CONTINUOUS DELIVERY

Hands on session

Agenda

Introduction

- Purpose
- Topics covered
- Scheduling
- Environment
- Sample application

Use cases description per level (description, tooltips, details)

Debriefing and conclusions

INTRODUCTION

Introduction : purpose

The hands-on session is aimed to deliver a global, practicial knowledge of continuous delivery processes in a realistic context:

- Self-organized teams (DEV/OPS, optionally BA/manager if any)
- Non static, changing needs and priorities (unexpected reviews)
- A toolset supporting the processes to masterize and set up
- An application to understand, extend and deliver through continuous delivery practices and tools

Introduction : topics covered

PILLAR	PROCESS/PRACTICE	MANDATORY TOOLSET
AGILITY	KANBAN-LIKE MANAGEMENT	ZENHUB.IO
AGILITY	INSTANT FEEDBACK	SLACK
AGILITY	SOCIAL CODING	GITHUB
CRAFTMANSHIP	CONTINUOUS BUILD/INTEGRATION	MAVEN / JENKINS / NEXUS
CRAFTMANSHIP	UNIT/INTEGRATION TESTS BDD/TDD CODE COVERAGE/MUTATION TESTS PERFORMANCE TESTS	JUNIT (SUREFIRE/FAILSAFE) JGIVEN JACOCO/PITEST GATLING
DEVOPS	SINGLE VERSION PACKAGING AUTODEPLOYMENT : LOCAL AUTODEPLOYMENT : CLOUD PIPELINE ORCHESTRATION	JENKINS / DOCKER DOCKER DOCKER / TUTUM JENKINS
MEASUREMENT	CODE QUALITY ANALYSIS PROCESS METRICS	SONARQUBE ELASTICSEARCH/KIBANA

Introduction : environment

A workstation per trainee is recommended, at least one per team :

- Recent Chrome/Firefox version
- Enough RAM to run Java IDE (IntelliJ / Eclipse)

At least two linux servers/VMs per team are required :

- 50 Gb HD, 8 Gb RAM, CentOS Core 7.2+ : build platform
- 16 Gb HD, 2 Gb RAM, Ubuntu 14.04+ : "cloud" deployment server

Full internet access from the servers/workstations is mandatory

Linux, maven and java skills in each team are highly recommended

Introduction : scheduling

The toolset previously mentioned is mandatory, except IDE

The use cases can be processed in any order ... when possible

Track your progression on the kanban board

If stuck:
- Help is available on lower slides, from expert to beginner level
- Feel free to ask questions ... through kanban/instant messaging tool

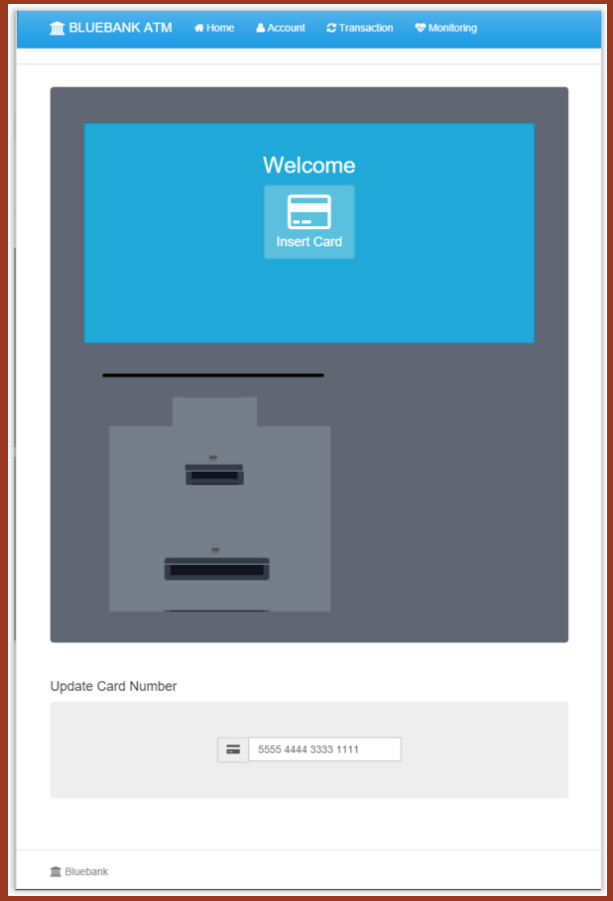
KEEP IT SIMPLE : we do not matter about security aspects here ...

Introduction : sample application

Bluebank : ATM application
(thanks to Riad)

3 main operations :
- cash deposit
- cash withdrawal
- account inquiry

Server : java standalone (port 8180)
UI : web site (requires HTTP server)



https://github.com/Finaxys/bluebank-atm-server

USE CASES DESCRIPTION

Use case #1 : Setup collaboration tools

To manage the team's follow-up during the hands-on session, set up the following items:

- ATM repository forked in this organization
- Collaborators included in this repository, including staff
- Assignement of issues in Zenhub (keep To Do/Progress/Done only)
- Set the start/due dates of the DEADLINE milestone
- Slack integration with the repository in a dedicated channel

Use case #1 : Tooltips

For Zenhub/Github, register your personal account using Finaxys email

Use cases already exist in the ATM repository as issues

For slack, we have a Finaxys domain : https://finaxys.slack.com

Use case #1 : Details 1/4

Github : source code management tool including social coding features

- Worldwide standard
- Git model (distributed versioning control system)
- Wiki/markdown online documentation
- Collaboration features : issues, forks, pull requests
- "Forking" the repository allows to work on a personal/organization copy with ability to merge (or not) changes later

Github account signup

Create organization and invite members

Fork the repository into your organization ==>

Assign the team to repository (Settings -> Collaborators and Teams)

Use case #1 : details 2/4

Zenhub : Chrome/Firefox extension handling Github issues "a la agile"

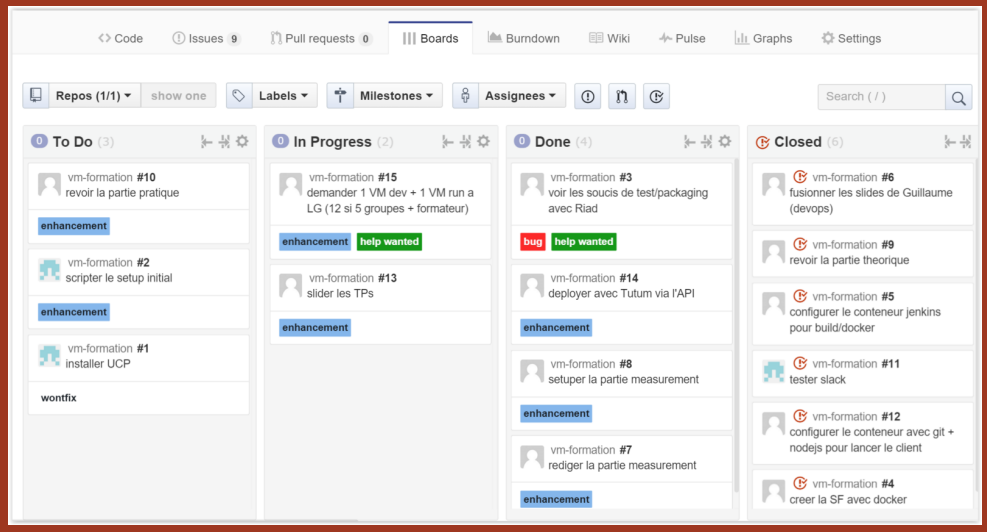
Install Zenhub on Firefox/Chrome (tour included)

On "Boards" screen keep only To Do/In Progress/Done/Closed stages

ASK FOR STAFF TO COPY ISSUES ON YOUR FORKED PROJECT

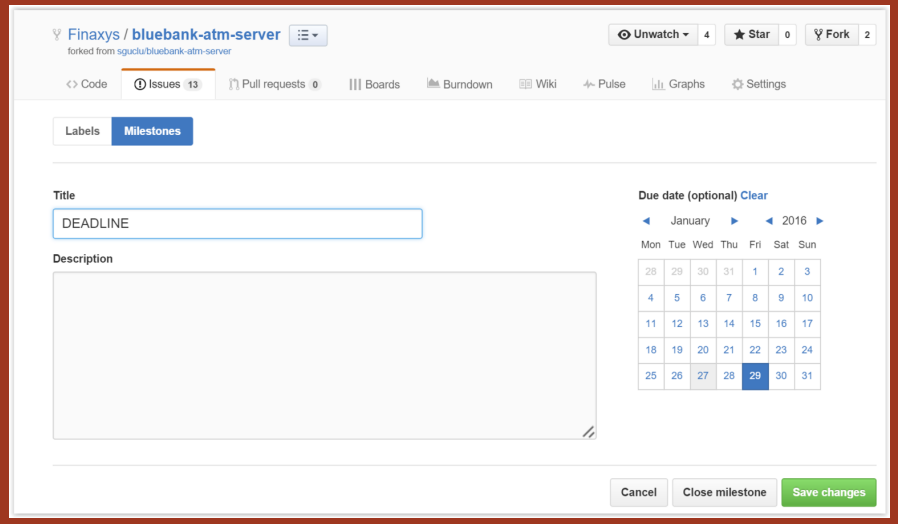
Each issue can be

- assigned
- slided between stages
- prioritized (milestone)
- estimated (workload)



Use case #1 : details 3/4

Assign, estime and define start and due dates of DEADLINE milestone (beginning-end of session)



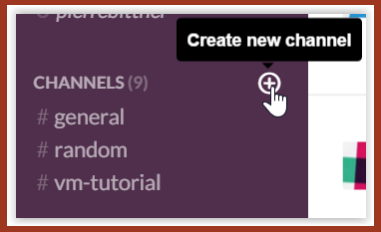
We are almost ready to go ...

Use case #1 : Details 3/3

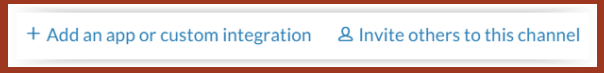
Slack : instant messaging for enterprises solution
- allows public/shared/private channels to communicate
- integrates with Github and other tools to notify changes

In not already done yet, create an account : https://finaxys.slack.com

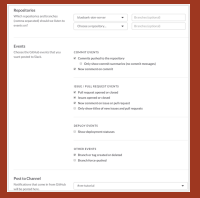
Create a channel for your team =>



Include github application =>



Map your repository to your channel =>



Use case #2 : Build locally

This part must be performed on a local workstation IDE

- Get the source code and make the application compile and package (without tests for the moment)

Use case #2 : tooltips

The compilation requires specific tools and configurations to find

The server part is handled by Maven

Use case #2 : Details 1/6

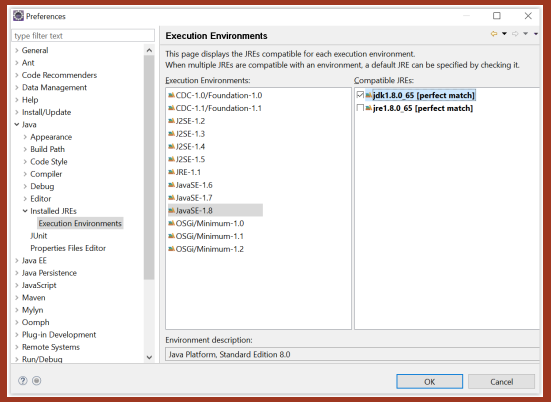
ATM server build requires on the workstation:

- A Java development environment (Eclipse/IntelliJ)
- A java 8 JDK compiler (current link here)
- Maven (included in Eclipse/IntelliJ) for build/test/package/deploy
- Protocol buffers compiler (message serialization)

The IDE configuration must be this one:

- JDK 8 / Maven 3.x
- Environment variable PIPELINE_VERSION enforced in maven build
- path to PROTOC executable available in the PATH

Use case #2 : Details 2/6



Build environment sample (Eclipse / Windows)

=> use JDK not JRE

Build is managed by maven

- #1 open-source build, test, package, deployment tool
- Command-line tools with many ways to operate (IDEs, CI tools...)
- Descriptive configuration : pom.xml (c.f. descriptor and XSD)
- Plenty of existing plugins and reusable components
- Handles dependencies between components
- Default lifecycles provided for components see here for details)

Each component has a unique GAV : groupId - artifactId - version

Use case #2 : Details 3/6

Protocol buffers : data serialization and interface framework (Google)

- generic, technology agnostic description of the data structure

```
<plugin>
  <groupId>com.google.protobuf.tools</groupId>
  <artifactId>maven-protoc-plugin</artifactId>
  <version>0.3.2</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <goals>
        <goal>compile</goal>
      </goals>
      <phase>generate-sources</phase>
      <configuration>
        <protoSourceRoot>${basedir}/src/main/resources/proto</protoSour
        <includes>
          <param>**/*.proto</param>
        </includes>
      </configuration>
    </execution>
  </executions>
  <configuration>
    <protocExecutable>${PROTOC_HOME}</protocExecutable>
  </configuration>
</plugin>
```

Use case #2 : Details 4/6

- interface classes generation for each language with protoc compiler
- sources in src/main/resources/proto/messages.proto
- interface classes compiled in target/generated-sources/protobuf/java

```
package org.bluebank.contract;

message ValidateCardRequest {
    optional string cardNumber = 1;
    optional string transactionId =
    optional string id = 3;
    optional string location = 4;
    optional string bankName = 5;
}

...
```

```
// Generated by the protocol buffer compiler
// source: messages.proto

package org.bluebank.contract;

public final class Messages {
  private Messages() {}
  public static void registerAllExtensions(
      com.google.protobuf.ExtensionRegistry
  }
  public interface ValidateCardRequestOrBuil
      // @@protoc_insertion_point(interface_
      com.google.protobuf.MessageOrBuilder {

    /**
     * <code>optional string cardNumber = 1;
     */
    boolean hasCardNumber();
  ...
```

COMPILER MUST BE INSTALLED & AVAILABLE IN THE PATH !

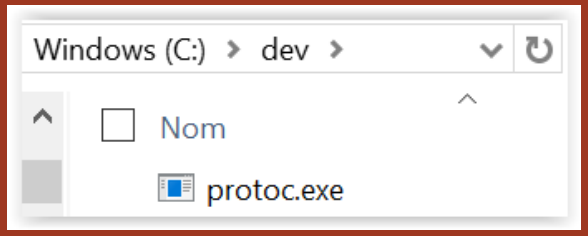
Use case #2 : Details 5/6

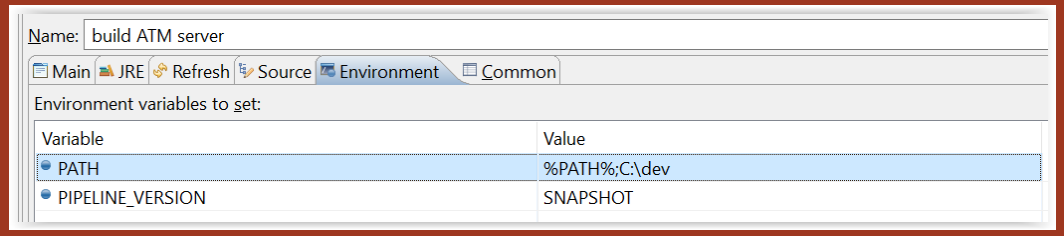
Here, an enforcer requires an environment variable to build version (unclean but useful for next parts)

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-enforcer-plugin</artifactId>
  <version>1.4.1</version>
  <executions>
    <execution>
      <id>enforce-property</id>
        <goals>
          <goal>enforce</goal>
        </goals>
        <configuration>
          <rules>
            <requireProperty>
              <property>project.version</property>
              <message>"Project version must be specified."</message>
              <regex>.*(\d|-SNAPSHOT)$</regex>
              <regexMessage>
                "Please set the PIPELINE_VERSION environment variable (number or -SNAPS
              </regexMessage>
            </requireProperty>
          </rules>
          <fail>true</fail>
        </configuration>
    </execution>
  </executions>
</plugin>
```

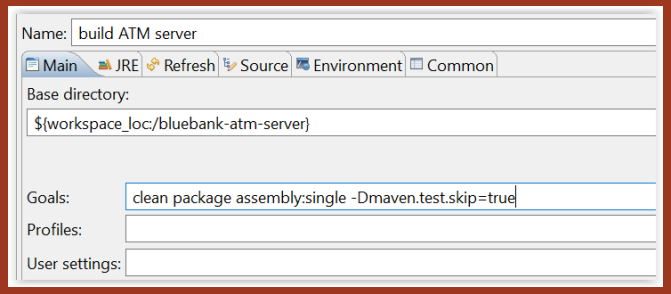
Use case #2 : Details 6/6

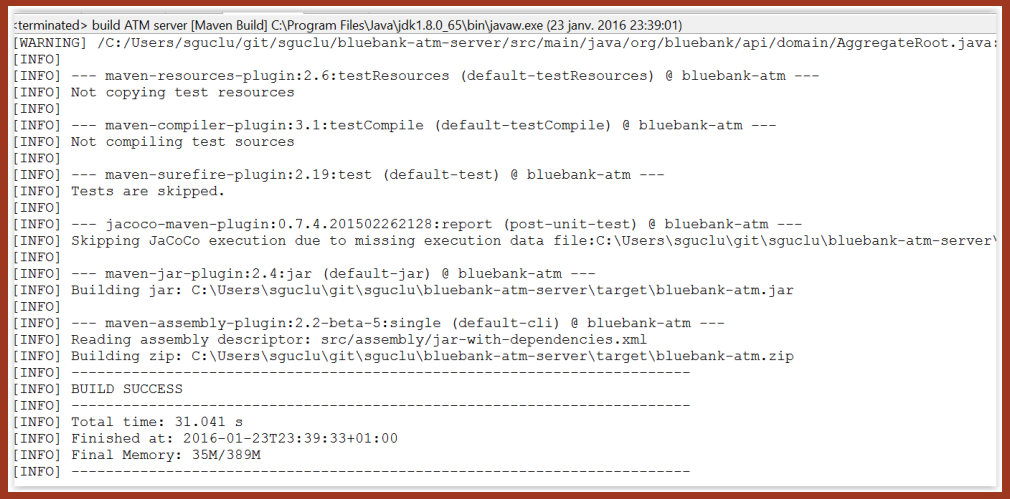
Here is a sample configuration to **build** and package the server/jar (Eclipse / Windows)





Maven goals : clean package assembly:single -Dmaven.test.skip=true





Use case #3 : Analyze locally

This part must be performed on a local workstation IDE

- Run unit tests (Junit)
- Run integration tests (failsafe)
- Run code coverage tests (jacoco)
- Run BDD scenarios (jgiven)
- Run mutation tests (pitest)
- Generate all reports for review

Use case #3 : tooltips

All test frameworks are available as maven dependencies

Unit tests call BDD/TDD scenarios (Jgiven/Junit + mockito)

Integration tests call a jetty server (failsafe)

A profile called "all_tests" activates unit and integration tests

Code coverage must be called explicitly (jacoco)

Mutation testing must be called explicitly (pitest)

Use case #3 : Details 1/2

Using same running config as for build, full analysis can be run:

```
clean verify -Pall-tests jacoco:report org.pitest:pitest-maven:mutationCoverage
```

Warning : as you will see, all tests cannot apply to all the code

Unit tests results (raw) : target/surefire-tests

```
-------------------------------------------------------------------------------
Test set: org.bluebank.atm.AtmHealthCheckTest
-------------------------------------------------------------------------------
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.103 sec - in org.bluebank.atm.AtmHealthCheckTest

```

Integration tests (raw) : target/failsafe-tests

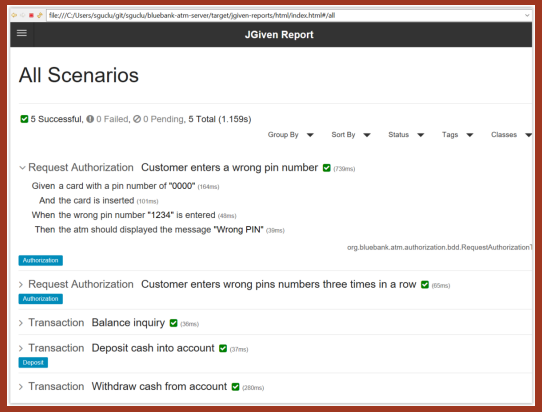
```
-------------------------------------------------------------------------------
Test set: org.bluebank.atm.AtmResourceIT
-------------------------------------------------------------------------------
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 6.883 sec - in org.bluebank.atm.AtmResourceIT

```

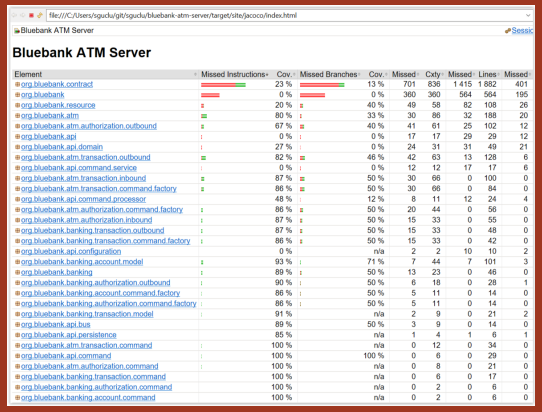
Use case #3 : Details 2/2

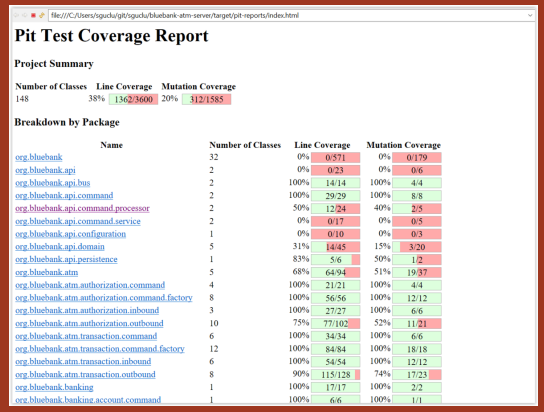
BDD HTML report in target/jgiven-reports/html

Coverage HTML reports in target/site/jacoco

Mutation testing reports in target/pit-reports







Use case #4 : Run ATM locally

This part must be performed on a local workstation IDE

Make the application run locally for review

- ATM server
- HTTP client

Use case #4 : Tooltips

The server part can be run as a standalone application locally by :

- running the main class
- running the jar (requires classpath setup)

The UI part (find out where it is located) requires an HTTP server

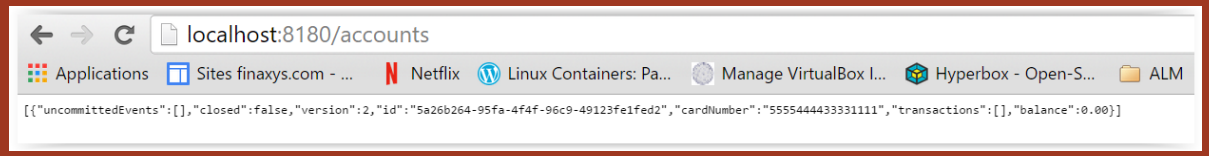
Use case #4 : Details 1/2

Here let's just run the main class as-is, jar run will be used later ...

The main class to run is org.bluebank.Application

```
run ATM server [Java Application] C:\Program Files\Java\jre1.8.0_65\bin\javaw.exe (24 janv. 2016 21:41:14)
21:41:16.850 [main] DEBUG o.e.j.u.component.AbstractLifeCycle - STARTED @2026ms o.e.j.s.ServletContextHandler@4b5d6a01{/,null,AVAILABLE}
21:41:16.850 [main] DEBUG o.e.j.u.component.AbstractLifeCycle - starting ServerConnector@4445629{HTTP/1.1,[http/1.1]}{0.0.0.0:8180}
21:41:16.860 [main] DEBUG o.e.j.u.component.ContainerLifeCycle - ServerConnector@4445629{HTTP/1.1,[http/1.1]}{0.0.0.0:8180} added {sun.nio.ch.ServerSocketChann
21:41:16.861 [main] DEBUG o.e.j.u.component.AbstractLifeCycle - starting org.eclipse.jetty.util.thread.ScheduledExecutorScheduler@33afa13b
21:41:16.861 [main] DEBUG o.e.j.u.component.AbstractLifeCycle - STARTED @2038ms org.eclipse.jetty.util.thread.ScheduledExecutorScheduler@33afa13b
21:41:16.861 [main] DEBUG o.e.j.u.component.AbstractLifeCycle - starting HttpConnectionFactory@1fe20588[HTTP/1.1]
21:41:16.861 [main] DEBUG o.e.j.u.component.AbstractLifeCycle - STARTED @2038ms HttpConnectionFactory@1fe20588[HTTP/1.1]
21:41:16.861 [main] DEBUG o.e.j.u.component.AbstractLifeCycle - starting org.eclipse.jetty.server.ServerConnector$ServerConnectorManager@54c562f7
21:41:16.866 [main] DEBUG o.e.j.u.component.AbstractLifeCycle - starting org.eclipse.jetty.io.ManagedSelector@251f7d26 id=0 keys=-1 selected=-1
21:41:16.881 [main] DEBUG o.e.j.u.component.AbstractLifeCycle - STARTED @2057ms org.eclipse.jetty.io.ManagedSelector@251f7d26 id=0 keys=0 selected=0
21:41:16.881 [main] DEBUG o.e.j.util.thread.QueuedThreadPool - queue org.eclipse.jetty.io.ManagedSelector@251f7d26 id=0 keys=0 selected=0
21:41:16.881 [main] DEBUG o.e.j.u.component.AbstractLifeCycle - starting org.eclipse.jetty.io.ManagedSelector@ecf9fb3 id=1 keys=-1 selected=-1
21:41:16.882 [qtp750468423-15] DEBUG o.e.j.util.thread.QueuedThreadPool - run org.eclipse.jetty.io.ManagedSelector@251f7d26 id=0 keys=0 selected=0
21:41:16.882 [qtp750468423-15] DEBUG o.e.j.u.t.s.ExecuteProduceConsume - EPR Idle/org.eclipse.jetty.io.ManagedSelector$SelectorProducer@7dce6799 execute
21:41:16.882 [qtp750468423-15] DEBUG o.e.j.u.t.s.ExecuteProduceConsume - EPR Prod/org.eclipse.jetty.io.ManagedSelector$SelectorProducer@7dce6799 produce enter
21:41:16.882 [qtp750468423-15] DEBUG o.e.j.u.t.s.ExecuteProduceConsume - EPR Prod/org.eclipse.jetty.io.ManagedSelector$SelectorProducer@7dce6799 producing
21:41:16.882 [qtp750468423-15] DEBUG org.eclipse.jetty.io.ManagedSelector - Selector loop waiting on select
21:41:16.882 [main] DEBUG o.e.j.u.component.AbstractLifeCycle - STARTED @2058ms org.eclipse.jetty.io.ManagedSelector@ecf9fb3 id=1 keys=0 selected=0
21:41:16.882 [main] DEBUG o.e.j.util.thread.QueuedThreadPool - queue org.eclipse.jetty.io.ManagedSelector@ecf9fb3 id=1 keys=0 selected=0
21:41:16.883 [main] DEBUG o.e.j.u.component.AbstractLifeCycle - STARTED @2059ms org.eclipse.jetty.server.ServerConnector$ServerConnectorManager@54c562f7
21:41:16.883 [qtp750468423-14] DEBUG o.e.j.util.thread.QueuedThreadPool - run org.eclipse.jetty.io.ManagedSelector@ecf9fb3 id=1 keys=0 selected=0
21:41:16.883 [qtp750468423-14] DEBUG o.e.j.u.t.s.ExecuteProduceConsume - EPR Idle/org.eclipse.jetty.io.ManagedSelector$SelectorProducer@ffeaa6e execute
21:41:16.883 [qtp750468423-14] DEBUG o.e.j.u.t.s.ExecuteProduceConsume - EPR Prod/org.eclipse.jetty.io.ManagedSelector$SelectorProducer@ffeaa6e produce enter
21:41:16.883 [qtp750468423-14] DEBUG o.e.j.u.t.s.ExecuteProduceConsume - EPR Prod/org.eclipse.jetty.io.ManagedSelector$SelectorProducer@ffeaa6e producing
21:41:16.883 [qtp750468423-14] DEBUG org.eclipse.jetty.io.ManagedSelector - Selector loop waiting on select
21:41:16.884 [main] DEBUG o.e.j.u.component.ContainerLifeCycle - ServerConnector@4445629{HTTP/1.1,[http/1.1]}{0.0.0.0:8180} added {acceptor-0@4593ff34,POJO}
21:41:16.884 [main] DEBUG o.e.j.util.thread.QueuedThreadPool - queue acceptor-0@4593ff34
21:41:16.884 [qtp750468423-16] DEBUG o.e.j.util.thread.QueuedThreadPool - run acceptor-0@4593ff34
21:41:16.884 [main] INFO  o.e.jetty.server.ServerConnector - Started ServerConnector@4445629{HTTP/1.1,[http/1.1]}{0.0.0.0:8180}
21:41:16.884 [main] DEBUG o.e.j.u.component.AbstractLifeCycle - STARTED @2060ms ServerConnector@4445629{HTTP/1.1,[http/1.1]}{0.0.0.0:8180}
21:41:16.885 [main] INFO  org.eclipse.jetty.server.Server - Started @2061ms
21:41:16.885 [main] DEBUG o.e.j.u.component.AbstractLifeCycle - STARTED @2061ms org.eclipse.jetty.server.Server@402e37bc
```

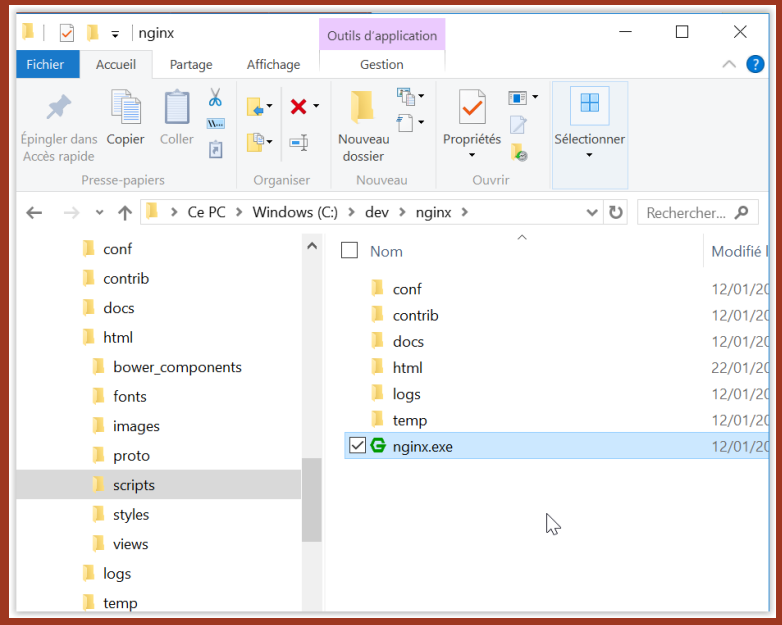
The server listens on port 8180, e.g. http://localhost:8180/accounts

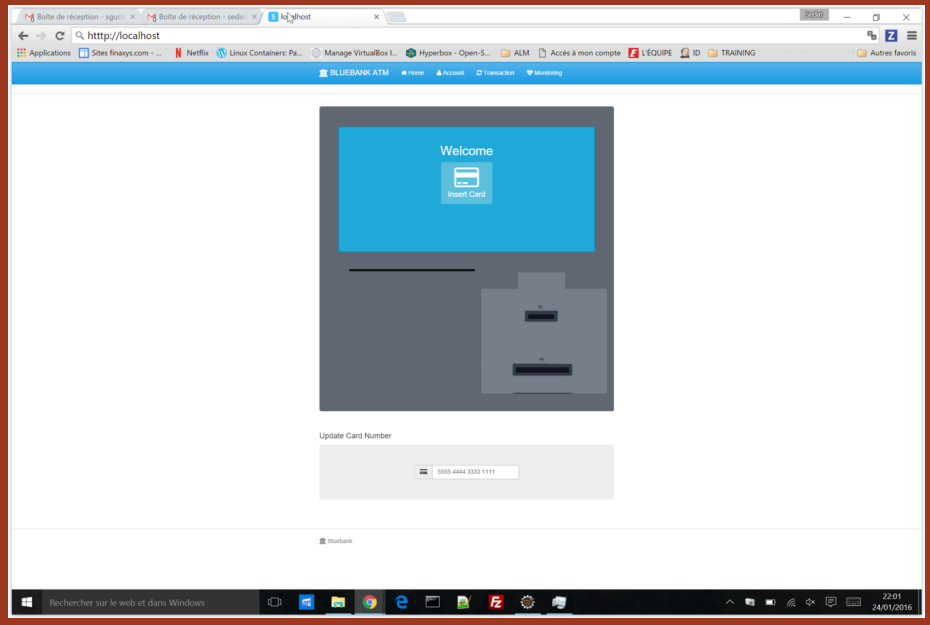


Use case #4 : Details 2/2

For static site, the static pages are located in node-client

A server like nginx is required to display pages (copy the content of node-client directly in *html* folder)





Play a bit to understand the design and features

Use case #5 : Install a software factory

This part must be performed on the virtual machine

Install and run a software factory including:

- jenkins : continuous integration engine version (jenkins:1.625.3)

- nexus : binary repository management tool (sonatype/nexus:2.11.2-06)

- sonarqube : code quality analysis, version (sonarqube:5.2)

- elasticsearch/logstash/kibana : metrology (sebp/elk:E1L1K4)

... using docker and docker-compose

Use case #5 : Tooltips 1/2

Docker is #1 container management solution

- Strong alternative to virtual machines (lighter, faster, better)

- Allows custom image build/commit, or reuse/override existing ones

- Can handle almost any kind of components/applications

First, an account on https://hub.docker.com/ will greatly help ...

Use case #5 : Tooltips 2/2

Docker-compose runs on top of docker

- Allows composition of containers (a.k.a. services)

- Manages services start/stop order based on services dependencies

- Handles network link and naming resolution between services

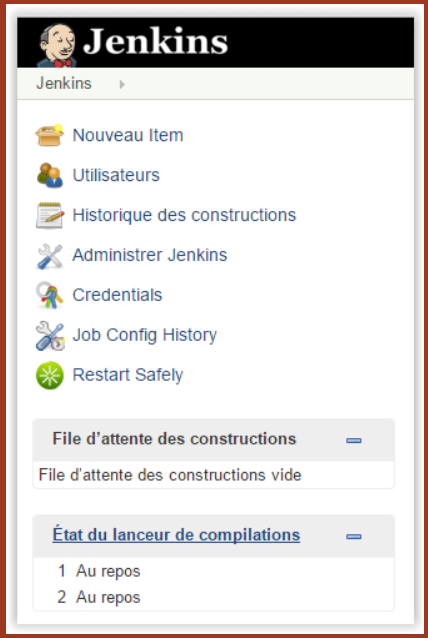
- Supports custom build and out-of-the box images as services

Use case #5 : Details 1/10

- On the virtual machine, connect to the docker hub `docker login`

- Try to run a Docker image containing Jenkins official image (https://github.com/jenkinsci/docker) `docker run -p 8080:8080 jenkins:1.625.`

- After a while, you should have this on http://<your_vm_address>:8080 :



- What happened there ?

Use case #5 : Details 2/10

- Docker downloaded the latest Jenkins image from https://hub.docker.com/

```
Unable to find image 'jenkins:latest' locally
latest: Pulling from library/jenkins

523ef1d23f22: Extracting 25.17 MB/51.35 MB
140f9bdfeb97: Download complete
5c63804eac90: Download complete
ce2b29af7753: Download complete
4ee671494b6b: Download complete
```

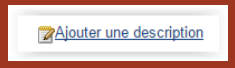
- Then Docker ran a container based on this image listening on http port 8080 (jenkins port mapped with the 8080 port of your VM)

And voila ... You can check the running container using: `docker ps`

```
[traineegrp@formation ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                                        NAMES
cale9611433c        jenkins             "/bin/tini -- /usr/lo"   6 minutes ago       Up 6 minutes        50000/tcp, 0.0.0.0:5432->8080/tcp            cranky_brahmagupta
```

WARNING : data belong to the running containers, not the images!

- Add a description (FIRST CONTAINER) on Jenkins:



Use case #5 : Details 3/10

- Stop the container, (ctrl + c) then launch the Docker run command again : description disappeared from the UI!

We just ran a new container with its own storage...

```
docker ps -a | grep jenki
```

```
[traineegrp@formation ~]$ docker ps -a  | grep jenkins
cale9611433c        jenkins                  "/bin/tini -- /usr/lo"   About an hour ago   Up About an hour              50000/tcp, 0.0.0.0:5432->8080/tcp                                cranky_brahmagupta
7c3cac9be7d8        jenkins                  "/bin/tini -- /usr/lo"   About an hour ago   Exited (130) About an hour ago                                                                 fervent_boyd
```

- Now, stop this new container and run the previous one using ids

```
docker stop <new_container_id> ; docker start <old_container_id>
```

Message is back ... because it was saved in the first container created

Use case #5 : Details 4/10

Containers persist data in the container by default, but what if we really need to keep the data ?

=> We need to "export" the datas in a volume on the host server

- Stop the container, and run it again with the volume mount option

```
docker stop <old_container_id>
sudo mkdir -p /volumes/jenkins
sudo chown 1000:1000 /volumes/jenkins
docker run -p 8080:8080 -v /volumes/jenkins:/var/jenkins_home jenkins:1.625.3
```

In this new container, var/jenkins_home folder (jenkins repository) is now hosted on the /volumes/jenkins folder of the virtual machine

=> persists across container restart/removals

=> take care of concurrent accesses through multiple containers!

Use case #5 : Details 5/10

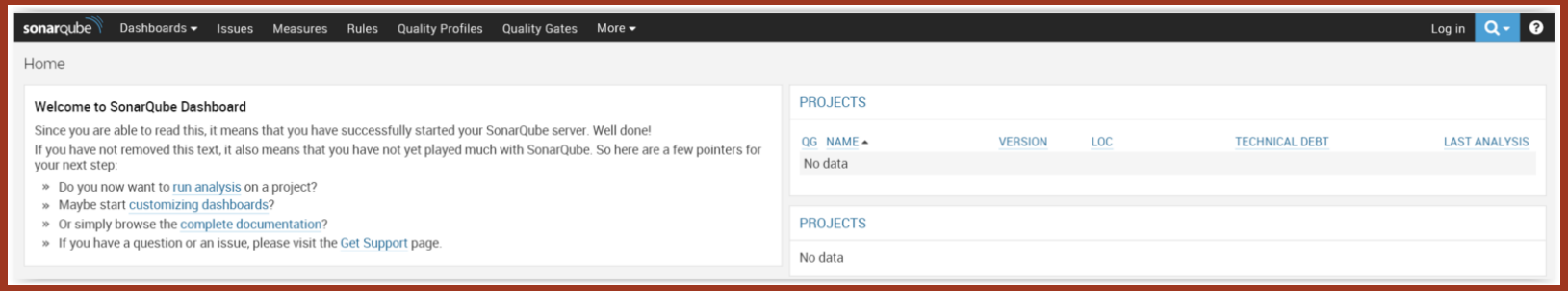
- In the same way, start a Sonarqube container with these properties:

port 9000 : HTTP port of the user interface

port 9092 : JDBC port of the internal database

/volumes/sonar/data:/opt/sonarqube/data (internal database)
/volumes/sonar/extensions:/opt/sonarqube/extensions (plugins)

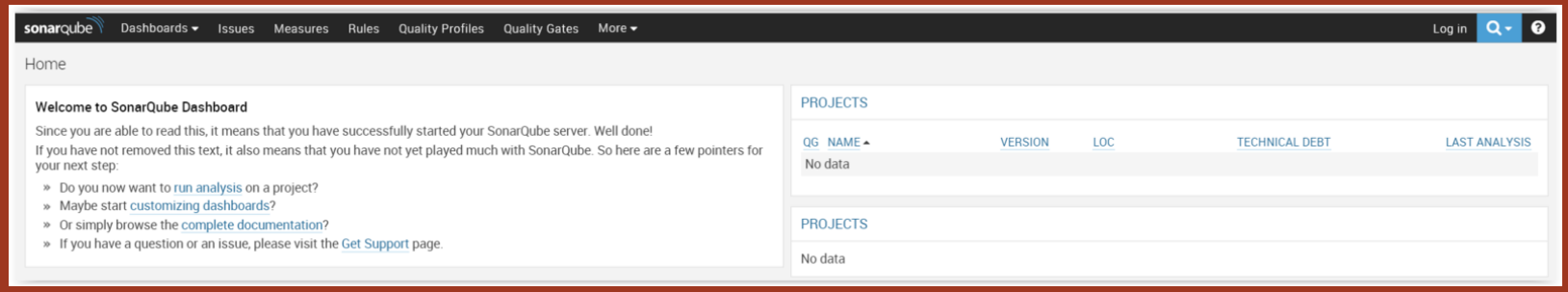
```
sudo mkdir -p /volumes/sonar/data /volumes/sonar/extensions
sudo chown -R 1000:1000 /volumes/sonar
docker run -p 9000:9000 -p 9092:9092 \
-v /volumes/sonar/data:/opt/sonarqube/data \
-v /volumes/sonar/extensions:/opt/sonarqube/extensions \
sonarqube:5.2
```



Use case #5 : Details 6/10

- Then, start a Nexus container with these properties:
 port 8081 : HTTP port of the user interface
/volumes/nexus:/sonatype-work (internal repository)

```
sudo mkdir /volumes/nexus
sudo chown 200:200 /volumes/nexus
docker run -p 8081:8081 \
-v /volumes/nexus:/sonatype-work \
sonatype/nexus:2.11.2-06
```



Use case #5 : Details 7/10

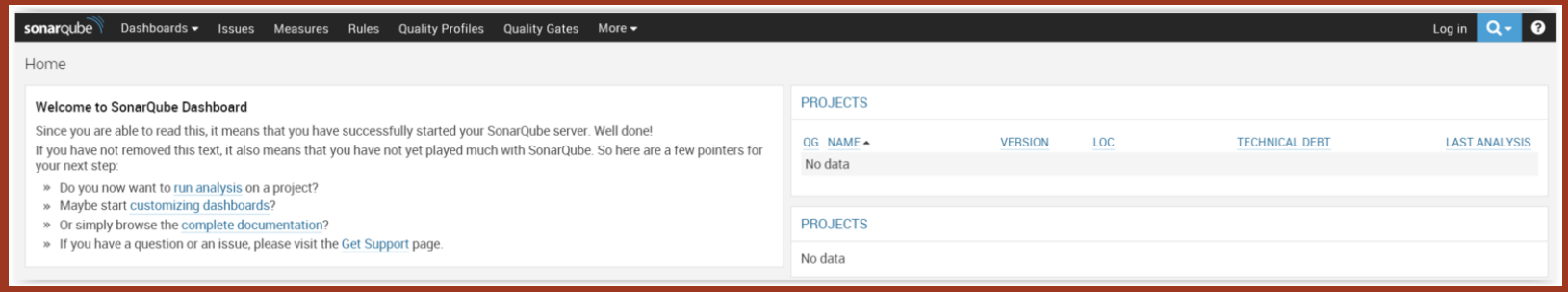
- Finally, run an Elastic/Kibana container with these properties:

port 9200, 9300 : Elasticsearch ports

port 5601 : Kibana user interface

/volumes/elk:/var/lib/elasticsearch (internal repository)

```
sudo mkdir /volumes/elk
sudo chown 103:107 /volumes/elk
docker run -p 9200:9200 -p 9300:9300 -p 5601:5601 \
-v /volumes/elk:/var/lib/elasticsearch \
sebp/elk:E1L1K4
```



Use case #5 : Details 8/10

It works fine, but using docker-compose:

- everything is started in the proper order

- ports/volumes are handled automatically

- using named containers, services can talk to each other

- existing, unchanged service is started only once

- Stop all the containers, then edit a file called docker-compose.yml

Use case #5 : details 9/10

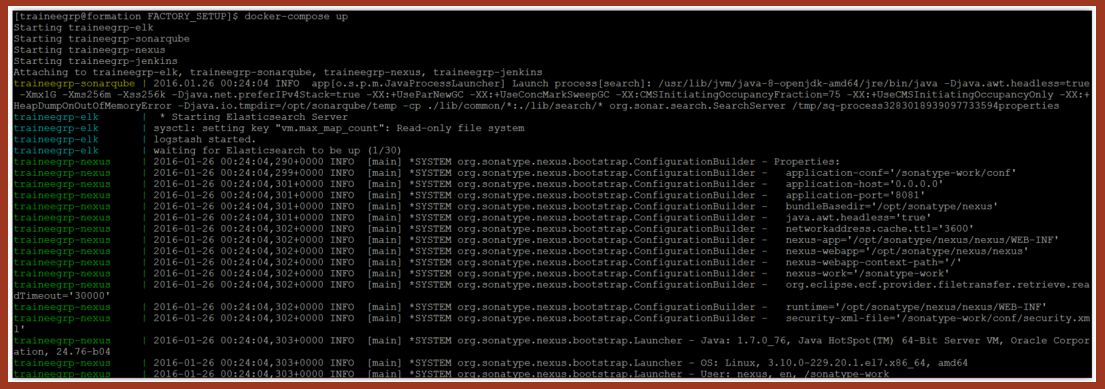
- Put the following content in it, then save (scroll!):

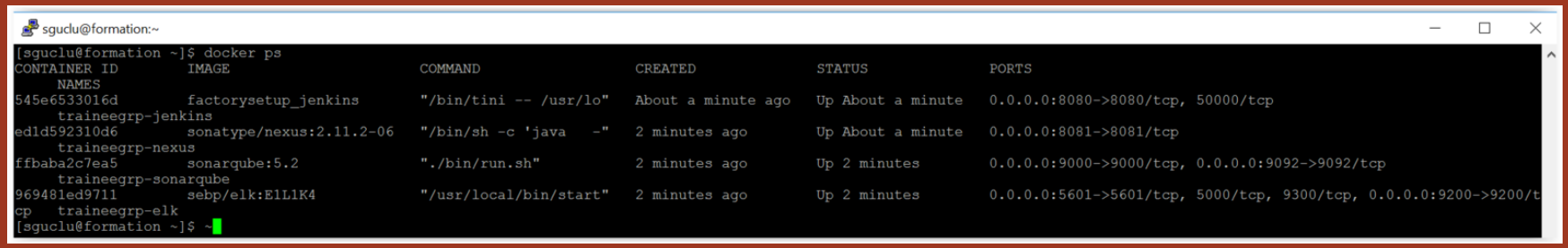
```
jenkins:
  image : "jenkins:1.625.3"
  ports:
   - "8080:8080"
  volumes:
   - /volumes/jenkins:/var/jenkins_home
  container_name: traineegrp-jenkins
  links:
  - sonar
  - nexus
  - elk

nexus:
  image: "sonatype/nexus:2.11.2-06"
  ports:
   - "8081:8081"
  volumes:
   - /volumes/nexus:/sonatype-work
  container_name: traineegrp-nexus
  links:
  - elk

sonar:
  image: "sonarqube:5.2"
  ports:
```

Use case #5 : details 10/10

- Run now this command and check what is going on : `docker-compose up`

Each service is launched as expected in the proper order





This software factory will be improved later ...

Use case #6 : Prepare the factory

Jenkins :

- Install tools : Gatling, Java 8, Maven 3, git client, Protocol Buffers
- install plugins : git, sonar, htmlpublisher, jobConfigHistory, saferestart, pitmutation, envinject, parameterized-trigger, build-pipeline-plugin, rebuild, mask-passwords, logstash, metadata
- Set the sonarqube, logstash and stash configuration in Jenkins
- Allow jenkins to run docker containers on the host

Sonarqube : install Checkstyle, PMD, Findbugs, Github et Pitest plugins

ELK : install the "HEAD" plugin on elasticsearch (indexes browsing)

Use case #6 : Tooltips 1/2

Sonar plugins have to be installed from the UI

Jenkins plugins can be installed:

- From the UI : convenient but not automated

- From https://updates.jenkins-ci.org/download/plugins/ and copied to the plugins folder : painful, no plugin dependencies managed

- Managed by jenkins-cli : better but requires post-start commands

- Included in the image build process : requires custom image build

Use case #6 : Tooltips 2/2

Jenkins tools can be installed:

- Manually in the container, then declared : painful, not versionned

- Auto-installed by jenkins when possible : better but requires post-start configuration on the slave

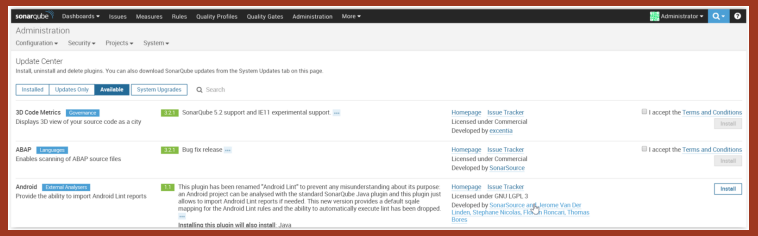
- Included in the image build process : requires custom image build

Use case #6 : Details 1/12

Sonarqube :

- log on http://<your_VM>:9000

user/password : admin / admin



- The available plugins can be installed

from update center : http://<your_VM>:9000/updatecenter/available

- Select and install the plugins, then restart the factory `docker-compose restar`

As long as the sonarqube database and extension folders are managed as volumes outside the container, we can consider it is safe to do so

```
[traineegrp@formation vm-formation]$ ls -l /volumes/sonar/extensions/plugins/
total 30072
-rw-r--r--. 1 root      root      5459683 Jan 26 12:37 sonar-checkstyle-plugin-2.4.jar
-rw-r--r--. 1 root      root      7080266 Jan 26 12:37 sonar-findbugs-plugin-3.3.jar
-rw-r--r--. 1 root      root      1601168 Jan 26 12:37 sonar-github-plugin-1.1.jar
-rw-r--r--. 1 root      root      3005657 Jan 26 12:37 sonar-java-plugin-3.9.jar
-rw-r--r--. 1 root      root        31567 Jan 26 12:37 sonar-pitest-plugin-0.6.jar
-rw-r--r--. 1 root      root      4366424 Jan 26 12:37 sonar-pmd-plugin-2.5.jar
-rw-r--r--. 1 lgrangeau lgrangeau 2680676 Oct 12 16:44 sonar-scm-git-plugin-1.0.jar
-rw-r--r--. 1 lgrangeau lgrangeau 6556136 Oct 12 16:44 sonar-scm-svn-plugin-1.2.jar
```

Use case #6 : Details 2/12

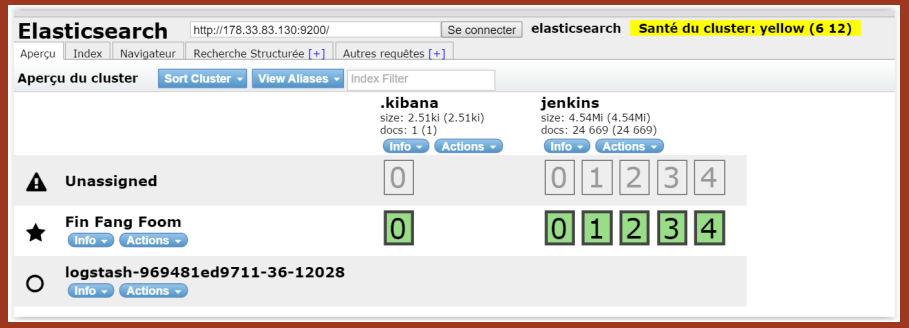
ELK :

- To install head plugin, the command-line is self-sufficient

```
docker exec -ti traineegrp-elk /usr/share/elasticsearch/bin/plugin -install mobz/elastics
```

```
[traineegrp@formation vm-formation]$ docker exec -ti traineegrp-elk /usr/share/elasticsearch/bin/plugin -install mobz/elasticsearch-head
-> Installing mobz/elasticsearch-head...
Trying https://github.com/mobz/elasticsearch-head/archive/master.zip...
Downloading ..........
.....
.....DONE
Installed mobz/elasticsearch-head into /usr/share/elasticsearch/plugins/head
```

This will allow to browse the elasticsearch indexes through a browser (not possible by default!) at http://<your_VM>:9200/_plugin/head/



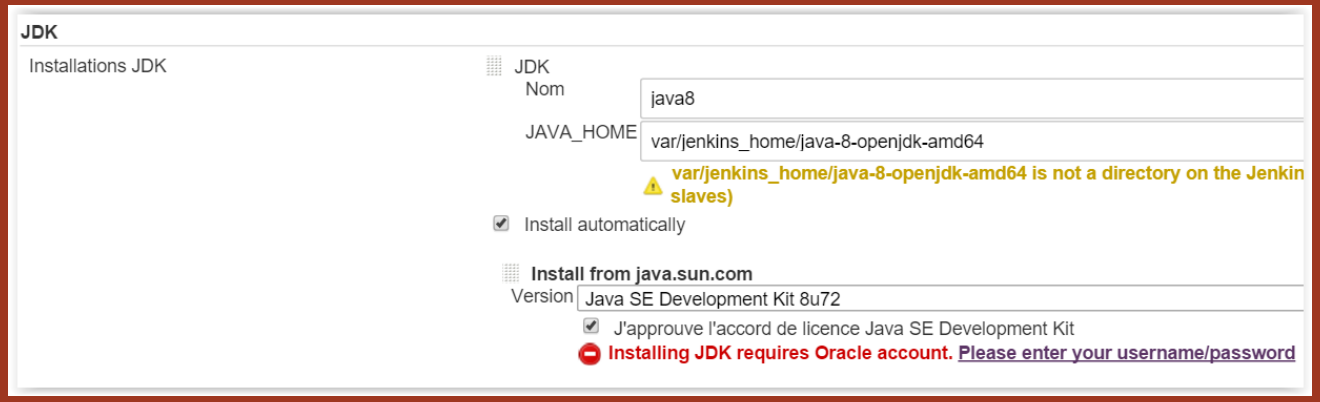
Use case #6 : Details 3/12

Jenkins:

- install the plugins using the Jenkins CLI

```
docker exec -ti traineegrp-jenkins java \
 -jar /var/jenkins_home/war/WEB-INF/jenkins-cli.jar -s http://localhost:8080 install-plugin
 git sonar htmlpublisher jobConfigHistory saferestart pitmutation envinject parameterized-t
 build-pipeline-plugin mask-passwords rebuild logstash metadata -restart
```

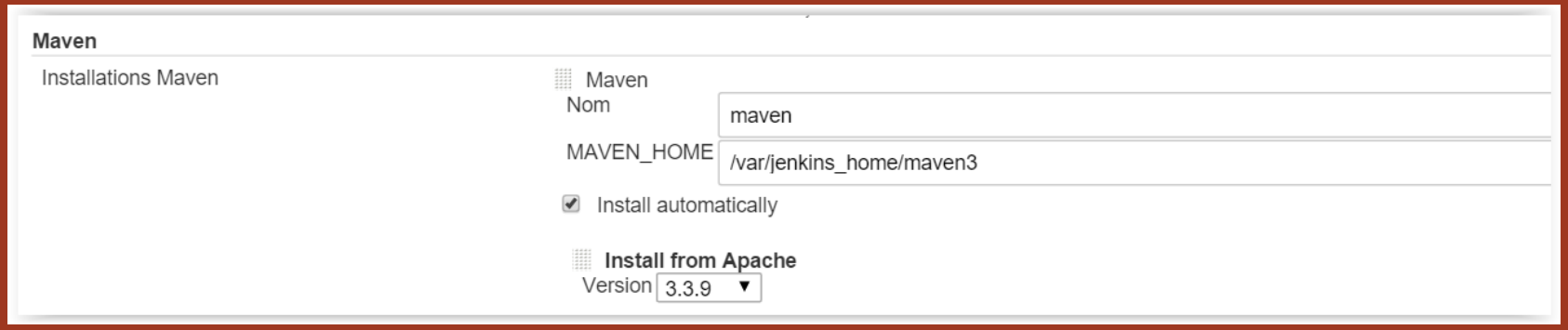
- Java can be "auto-installed" using the appropriate feature in Jenkins



- Oracle registration : https://login.oracle.com/mysso/signon.jsp

Use case #6 : Details 4/12

- Similarly, Maven can be auto-installed



- Protocol buffer cannot be installed as-is : the package must be installed specifically

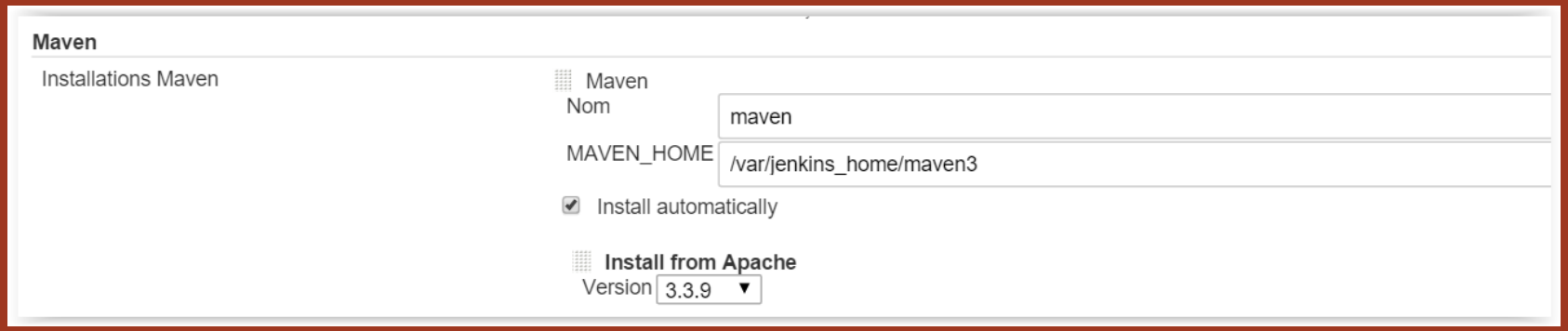
- The same for Gatling

- Allowing jenkins to run docker means install docker + specific configuration ...

Standard image is NOT enough for that anymore ...

Use case #6 : Details 5/12

- Similarly, Maven can be auto-installed



- Protocol buffer cannot be installed as-is : the package must be installed specifically

- The same for Gatling

- Allowing jenkins to run docker means install docker + specific configuration ...

Standard image is NOT enough for that anymore ...

Use case #6 : Details 6/12

- Modify the jenkins definition in docker-compose.yml file as-is:

```
jenkins:
  build: ./01-custom-jenkins
  ports:
   - "8080:8080"
  volumes:
   - /volumes/jenkins:/var/jenkins_home
   - /var/run/docker.sock:/var/run/docker.sock
  container_name: traineegrp-jenkins
  privileged: true
  links:
  - sonar
  - nexus
  - elk
```

Jenkins service will now be built not run from an existing image

The docker.sock share and privileged options allows jenkins to run containers on the host itself instead of inside jenkins itself

Use case #6 : Details 7/12

- Create a folder 01-custom-jenkins and a file Dockerfile in it:

```
FROM jenkins:1.625.3

# NEED TO BE ROOT FOR APT INSTALLS
USER root

# MODULES INSTALL : GIT, PROTOBUF, PUPPET, SUDO
RUN apt-get update && apt-get install -y git protobuf-compiler apt-transport-https sudo wge

# DOCKER REPOS SETUP - YEAH, DOCKER WITH DOCKER ;)
RUN apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys 58118E89F3A912
    echo "deb https://apt.dockerproject.org/repo debian-jessie main" > /etc/apt/sources.lis
    apt-get update && apt-cache policy docker-engine

# INSTALL AND ACTIVATE DOCKER
RUN apt-get install -y docker-engine && \
    adduser jenkins sudo && \
    echo "jenkins ALL=NOPASSWD:ALL" | (EDITOR="tee -a" visudo) && usermod -aG docker jenkin

# GATLING INSTALL "A LA MANO"
RUN wget https://oss.sonatype.org/content/repositories/releases/io/gatling/highcharts/gatli
  unzip gatling-charts-highcharts-bundle-2.1.7-bundle.zip

# GET BACK TO STANDARD USER
USER jenkins
```

Use case #6 : Details 8/12

This describes an new Jenkins image overriding official one with:

- additional tools installed with aptitude (ubuntu originated)

- registering of ubuntu packages for docker

- install of the docker-engine allowed for jenkins user as sudo (required to launch containers on the host VM itself)

- Gatling distribution installed

Use case #6 : Details 9/12

- Now, let's build the factory ... `docker-compose build`

Jenkins image is built from the Dockerfile specification layer per layer

```
[traineegrp@formation FACTORY_SETUP]$ docker-compose build
elk uses an image, skipping
sonar uses an image, skipping
nexus uses an image, skipping
Building jenkins
Step 1 : FROM jenkins:1.625.3
 ---> eb09f4d99ddf
Step 2 : USER root
 ---> Using cache
 ---> 34eafc41cf78
Step 3 : RUN apt-get update && apt-get install -y git protobuf-compiler apt-transport-https sudo wget telnet vim
 ---> Running in 56835c13fb2a
Get:1 http://security.debian.org jessie/updates InRelease [63.1 kB]
Ign http://httpredir.debian.org jessie InRelease
Get:2 http://httpredir.debian.org jessie-updates InRelease [136 kB]
Get:3 http://httpredir.debian.org jessie-backports InRelease [166 kB]
```

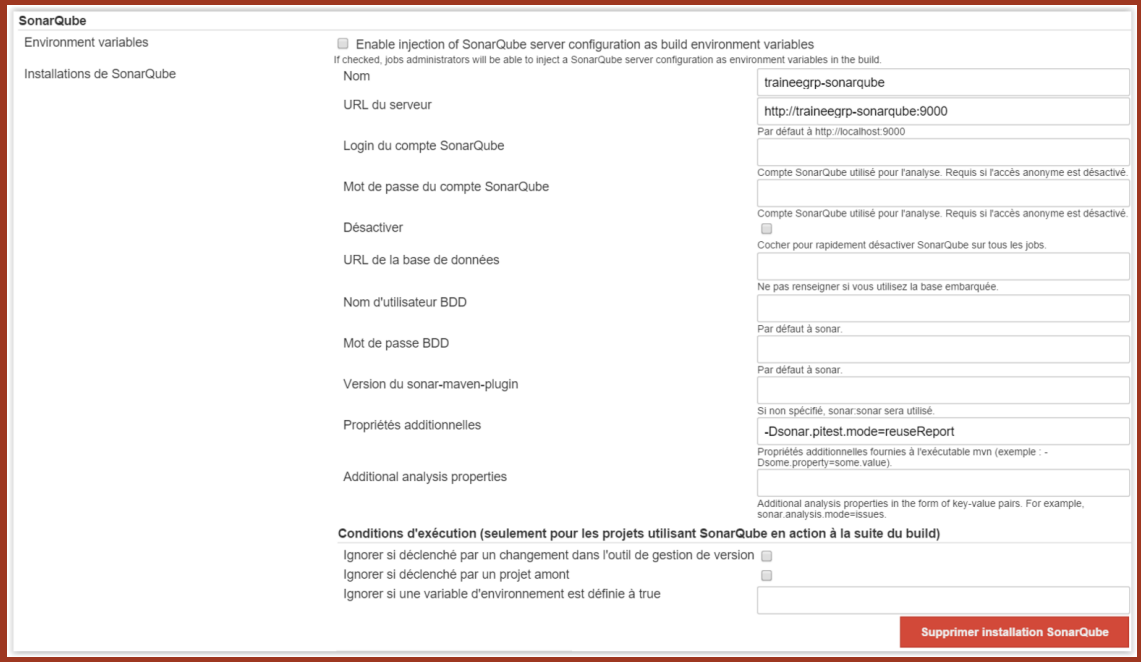
Now, restart the factory : docker-compose knows that Jenkins container is obsolete and runs it

instead of restarting old one

```
[traineegrp@formation FACTORY_SETUP]$ docker-compose up
Starting traineegrp-elk
Starting traineegrp-sonarqube
Starting traineegrp-nexus
Recreating traineegrp-jenkins
```

Use case #6 : Details 10/12

- Configure now the Sonarqube component in Jenkins admin UI



Thanks to the links, Jenkins accesses Sonarqube using its name, allowing code quality analyses pushed on demand

Use case #6 : Details 11/12

- Configure now the Logstash component in Jenkins admin UI

Once again, Jenkins accesses smoothly to ELK container using its name, allowing jobs status reporting in elasticsearch database

Use case #6 : Details 12/12

- Configure now the Slack component in Jenkins admin UI and test connexion

Take the channel and integration tokens from your stash configuration

Thanks to the links, Jenkins accesses Slack to notify instantly the build events activated on the jobs (start/stop/failed...)

Use case #7 : Create QUALIFY job

Create a freestyle job called "QUALIFY" which:

- Sets the value of PIPELINE_VERSION to internal build number and puts the version information on the application main page

- Builds/tests/packages and deploys the application in Nexus

- Publishes the Junit/Jgiven/Pitest reports in Jenkins UI and Sonarqube

- Pushes a new github tag : ATMSERVER-${PIPELINE_VERSION}

- Sends status and reports to ELK and Stash

Use case #7 : Tooltips

A fully installed an configured software factory is required here...

The best place to put the version is in the page title or in the banner

Build/tests should work in the same way as on your workstation

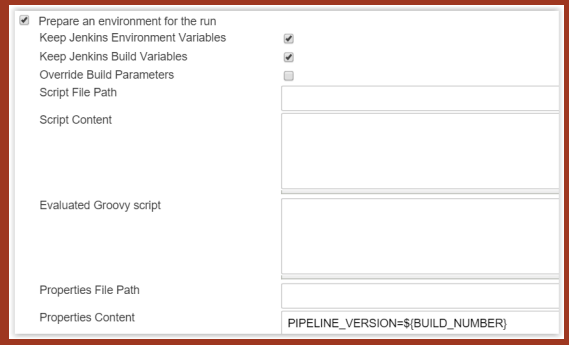
The deployment in Nexus can be done with HTTP connexion parameters to simplify the maven part (with password masking!)

No specific feature for Jgiven publish, use HTML instead (bugged)

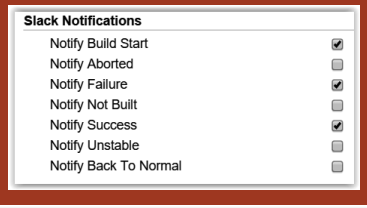
- The tag in github needs configuration to work

Use case #7 : Details 1/5

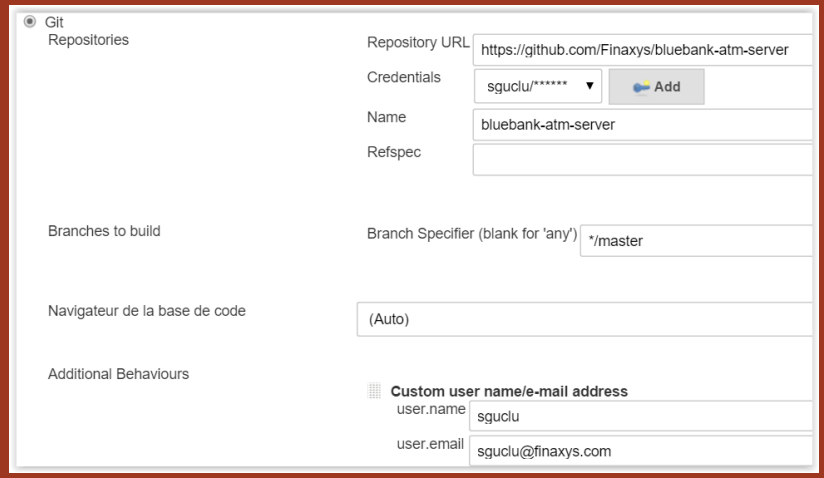
- The pipeline version is configured at the beginning using the Jenkins ${BUILD_NUMBER} variable



- Slack can be notified in case of a job starts/succeeds/fails

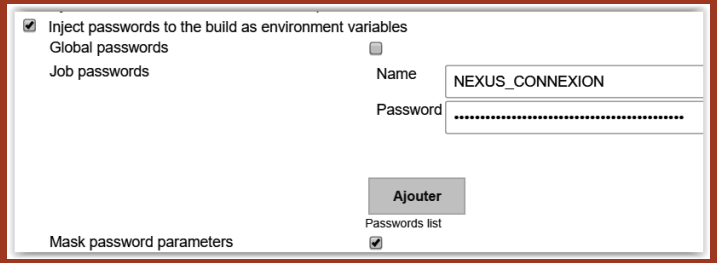


- To push the tag to Github, the SCM configuration requires a proper authentication/autorization on the forked repository URL, and also user name/email for Github checks

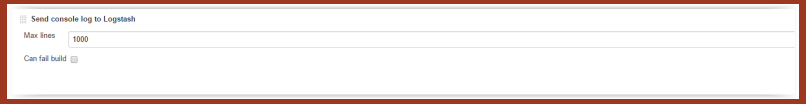


Use case #7 : Details 2/5

- HTTP connexion to Nexus defaults with "admin:admin123", however to hide this info from logs/UI, we inject and mask it



- Elasticsearch can be notified of all events related to this job (don't forget the post-task!)



- Using a shell step, here is a cheap way to inject versions in the application:

```
sed -i "s|<title></title>|<title>ATM-${PIPELINE_VERSION}</title>|g" node-client/index
sed -i "s|BLUEBANK ATM|BLUEBANK ATM v${PIPELINE_VERSION}|g" node-client/index.html
```

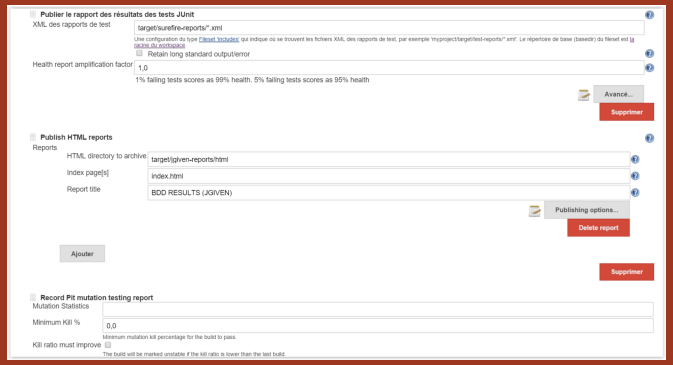
Use case #7 : Details 3/5

- The build is quite similar to the workstation one, except the deployment part using the password obfuscation

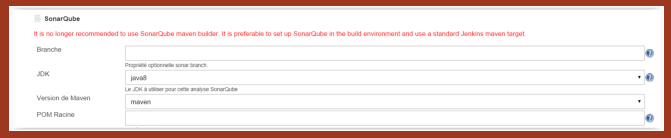
```
ean
ckage
sembly:single
ploy
all-tests jacoco:report org.pitest:pitest-maven:mutationCoverage
altDeploymentRepository=releases::default::http://${NEXUS_CONNEXION}@traineegrp-nexus:8081/content/repos
```

- Jenkins reports publish

is pretty self-explanatory

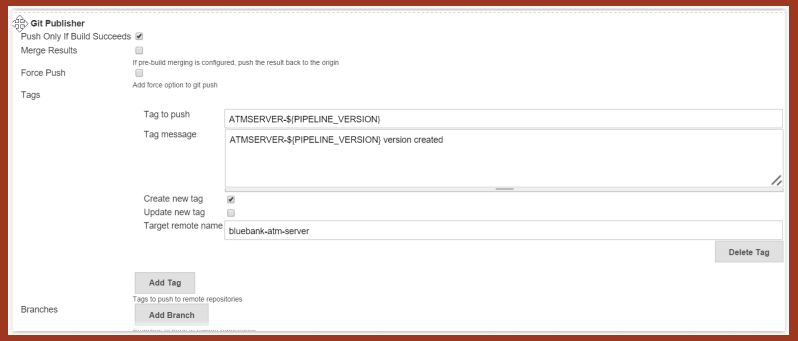


- Add sonarqube invocation

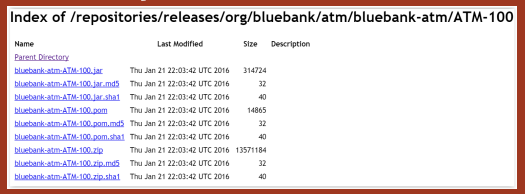


Use case #7 : Details 4/5

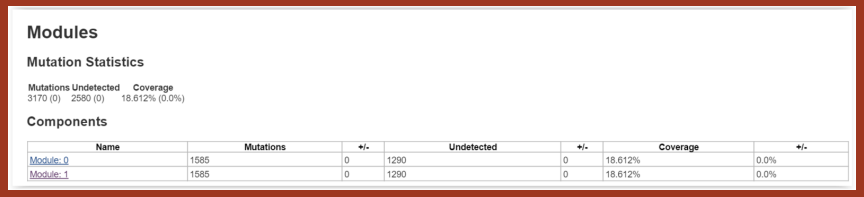
- We push only the tag based on the repository name (no code changed, even the HTML page)

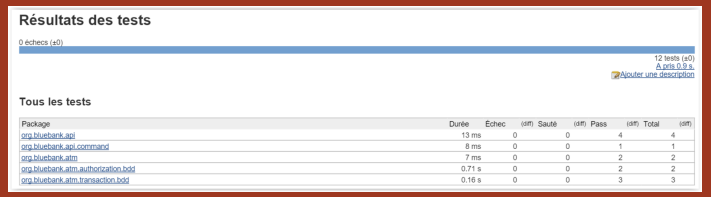


- Nexus URL (http://<your_VM>:8081/content/repositories/releases) should contain the binaries produced



- When executed, the build/tests must all pass as seen in the reports (for Jgiven, get the zip and open it extracted outside Jenkins)

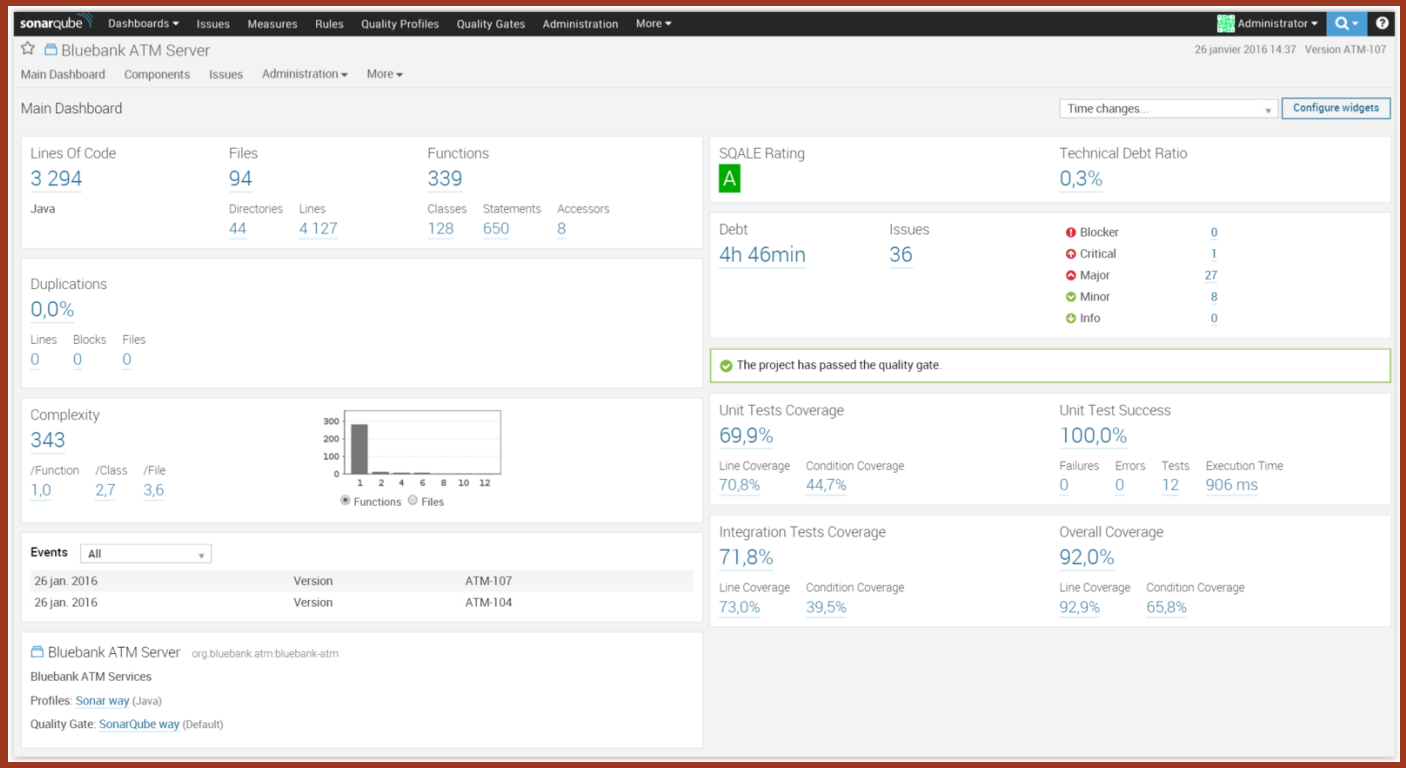




Use case #7 : Details 5/5

- On Sonarqube (http://<your_VM>:9000) you can add the Pitest and Integration tests widgets to have a full status of the code quality

(N.B. : bug on Pitest reports with Sonarqube 5.2 already known)



Use case #8 : Create PACKAGE job

Create a job called "PACKAGE" which:

- Run only if the "QUALIFY" passed, and use the same source/version

- Creates a docker container derived from "codenvy/debian_jdk8"

- Installs the tools required to run the ATM server (JVM) and client (web site)

- Retrieves, installs and run the application packaged from Nexus

- Sends status and reports to ELK and Slack

Use case #8 : Tooltips

The SCM, Logstash and Slack configuration are the same

A link between both jobs passing the PIPELINE_VERSION is required

To retrieve the binary, a "simple" wget is already present in the container

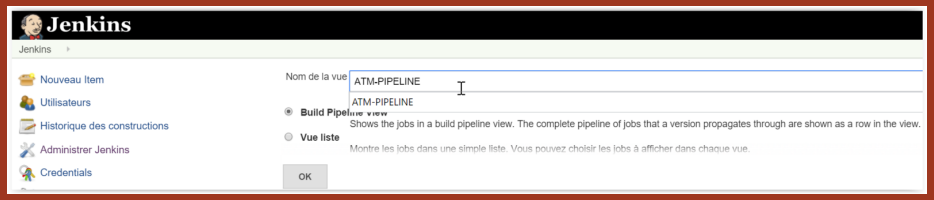
An HTTP client such a Nginx to install implies a custom Dockerfile

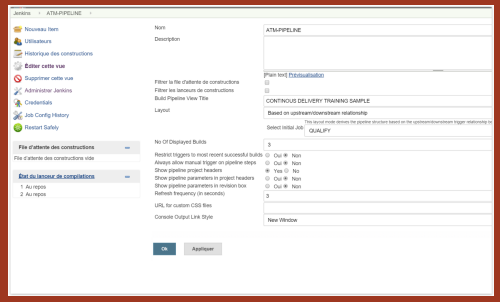
To run the container processes, we will need an entry point

To chain the jobs visually, a pipeline view is perfect ...

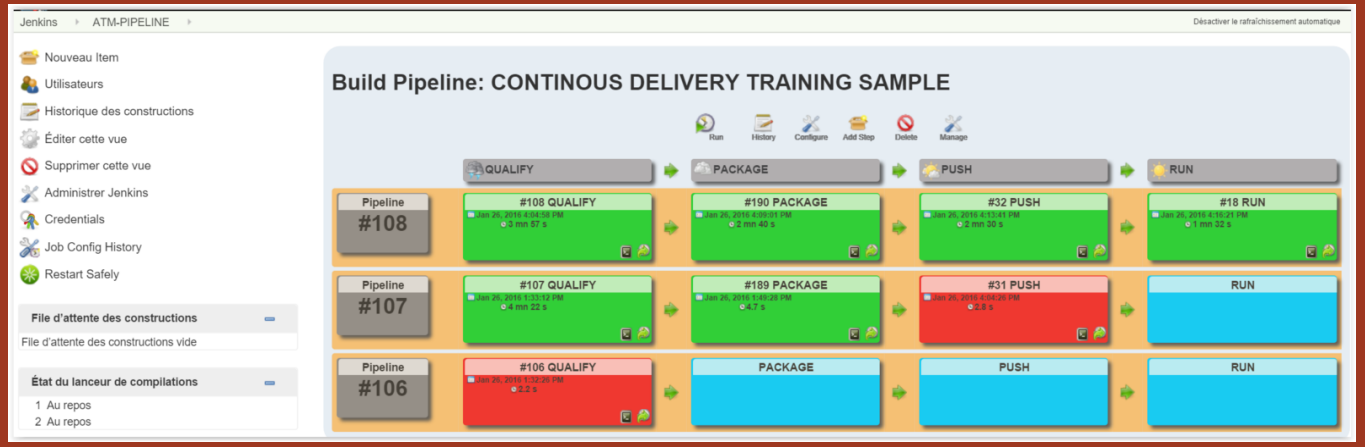
Use case #8 : Details 1/7

- First, create a pipeline view called "ATM-PIPELINE"n this will ease the next steps (set first job as "QUALIFY")





The target (if everything works as expected) :

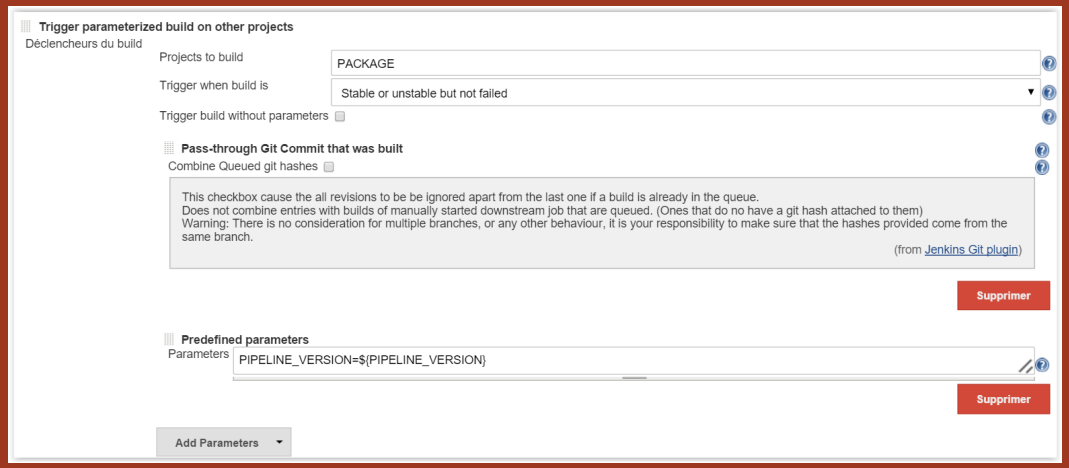


Use case #8 : Details 2/7

- Create now a new freestyle job called "PACKAGE"

- Reuse the SCM, Logstash, Slack configuration and save

- In QUALIFY, add a trigger to PACKAGE and pass PIPELINE_VERSION



- Save and go back to PACKAGE one

Use case #8 : Details 3/7

- Add a shell build step used to retrieve the binaries (scroll!)

```
wget --no-verbose http://traineegrp-nexus:8081/content/repositories/releases/org/bluebank/a
```

The tricky part is that:

- The Codenvy image does not have Nginx installed
- The application must be copied, extracted and run in the container

The https://github.com/nginxinc/docker-nginx.git shows how to install and run nginx, it is a good basis for our own container

On the Github repository, add a "Dockerfile" file on the root directory containing these entries (scroll!) :

Use case #8 : Details 4/7

```
FROM codenvy/debian_jdk8

USER root

RUN apt-key adv --keyserver hkp://pgp.mit.edu:80 --recv-keys 573BFD6B3D8FBC641079A6ABABF5BD
RUN echo "deb http://nginx.org/packages/mainline/debian/ jessie nginx" >> /etc/apt/sources.

# server ports (8180 for server and 80 for nginx)
EXPOSE 8180 80

RUN apt-get update && apt-get install -y ca-certificates nginx && rm -rf /var/lib/apt/lists

# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log && ln -sf /dev/stderr /var/log/nginx/error

# GET ATM server/client code
COPY ./bluebank-atm.zip /tmp/

# extract the war to run server/UI in standalone
RUN rm -rf /tmp/ATM-SERVER && mkdir -p /tmp/ATM-SERVER && cd /tmp/ATM-SERVER && unzip /tmp/

# COPY THE UI DISTRIBUTION IN NGINX HTTP FOLDER
RUN cp -r /tmp/ATM-SERVER/node-client/* /usr/share/nginx/html/

# GET entry point script running server and client binaries
COPY ./atm-startup.sh /tmp/
RUN whoami && ls -l /tmp/atm-startup.sh && chmod +x /tmp/atm-startup.sh
```

Use case #8 : Details 5/7

This configuration :

- Adds Nginx package sources to authorized list
- Exposes the HTTP port (80) and the ATM server port (8180)
- Retrieves and install Nginx tool
- Copy the ATM bundle from the workspace to the container
- Extracts the bundle (server and UI) and copies the UI in Nginx
- Copies the entry point script from the workspace to the container
- Associates it with the image (called by docker run/start)

To be able to run as expected, add near the Dockerfile the entry point script atm-startup.sh :

Use case #8 : Details 6/7

```
#! /bin/bash

# ATM run server
cd /tmp/ATM-SERVER/lib
runningJar=$(ls bluebank-atm-*.jar)

echo " -- launching ATM CLIENT (NGINX) : nginx"
nginx

echo " -- launching ATM SERVER : java -jar ${runningJar} org.bluebank.Application"
java -jar ${runningJar} org.bluebank.Application
```

This script find and runs the auto-executable jar and nginx

-> UI should be available on http 80 port

-> Server should be available on websocket 8081 port

- Commit and push these files in the github project

Now we have to build and run it for real...

Use case #8 : Details 7/7

- Set a password variable (value is your Docker hub login)

- Add a shell step building the image (sudo required)

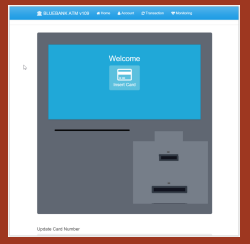
```
sudo docker build --tag=${LOGIN_DOCKER}/atm:${PIPELINE_VERSION} .
```

- Add another shell step running this image in a container

```
sudo docker run --name ${LOGIN_DOCKER}-atm-${PIPELINE_VERSION} -itd -p 80:80 -p
  ${LOGIN_DOCKER}/atm:${PIPELINE_VERSION}
```

- Launch the whole pipeline ...

The ATM server should be available on http://<your_VM>



Use case #9 : Create PUSH job

Create a job called "PUSH" which:

- Stop/removes the ATM container tested before

- Adds a "latest" tag ATM image for continuous updates

- Logs on the your public Docker Hub registry

- Pushes the ATM image to the public Docker Hub registry

- Sends status and reports to ELK and Slack

Use case #9 : Tooltips

Latest is a similar concept as maven snapshots

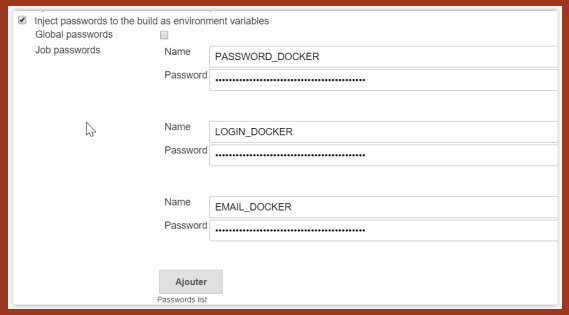
This job will be set as manual in the pipeline
=> publication requires manual approval process

Use case #9 : Details 1/2

- Create a new freestyle job called "PUSH"

- Activate Logstash and Slack pushes as before

- Define your Docker user/password/email as password entries



- Create a shell step used to stop/remove the previous container

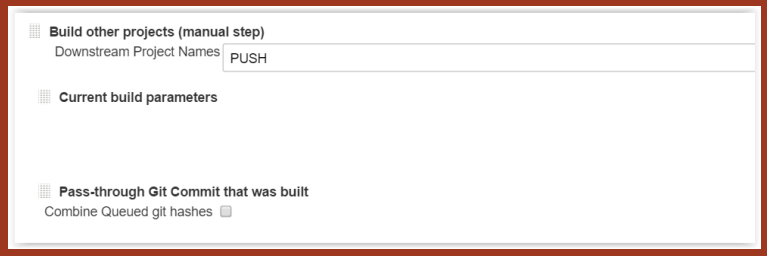
```
sudo docker stop ${LOGIN_DOCKER}-atm-${PIPELINE_VERSION} || true
sudo docker rm   ${LOGIN_DOCKER}-atm-${PIPELINE_VERSION} || true
```

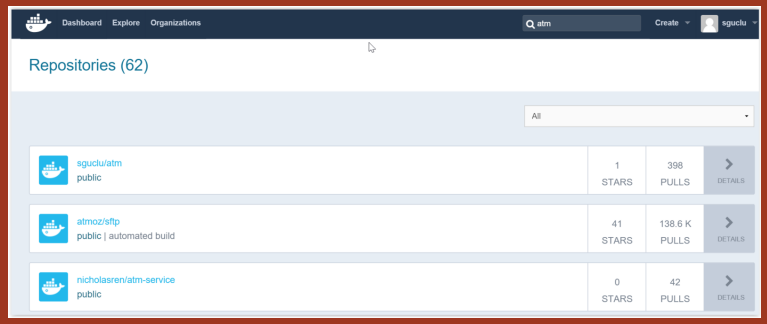
Use case #9 : Details 2/2

- Add a new shell step to tag the image as "latest", then log on and push the images to your public space on Docker hub

```
sudo docker tag -f ${LOGIN_DOCKER}/atm:${PIPELINE_VERSION} ${LOGIN_DOCKER}/atm:latest
sudo docker login --username=${LOGIN_DOCKER} --password=${PASSWORD_DOCKER} --email=${EMA
sudo docker push ${LOGIN_DOCKER}/atm
```

- Create a manual downstream link between PACKAGE and PUSH jobs => application is exposed only after manual approval process





- Run the job ... and voila

Use case #10 : Create RUN job

Configure your tutum account and "cloud" virtual machine

Create a job called "RUN" which:

- Adds a new project/tag label on the latest image

- Pushed this image to a docker cloud private repository

- Deploys automatically new images on the "run" VM

- Sends status and reports to ELK and Stash

Use case #10 : Tooltips

Docker Cloud is a container deployment and runtime service

- Formerly known as Tutum, acquired by Docker inc. in 2015

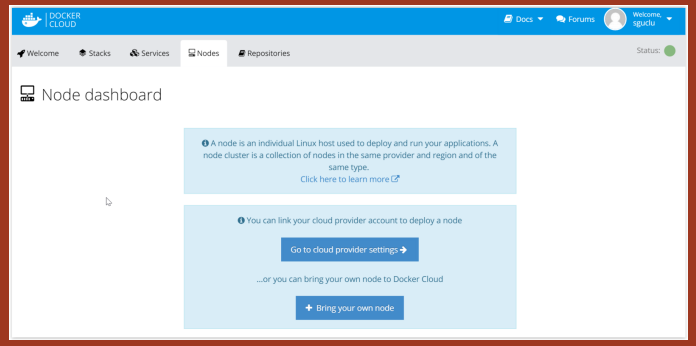
- Provides its own private registry

- Can address multiple clouds (AWS, Azure) including private ones ("Bring Your Own Cloud" feature)

- Provides many cool features to manage a docker-based infrastructure : scalability on the fly, auto-redeployments ...

Use case #10 : Details 1/5

- Create an account on https://cloud.docker.com with your Docker IDs



- On the "Bring your own node" tab, declare your "run" virtual machine

- You will have to run the command provided in the UI to install the tutum agent on the "run" VM

```
curl -Ls https://get.cloud.docker.com/ | sudo -H sh -s **********
```

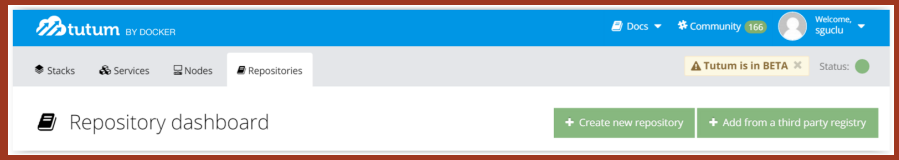
If something goes wrong, install can be removed before retry :

```
apt-get remove dockercloud-agent && rm -rf /etc/dockercloud-ag
```

Use case #10 : Details 2/5

- After a while, the node should be available to accept containers

- Create a private repository called <your_LOGIN>/atmcloud: it will host the ATM images with autodeploy features



- On Jenkins, create a new freestyle job called "RUN"

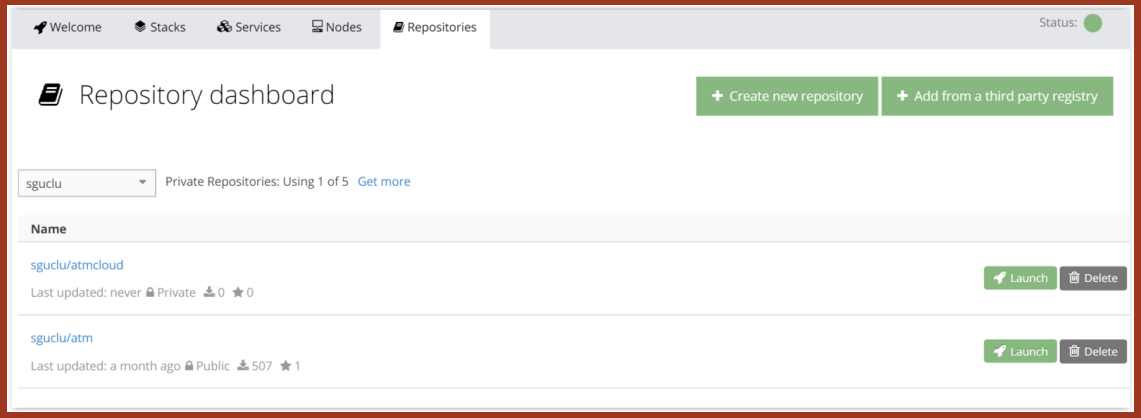
- Define the connexion parameters for Docker Cloud (same as Docker)

Use case #10 : Details 3/5

- Create a shell step used to tag the latest image with atmcloud name, log on docker cloud private registry and push the image

```
sudo docker tag -f ${LOGIN_DOCKER}/atm:latest ${LOGIN_DOCKER}/atmcloud:latest
sudo docker login --username=${LOGIN_DOCKER} --password=${PASSWORD_DOCKER} --email=${EMAIL_DOCKE
sudo docker push ${LOGIN_DOCKER}/atmcloud:latest
```

- Run the job : the latest ATM image should be on Tutum registry

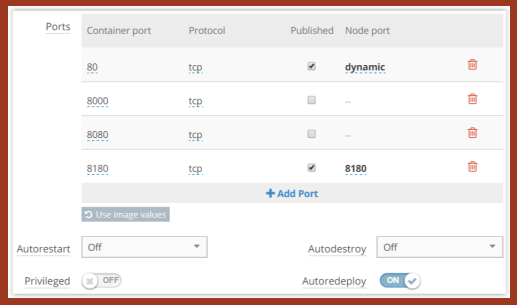


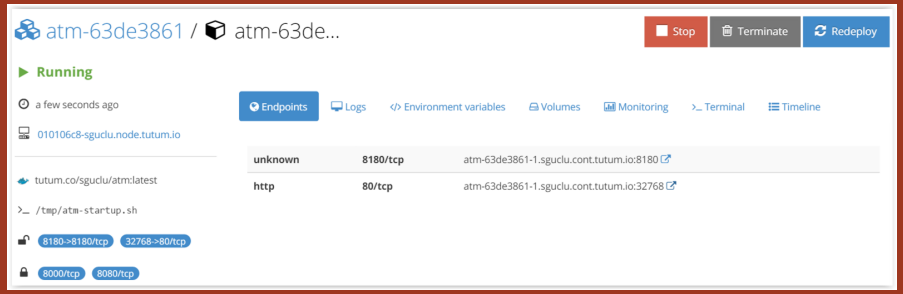
- We will now try to deploy manually this image ...

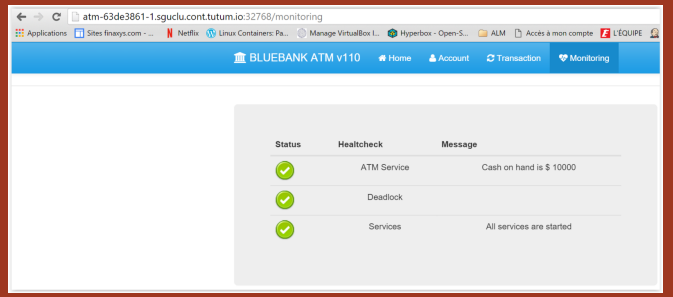
Use case #10 : Details 4/5

- Using the "Launch" button, try to run the container on the node with mapped 80 (32768)/8180 (8180) ports, and autoredeploy activated

- Select "Create and deploy" ... after a while the application should be online "on the cloud"







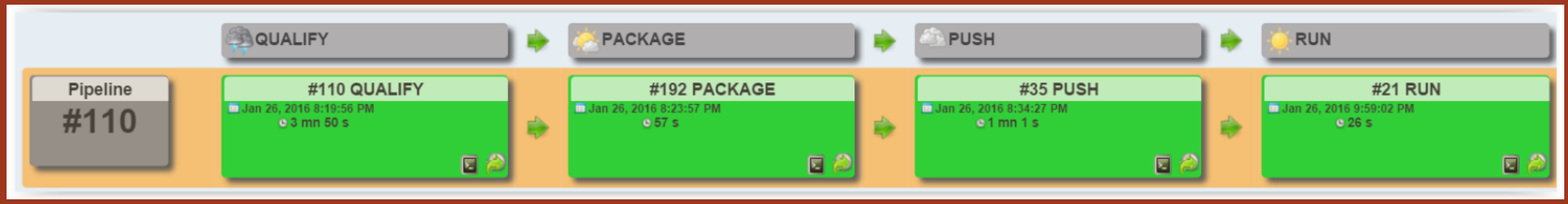
Now, let's automate the redeployment ...

Use case #10 : Details 5/5

- On the PUSH job, add RUN a downstream job

- Launch the pipeline, then launch manually the PUSH job :

ATM should be updated automatically on the "cloud" VM



Use case #11 : Performance test

This part will be performed on your workstation

For the most courageous/advanced, you can try to execute it in the pipeline too...

- Set up and record a Gatling scenario

- Play it against the ATM server packaged in PACKAGE job

Use case #11 : Tooltips

- Gatling provides a recorder to register a scenario (use browser in proxy mode)

- Once recorded (Scala code), the scenario can be replayed in standalone mode

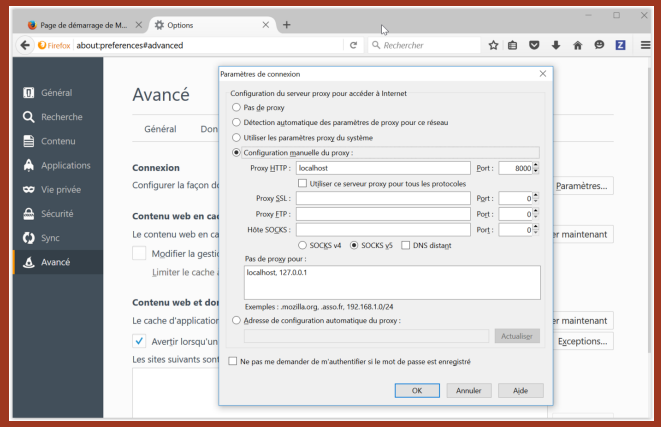
- Reports can be generated and displayed too

Use case #11 : Details 1/4

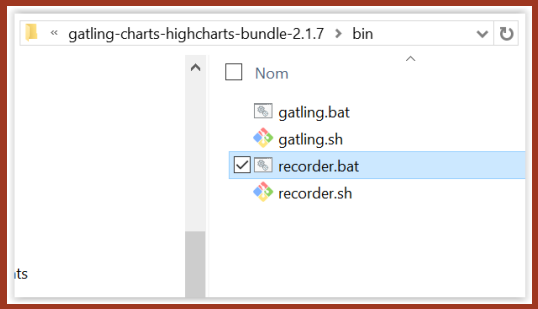
- Start the ATM server and the UI on your workstation

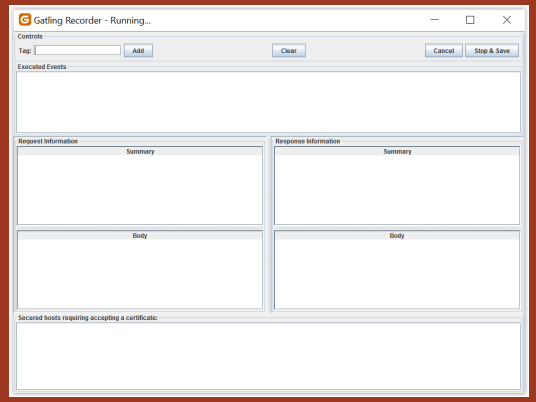
- Install gatling on your workstation

- Configure your browser to use a proxy on localhost, port 8000, e.g. F irefox



- Run the recorder in proxy mode



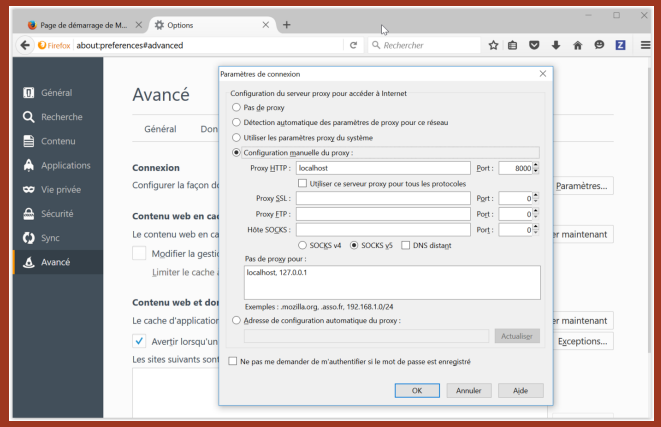


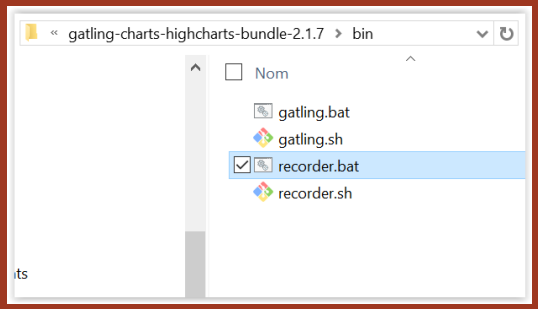
Use case #11 : Details 2/4

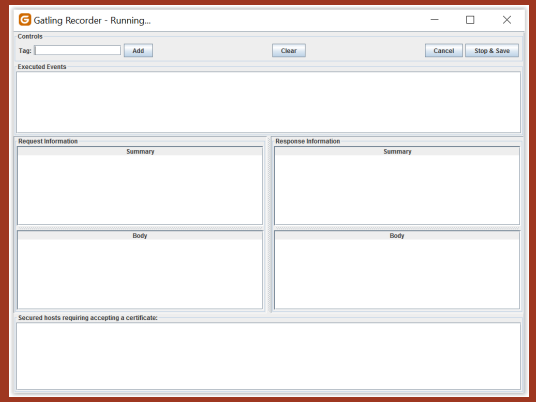
- Start the ATM server and the UI on your workstation

- Install gatling on your workstation

- Configure your browser to use a proxy on localhost, port 8000, e.g. F irefox

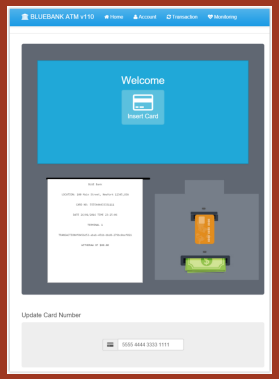






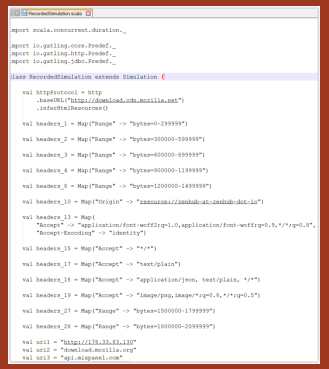
Use case #11 : Details 3/4

- Access the ATM server running on http://<your_VM> with the proxified browser



- Enter PIN (1234) and select a 80$ withdraw

- Stop the recording, and open the produced simulation file \user-files\simulations\RecordedSimulation.scala)



```
import scala.concurrent.duration._

import io.gatling.core.Predef._
import io.gatling.http.Predef._
import io.gatling.jdbc.Predef._

class RecordedSimulation extends Simulation {

  val httpProtocol = http
    .baseURL("http://download.cdn.mozilla.net")
    .inferHtmlResources()

  val headers_1 = Map("Range" -> "bytes=0-299999")

  val headers_2 = Map("Range" -> "bytes=300000-599999")

  val headers_3 = Map("Range" -> "bytes=600000-899999")

  val headers_4 = Map("Range" -> "bytes=900000-1199999")

  val headers_8 = Map("Range" -> "bytes=1200000-1499999")

  val headers_10 = Map("Origin" -> "resource://zenhub-at-zenhub-dot-io")

  val headers_13 = Map(
    "Accept" -> "application/font-woff2;q=1.0,application/font-woff;q=0.9,*/*;q=0.8",
    "Accept-Encoding" -> "identity")

  val headers_15 = Map("Accept" -> "*/*")

  val headers_17 = Map("Accept" -> "text/plain")

  val headers_18 = Map("Accept" -> "application/json, text/plain, */*")

  val headers_19 = Map("Accept" -> "image/png,image/*;q=0.8,*/*;q=0.5")

  val headers_27 = Map("Range" -> "bytes=1500000-1799999")

  val headers_28 = Map("Range" -> "bytes=1800000-2099999")

  val uri1 = "http://178.33.83.130"
  val uri2 = "download.mozilla.org"
  val uri3 = "api.wisgamel.com"
```

- Using the recorded session, it is possible to run the simulation again and produce reports

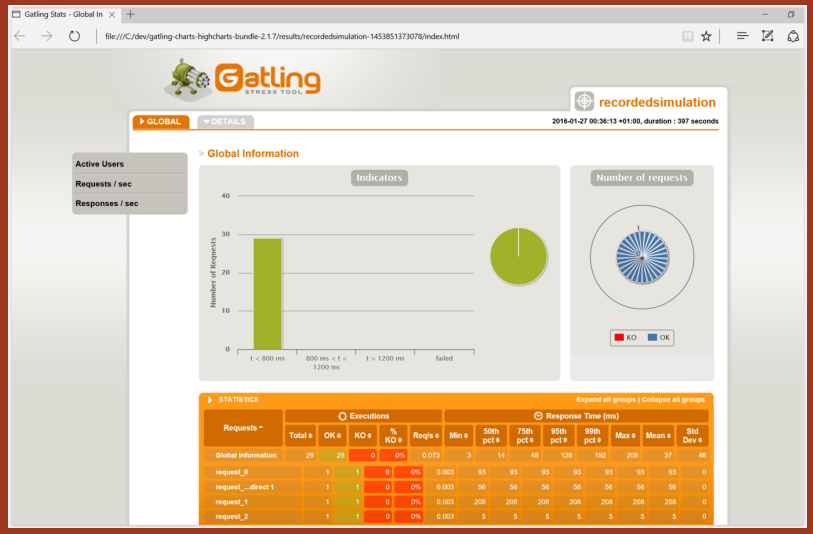
Use case #11 : Details 4/4

```
gatling --simulation RecordedSimulation
```

- Most interesting, gatling produces plenty reports about the tests performed

```
Reports generated in 2s.
Please open the following file: C:\dev\gatling-charts-highcharts-bundle-2.1.7\results\recordedsimulation-1453851373078\index.html
```

- Including scala and other recorded resources in the project source,
it is possible to play the scenario
in the pipeline (after container start
in PACKAGE job ?)



Use case #12 : Get some metrics

Based on the data collected previously, generate some reports about the activity around Jenkins per version produced

- History of successful steps

- Simplified time-to-release per version (cumulated duration between PACKAGE & RUN per version)

Use case #12 : Tooltips

Data collected by each build will be VERY useful here

Some sensitive data appear there : a filtering can be added using logstash before inserting into elasticsearch base (not covered here)

Kibana can process indexed data out-of-the-box as long as we define an index based on the timestamp

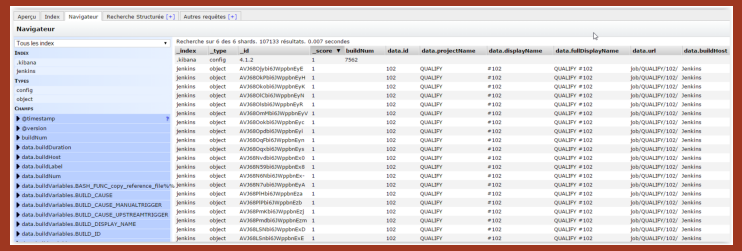
Use case #12 : Details 1/

Open elasticsearch "head" UI

- http://<your_VM>:9200/_plugin/head

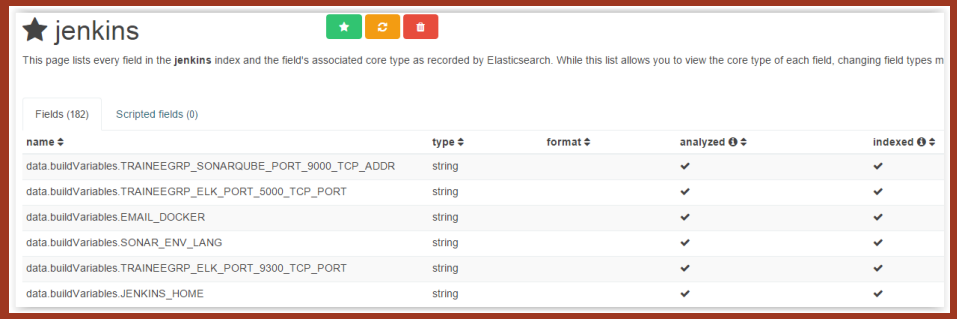
 => the index name to process is jenkins

 => the timestamp field is @timestamp



- Open now Kibana UI at http://<your_VM>:5601/
- Define an index pattern "jenkins" based on @timestamp and save

=> available fields are available now for reporting



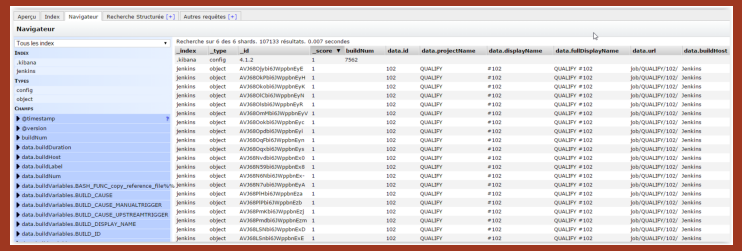
Use case #12 : Details 2/

Open elasticsearch "head" UI

- http://<your_VM>:9200/_plugin/head

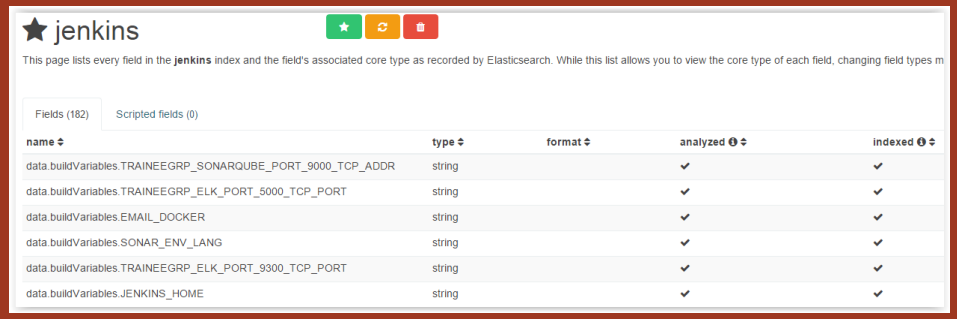
 => the index name to process is jenkins

 => the timestamp field is @timestamp



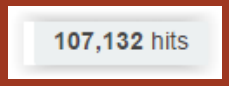
- Open now Kibana UI at http://<your_VM>:5601/
- Define an index pattern "jenkins" based on @timestamp and save

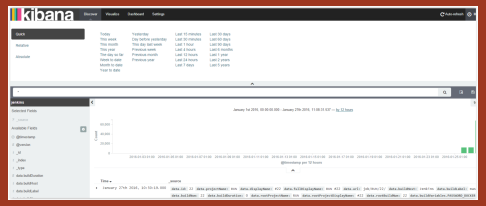
=> available fields are available now for reporting



Use case #12 : Details 3/

Warning : some sensitive data appear!

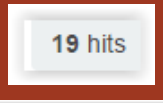


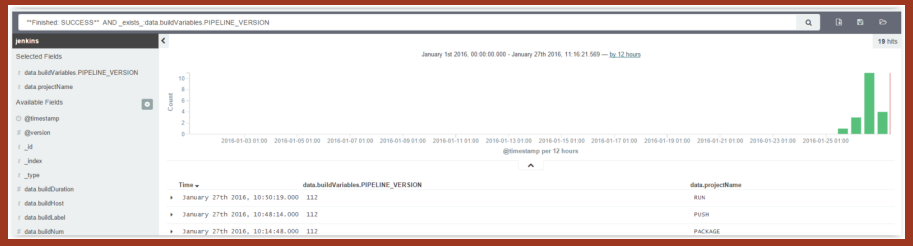


- Update the query field (*) by this one and refresh the search

```
"*Finished: SUCCESS*"  AND _exists_:data.buildVariables.PIPELINE_VE
```

=> Few lines should appear now (only successful builds)





- Pick the following fields:

data.projectName

data.buildVariables.PIPELINE_VERSION

- Save the query HISTORY_PER_VERSION => successful builds/version

Use case #12 : Details 4/

- Update the query field (*) by this one and refresh the search

```
("*Finished: FAILED*" OR "*Finished: SUCCESS*") AND _exists_:data.buildVariables.PIPELINE
```

- Put the following fields :

data.buildVariables.PIPELINE_VERSION

data.buildDuration

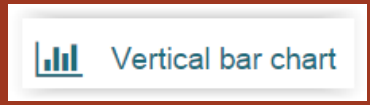
data.projectName

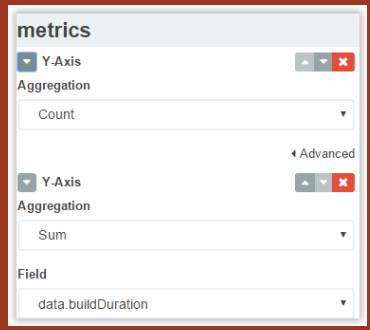
- Save the search as as **FINISHED_BUILDS_DURATION**

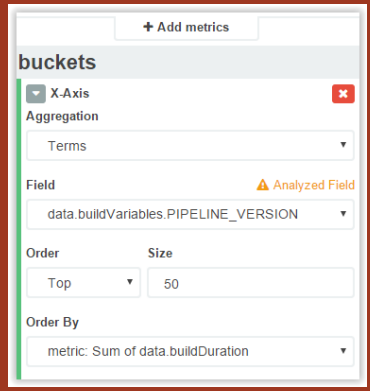
=> Duration steps for a version on PACKAGE/PUSH/RUN steps

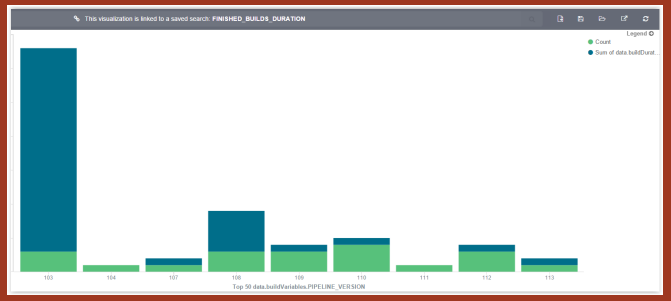
Use case #12 : Details 5/6

- Create a visualization of type "Vertical Bar Chart"
- Set count on Y-axis
- Add an aggregation on Y-Axis,
summing data.build.Duration values
- As a bucket, aggregate X-Axis on the term
data.buildVariables.PIPELINE_VERSION
(keep top 50 values)
- Save the view as **BUILD_DURATION_PER_VERSION**









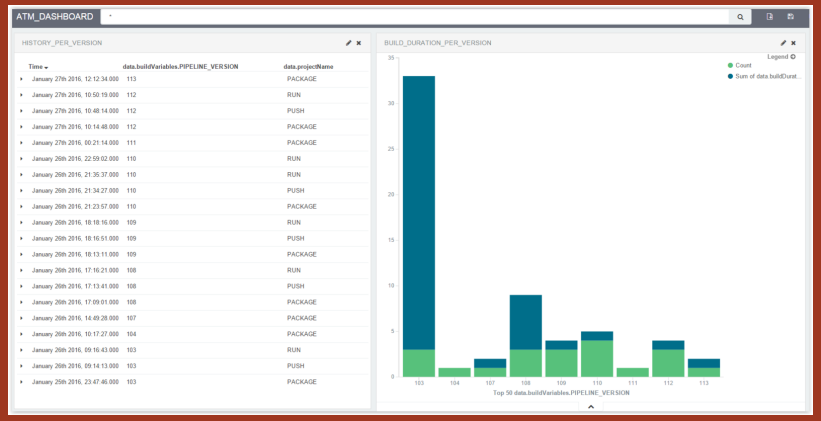
Use case #12 : Details 6/6

- Finally, create a dashboard including : HISTORY_PER_VERSION query on left and BUILD_DURATION_PER_VERSION visualization on right

-Save it as ATM_DASHBOARD ... and voila

Not a really accurate measurement (QUALIFY step not measured, nor time to qualify the issue ...) but a way to keep an eye on the release efficiency

Going further needs data process during collection (e.g. logstash)



Use case #13 : Implement a feature

Based on the toolset and practices, implement the following controls:

- Prevents negative accounts (refuse withdraw if > balance)
- Prevents banqueroute (refuse withdraw if > bank balance)

Define the BDD scenario, the tests to perform, the code to implement

Implement, test and validate everything using your software factory

Package and deploy on public registry and on cloud

Use case #13 : Tooltips

The BDD scenario for first business rule should be something like this:

```
No negative account
   Given the account balance is 100 dollars
     And the valid pin number has been entered
    When a withdraw of 120.00 dollars is made
    Then the ATM rejects the withdraw
     And the balance is 100 dollars
     And the card is returned
     And a receipt is printed

      | LOCATION: 100 Main Street, NewYork 12345,USA
      | CARD NO: 123456
      | DATE 01/07/2015 TIME 17:05:00
      | TERMINAL 47
      | TRANSACTION#00000000-0000-0001-0000-000000000002
      | WITHDRAW OF $120.00
      | --------- REJECTED : ONLY 100 USD LEFT ---------
```

Use case #13 : Details

This part is yours ....

Questions ?

Thank you