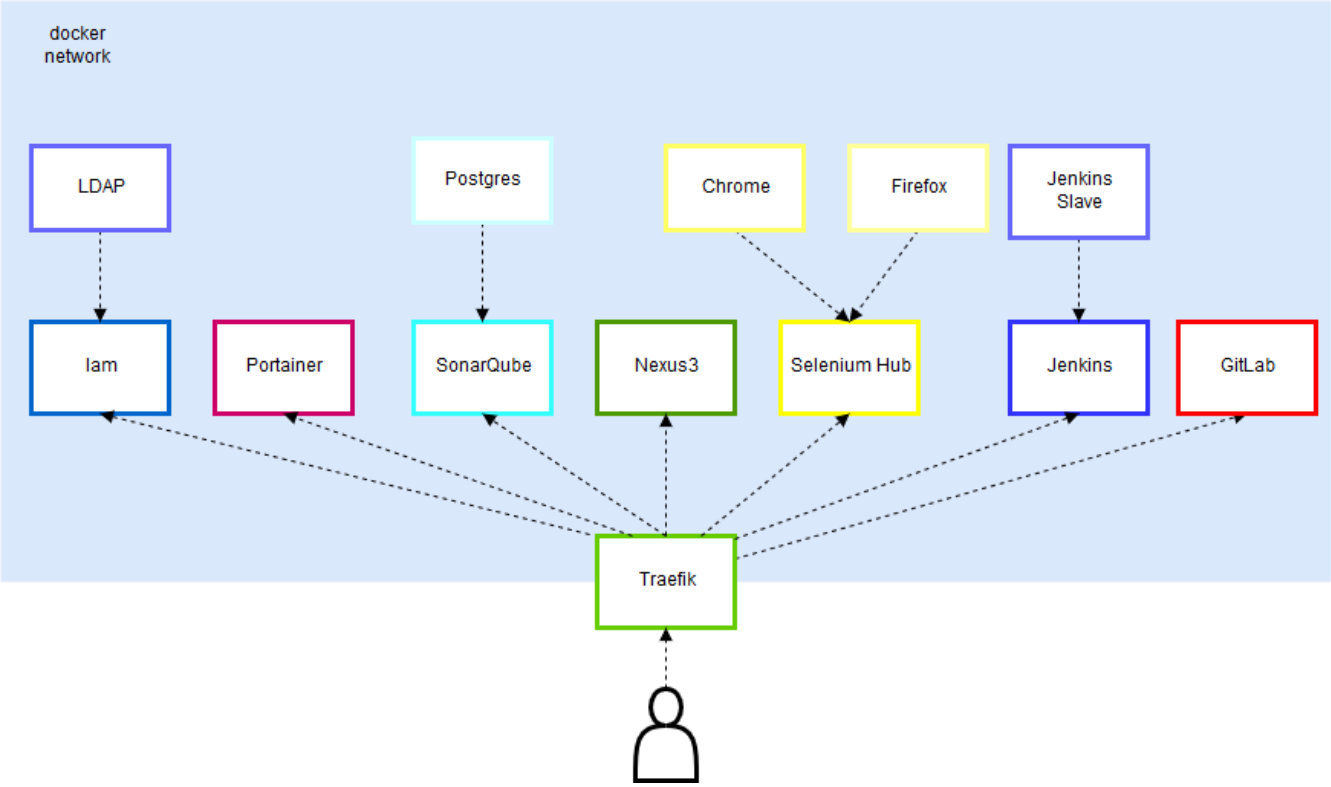
You can order your Production Line environment instance following the official guide. Remember that you need to order at least the next tools: * Jenkins * GitLab * SonarQube * Nexus

---

Back.

2.2. dsf4docker provisioning environment

2.2.1. Architecture overview



2.2.2. Prerequisite

To use dsf4docker provisioning environment you need a remote server and you must clone or download devonfw shop floor.

2.2.3. How to use it

Navigate to `./devonfw-shop-floor/dsf4docker/environment` and here you can find one scripts to install it, and another one to unistall it.

Install devonfw shop floor 4 Docker

There is an installation script to do so, so the complete installation should be completed by running it. Make sure this script has execution permissions in the Docker Host:

```
# chmod +x dsf4docker-install.sh
# sudo ./dsf4docker-install.sh
```

This script, besides the container "installation" itself, will also adapt the `docker-compose.yml` file to your host (using `sed` to replace the **IP_ADDRESS** word of the file for your real Docker Host's IP address).

Uninstall devonfw shop floor 4 Docker

As well as for the installation, if we want to remove everything concerning **devonfw shop floor 4 Docker** from our Docker Host, we'll run this script:

```
# chmod +x dsf4docker-uninstall.sh
# sudo ./dsf4docker-uninstall.sh
```

2.2.4. A little history

The **Docker** part of the shop floor is created based on the experience of the environment setup of the project **Mirabaud Advisory**, and intended to be updated to latest versions. Mirabaud Advisory is a web service developed with devonfw (Java) that, alongside its own implementation, it needed an environment both for the team to follow CICD rules through their 1-week-long sprints and for the client (Mirabaud) to check the already done work.

There is a practical experience about the Mirabaud Case.

Back.

3. Configuration and services integration

3.1. Nexus Configuration

In this document you will see how you can configure Nexus repository and how to integrate it with jenkins.

3.1.1. Prerequisites

Repositories

You need to have one repository for snapshots, another for releases and another one for release-candidates. Normally you use maven2 (hosted) repositories and if you are going to use a docker registry, you need docker (hosted) too.

To create a repository in Nexus go to the administration clicking on the gear icon at top menu bar. Then on the left menu click on Repositories and press the ***Create repository*** button.

[nexus create repository] | *./images/configuration/nexus-create-repository.png*

Now you must choose the type of the repository and configure it. This is an example for Snapshot:

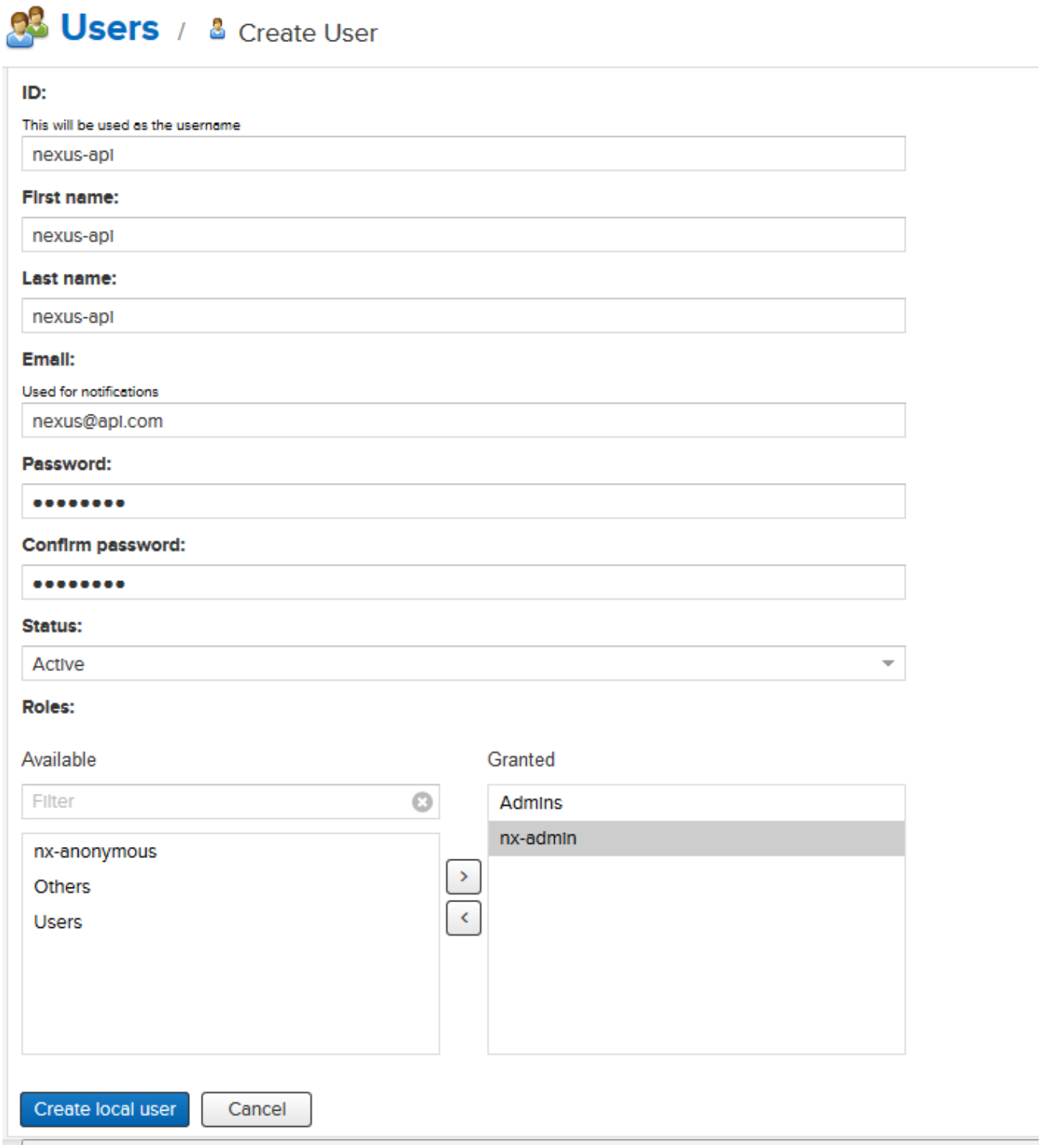
[nexus create repository form] | *./images/configuration/nexus-create-repository-form.png*

[[dsf-configure-nexus.asciidoc_create-user-to-upload/download-content]] == Create user to upload/download content

Once you have the repositories, you need a user to upload/download content. To do it go to the administration clicking on the gear icon at top menu bar. Then on the left menu click on Users and press the ***Create local*** user button.

[nexus create user] | *./images/configuration/nexus-create-user.png*

Now you need to fill a form like this:

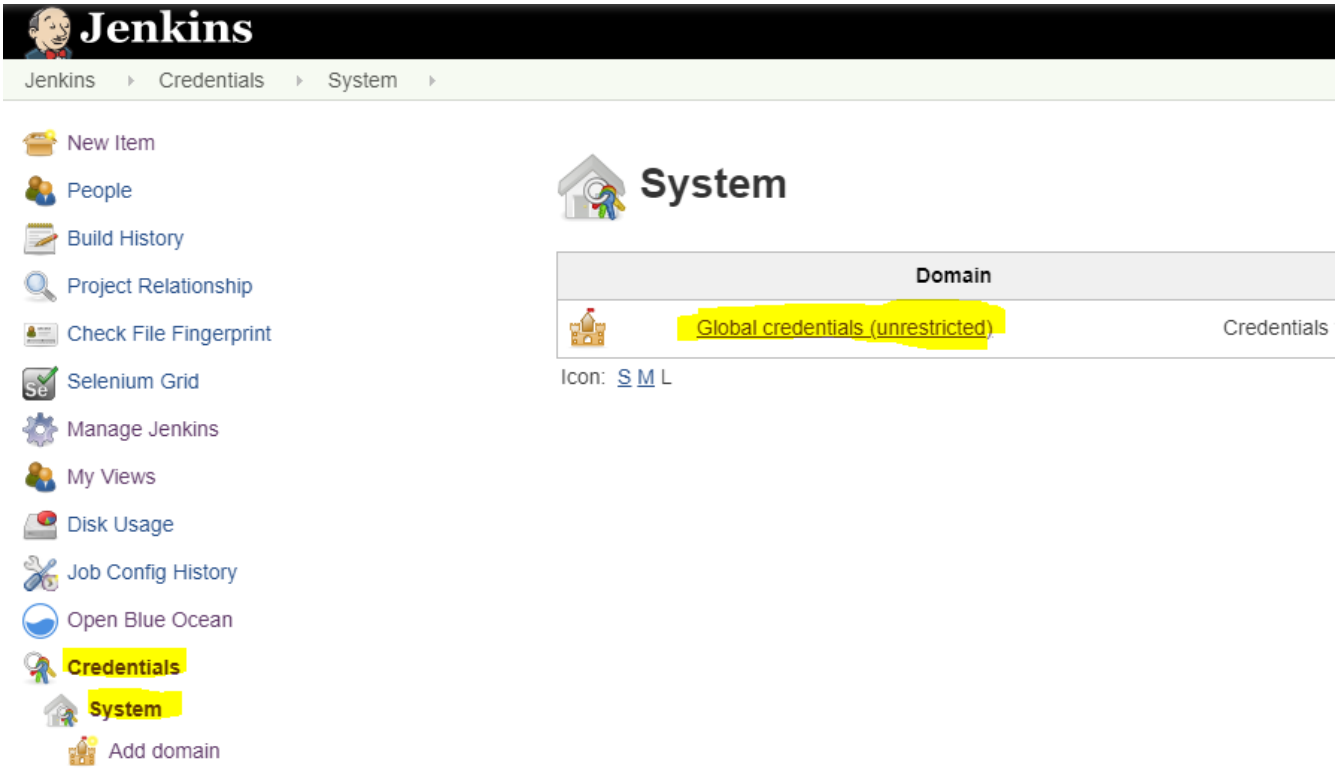


3.1.2. Jenkins integration

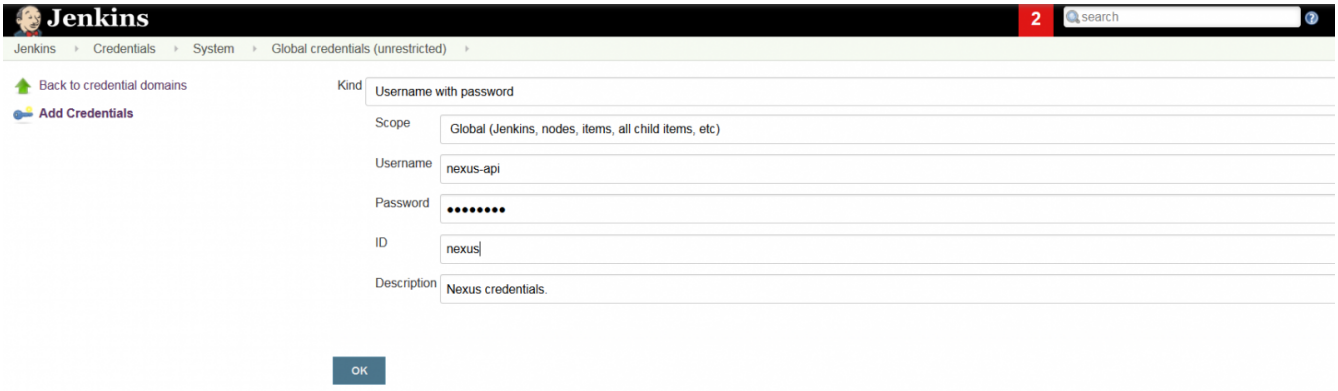
To use Nexus in our pipelines you need to configure Jenkins.

Add nexus user credentials

First of all you need to add the user created in the step before to Jenkins. To do it (on the left menu) click on Credentials, then on System. Now you could access to ***Global credentials (unrestricted)***.

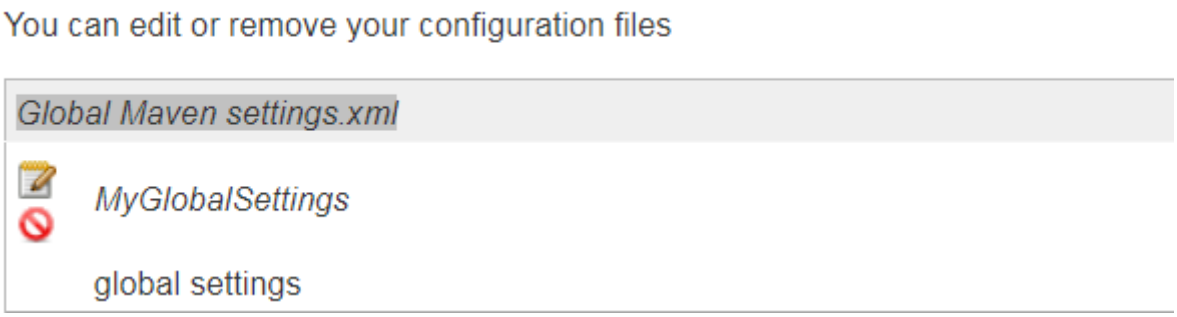


Enter on it and you could see a button on the left to ***Add credentials***. Click on it and fill a form like this:

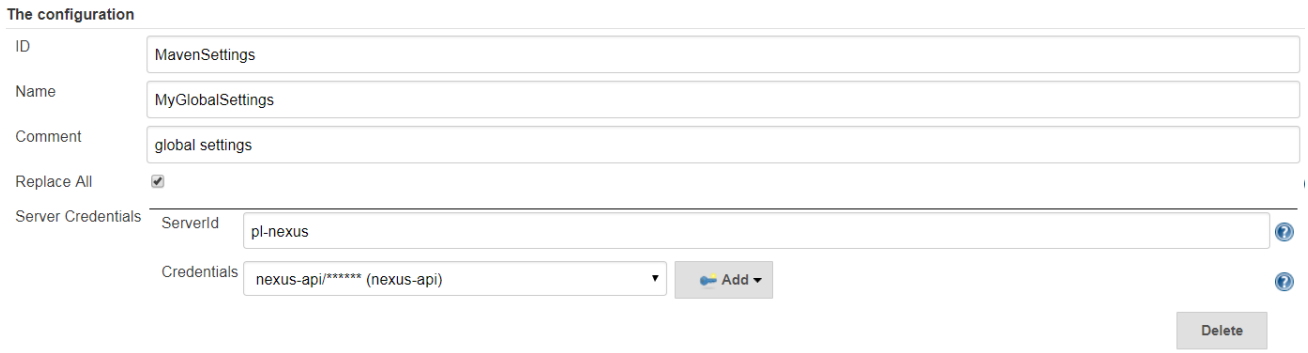


Add the nexus user to maven global settings

Now you need to go to Manage Jenkins clicking on left menu and enter in ***Managed files***.



Edit the Global Maven settings.xml to add your nexus repositories credentials as you could see in the next image:



And you are done.

3.2. SonarQube Configuration

To use SonarQube you need to use a token to connect, and to know the results of the analysis you need a webhook. Also, you need to install and configure SonarQube in Jenkins.

3.2.1. Generate user token

To generate the user token, go to your account clicking in the left icon on the top menu bar.

[sonarqube administration] | *./images/configuration/sonarqube-administration.png*

Go to security tab and generate the token.

[sonarqube token] | *./images/configuration/sonarqube-token.png*

3.2.2. Webhook

When you execute our SonarQube scanner in our pipeline job, you need to ask SonarQube if the quality gate has been passed. To do it you need to create a webhook.

Go to administration clicking the option on the top bar menu and select the tab for Configuration.

Then search in the left menu to go to webhook section and create your webhook.

An example for Production Line:

[sonarqube webhook] | *./images/configuration/sonarqube-webhook.png*

3.2.3. Jenkins integration

To use SonarQube in our pipelines you need to configure Jenkins to integrate SonarQube.

SonarQube Scanner

First, you need to configure the scanner. Go to Manage Jenkins clicking on left menu and enter in ***Global Tool Configuration***.

Go to SonarQube Scanner section and add a new SonarQube scanner like this.

[sonarqube jenkins scanner] | *./images/configuration/sonarqube-jenkins-scanner.png*

SonarQube Server

Now you need to configure where is our SonarQube server using the user token that you create before. Go to Manage Jenkins clicking on left menu and enter in ***Configure System***.

For example, in ProductionLine the server is the next:

[sonarqube jenkins server] | *./images/configuration/sonarqube-jenkins-server.png*

Remember, the token was created at the beginin of this SonarQube configuration.

4. Create project

4.1. Create and integrate git repository

include::dsf-configure-gitlab.asciidoc[leveloffset=2].

4.2. start new devonfw project

It is time to create your devonfw project:

- visit our devon4ng guide.
- visitr our devon4j guide.

4.3. cicd configuration

4.3.1. Manual configuration

Jenkinsfile

include::dsf-configure-jenkins.asciidoc[leveloffset=2].

5. Deployment

include::dsf-deployment-dsf4openshift.asciidoc[leveloffset=2].

6. Annexes

6.1. Custom Services

6.1.1. BitBucket

[Under construction]

The purpose of the present document is to provide the basic steps carried out to setup a BitBucket server in OpenShift.

Introduction

BitBucket is the Atlassian tool that extends the Git functionality, by adding integration with JIRA, Confluence, or Trello, as well as incorporates extra features for security or management of user accounts (See BitBucket).

BitBucket server is the Atlassian tool that runs the BitBucket services (See BitBucket server).

The followed approach has been not using command line, but OpenShift Web Console, by deploying the Docker image **atlassian/bitbucket-server** (available in Docker Hub) in the existing project **Deployment**.

The procedure below exposed consists basically in three main steps:

1. Deploy the BitBucket server image (from OpenShift web console)
2. Add a route for the external traffic (from OpenShift web console)
3. Configure the BitBucket server (from BitBucket server web console)

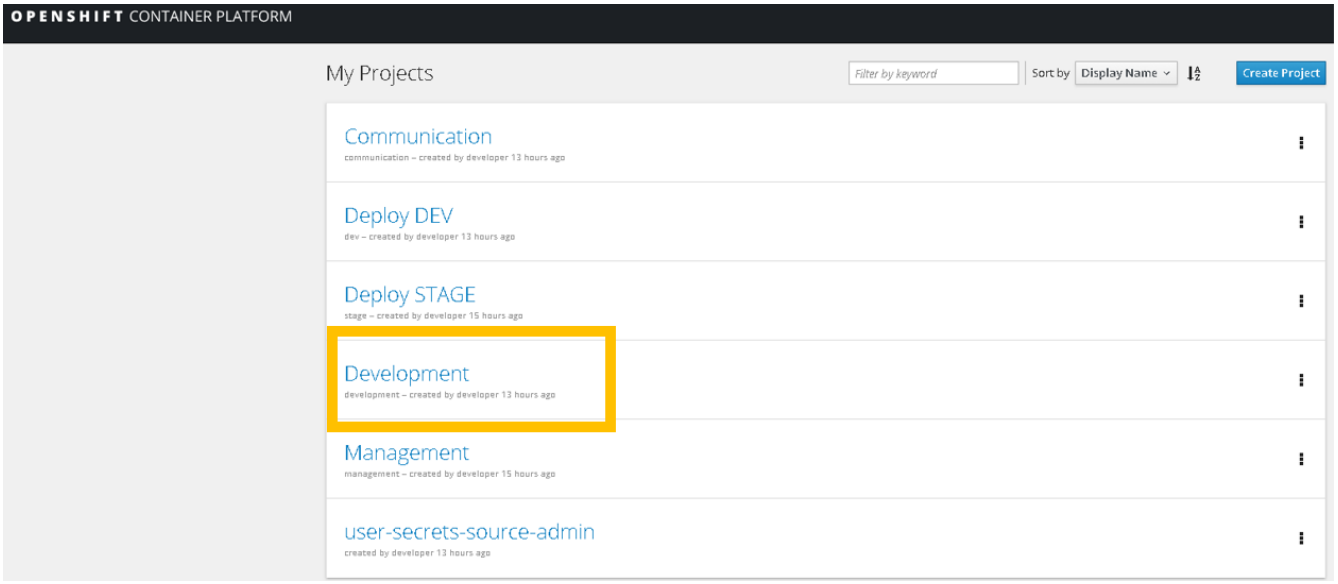
Prerequisites

- OpenShift up & running
- Atlassian account (with personal account key). Not required for OpenShift, but for the initial BitBucket server configuration.

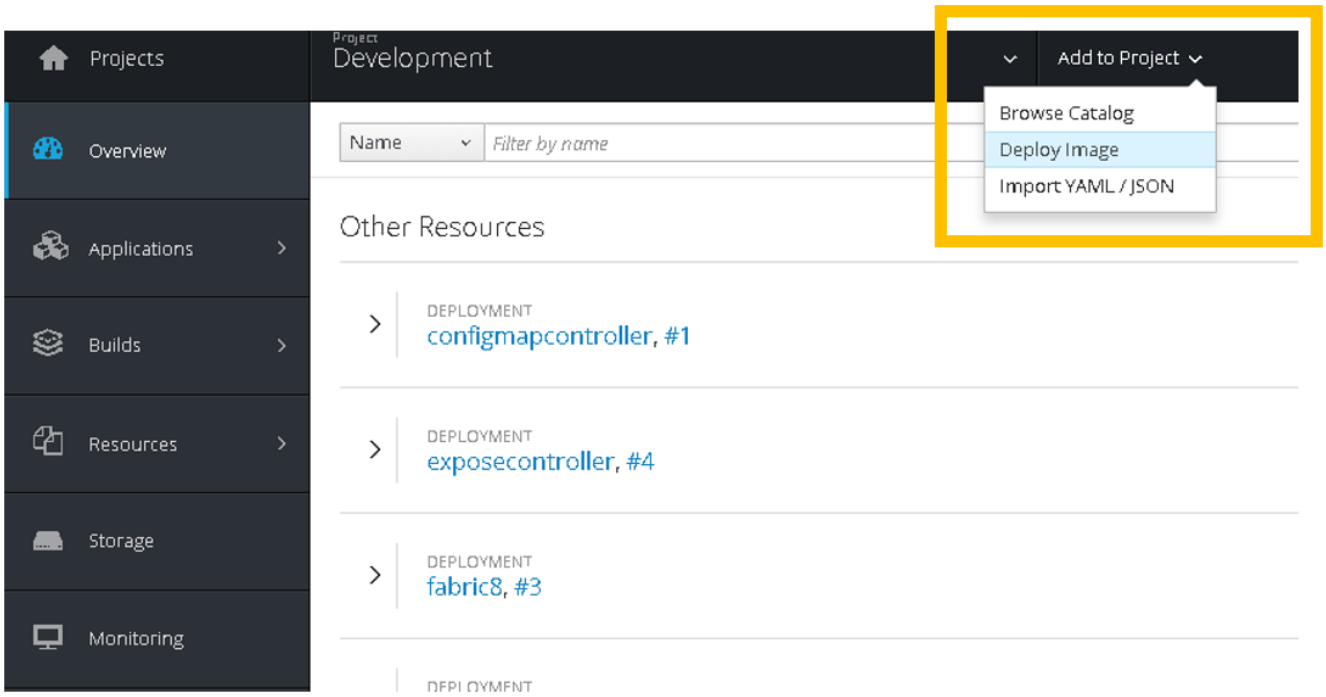
Procedure

[[dsf-openshift-services-bitbucket-basic-server-setup.asciidoc_step-0-log-into-our-linkhttps//10.68.26.1638443/console/logout[openshift-web-console]]] === Step 0: Log into our OpenShift Web console image::./images/others/bitbucket/step0.png[]

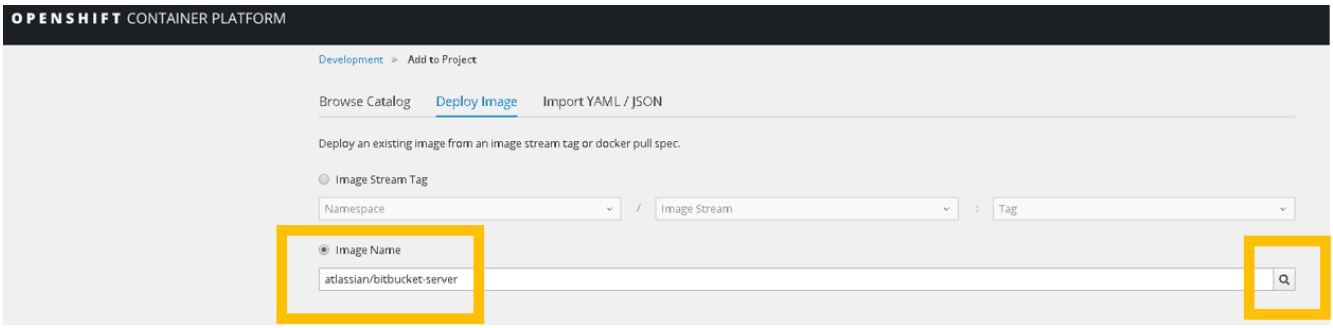
Step 1: Get into Development project



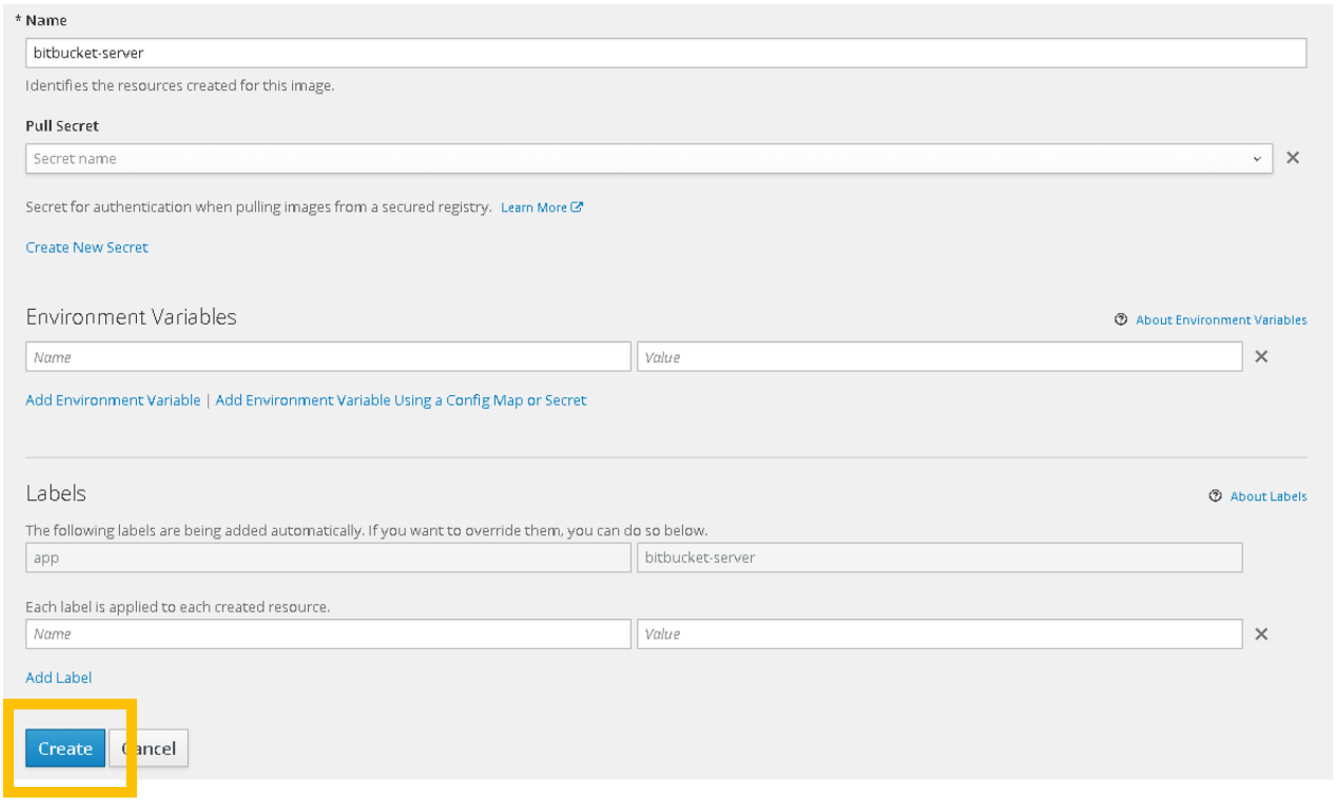
Step 2.1: Deploy a new image to the project



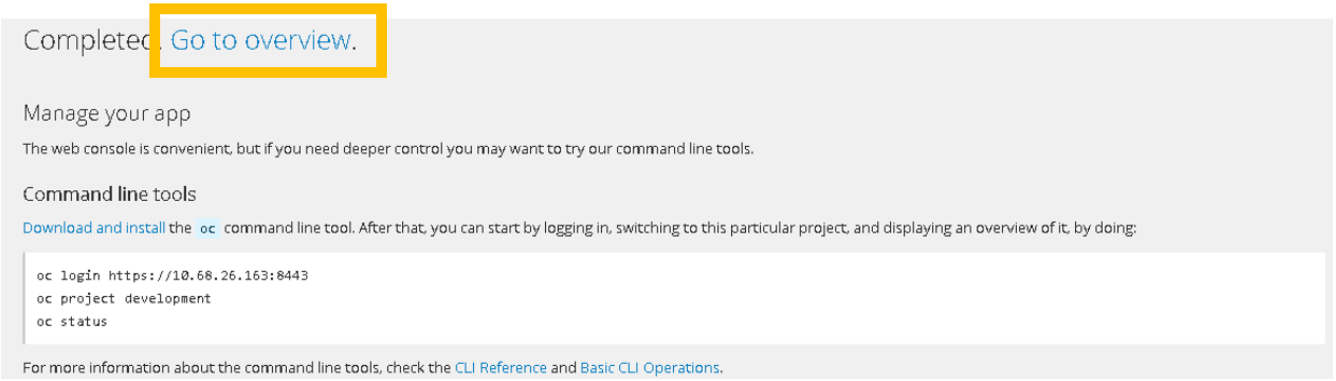
[[dsf-openshift-services-bitbucket-basic-server-setup.asciidoc_step-2.2-introduce-the-image-name-available-in-linkhttps//hub.docker.com/r/atlassian/bitbucket-server/[docker-hub]-and-search]] === Step 2.2: Introduce the image name (available in Docker Hub) and search Image name: **atlassian/bitbucket-server**



Step 2.3: Leave by the moment the default config. since it is enough for the basic setup. Press Create

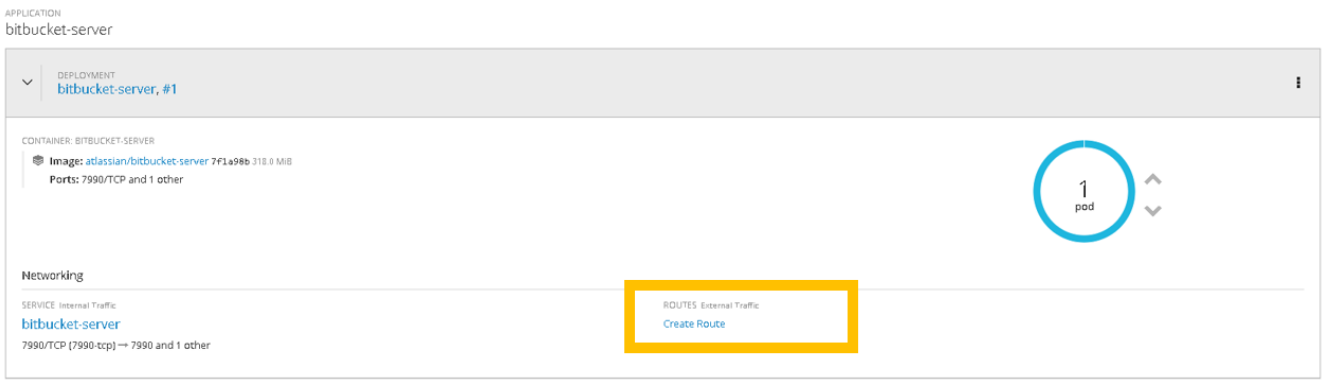


Step 2.4: Copy the oc commands in case it is required to work via command line, and Go to overview



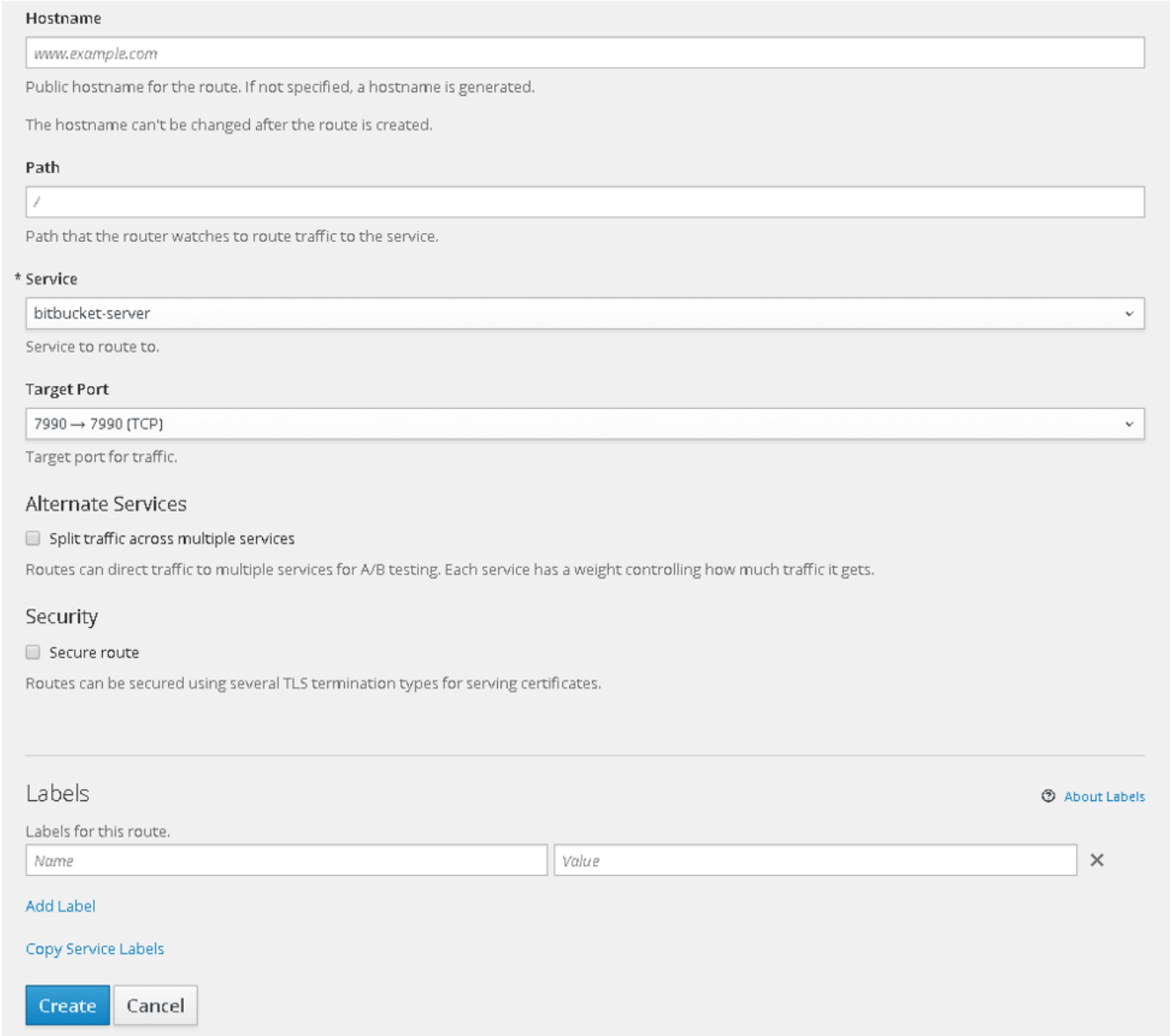
Step 2.5: Wait until OpenShift deploys and starts up the image. All the info will be available.

Please notice that there are no pre-configured routes, hence the application is not accessible from outside the cluster.



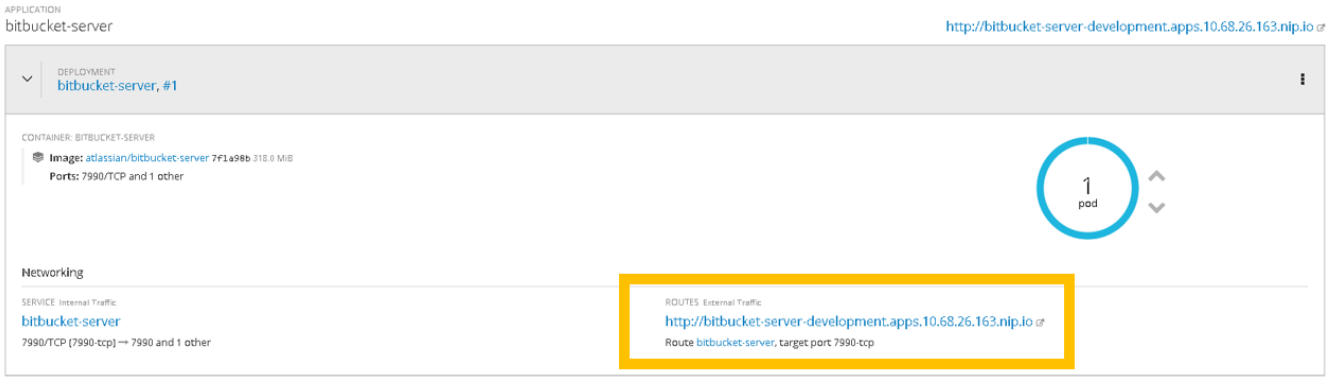
Step 3: Create a route in order for the application to be accessible from outside the cluster (external traffic). Press Create

Please notice that there are different fields that can be specified (hostname, port). If required, the value of those fields can be modified later.



Leave by the moment the default config. as it is enough for the basic setup.

The route for external traffic is now available.

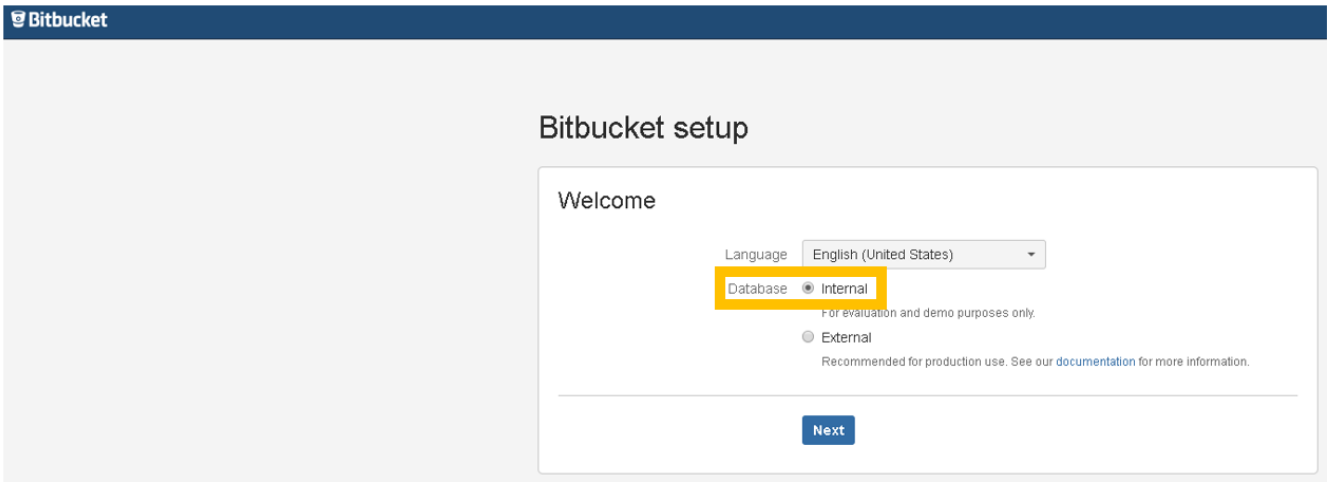


Now the BitBucker server container is up & running in our cluster.

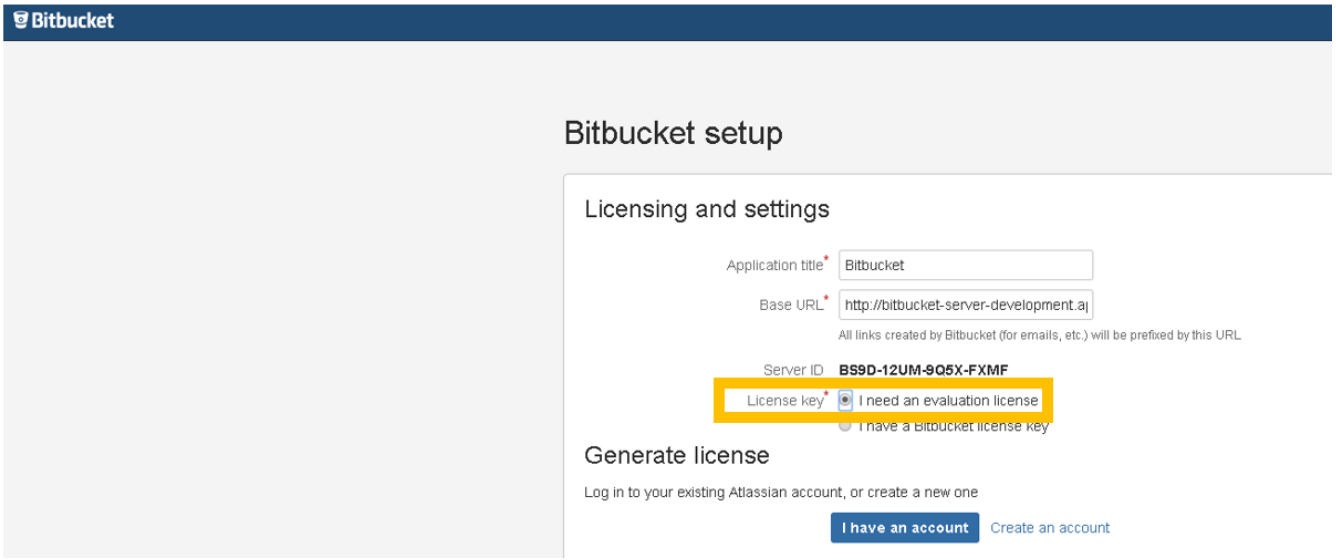
The below steps correspond to the basic configuration of our BitBucket server.

Step 4.1: Click on the link of the external traffic route. This will open our BitBucket server setup wizard

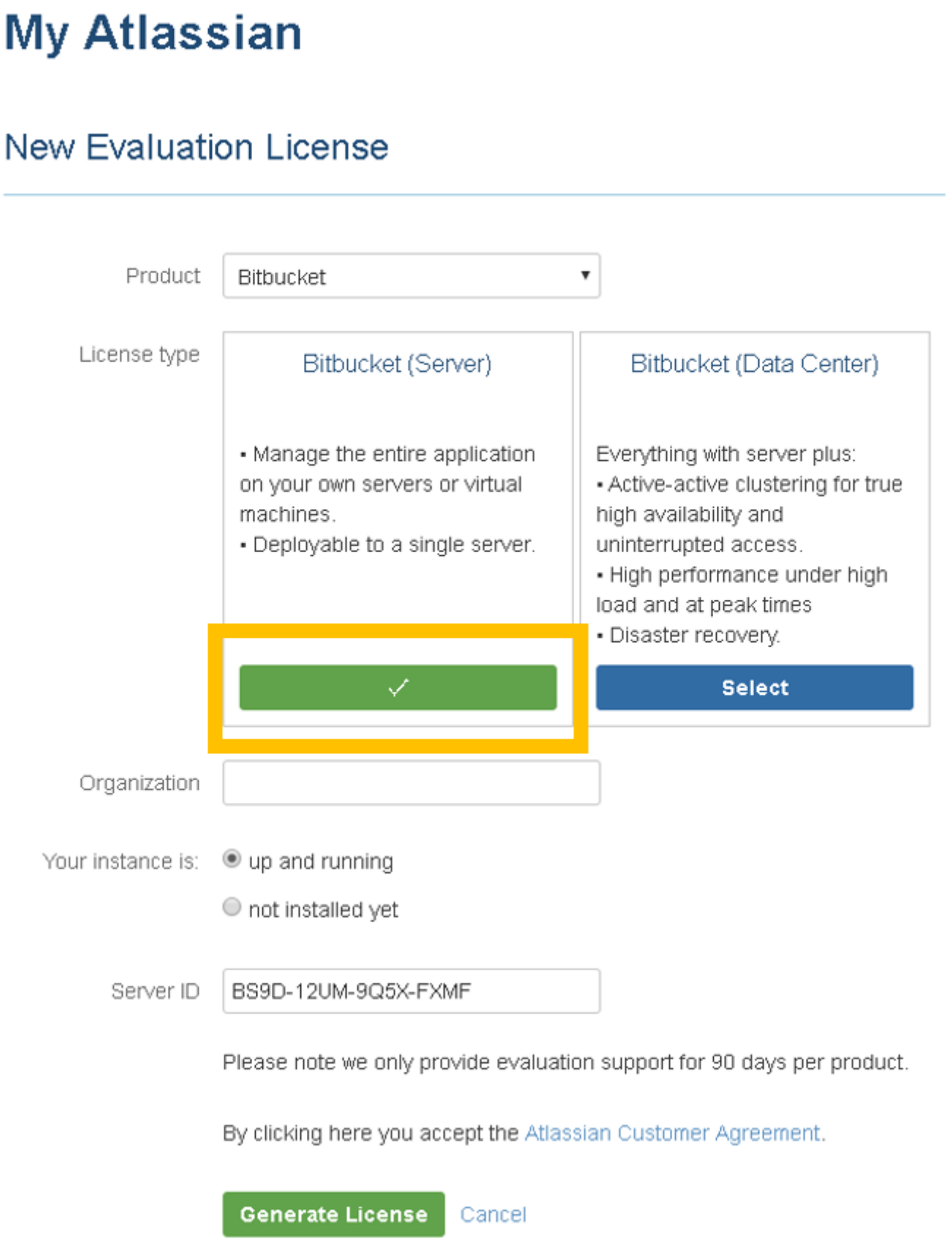
Step 4.2: Leave by the moment the Internal database since it is enough for the basic setup (and it can be modified later), and click Next



Step 4.3: Select the evaluation license, and click I have an account

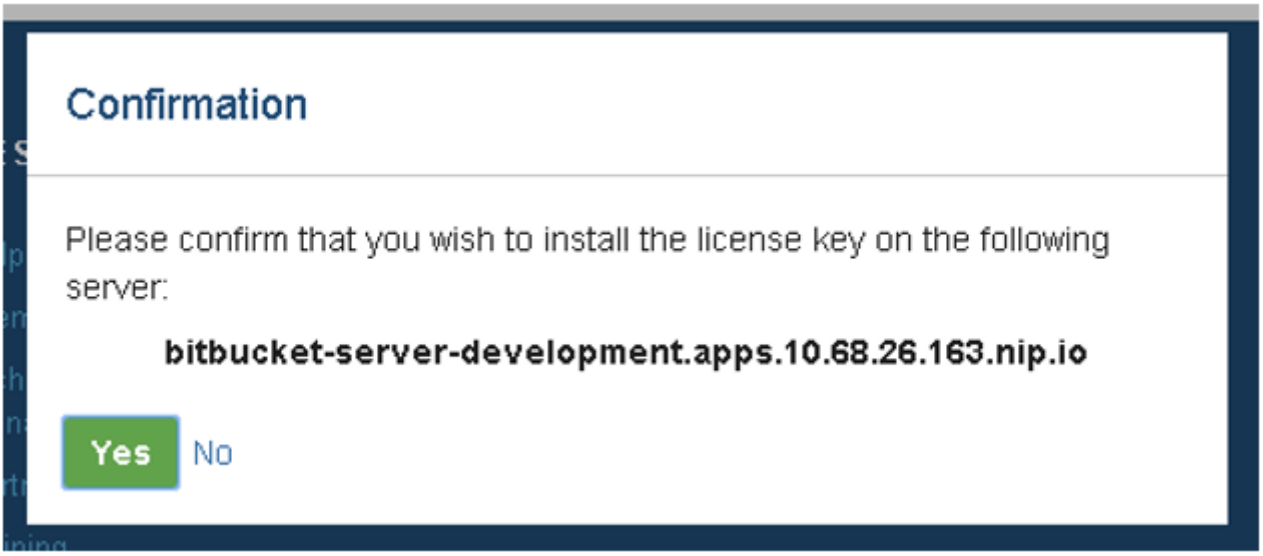


Step 4.4: Select the option Bitbucker (Server)



Step 4.5: Introduce your organization (Capgemini), and click Generate License

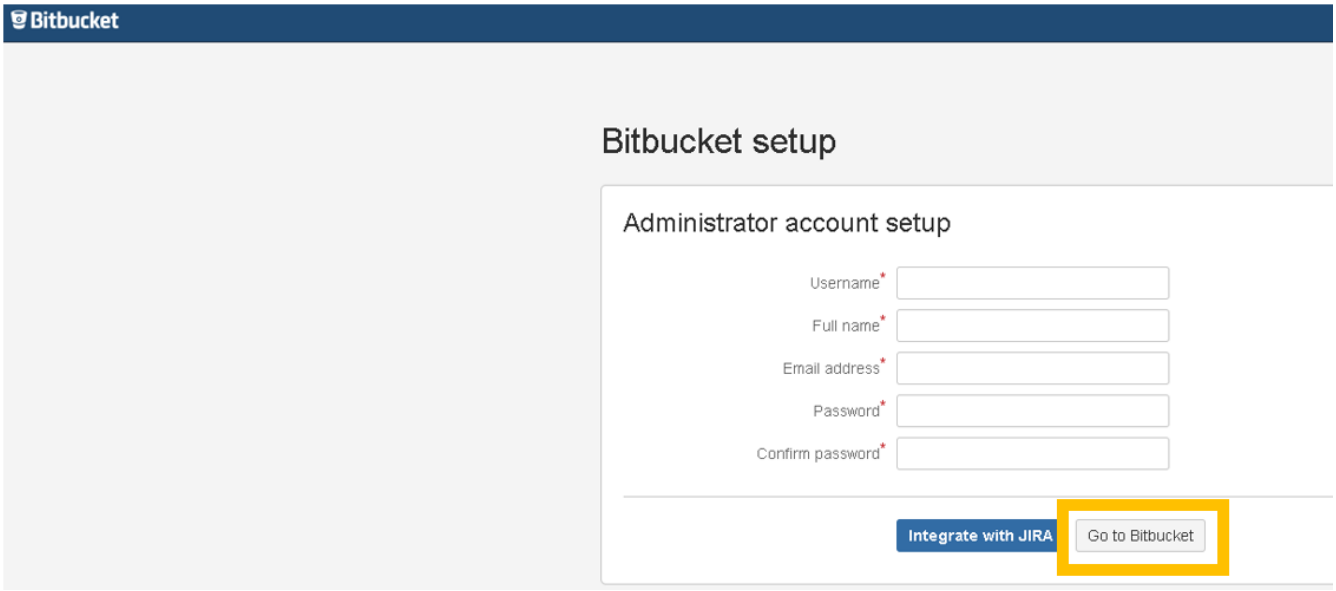
Step 4.6: Confirm that you want to install the license on the BitBucket server



The license key will be automatically generated. Click **Next**

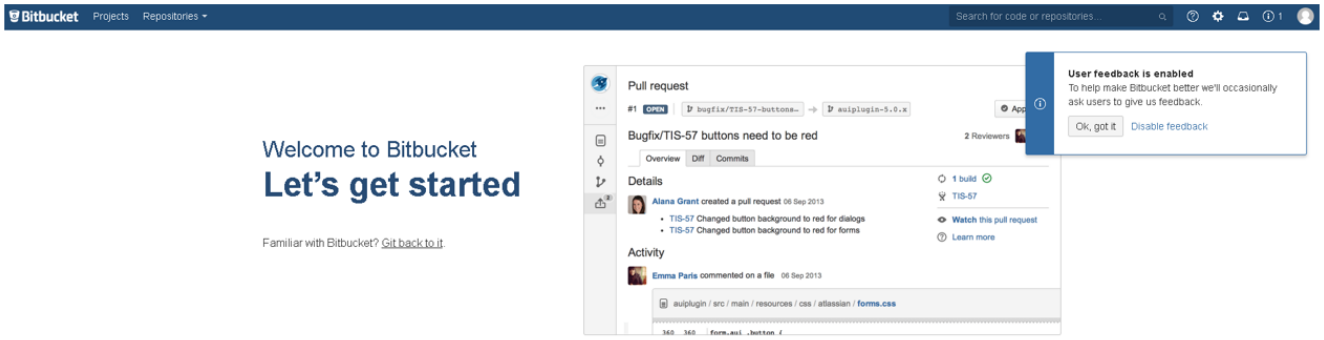
Step 4.7: Introduce the details of the Administration account.

Since our BitBucket server is not going to be integrated with JIRA, click on Go to Bitbucket. The integration with JIRA can be configured later.



Step 4.8: Log in with the admin account that has been just created

DONE !!



[Under construction]

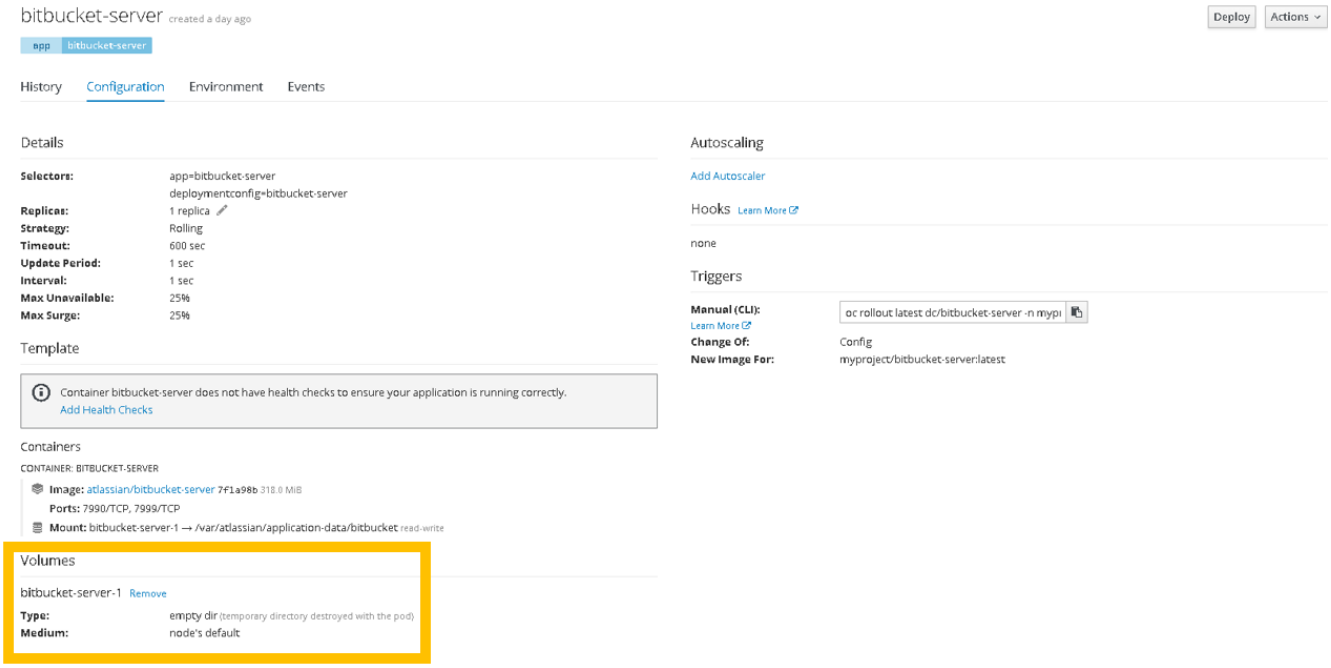
The purpose of the present document is to provide the basic steps carried out to improve the configuration of BitBucket server in OpenShift.

The improved configuration consists on:

- Persisten Volume Claims
- Health Checks *(pending to be completed)*

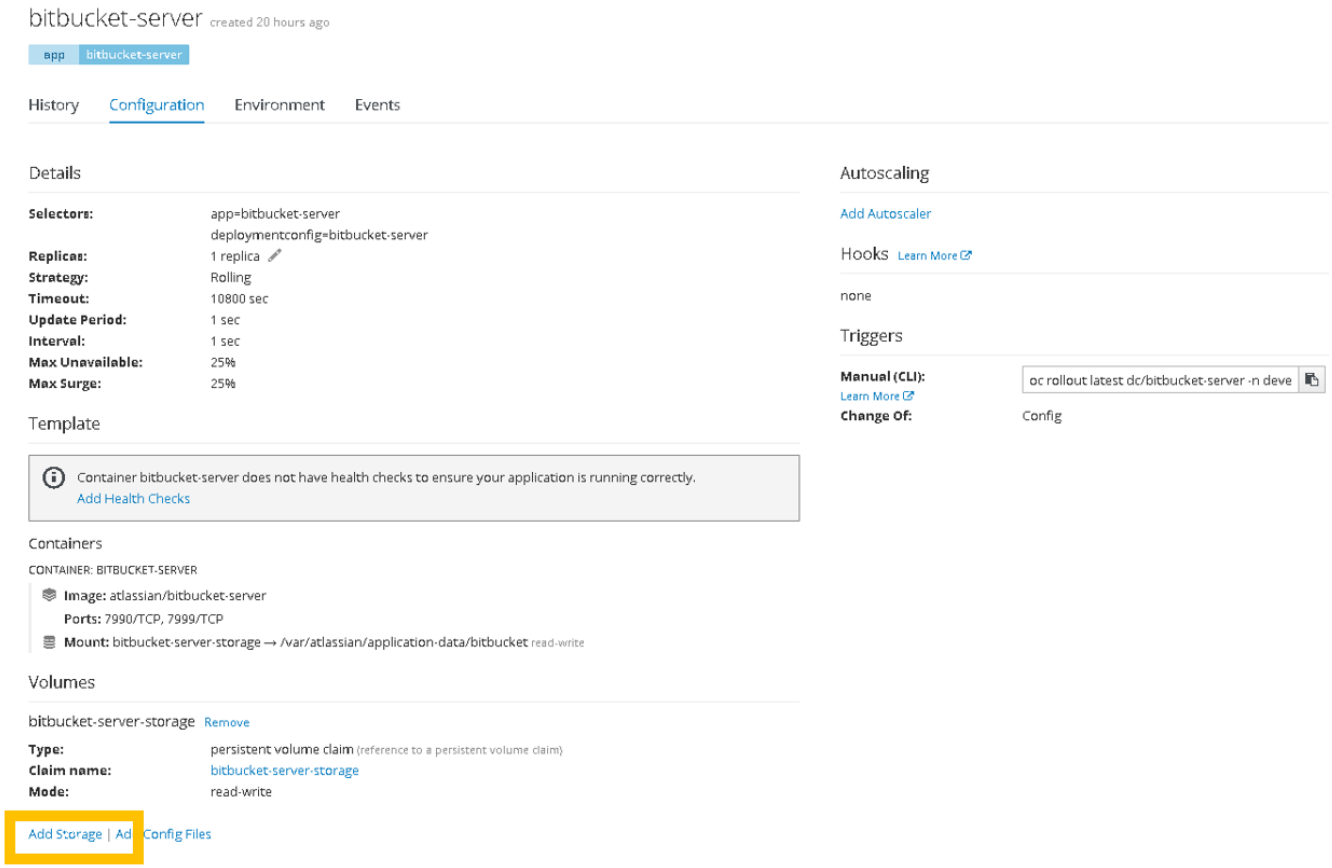
Persisten Volume Claims.

Please notice that the BitBucket server container does not use persistent volume claims by default, which means that the data (e.g.: BitBucket server config.) will be lost from one deployment to another.

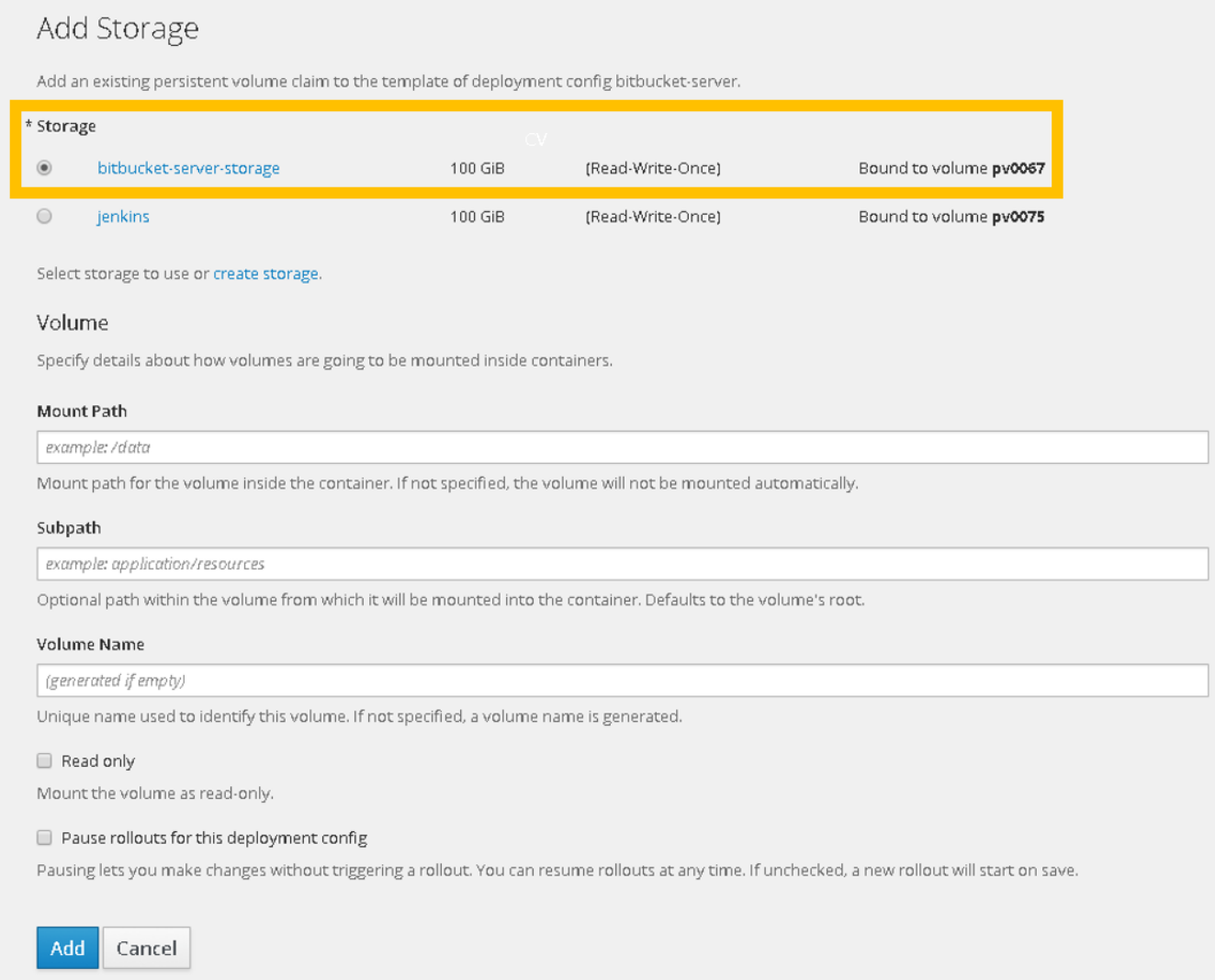


It is very important to create a persisten volume claim in order to prevent the mentioned loss of data.

Step 1: Add storage

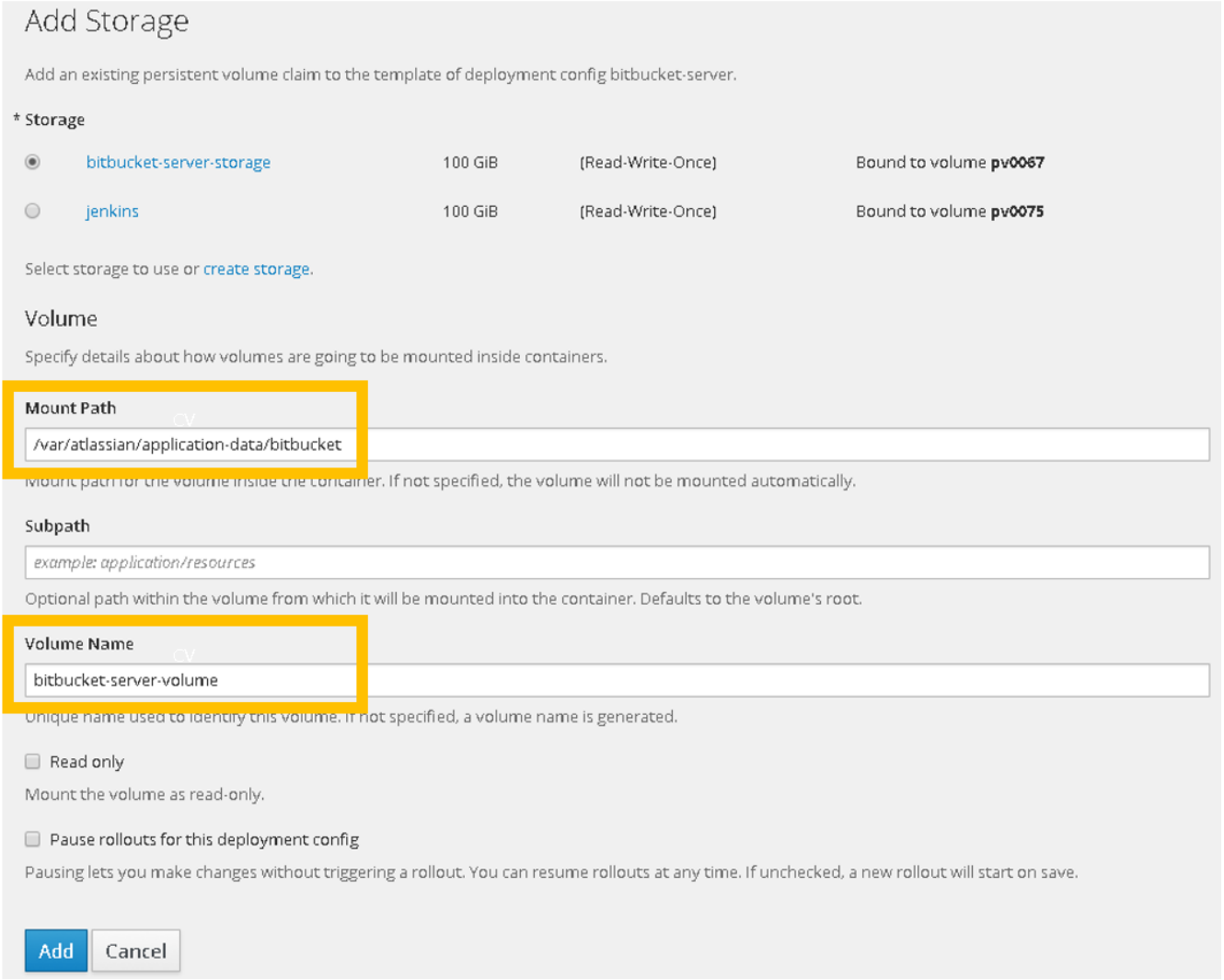


Step 2: Select the appropriate storage, or create it from scratch if necessary

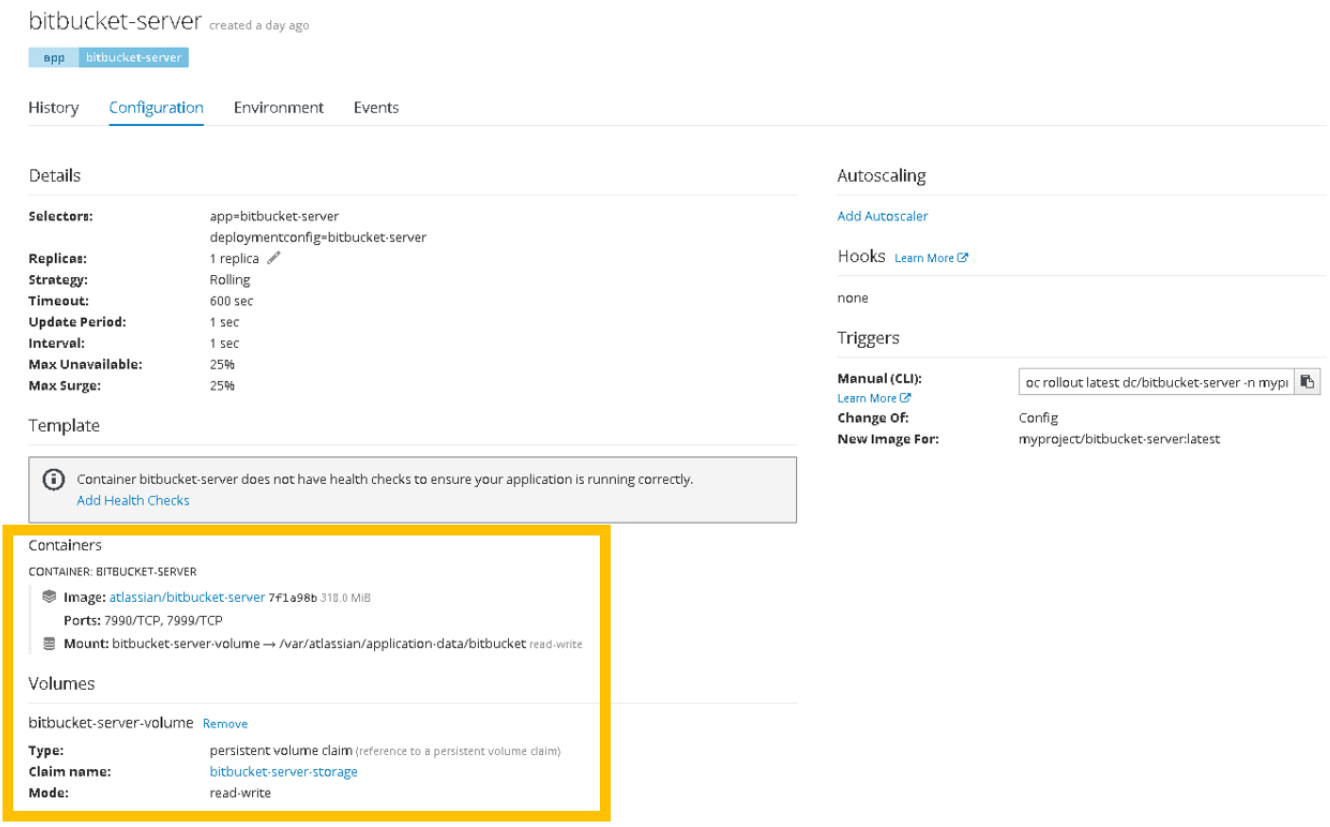


Step 3: Introduce the required information

- **Path** as it is specified in the BitBucket server Docker image (/var/atlassian/application-data/bitbucket)
- **Volume name** with a unique name to clearly identify the volume



The change will be inmediately applied



6.2. Mirabaud CICD Environment Setup

Initial requirements:

- **OS**: RHEL 6.5

Remote setup in CI machine (located in the Netherlands)

```
    - Jenkins
    - Nexus
    - GitLab
    - Mattermost
    - Atlassian Crucible
    - SonarQube
```

6.2.1. 1. Install Docker and Docker Compose in RHEL 6.5

Docker

Due to that OS version, the only way to have Docker running in the CI machine is by installing it from the **EPEL** repository (Extra Packages for Enterprise Linux).

1. Add EPEL

```
# rpm -iUvh http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
```

2. Install `docker.io` from that repository

```
# yum -y install docker-io
```

3. Start Docker daemon

```
# service docker start
```

4. Check the installation

```
# docker -v
Docker version 1.7.1, build 786b29d/1.7.1
```

Docker Compose

Download and install it via **curl**. It will use this site.

```
# curl -L https://github.com/docker/compose/releases/download/1.5.0/docker-compose-
`uname -s`-`uname -m` > /usr/local/bin/docker-compose

# chmod +x /usr/local/bin/docker-compose
```

Add it to your `sudo` path:

1. Find out where it is:

```
# echo $PATH
```

2. Copy the `docker-compose` file from `/usr/local/bin/` to your `sudo` PATH.

```
# docker-compose -v
docker-compose version 1.5.2, build 7240ff3
```

6.2.2. 2. Directories structure

Several directories had been added to organize some files related to docker (like `docker-compose.yml`) and docker volumes for each service. Here's how it looks:

```
/home
    /[username]
        /jenkins
            /volumes
                /jenkins_home
        /sonarqube
            /volumes
                /conf
                /data
                /extensions
                /lib
                    /bundled-plugins
        /nexus
            /volumes
                /nexus-data
        /crucible
            /volumes
                /
        /gitlab
            docker-compose.yml
            /volumes
                /etc
                    /gitlab
                /var
                    /log
                    /opt
        /mattermost
            docker-compose.yml
            /volumes
                /db
                    /var
                        /lib
                            /postgresql
                                /data
                /app
                    /mattermost
                        /config
                        /data
                        /logs
                /web
                    /cert
```

6.2.3. 3. CICD Services with Docker

Some naming conventions had been followed as naming containers as `mirabaud_[service]`.

Several folders have been created to store each service's volumes, `docker-compose.yml`(s), extra configuration settings and so on:

Jenkins

Command

```
# docker run -d -p 8080:8080 -p 50000:50000 --name=mirabaud_jenkins \
    -v /home/[username]/jenkins/volumes/jenkins_home:/var/jenkins_home \
    jenkins
```

Generate keystore

```
keytool -importkeystore -srckeystore server.p12 -srcstoretype pkcs12 -srcalias 1
-destkeystore newserver.jks -deststoretype jks -destalias server
```

Start jekins with SSL (TODO: make a docker-compose.yml for this):

```
sudo docker run -d --name mirabaud_jenkins -v /jenkins:/var/jenkins_home -p 8080:8443
jenkins --httpPort=-1 --httpsPort=8443
--httpsKeyStore=/var/jenkins_home/certs/keystore.jks
--httpsKeyStorePassword=Mirabaud2017
```

Volumes

```
volumes/jenkins_home:/var/jenkins_home
```

SonarQube

Command

```
# docker run -d -p 9000:9000 -p 9092:9092 --name=mirabaud_sonarqube \
    -v /home/[username]/sonarqube/volumes/conf:/opt/sonarqube/conf \
    -v /home/[username]/sonarqube/volumes/data:/opt/sonarqube/data \
    -v /home/[username]/sonarqube/volumes/extensions:/opt/sonarqube/extensions \
    -v /home/[username]/sonarqube/volumes/lib/bundled-
plugins:/opt/sonarqube//lib/bundled-plugins \
    sonarqube
```

Volumes

```
volumes/conf:/opt/sonarqube/conf
volumes/data:/opt/sonarqube/data
volumes/extensions:/opt/sonarqube/extensions
volumes/lib/bundled-plugins:/opt/sonarqube/lib/bundled-plugins
```

Nexus

Command

```
# docker run -d -p 8081:8081 --name=mirabaud_nexus\
    -v /home/[username]/nexus/nexus-data:/sonatype-work
    sonatype/nexus
```

Volumes

```
volumes/nexus-data/:/sonatype-work
```

Atlassian Crucible

Command

```
# docker run -d -p 8084:8080 --name=mirabaud_crucible \
    -v /home/[username]/crucible/volumes/data:/atlassian/data/crucible
    mswinarski/atlassian-crucible:latest
```

Volumes

```
volumes/data:/atlassian/data/crucible
```

6.2.4. 4. CICD Services with Docker Compose

Both Services had been deploying by using the `# docker-compose up -d` command from their root directories (`/gitlab` and `/mattermost`). The syntax of the two `docker-compose.yml` files is the one corresponding with the 1st version (due to the `docker-compose v1.5`).

GitLab

[[dsf-mirabaud-cicd-environment-setup.asciidoc_`docker-compose.yml`]] ==== `docker-compose.yml`

```
mirabaud:
    image: 'gitlab/gitlab-ce:latest'
    restart: always
    ports:
            - '8888:80'
    volumes:
            - '/home/[username]/gitlab/volumes/etc/gilab:/etc/gitlab'
            - '/home/[username]/gitlab/volumes/var/log:/var/log/gitlab'
            - '/home/[username]/gitlab/volumes/var/opt:/var/opt/gitlab'
```

Command (docker)

```
docker run -d -p 8888:80 --name=mirabaud_gitlab \
    -v /home/[username]/gitlab/volumes/etc/gitlab/:/etc/gitlab \
    -v /home/[username]/gitlab/volumes/var/log:/var/log/gitlab \
    -v /home/[username]/gitlab/volumes/var/opt:/var/opt/gitlab \
    gitlab/gitlab-ce
```

Volumes

```
volumes/etc/gitlab:/etc/gitlab
volumes/var/opt:/var/log/gitlab
volumes/var/log:/var/log/gitlab
```

Mattermost

[[dsf-mirabaud-cicd-environment-setup.asciidoc_`docker-compose.yml`]] ==== `docker-compose.yml`:

```
db:
  image: mattermost/mattermost-prod-db
  restart: unless-stopped
  volumes:
    - ./volumes/db/var/lib/postgresql/data:/var/lib/postgresql/data
    - /etc/localtime:/etc/localtime:ro
  environment:
    - POSTGRES_USER=mmuser
    - POSTGRES_PASSWORD=mmuser_password
    - POSTGRES_DB=mattermost

app:
  image: mattermost/mattermost-prod-app
  links:
    - db:db
  restart: unless-stopped
  volumes:
    - ./volumes/app/mattermost/config:/mattermost/config:rw
    - ./volumes/app/mattermost/data:/mattermost/data:rw
    - ./volumes/app/mattermost/logs:/mattermost/logs:rw
    - /etc/localtime:/etc/localtime:ro
  environment:
    - MM_USERNAME=mmuser
    - MM_PASSWORD=mmuser_password
    - MM_DBNAME=mattermost

web:
  image: mattermost/mattermost-prod-web
  ports:
    - "8088:80"
    - "8089:443"
  links:
    - app:app
  restart: unless-stopped
  volumes:
    - ./volumes/web/cert:/cert:ro
    - /etc/localtime:/etc/localtime:ro
```

SSL Certificate

How to generate the certificates:

Get the **crt** and **key** from CA or **generate a new one self-signed**. Then:

```
// 1. create the p12 keystore
# openssl pkcs12 -export -in cert.crt -inkey mycert.key -out certkeystore.p12

// 2. export the pem certificate with password
# openssl pkcs12 -in certkeystore.p12 -out cert.pem

// 3. export the pem certificate without password
# openssl rsa -in cert.pem -out key-no-password.pem
```

SSL:

Copy the cert and the key without password at:

`./volumes/web/cert/cert.pem`

and

`./volumes/web/cert/key-no-password.pem`

Restart the server and the SSL should be enabled at port **8089** using **HTTPS**.

Volumes

```
-- db --
volumes/db/var/lib/postgresql/data:/var/lib/postgresql/data
/etc/localtime:/etc/localtime:ro                                    # absolute path

-- app --
volumes/app/mattermost/config:/mattermost/config:rw
volumes/app/mattermost/data:/mattermost/data:rw
volumes/app/mattermost/logs:/mattermost/logs:rw
/etc/localtime:/etc/localtime:ro                                    # absolute path

-- web --
volumes/web/cert:/cert:ro
/etc/localtime:/etc/localtime:ro                                    # absolute path
```

6.2.5. 5. Service Integration

All integrations had been done following **CICD Services Integration** guides:

- Jenkins - Nexus integration
- Jenkins - GitLab integration
- Jenkins - SonarQube integration

These guides may be obsolete. You can find here the official configuration guides,

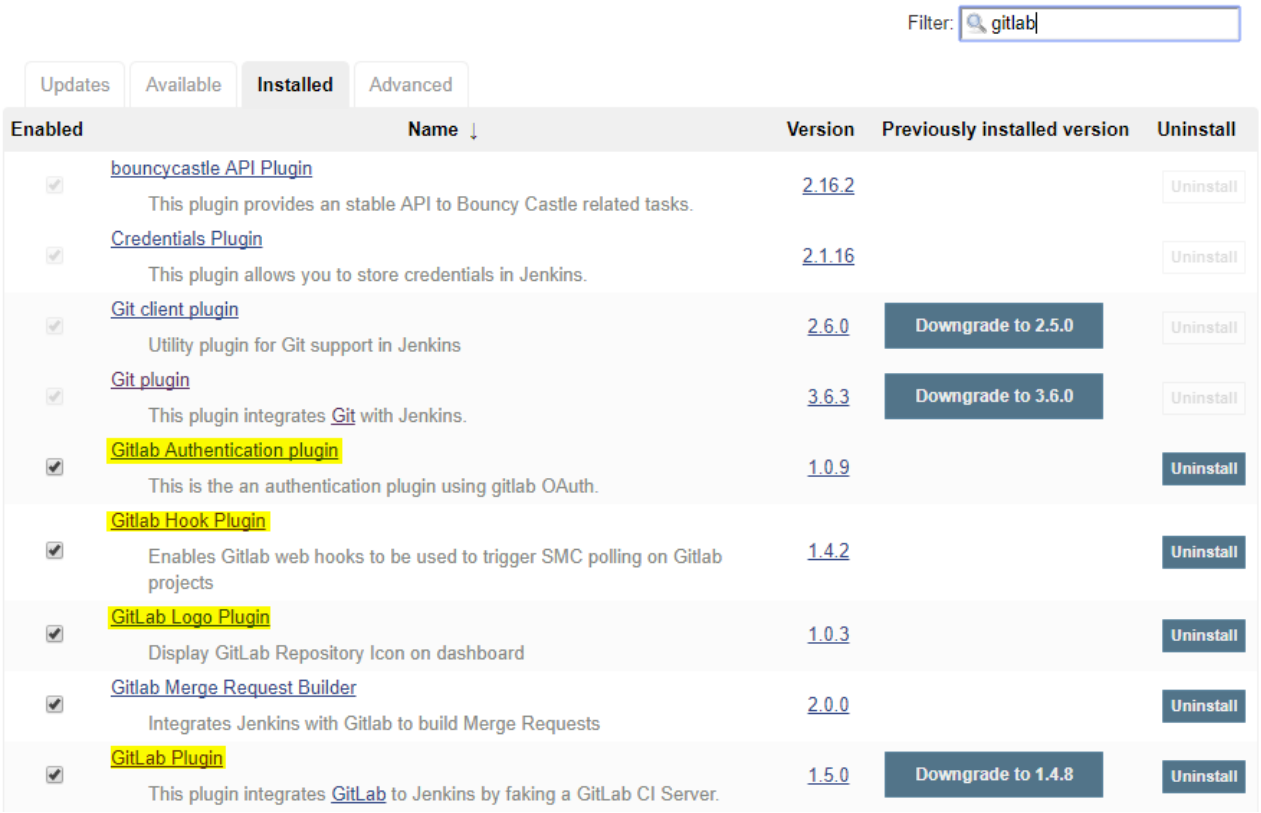
6.2.6. Jenkins - GitLab integration

The first step to have a Continuous Integration system for your development is to make sure that all your changes to your team's remote repository are evaluated by the time they are pushed. That usually implies the usage of so-called *webhooks*. You'll find a fancy explanation about what Webhooks are in here.

To resume what we're doing here, we are going to prepare our Jenkins and our GitLab so when a developer pushes some changes to the GitLab repository, a pipeline in Jenkins gets triggered. Just like that, in an automatic way.

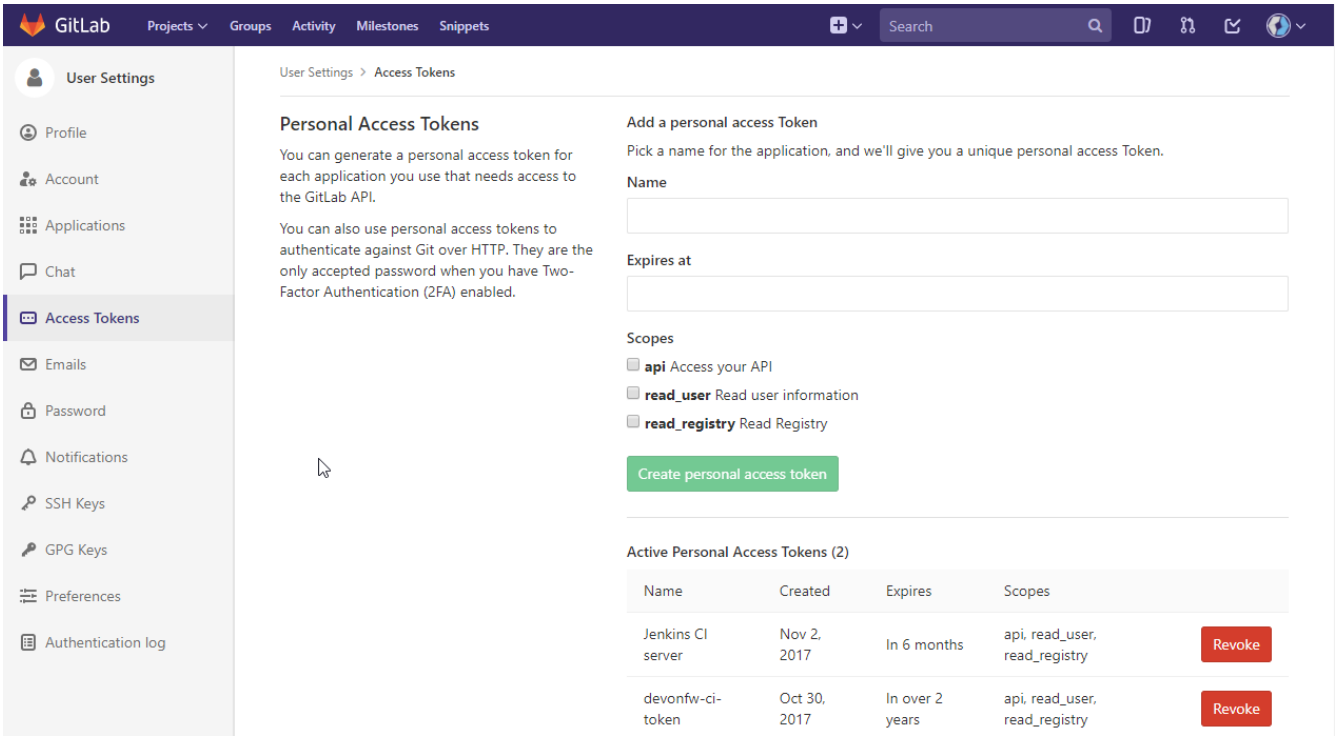
1. Jenkins GitLab plugin

As it usually happens, some Jenkins plug-in(s) must be installed. In this case, let's install those related with GitLab:

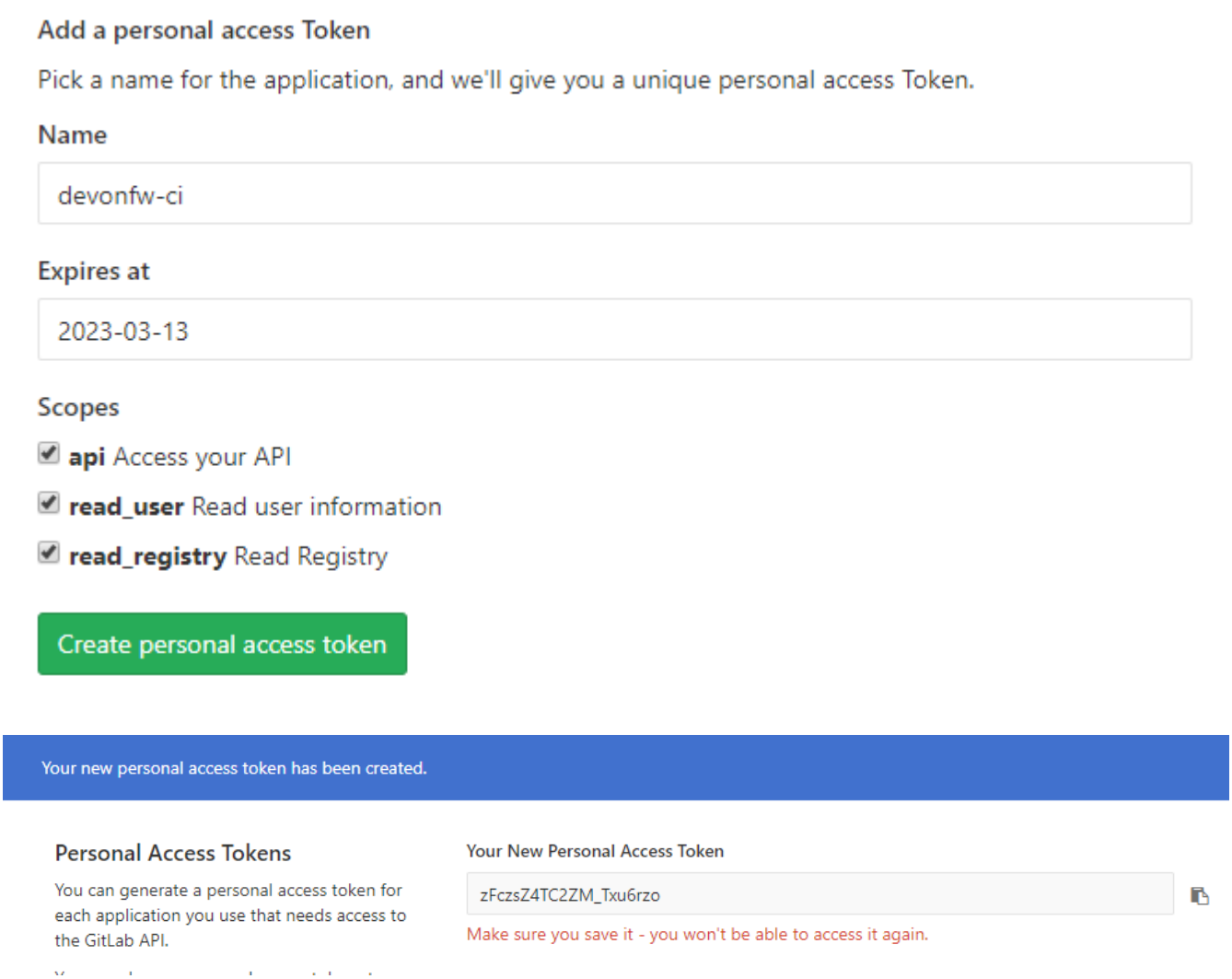


2. GitLab API Token

To communicate with GitLab from Jenkins, we will need to create an authentication token from your GitLab user settings. A good practice for this would be to create it from a *machine user*. Something like (i.e.) `devonfw-ci/******`.



Simply by adding a name to it and a date for it expire is enough:

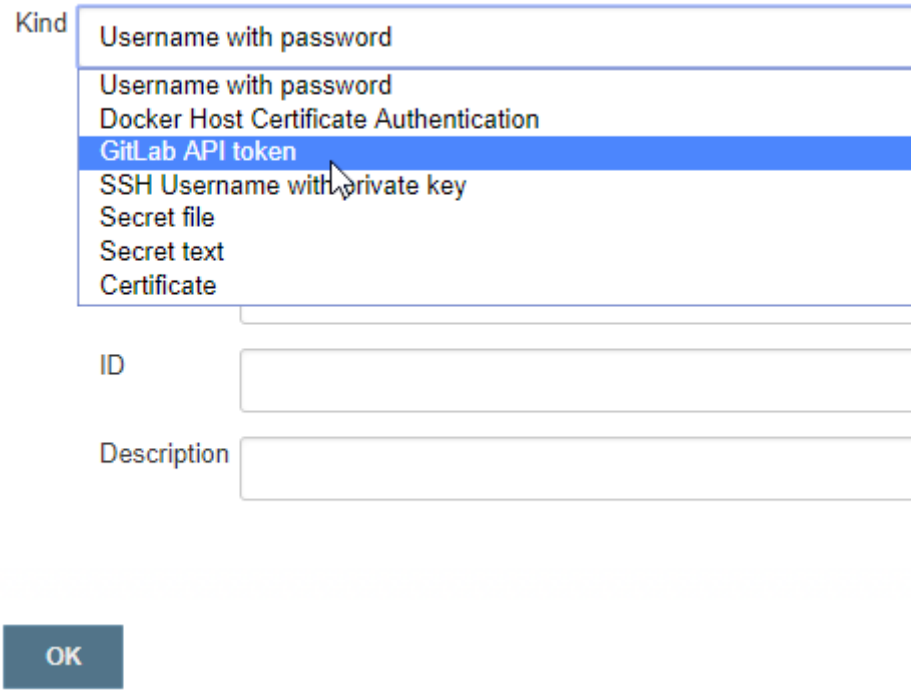


As GitLab saids, you should make sure you don't lose your token. Otherwise you would need to create a new one.

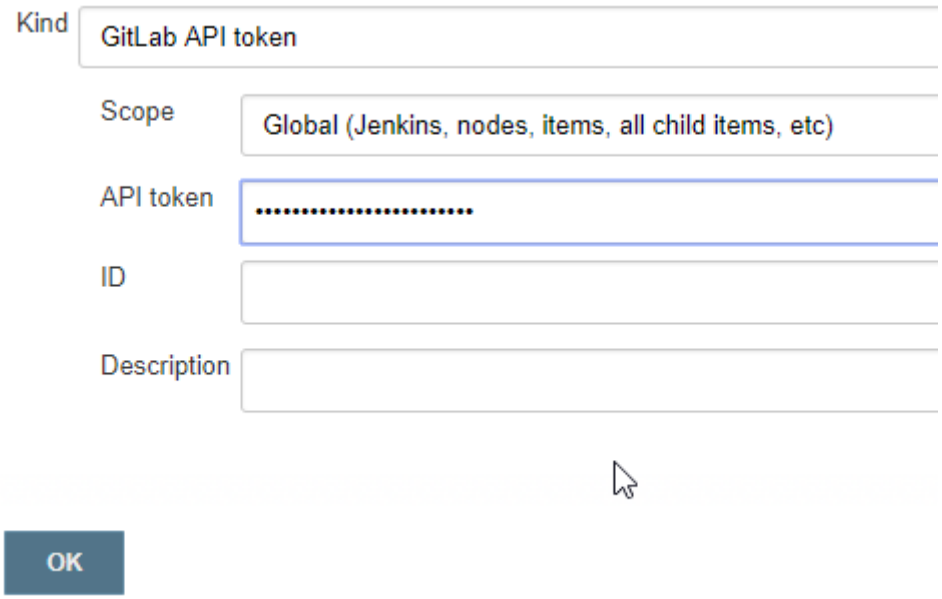
This will allow Jenkins to connect with right permissions to our GitLab server.

[[dsf-mirabaud-jenkins-gitlab-integration.asciidoc_3.-create-"gitlab-api"-token-credentials]] == 3. Create "GitLab API" Token credentials

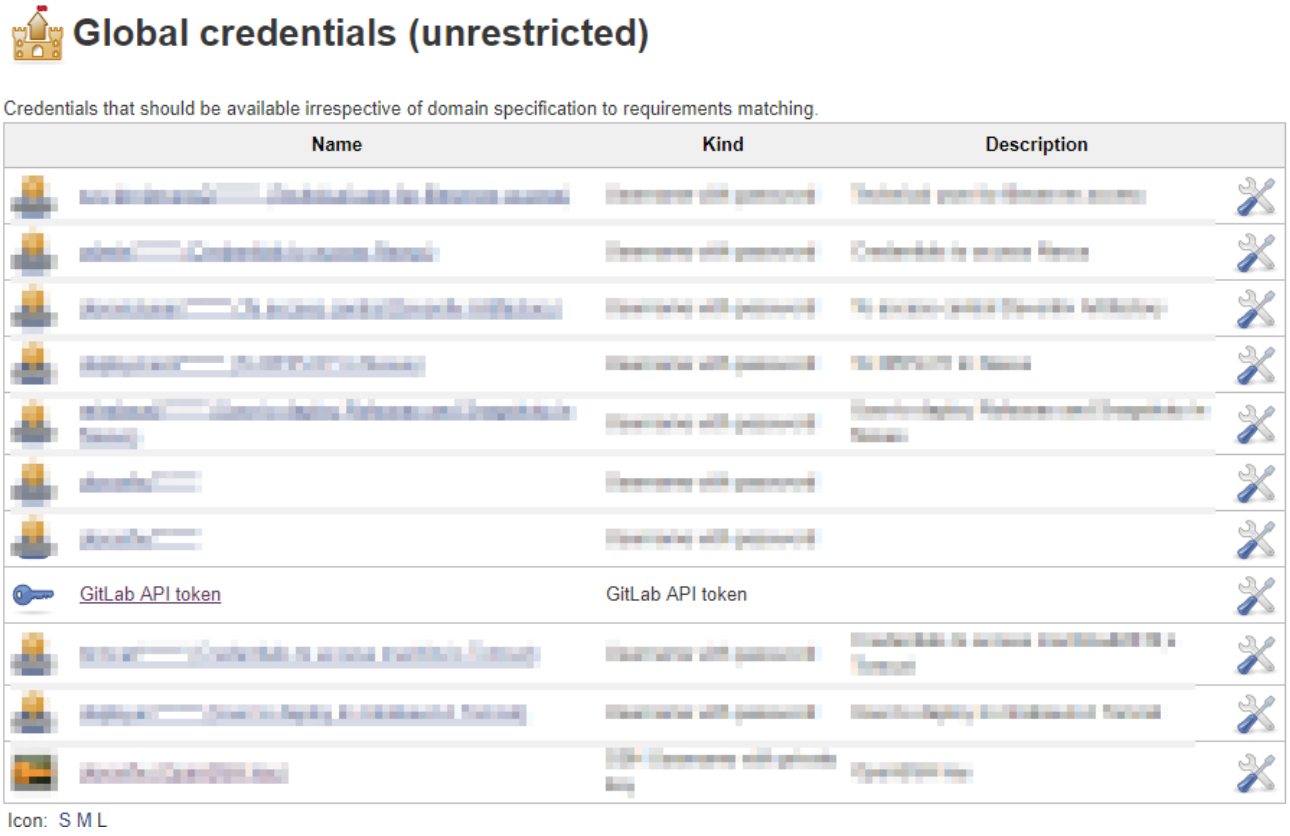
Thos credentials will use that token already generated in GitLab to connect once we declare the GitLab server in the Global Jenkins configuration. Obviously, those credentials must be **GitLab API token**-like.



Then, we add the generated token in the `API token` field:



Look in your Global credentials if they had been correctly created:



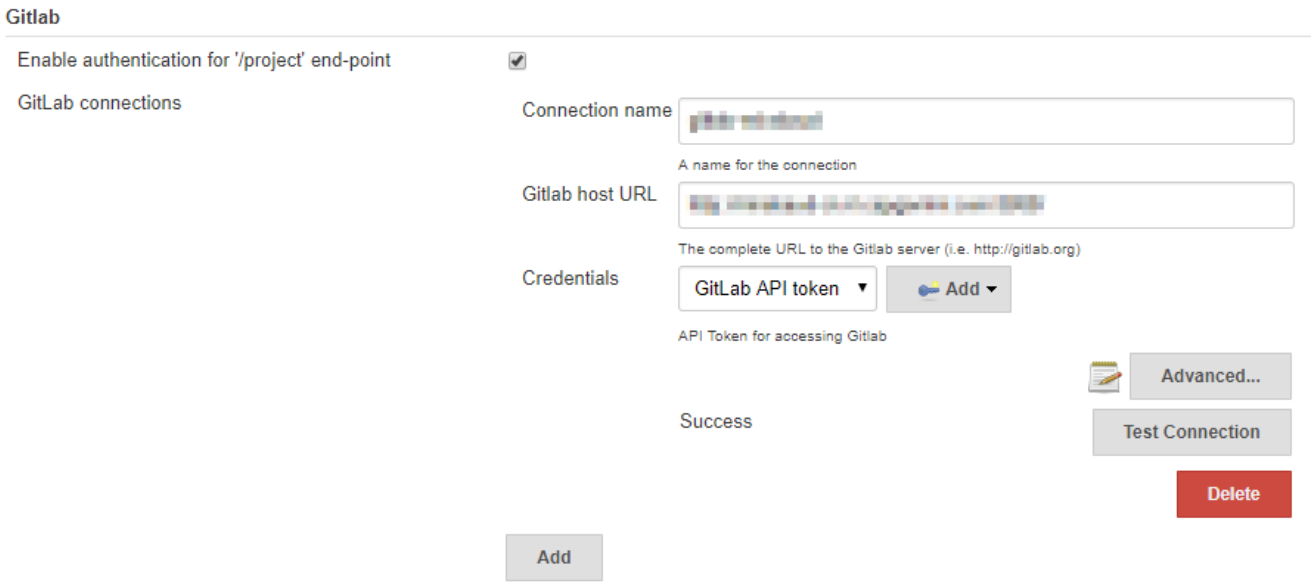
4. Create GitLab connection in Jenkins

Specify a GitLab connection in your Jenkins's `Manage Jenkins > Configure System` configuration. This will tell Jenkins where is our GitLab server, a user to access it from and so on.

You'll need to give it a name, for example, related with what this GitLab is dedicated for (specific clients, internal projects...). Then, the `Gitlab host URL` is just where your GitLab server is. If you have it locally, that field should look similar to:

- Connection name: `my-local-gitlab`
- Gitlab host URL: `http://localhost:[dsf-mirabaud-jenkins-gitLab-integration.asciidoc_PORT_NUMBER]`

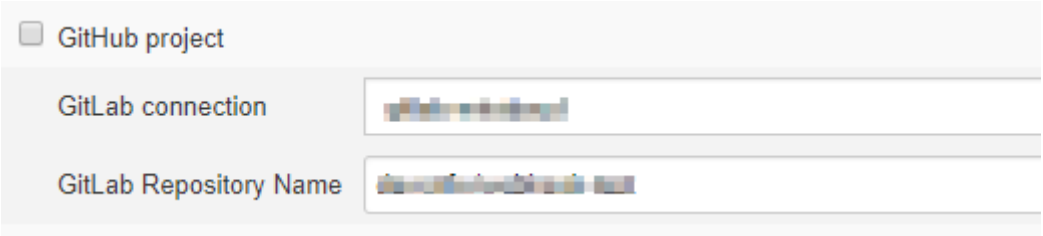
Finally, we select our recently GitLab API token as credentials.



5. Jenkins Pipeline changes

5.1 Choose GitLab connection in Pipeline's General configuration

First, our pipeline should allow us to add a GitLab connection to connect to (the already created one).

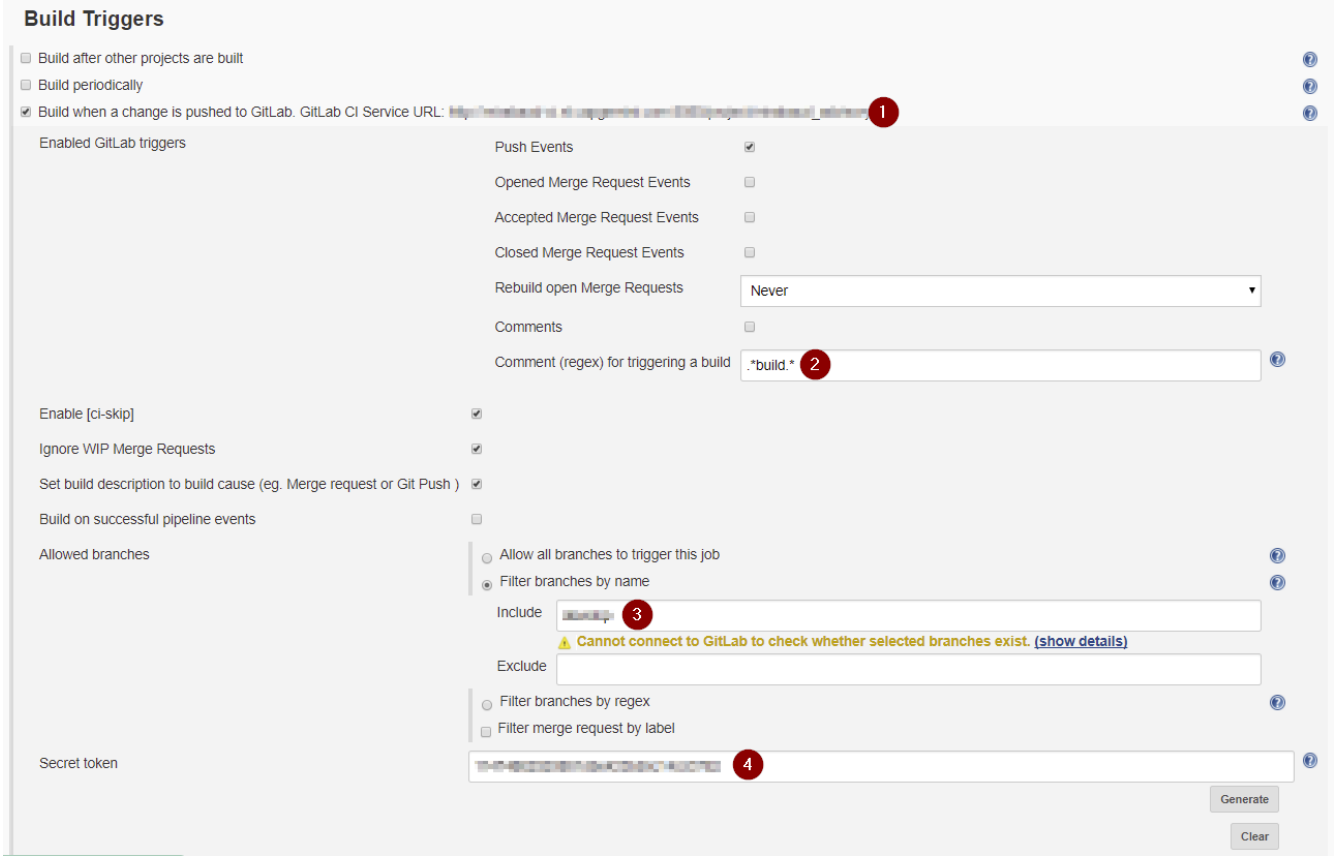


In the case of the local example, could be like this:

- GitLab connection: `my-local-gitlab`
- GitLab Repository Name: `myusername/webhook-test` (for example)

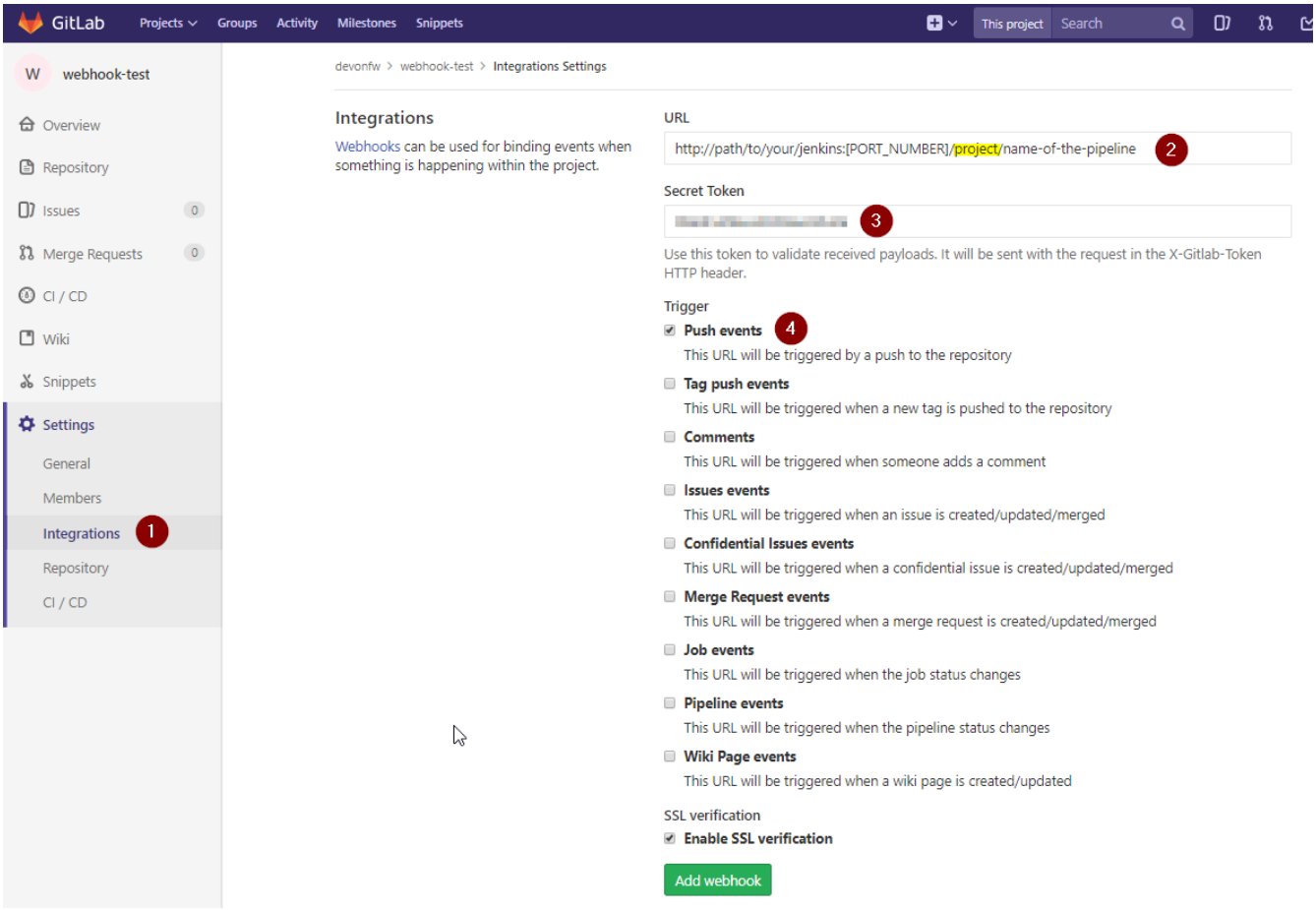
5.2 Create a Build Trigger

1. You should already see your GitLab project's URL (as you stated in the General settings of the Pipeline).
2. Write `.*build.*` in the comment for triggering a build
3. Specify or filter the branch of your repo you want use as target. That means, whenever a git action is done to that branch (for exapmle, `master`), this Pipeline is going to be built.
4. Generate a Secret token (to be added in the yet-to-be-created GitLab webhook).



6. GitLab Webhook

1. Go to you GitLab project's `Settings > Integration` section.
2. Add the path to your Jenkins Pipeline. Make sure you add **project** instead of **job** in the path.
3. Paste the generated Secret token of your Jenkins pipeline
4. Select your git action that will trigger the build.



7. Results

After all those steps you shoul have a result similar to this in your Pipeline:

Enjoy the Continuous Integration! :)

6.2.7. Jenkins - Nexus integration

Nexus is used to both host dependencies for devonfw projects to download (common Maven ones, custom ones such as `ojdb` and even devonfw so-far-IP modules). Moreover, it will host our projects' build artifacts (`.jar`, `.war`, ...) and expose them for us to download, wget and so on. A team should have a bidirectional relation with its Nexus repository.

1. Jenkins credentials to access Nexus

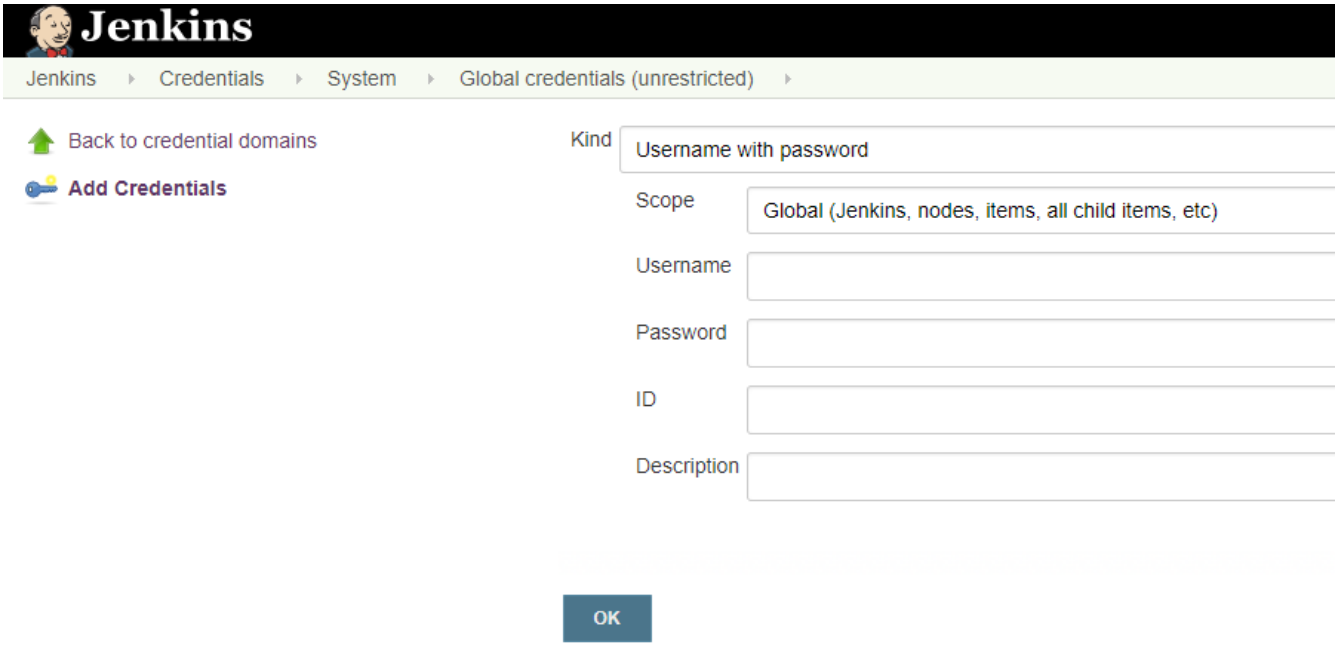
By default, when Nexus is installed, it contains 3 user credentials for different purposes. The admin ones look like this: `admin/admin123`. There are also other 2: `deployment/deployment123` and `TODO`.

```
// ADD USER TABLE IMAGE FROM NEXUS
```

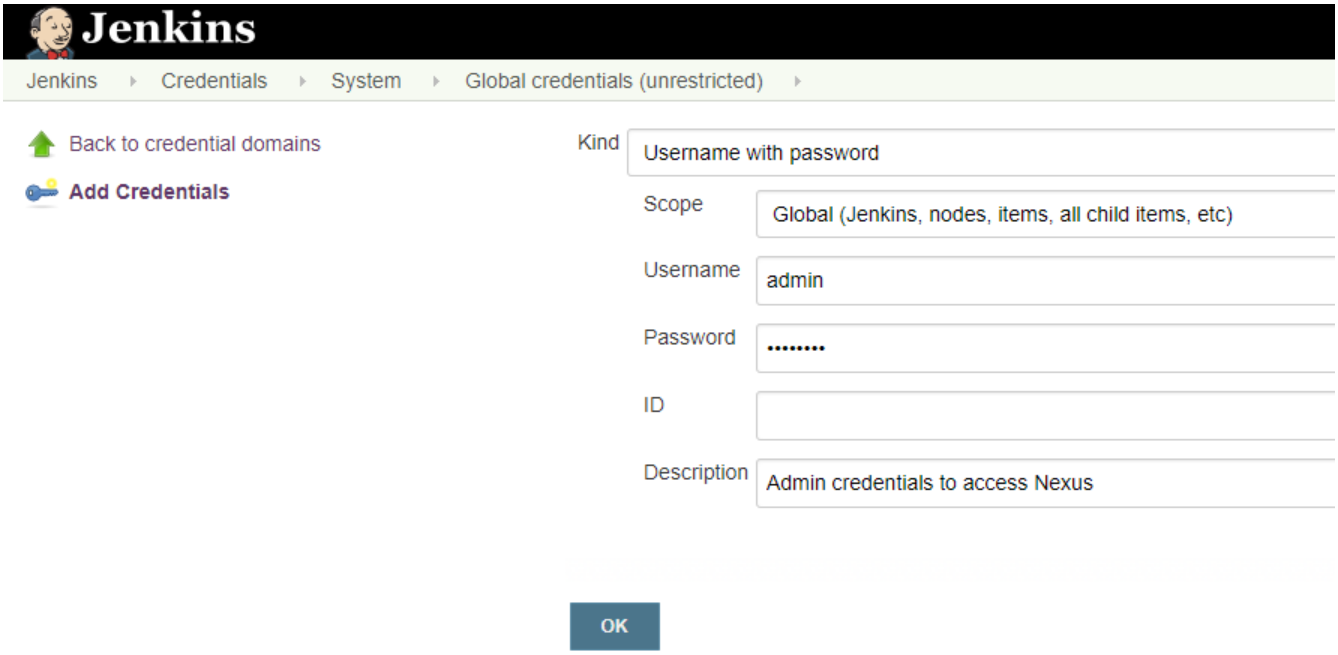
In this case, let's use the ones with the greater permissions: `admin/admin123`.

Go to `Credentials > System` (left sidebar of Jenkins) then to `Global credentials (unrestricted)` on the page table and on the left sidebar again click on `Add Credentials`.

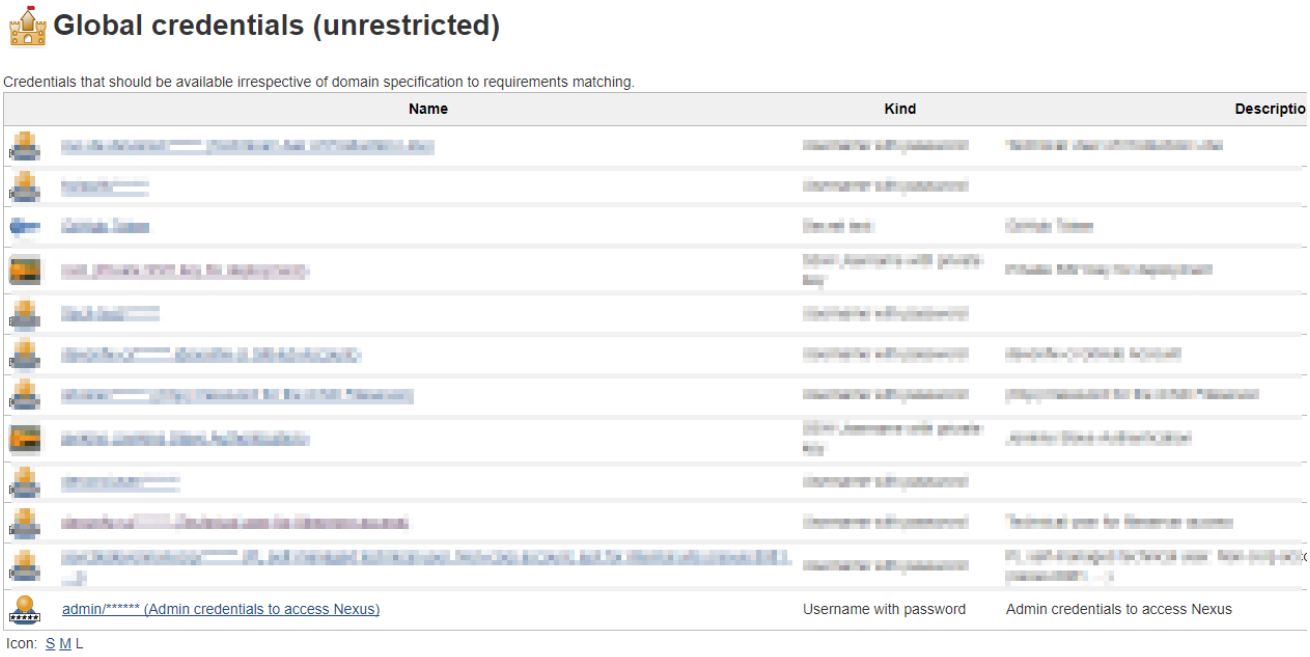
This should be shown in your Jenkins:



Fill the form like this:



And click in OK to create them. Check if the whole thing went as expected:

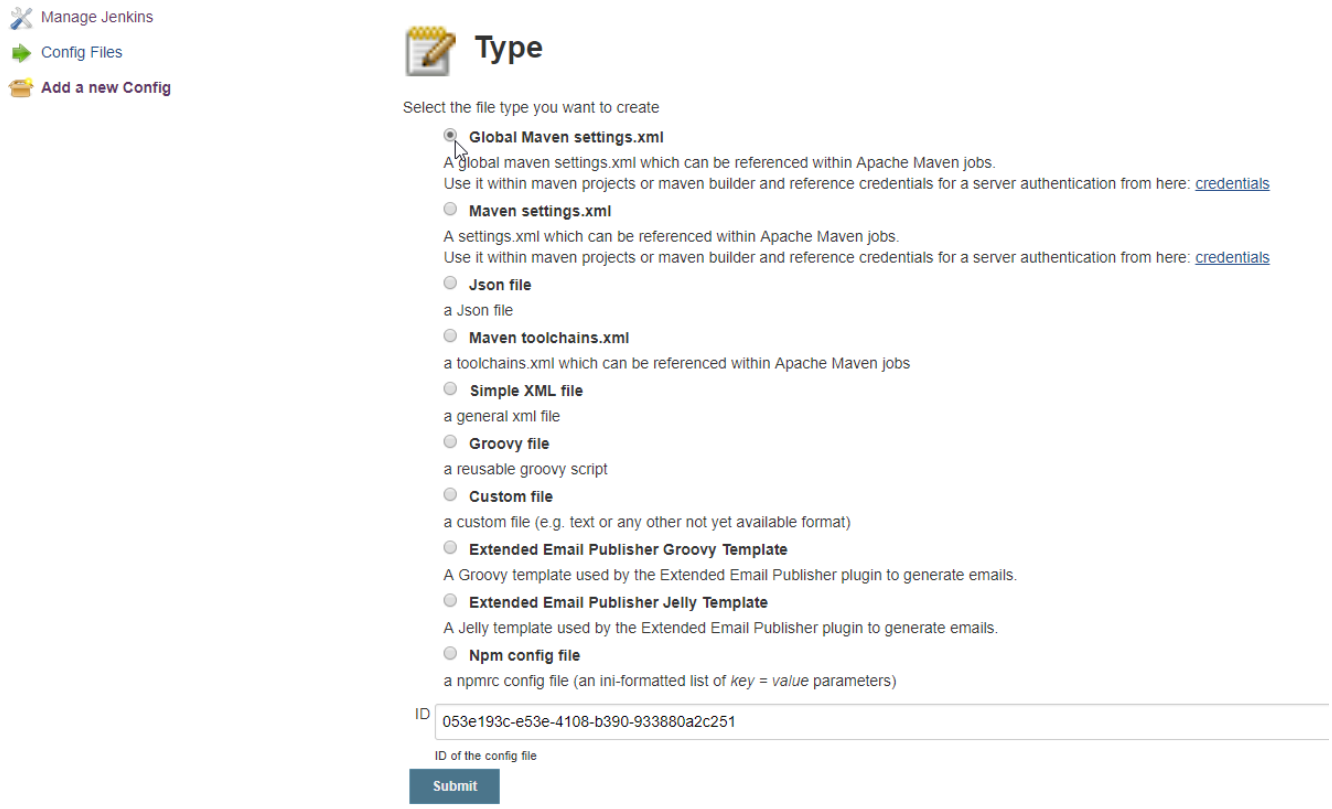


2. Jenkins Maven Settings

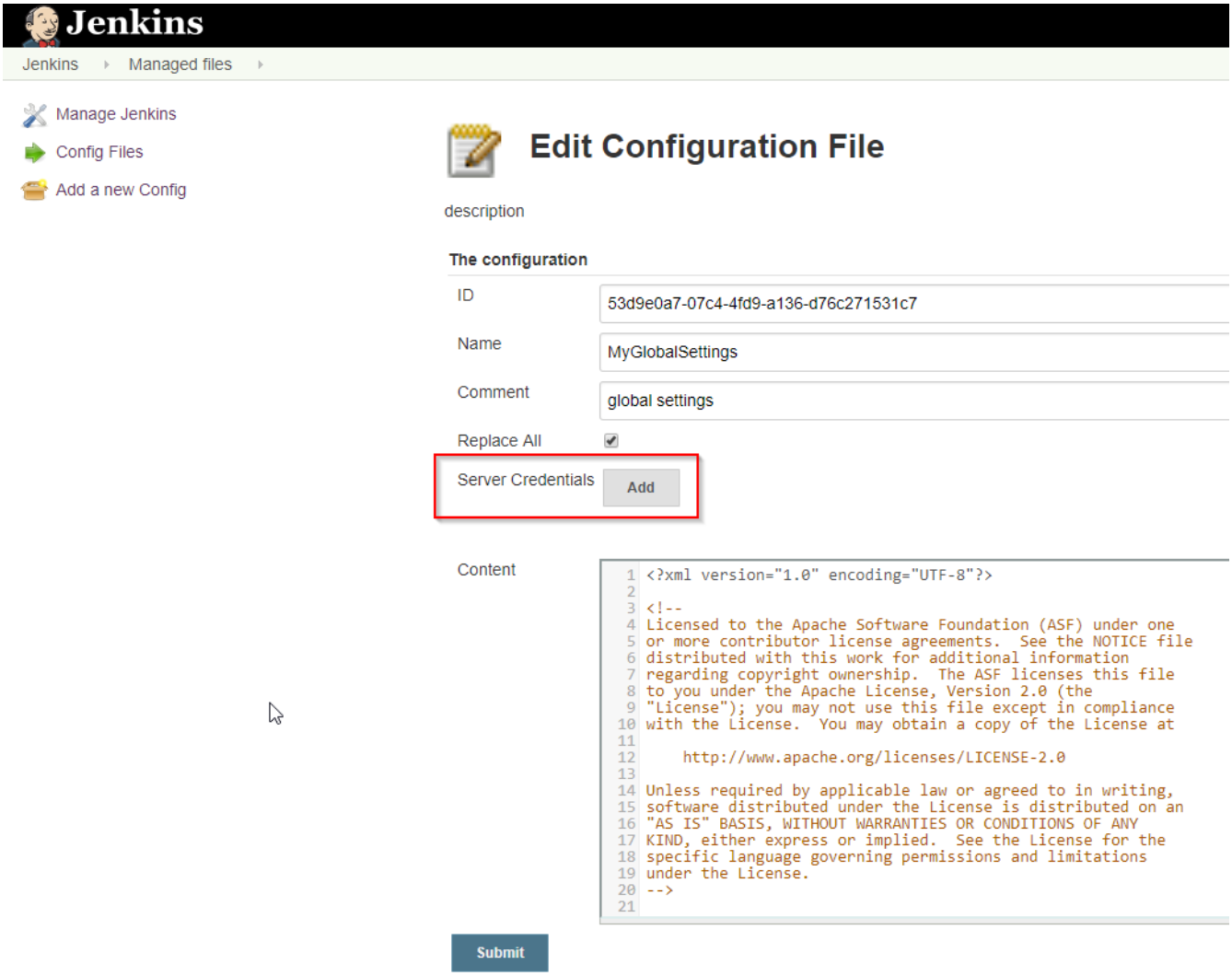
Those settings are also configured (or maybe not-yet-configured) in our **devonfw distributions** in:

```
/${devonfw-dist-path}
    /software
        /maven
            /conf
                settings.xml
```

Go to `Manage Jenkins > Managed files` and select `Add a new Config` in the left sidebar.



The ID field will get automatically filled with a unique value if you don't set it up. No problems about that. Click on `Submit` and let's create some Servers Credentials:



Those **Server Credentials** will allow Jenkins to access to the different repositories/servers that are going to be declared afterwards.

Let's create 4 server credentials.

- `my.nexus`: Will serve as general profile for **Maven**.
- `mynexus.releases`: When a `mvn deploy` process is executed, this will tell **Maven** where to push **releases** to.
- `mynexus.snapshots`: The same as before, but with **snapshots** instead.
- `mynexus.central`: Just in case we want to install an specific dependency that is not by default in the Maven Central repository (such as `ojdbc`), Maven will point to it instead.

A more or less complete Jenkins Maven settings would look look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">

    <mirrors>
        <mirror>
            <id>mynexus.central</id>
            <mirrorOf>central</mirrorOf>
            <name>central</name>
            <url>http://${URL-TO-YOUR-NEXUS-REPOS}/central</url>
        </mirror>
    </mirrors>

    <profiles>
        <profile>
            <id>my.nexus</id>
            <!-- 3 REPOS ARE DECLARED -->
            <repositories>
                <repository>
                    <id>mynexus.releases</id>
                    <name>mynexus Releases</name>
                    <url>http://${URL-TO-YOUR-NEXUS-REPOS}/releases</url>
                    <releases>
                        <enabled>true</enabled>
                        <updatePolicy>always</updatePolicy>
                    </releases>
                    <snapshots>
                        <enabled>false</enabled>
                        <updatePolicy>always</updatePolicy>
```

```
                    </snapshots>
                </repository>
                <repository>
                    <id>mynexus.snapshots</id>
                    <name>mynexus Snapshots</name>
                    <url>http://${URL-TO-YOUR-NEXUS-REPOS}/snapshots</url>
                    <releases>
                        <enabled>false</enabled>
                        <updatePolicy>always</updatePolicy>
                    </releases>
                    <snapshots>
                        <enabled>true</enabled>
                        <updatePolicy>always</updatePolicy>
                    </snapshots>
                </repository>
            </repositories>
            <pluginRepositories>
                <pluginRepository>
                    <id>public</id>
                    <name>Public Repositories</name>
                    <url>http://${URL-TO-YOUR-
NEXUS}/nexus/content/groups/public/</url>
                    <releases>
                        <enabled>true</enabled>
                        <updatePolicy>always</updatePolicy>
                    </releases>
                    <snapshots>
                        <enabled>true</enabled>
                        <updatePolicy>always</updatePolicy>
                    </snapshots>
                </pluginRepository>
            </pluginRepositories>
        </profile>
    </profiles>
    <!-- HERE IS WHERE WE TELL MAVEN TO CHOOSE THE my.nexus PROFILE -->
    <activeProfiles>
        <activeProfile>my.nexus</activeProfile>
    </activeProfiles>
</settings>
```

3. Use it in Jenkins Pipelines

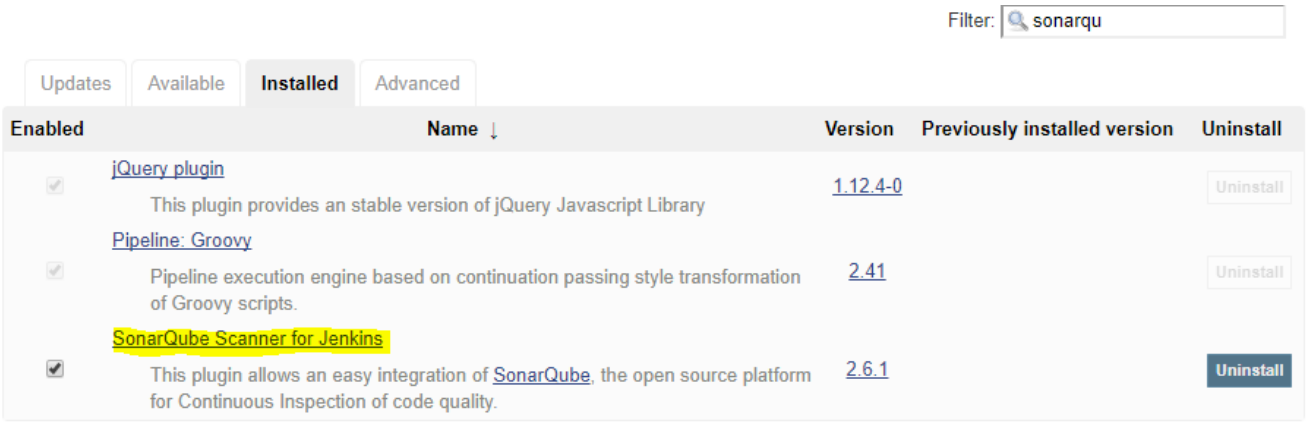
6.2.8. Jenkins - SonarQube integration

First thing is installing both tools by, for example, Docker or Docker Compose. Then, we have to think about how they should collaborate to create a more efficient Continuous Integration process.

Once our project's pipeline is triggered (it could also be triggered in a fancy way, such as when a merge to the `develop` branch is done).

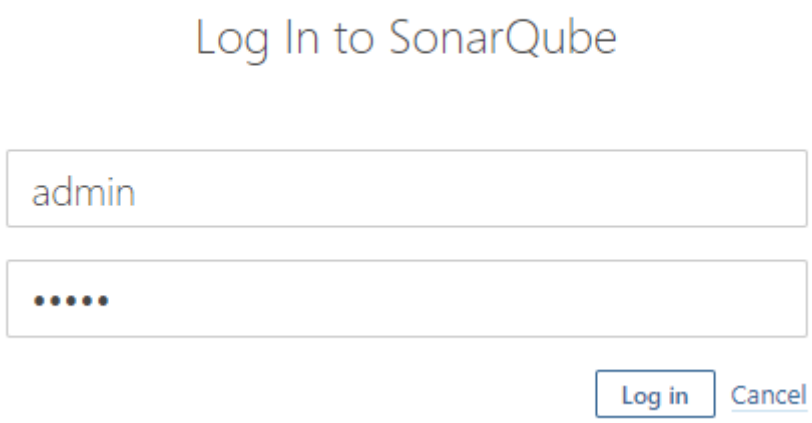
1. Jenkins SonarQube plugin

Typically in those integration cases, Jenkins plug-in installations become a **must**. Let's look for some available SonarQube plug-in(s) for Jenkins:

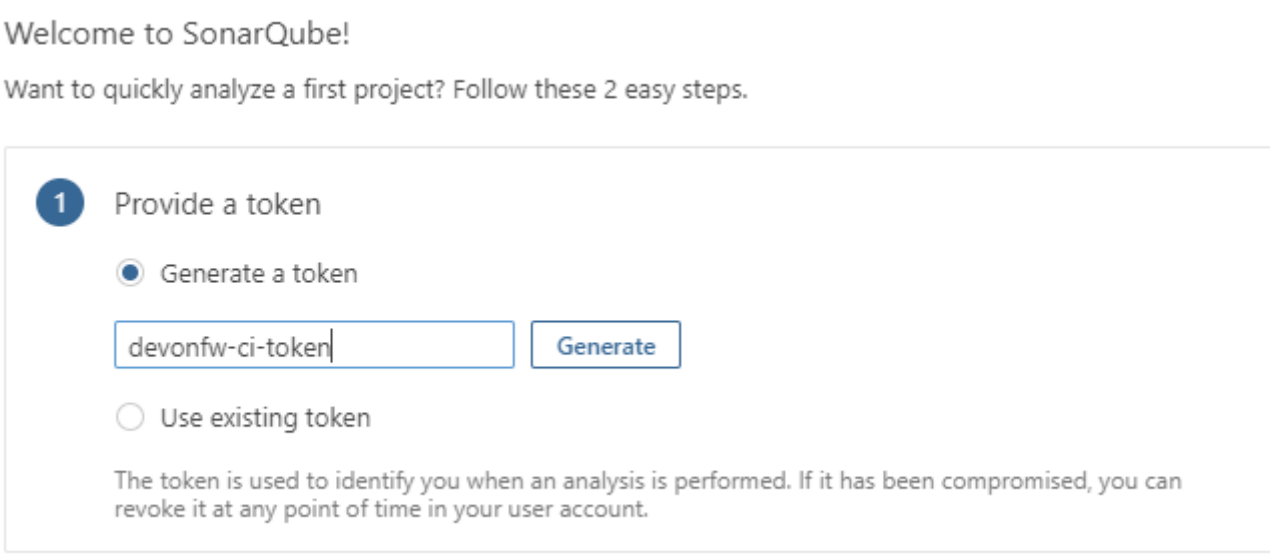


2. SonarQube token

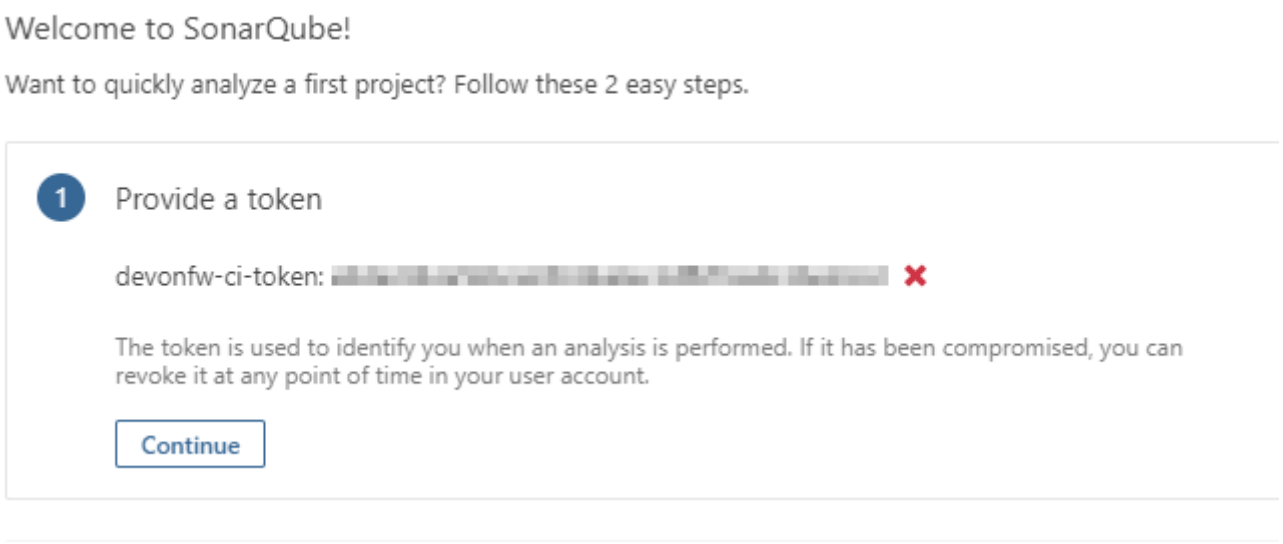
Once installed let's create a **token** in SonarQube so that Jenkins can communicate with it to trigger their Jobs. Once we install SonarQube in our CI/CD machine (ideally a remote machine) let's login with `admin/admin` credentials:



Afterwards, SonarQube itself asks you to create this token we talked about (the name is up to you):



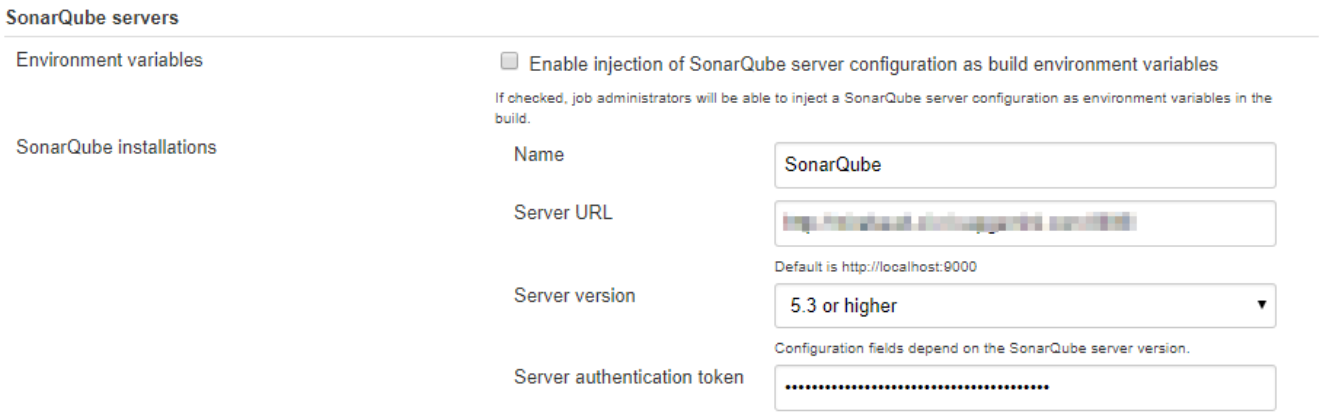
Then a token is generated:



You click in "continue" and the token's generation is completed:

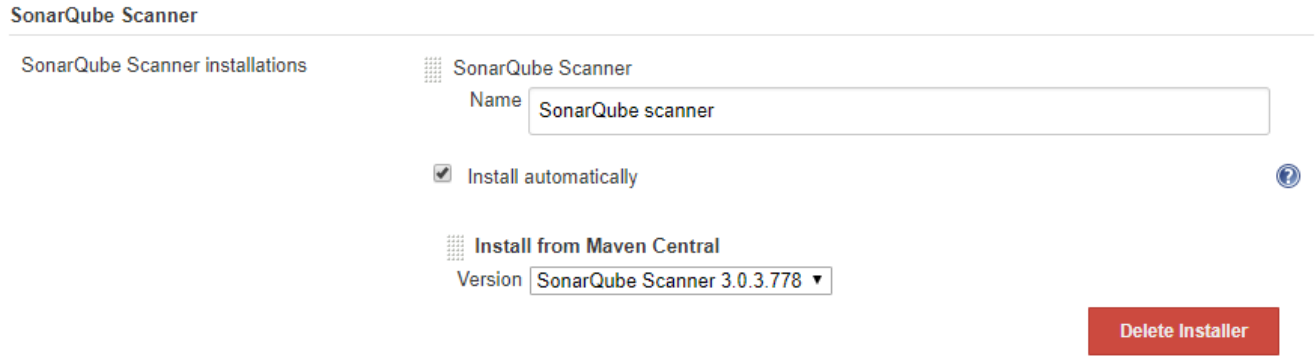
3. Jenkins SonarQube Server setup

Now we need to tell Jenkins where is SonarQube and how to communicate with it. In `Manage Jenkins > Configure Settings`. We add a name for the server (up to you), where it is located (URL), version and the Server authentication token created in point 2.



4. Jenkins SonarQube Scanner

Install a SonarQube Scanner as a Global tool in Jenkins to be used in the project's pipeline.



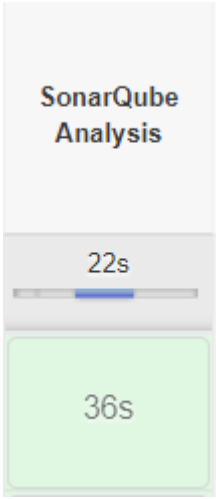
5. Pipeline code

Last step is to add the SonarQube process in our project's Jenkins pipeline. The following code will trigger a SonarQube process that will evaluate our code's quality looking for bugs, duplications, and so on.

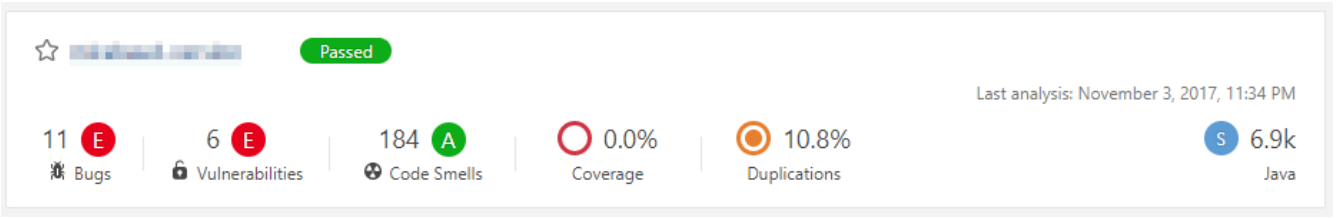
```
    stage 'SonarQube Analysis'
        def scannerHome = tool 'SonarQube scanner';
        sh "${scannerHome}/bin/sonar-scanner \
            -Dsonar.host.url=http://url-to-your-sq-server:9000/ \
            -Dsonar.login=[SONAR_USER] -Dsonar.password=[SONAR_PASS] \
            -Dsonar.projectKey=[PROJECT_KEY] \
            -Dsonar.projectName=[PROJECT_NAME]
-Dsonar.projectVersion=[PROJECT_VERSION] \
            -Dsonar.sources=. -Dsonar.java.binaries=. \
            -Dsonar.java.source=1.8 -Dsonar.language=java"
```

6. Results

After all this, you should end up having something like this in Jenkins:



And in SonarQube:



7. Changes in a devonfw project to execute SonarQube tests with Coverage

The plugin used to have Coverage reports in the SonarQube for devonfw projects is **Jacoco**. There are some changes in the project's parent `pom.xml` that are mandatory to use it.

Inside of the `<properties>` tag:

```
<properties>

    (...)

    <sonar.jacoco.version>3.8</sonar.jacoco.version>
    <sonar.java.coveragePlugin>jacoco</sonar.java.coveragePlugin>
    <sonar.core.codeCoveragePlugin>jacoco</sonar.core.codeCoveragePlugin>
    <sonar.dynamicAnalysis>reuseReports</sonar.dynamicAnalysis>
    <sonar.language>java</sonar.language>
    <sonar.java.source>1.7</sonar.java.source>
    <sonar.junit.reportPaths>target/surefire-reports</sonar.junit.reportPaths>
    <sonar.jacoco.reportPaths>target/jacoco.exec</sonar.jacoco.reportPaths>
    <sonar.sourceEncoding>UTF-8</sonar.sourceEncoding>
    <sonar.exclusions>
        **/generated-sources/**/*,
        **io/oasp/mirabaud/general/**/*,
        **/*Dao.java,
        **/*Entity.java,
        **/*Cto.java,
        **/*Eto.java,
        **/*SearchCriteriaTo.java,
        **/*management.java,
        **/*SpringBootApp.java,
        **/*SpringBootBatchApp.java,
        **/*.xml,
        **/*.jsp
    </sonar.exclusions>
    <sonar.coverage.exclusions>
        **io/oasp/mirabaud/general/**/*,
        **/*Dao.java,
        **/*Entity.java,
        **/*Cto.java,
        **/*Eto.java,
        **/*SearchCriteriaTo.java,
        **/*management.java,
        **/*SpringBootApp.java,
        **/*SpringBootBatchApp.java,
        **/*.xml,
        **/*.jsp
    </sonar.coverage.exclusions>
    <sonar.host.url>http://${YOUR_SONAR_SERVER_URL}/</sonar.host.url>
    <jacoco.version>0.7.9</jacoco.version>

    <war.plugin.version>3.2.0</war.plugin.version>
    <assembly.plugin.version>3.1.0</assembly.plugin.version>
</properties>
```

Of course, those `sonar` amd `sonar.coverage` can/must be changed to fit with other projects.

Now add the **Jacoco Listener** as a dependency:

```
<dependencies>
    <dependency>
        <groupId>org.sonarsource.java</groupId>
        <artifactId>sonar-jacoco-listeners</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Plugin Management declarations:

```
<pluginManagement>
    <plugins>
        <plugin>
            <groupId>org.sonarsource.scanner.maven</groupId>
            <artifactId>sonar-maven-plugin</artifactId>
            <version>3.2</version>
        </plugin>
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>${jacoco.version}</version>
        </plugin>
    </plugins>
<pluginManagement>
```

Plugins:

```
<plugins>

    (...)

    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.20.1</version>
        <configuration>
            <argLine>-XX:-UseSplitVerifier -Xmx2048m ${surefireArgLine}</argLine>
            <testFailureIgnore>false</testFailureIgnore>
            <useFile>false</useFile>
            <reportsDirectory>
${project.basedir}/${sonar.junit.reportPaths}</reportsDirectory>
            <argLine>${jacoco.agent.argLine}</argLine>
            <excludedGroups>${oasp.test.excluded.groups}</excludedGroups>
            <alwaysGenerateSurefireReport>true</alwaysGenerateSurefireReport>
            <aggregate>true</aggregate>
            <properties>
                <property>
                    <name>listener</name>
```

```
                    <value>org.sonar.java.jacoco.JUnitListener</value>
                </property>
            </properties>
        </configuration>
    </plugin>
    <plugin>
        <groupId>org.jacoco</groupId>
        <artifactId>jacoco-maven-plugin</artifactId>
        <configuration>
            <argLine>-Xmx128m</argLine>
            <append>true</append>
            <propertyName>jacoco.agent.argLine</propertyName>
            <destFile>${sonar.jacoco.reportPath}</destFile>
            <excludes>
                <exclude>**/generated-sources/**/*,</exclude>
                <exclude>**io/oasp/${PROJECT_NAME}/general/**/*</exclude>
                <exclude>**/*Dao.java</exclude>
                <exclude>**/*Entity.java</exclude>
                <exclude>**/*Cto.java</exclude>
                <exclude>**/*Eto.java</exclude>
                <exclude>**/*SearchCriteriaTo.java</exclude>
                <exclude>**/*management.java</exclude>
                <exclude>**/*SpringBootApp.java</exclude>
                <exclude>**/*SpringBootBatchApp.java</exclude>
                <exclude>**/*.class</exclude>
            </excludes>
        </configuration>
        <executions>
            <execution>
                <id>prepare-agent</id>
                <phase>initialize</phase>
                <goals>
                    <goal>prepare-agent</goal>
                </goals>
                <configuration>
                    <destFile>${sonar.jacoco.reportPath}</destFile>
                    <append>true</append>
                </configuration>
            </execution>
            <execution>
                <id>report-aggregate</id>
                <phase>verify</phase>
                <goals>
                    <goal>report-aggregate</goal>
                </goals>
            </execution>
            <execution>
                <id>jacoco-site</id>
                <phase>verify</phase>
                <goals>
                    <goal>report</goal>
```

```
                </goals>
            </execution>
        </executions>
    </plugin>
</plugins>
```

Jenkins SonarQube execution

If the previous configuration is already setup, once Jenkins execute the sonar maven plugin, it will automatically execute coverage as well.

This is an example of a block of code from a devonfw project's `Jenkinsfile`:

```
    withMaven(globalMavenSettingsConfig: 'YOUR_GLOBAL_MAVEN_SETTINGS', jdk: 'OpenJDK
1.8', maven: 'Maven_3.3.9') {
        sh "mvn sonar:sonar -Dsonar.login=[USERNAME] -Dsonar.password=[PASSWORD]"
    }
```

6.3. OKD *(OpenShift Origin)*

6.3.1. What is OKD

OKD is a distribution of Kubernetes optimized for continuous application development and multi-tenant deployment. OKD is the upstream Kubernetes distribution embedded in Red Hat OpenShift.

OKD embeds Kubernetes and extends it with security and other integrated concepts. OKD is also referred to as Origin in github and in the documentation.

OKD provides a complete open source container application platform. If you are looking for enterprise-level support, or information on partner certification, Red Hat also offers Red Hat OpenShift Container Platform.

Continue reading...

- How to install Openshift Origin
- Initial setup
  - s2i
  - templates
  - Customize Openshift
    - Customize icons
    - Customize catalog

6.3.2. Install OKD *(Openshift Origin)*

Pre-requisites

Install docker

https://docs.docker.com/engine/installation/linux/docker-ce/debian/#set-up-the-repository

```
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
```

Download Openshift Origin Client

Download Openshift Origin Client from here

When the download it's complete, only extract it on the directory that you want, for example `/home/administrador/oc`

Add oc to path

```
$ export PATH=$PATH:/home/administrador/oc
```

Install Openshift Cluster

Add the insecure registry

Create file `/etc/docker/daemon.json` with the next content:

```
{
    "insecure-registries" : [ "172.30.0.0/16" ]
}
```

Download docker images for openshift

```
$ oc cluster up
```

Install Oc Cluster Wrapper

To manage easier the cluster persistent, we are going to use oc cluster wrapper.

```
cd /home/administrador/oc
wget https://raw.githubusercontent.com/openshift-evangelists/oc-cluster-
wrapper/master/oc-cluster
```

oc-cluster up devonfw-shop-floor --public-hostname X.X.X.X

Configure iptables

We must create iptables rules to allow traffic from other machines.

```
- The next commands it's to let all traffic, don't do it on a real server.

- $ iptables -F
- $ iptables -X
- $ iptables -t nat -F
- $ iptables -t nat -X
- $ iptables -t mangle -F
- $ iptables -t mangle -X
- $ iptables -P INPUT ACCEPT
- $ iptables -P OUTPUT ACCEPT
- $ iptables -P FORWARD ACCEPT
```

6.3.3. How to use Oc Cluster Wrapper

With oc cluster wrapper we could have a different clusters with different context.

Cluster up

```
$ oc-cluster up devonfw-shop-floor --public-hostname X.X.X.X
```

Cluster down

```
$ oc-cluster down
```

Use non-persistent cluster

```
oc cluster up --image openshift/origin --public-hostname X.X.X.X --routing-suffix
apps.X.X.X.X.nip.io
```

6.3.4. devonfw Openshift Origin Initial Setup

This is a scripts to customize an Openshift cluster to be a devonfw Openshift.

How to use

Prerequisite: Customize Openshift

devonfw Openshift Origin use custom icons, and we need to add it to openshift. More information:

- Customize Openshift

Script initial-setup

Download this script and execute it.

More information about what this script does here.

Known issues

Failed to push image

If you recive an error like this:

```
error: build error: Failed to push image: After retrying 6 times, Push image still
failed due to error: Get http://172.30.1.1:5000/v2/:  dial tcp 172.30.1.1:5000:
getsockopt: connection refused
```

It's because the registry isn't working, go to openshift console and enter into the **default** project `https://x.x.x.x:8443/console/project/default/overview` and you must see two resources, **docker-registry** and **router** they must be running. If they don't work, try to deploy them and look at the logs what is happen.

6.3.5. s2i devonfw

This are the s2i souce and templates to build an s2i images. It provides OpenShift builder images for components of the devonfw (at this momento only for angular and java).

This work is totally based on the implementation of Michael Kuehl from RedHat for Oasp s2i.

All this information is used as a part of the initial setup for openshift.

Previous setup

In order to build all of this, it will be necessary, first, to have a running OpenShift cluster. How to install it here.

Usage

Before using the builder images, add them to the OpenShift cluster.

Deploy the Source-2-Image builder images

First, create a dedicated `devonfw` project as admin.

```
$ oc new-project devonfw --display-name='devonfw' --description='devonfw Application
Standar Platform'
```

Now add the builder image configuration and start their build.

```
oc create -f https://raw.githubusercontent.com/devonfw/devonfw-shop-
floor/master/dsf4openshift/openshift-devonfw-deployment/s2i/java/s2i-devonfw-java-
imagestream.json --namespace=devonfw
oc create -f https://raw.githubusercontent.com/devonfw/devonfw-shop-
floor/master/dsf4openshift/openshift-devonfw-deployment/s2i/angular/s2i-devonfw-
angular-imagestream.json --namespace=devonfw
oc start-build s2i-devonfw-java --namespace=devonfw
oc start-build s2i-devonfw-angular --namespace=devonfw
```

Make sure other projects can access the builder images:

```
oc policy add-role-to-group system:image-puller system:authenticated
--namespace=devonfw
```

That's all !

Deploy devonfw templates

Now, it's time to create devonfw templates to use this s2i and add it to the browse catalog. More information here.

Build All

Use this script to automatically install and build all image streams. The script also creates templates devonfw-angular and devonfw-java inside the project 'openshift' to be used by everyone.

1. Open a bash shell as Administrator
2. Execute shell file:

```
$ /PATH/TO/BUILD/FILE/initial-setup.sh
```

More information about what this script does here.

Links & References

This is a list of useful articels etc I found while creating the templates.

- Template Icons
- Red Hat Cool Store Microservice Demo
- Openshift Web Console Customization

6.3.6. devonfw templates

This are the devonfw templates to build devonfw apps for Openshift using the s2i images. They are based on the work of Mickuehl in Oasp templates/mythaistar for deploy My Thai Star.

- Inside the `example-mythaistar` we have an example to deploy My Thai Star application using devonfw templates.

All this information is used as a part of the initial setup for openshift.

How to use

Previous requirements

Deploy the Source-2-Image builder images

Remember that this templates need a build image from s2i-devonfw-angular and s2i-devonfw-java. More information:

- Deploy the Source-2-Image builder images.

Customize Openshift

Remeber that this templates also have a custom icons, and to use it, we must modify the master-config.yml inside openshift. More information:

- Customize Openshift.

Deploy devonfw templates

Now, it's time to create devonfw templates to use this s2i and add it to the browse catalog.

To let all user to use this templates in all openshift projects, we should create it in an openshift namespace. To do that, we must log in as an admin.

```
oc create -f https://raw.githubusercontent.com/devonfw/devonfw-shop-
floor/master/dsf4openshift/openshift-devonfw-deployment/templates/devonfw-java-
template.json --namespace=openshift
oc create -f https://raw.githubusercontent.com/devonfw/devonfw-shop-
floor/master/dsf4openshift/openshift-devonfw-deployment/templates/devonfw-angular-
template.json --namespace=openshift
```

When it finish, remember to logout as an admin and enter with our normal user.

```
$ oc login
```

How to use devonfw templates in openshift

To use this templates with openshift, we can override any parameter values defined in the file by adding the --param-file=paramfile option.

This file must be a list of <name>=<value> pairs. A parameter reference may appear in any text field inside the template items.

The parameters that we must override are the following

```
$ cat paramfile
  APPLICATION_NAME=app-Name
  APPLICATION_GROUP_NAME=group-Name
  GIT_URI=Git uri
  GIT_REF=master
  CONTEXT_DIR=/context
```

The following parameters are optional

```
$ cat paramfile
  APPLICATION_HOSTNAME=Custom hostname for service routes. Leave blank for default
hostname, e.g.: <application-name>.<project>.<default-domain-suffix>,
  # Only for angular
  REST_ENDPOINT_URL=The URL of the backend's REST API endpoint. This can be declared
after,
  REST_ENDPOINT_PATTERN=The pattern URL of the backend's REST API endpoint that must
be modify by the REST_ENDPOINT_URL variable,
```

For example, to deploy My Thai Star Java

```
$ cat paramfile
  APPLICATION_NAME="mythaistar-java"
  APPLICATION_GROUP_NAME="My-Thai-Star"
  GIT_URI="https://github.com/oasp/my-thai-star.git"
  GIT_REF="develop"
  CONTEXT_DIR="/java/mtsj"

$ oc new-app --template=devonfw-java --namespace=mythaistar --param-file=paramfile
```

6.3.7. Customize Openshift Origin for devonfw

This is a guide to customize Openshift cluster.

Images Styles

The icons for templates must measure the same as below or the images don't show right:

- `Openshift logo`: 230px x 40px.
- `Template logo`: 50px x 50px.
- `Category logo`: 110px x 36px.

How to use

To use it, we need to enter in openshift as an admin and use the next command:

```
$ oc login

$ oc edit configmap/webconsole-config -n openshift-web-console
```

After this, we can see in our shell the webconsole-config.yaml, we only need to navegate untill **extensions** and add the url for our own `css` in the **stylesheetURLs** and `javascript` in the **scriptURLs** section.

IMPORTANT: Scripts and stylesheets must be served with the correct content type or they will not be run by the browser. Scripts must be served with Content-Type: application/javascript and stylesheets with Content-Type: text/css.

In git repositories, the content type of raw is text/plain. You can use rawgit to convert a raw from a git repository to the correct content type.

Example:

```
webconsole-config.yaml: |
  [...]
    extensions:
      scriptURLs:
        - https://cdn.rawgit.com/devonfw/devonfw-shop-
floor/master/dsf4openshift/openshift-cluster-setup/initial-
setup/customizeOpenshift/scripts/catalog-categories.js
      stylesheetURLs:
        - https://cdn.rawgit.com/devonfw/devonfw-shop-
floor/master/dsf4openshift/openshift-cluster-setup/initial-
setup/customizeOpenshift/stylesheet/icons.css
  [...]
```

More information

- Customize icons for Openshift.
- Customize catalog for Openshift.
- Openshift docs about customization.

Old versions

- Customize Openshift for version 3.7.

How to add Custom Icons inside openshift

This is a guide to add custom icons into an Openshift cluster.

Here we can find an icons.css example to use the devonfw icons.

Images Styles

The icons for templates must measure the same as below or the images don't show right:

- `Openshift logo`: 230px x 40px.
- `Template logo`: 50px x 50px.
- `Category logo`: 110px x 36px.

Create a css

Custom logo for openshift cluster

For this example, we are going to call the css icons.css but you can call as you wish. Openshift cluster draw their icon by the id header-logo, then we only need to add to our icons.css the next Style Attribute ID

```
#header-logo {
  background-image: url("https://raw.githubusercontent.com/devonfw/devonfw-shop-floor/master/dsf4openshift/openshift-cluster-setup/initial-setup/customizeOpenshift/images/devonfw-openshift.png);
  width: 230px;
  height: 40px;
}
```

Custom icons for templates

To use a custom icon to a template openshift use a class name. Then, we need to insert inside our icons.css the next Style Class

```
.devonfw-logo {
  background-image: url("https://raw.githubusercontent.com/devonfw/devonfw-shop-floor/master/dsf4openshift/openshift-cluster-setup/initial-setup/customizeOpenshift/images/devonfw.png");
  width: 50px;
  height: 50px;
}
```

To show that custom icon on a template, we only need to write the name of our class in the tag "iconClass" of our template.

```
{
    ...
    "items": [
        {
            ...
            "metadata": {
                ...
                "annotations": {
                    ...
                    "iconClass": "devonfw-logo",
                    ...
                }
            },
            ...
        }
    ]
}
```

Use our own css inside openshift

To do that, we need to enter in openshift as an admin and use the next command:

```
$ oc login
$ oc edit configmap/webconsole-config -n openshift-web-console
```

After this, we can see in our shell the webconsole-config.yaml, we only need to navegate untill **extensions** and add the url for our own `css` in the **stylesheetURLs** section.

IMPORTANT: Scripts and stylesheets must be served with the correct content type or they will not be run by the browser. stylesheets must be served with Content-Type: text/css.

In git repositories, the content type of raw is text/plain. You can use rawgit to convert a raw from a git repository to the correct content type.

Example:

```
webconsole-config.yaml: |
    [...]
    extensions:
      stylesheetURLs:
        - https://cdn.rawgit.com/devonfw/devonfw-shop-
floor/master/dsf4openshift/openshift-cluster-setup/initial-
setup/customizeOpenshift/stylesheet/icons.css
    [...]
```

How to add custom catalog categories inside openshift

This is a guide to add custom `Catalog Categories` into an Openshift cluster.

Here we can find a catalog-categories.js example to use the devonfw catalog categories.

Create a scrip to add custom lengauges and custom catalog categories

Custom language

For this example, we are going add a new language into the languages category. To do that we must created a script and we named as catalog-categories.js

```
// Find the Languages category.
var category = _.find(window.OPENSHIFT_CONSTANTS.SERVICE_CATALOG_CATEGORIES,
                      { id: 'languages' });
// Add Go as a new subcategory under Languages.
category.subCategories.splice(2,0,{ // Insert at the third spot.
  // Required. Must be unique.
  id: "devonfw-languages",
  // Required.
  label: "devonfw",
  // Optional. If specified, defines a unique icon for this item.
  icon: "devonfw-logo-language",
  // Required. Items matching any tag will appear in this subcategory.
  tags: [
    "devonfw",
    "devonfw-angular",
    "devonfw-java"
  ]
});
```

Custom category

For this example, we are going add a new category into the category tab. To do that we must created a script and we named as catalog-categories.js

```
// Add a Featured category as the first category tab.
window.OPENSHIFT_CONSTANTS.SERVICE_CATALOG_CATEGORIES.unshift({
  // Required. Must be unique.
  id: "devonfw-featured",
  // Required
  label: "devonfw",
  subCategories: [
    {
      // Required. Must be unique.
      id: "devonfw-languages",
      // Required.
      label: "devonfw",
      // Optional. If specified, defines a unique icon for this item.
      icon: "devonfw-logo-language",
      // Required. Items matching any tag will appear in this subcategory.
      tags: [
        "devonfw",
        "devonfw-angular",
        "devonfw-java"
      ]
    }
  ]
});
```

Use our own javascript inside openshift

To do that, we need to enter in openshift as an admin and use the next command:

```
$ oc login
$ oc edit configmap/webconsole-config -n openshift-web-console
```

After this, we can see in our shell the webconsole-config.yaml, we only need to navegate untill **extensions** and add the url for our own `javascript` in the **scriptURLs** section.

IMPORTANT: Scripts and stylesheets must be served with the correct content type or they will not be run by the browser. Scripts must be served with Content-Type: application/javascript.

In git repositories, the content type of raw is text/plain. You can use rawgit to convert a raw from a git repository to the correct content type.

Example:

```
webconsole-config.yaml: |
  [...]
    extensions:
      scriptURLs:
        - https://cdn.rawgit.com/devonfw/devonfw-shop-
floor/master/dsf4openshift/openshift-cluster-setup/initial-
setup/customizeOpenshift/scripts/catalog-categories.js
  [...]
```

Customize Openshift Origin v3.7 for devonfw

This is a guide to customize Openshift cluster. For more informaticon read the next:

- Openshift docs customization for the version 3.7.

Images Styles

The icons for templates must measure the same as below or the images don't show right:

- `Openshift logo`: 230px x 40px.
- `Template logo`: 50px x 50px.
- `Category logo`: 110px x 36px.

Quick Use

This is a quick example to add custom icons and categories inside openshift.

To modify the icons inside openshift, we must to modify our master-config.yaml of our openshift cluster. This file is inside the openshift container and to obtain a copy of it, we must to know what's our openshift container name.

Obtain the master-config.yaml of our openshift cluster

Obtain the name of our openshift container

To obtain it, we can know it executing the next:

```
$ docker container ls
CONTAINER ID        IMAGE                                          COMMAND
CREATED             STATUS              PORTS
NAMES
83a4e3acda5b        openshift/origin:v3.7.0
"/usr/bin/openshift …"   6 days ago          Up 6 days
origin
```

Here we can see that the name of the container is origin. Normaly the container it's called as origin.

Copy the master-config.yaml of our openshift container to our directory

This file is inside the openshift container in the next directory: /var/lib/origin/openshift.local.config/master/`master-config.yaml` and we can copy it with the next command:

```
$ docker cp origin:/var/lib/origin/openshift.local.config/master/master-config.yaml ./
```

Now we have a file with the configuration of our openshift cluster.

Copy all customize files inside the openshift container

To use our customization of devonfw Openshift, we need to copy our files inside the openshift container.

To do this we need to copy the images, scripts and stylesheets from here inside openshift container, for example, we could put it all inside a folder called openshift.local.devonfw. On the step one we obtain the name of this container, for this example we assume that it's called origin. Then our images are located inside openshift container and we can see an access it in `/var/lib/origin/openshift.local.devonfw/images`.

```
$ docker cp ./openshift.local.devonfw origin:/var/lib/origin/
```

Edit and copy the master-config.yaml to use our customize files

The master-config.yaml have a sections to charge our custom files. All this sections are inside the `assetConfig` and their names are the next:

- The custom stylessheets are into `extensionStylesheets`.
- The custom scripts are into `extensionScripts`.
- The custom images are into `extensions`.

To use all our custom elements only need to add the directory routes of each element in their appropriate section of the master-config.yaml

```
...
assetConfig:
  ...
  extensionScripts:
  - /var/lib/origin/openshift.local.devonfw/scripts/catalog-categories.js
  extensionStylesheets:
  - /var/lib/origin/openshift.local.devonfw/stylesheet/icons.css
  extensions:
  - name: images
    sourceDirectory: /var/lib/origin/openshift.local.devonfw/images
  ...
...
```

Now we only need to copy that master-config.yaml inside openshift, and restart it to load the new configuration. To do that execute the next:

```
$ docker cp ./master-config.yaml
origin:/var/lib/origin/openshift.local.config/master/master-config.yaml
```

To re-start openshift do `oc cluster down` and start again your persistent openshift cluster.

More information

- Customize icons for Openshift.
- Customize catalog for Openshift.
- Openshift docs about customization.