

Software Engineering and Architecture

Continuous Delivery, *getting our hands dirty*

Olivier Liechti

HEIG-VD

olivier.liechti@heig-vd.ch



MASTER OF SCIENCE
IN ENGINEERING

Quick recap (concepts)

Quick recap - what did we talk about so far?

- **Software is not a pack of bits that sits on a shelf.** *As responsible software engineers, we have to think about its entire lifecycle (operations, non-functional aspects, etc.).*
- **There is a growing realization of that issue in the industry.** *It has organizational, architectural and tooling impacts.*



Someone planted a seed.

Who is responsible to think about watering the plant?

Quick recap - what did we talk about so far?

- The **DevOps** movement captures a **big cultural change**. It emphasizes the continuous collaboration between software engineers and IT operations engineers.
- With DevOps, we want to **break the organizational silos** that exist between large “development” and “IT operations” department. We prefer to have **small cross-functional teams** and to give them **autonomy** and **responsibility**.



Quick recap - what did we talk about so far?

- DevOps is also associated with **automation**. Automation of every aspect of software production, validation and operations.
- Since the end of the nineties, developers know value of **continuous integration**. They value the ability to get a **daily** confirmation that their software modules can be **integrated** into a software system (avoid a lengthy “big bang” integration at the end of the project).



Quick recap - what did we talk about so far?

- **Continuous delivery** and **continuous deployment** go further:
 - With **continuous delivery**, we want to **quickly** know whether a **specific** change still means that our software **can be** deployed into production.
 - To have **confidence**, we need more than unit and integration tests. We don't have the time to rely on many manual tasks.
 - With **continuous deployment**, every “green” change is immediately and automatically deployed into production.



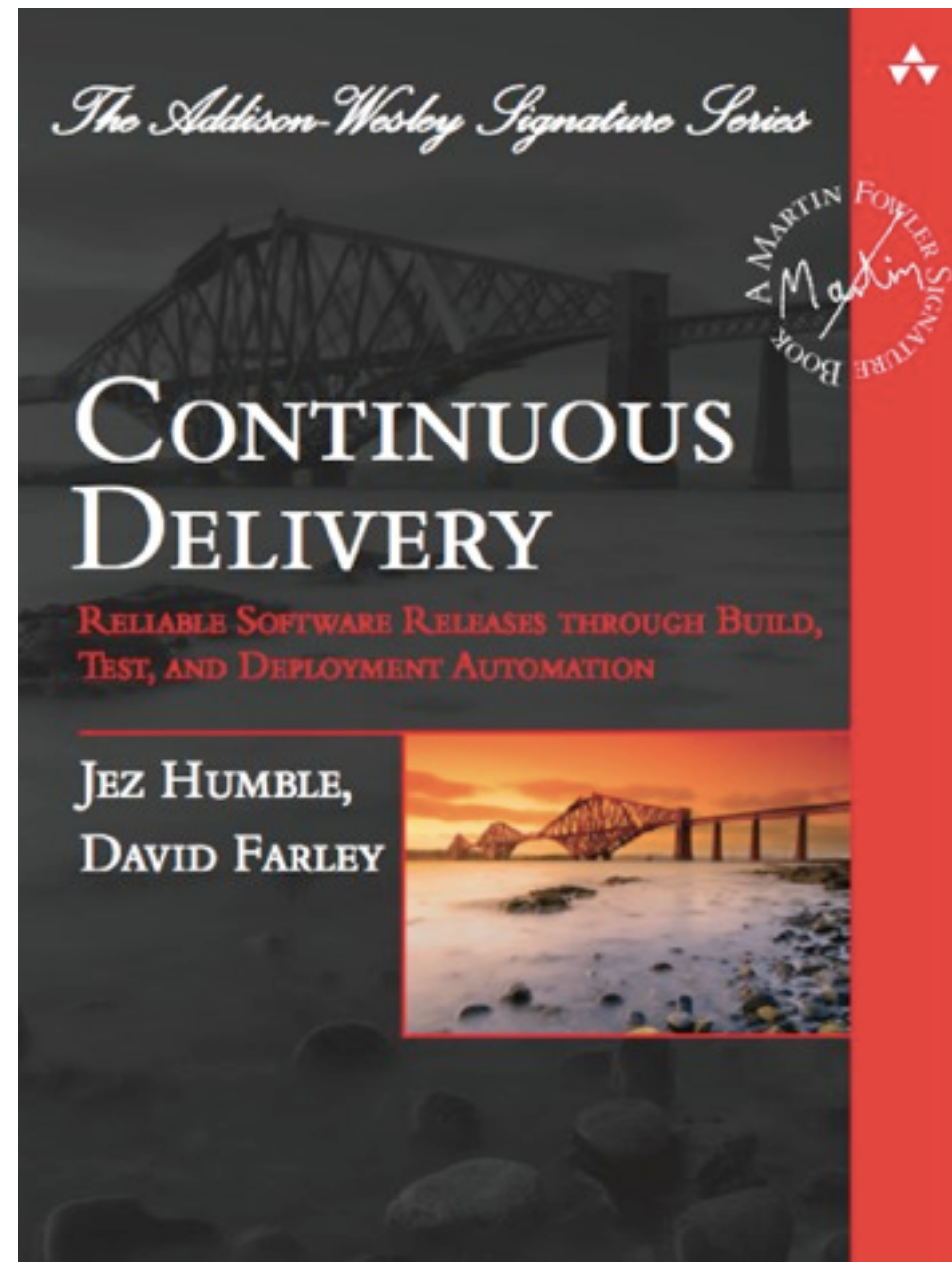
Quick recap - what did we talk about so far?

- **Most organizations** who evolve towards continuous delivery or continuous deployment build “**delivery pipelines**”.
- A delivery pipeline is made of several stages, where different tasks are performed. As many as possible of these tasks are automated.
- A **simple** delivery pipeline already has a lot of benefits. It saves time. It increases reproducibility and reliability.
- As time goes by, the delivery pipeline **evolves** and integrates **other types of tests**. These tests make it possible to evaluate **other aspects of the software quality**. Although non-functional testing is important, it is hard. This is one element that defines the **maturity level** of the continuous delivery process.



Figure 1.1 *The deployment pipeline*

Continuous Delivery



The slides are based on this book (ISBN:)

Free chapter: <http://www.informit.com/articles/article.aspx?p=1621865>

Exercise debrief

Exercise debrief

- By doing the simple exercise, we have seen a couple software engineering principles, techniques and tools in action (1):
 - As developers, you have received **executable specifications** in the form of unit tests. The name of the test methods were long and included the term “should”. This is an example of **Behavior Driven Development**, which we will discuss later during the semester.
 - The **JUnit framework** has been used to write and execute the unit tests. Equivalent frameworks exist for every programming language.

JUnit



Exercise debrief

- By doing the simple exercise, we have seen a couple software engineering principles, techniques and tools in action (2):
 - We have used **apache maven** as our “build tool”.
 - A build tool **drives** the “**micro-level**” construction of a system component. It needs a “recipe” to know what it should do. For maven, it is the pom.xml file.
 - A build tool **supervises** the first steps of the delivery pipeline: compilation, code analysis, unit testing, packaging, etc. The **real work**, however, is done in the supervised **plugins**.
 - Most build tool also take care of **dependencies**.
 - There other build tools for Java (e.g. Gradle) and there are build tools for other programming platforms (Grunt and Gulp for JavaScript, Rake for Ruby, etc.)
 - Today, we will introduce Jenkins, another tool which drives the process at the “**macro-level**” (i.e. we will see that Jenkins delegates work to maven).

Exercise debrief

- By doing the simple exercise, we have seen a couple software engineering principles, techniques and tools in action (3):
 - **The entire process has been automated:**
 - GitHub was used to host the branches of our codebase.
 - GitHub exposes a REST API, which means that we can write programs that identify branches, fetch them, identify contributors, etc.
 - It is therefore possible to automate (part of) the grading process. As a teacher, I can automatically run my unit tests against every single student solution and get the number of passing/failing tests.
 - This is the kind of things that you would do in a continuous delivery pipeline: use the APIs provided by your infrastructure to automate some of the construction, validation and deployment phases.

Today's objectives

Today's objectives

- **Implement the embryo** of our delivery pipeline. We will implement one part of the “commit stage”.
- **Introduce Docker as an enabling technology.** Today, it will allow use a continuous delivery server (Jenkins) without installing it ourselves on our machine.
- **Introduce Jenkins** as the orchestrator for our continuous delivery pipeline.
- **At the end, we want to have one Jenkins running on our laptop. We want to be able to open its Web UI and push a “Build button”, see that the last version of our “Instruments” project is fetched, compiled and unit tested.**



Figure 1.1 *The deployment pipeline*

The Deployment Pipeline

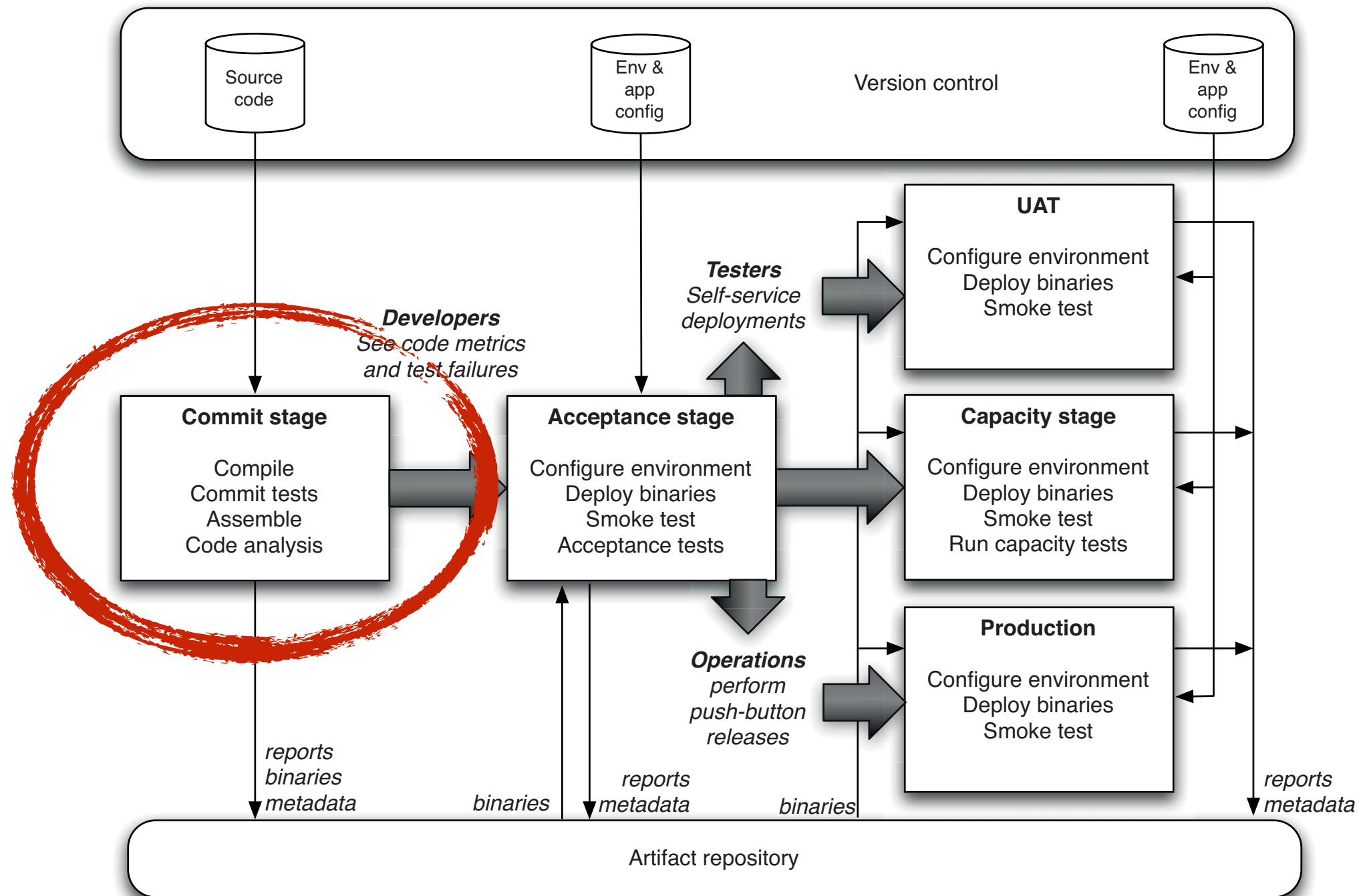


Figure 5.4 Basic deployment pipeline

Tools

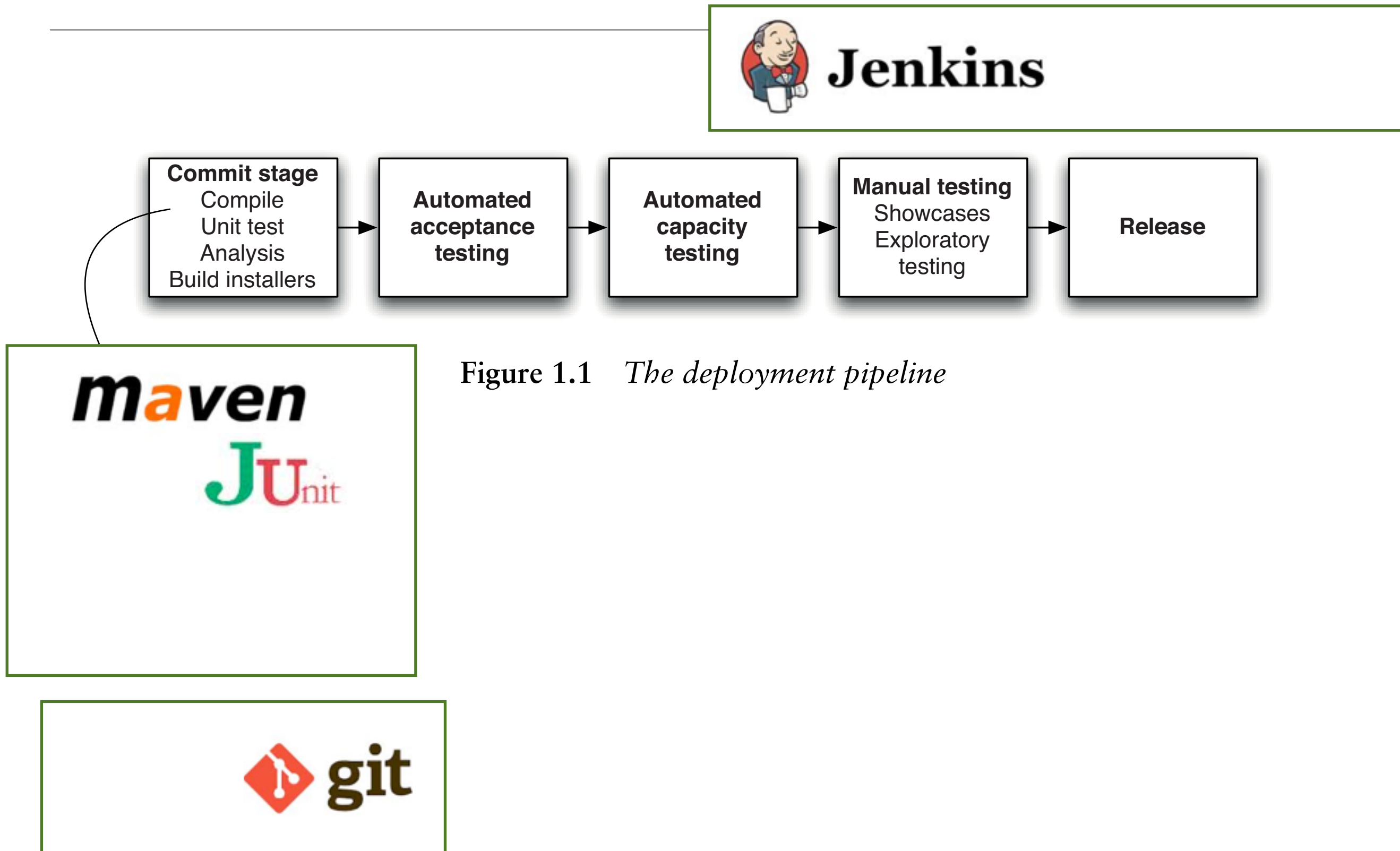
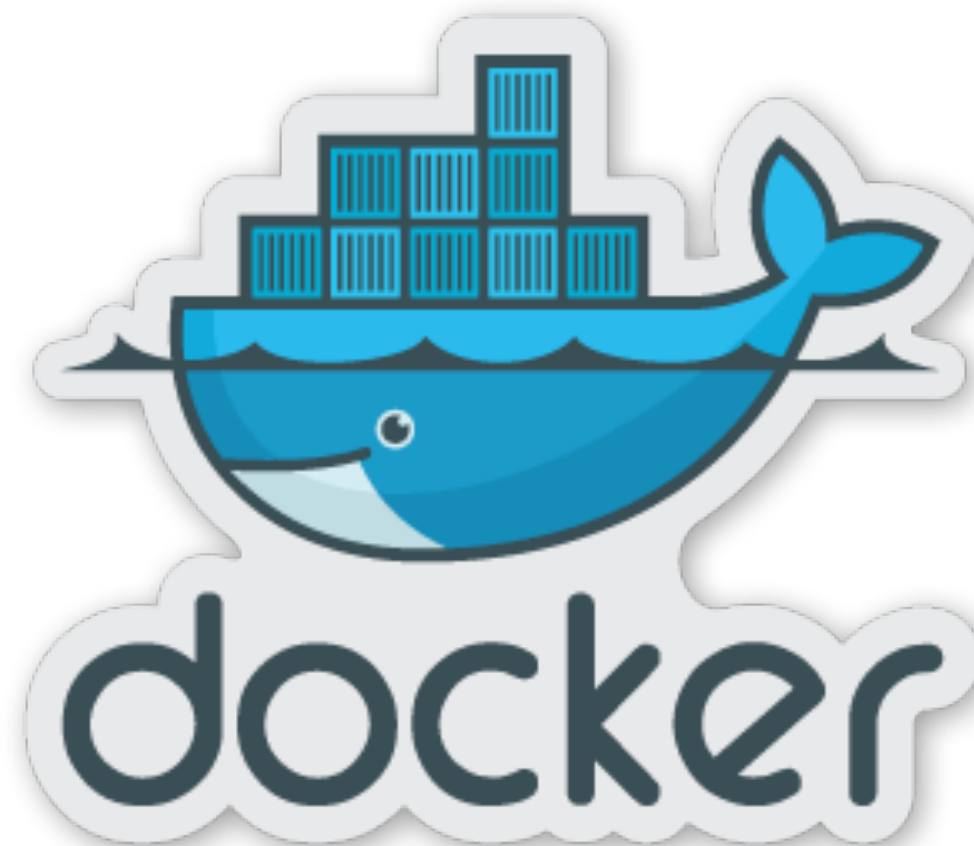
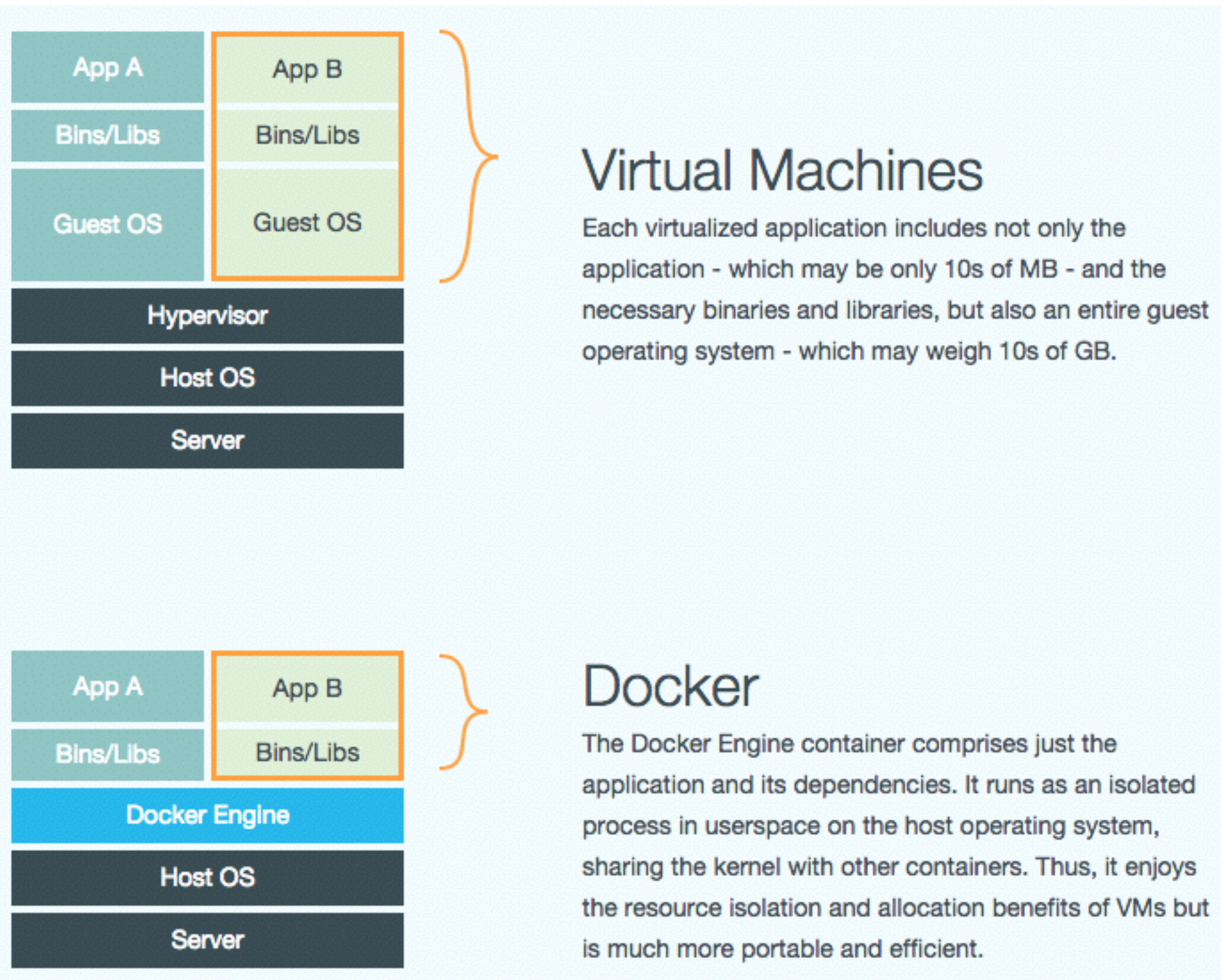


Figure 1.1 *The deployment pipeline*





<https://www.docker.com/whatisdocker/>



Docker image

Docker container

Why Docker in this course?

- Docker is a **lightweight virtualization technology**.
- It enables a new way for **packaging, distributing and running applications**.
- There are **different use cases** for the technology. Continuous delivery is one of the use cases.

Docker on your machine

- You have two options for using Docker:
 - Firstly, you can **use the Vagrant VM**. Docker is installed (and a couple of images have been downloaded already). Since the VM was prepared at the beginning of the semester, it is not the latest version of Docker.
 - You can install “**Docker Machine**” on your host. This also works if you have a Windows or a Mac OS machine. In this setup, Docker Machine actually uses a VM similar to the Vagrant VM (it is smaller).

Docker containers (1)

- You can think of a container as a **lightweight** virtual machine. Each container is isolated from the others and has its own IP address.
- Each container is created from a docker image (one could say that a container is a **running instance of an image**).
- There are **commands** to start, list, stop and delete containers.

```
# Start a container (more on this later)
```

```
$ docker run
```

```
# List running containers
```

```
$ docker ps
```

```
# List all containers
```

```
$ docker ps -a
```

```
# Delete a container
```

```
$ docker rm
```

```
# Display logs produced by a container
```

```
$ docker logs
```

Docker containers (2)

- With Docker, the philosophy is to have **one application-level service per container** (it is not a strict rule).
- With Docker, the philosophy is also to **run several (many) containers on the same machine**.
- If you think of a typical **Web Application infrastructure**, you would have one or more containers for the apache web server, one container for the database, one container for the reverse proxy, etc.
- With Docker, containers tend also to be **short-lived**. Each container has an **entry point**, i.e. a command that is executed at startup. When this command returns, the container is stopped (and will typically be removed).
- **If a container dies, it should not be a big deal**. Instead of trying to fix it, one will create a new one (from the same image).

Docker images (1)

- **A Docker image is a template**, which is used to create containers.
- Every image is **built from a base image** and adds its own configuration and content.
- With Vagrant, we use a file named Vagrantfile to configure and provision a Vagrant box. With Docker, we use a file name **Dockerfile** to create an image. The file contains statements (FROM, RUN, COPY, ADD, EXPOSE, CMD, VOLUME, etc.)
- Just like the community is sharing Vagrant boxes, **the community is sharing Docker images**. This happens on the Docker Hub registry (<https://registry.hub.docker.com/>).
- You can build docker images locally (with the `docker build` command). You can also build them on the **Docker Hub** cloud service.

Docker images (2)

- Here is an example for a Dockerfile

```
# This image is based on another image

FROM dockerfile/nodejs:latest

# For information: who maintains this Dockerfile?

MAINTAINER Olivier Liechti

# When we create the image, we copy files from the host into
# the image file system. This is NOT a shared folder!

COPY file_system /opt/res/

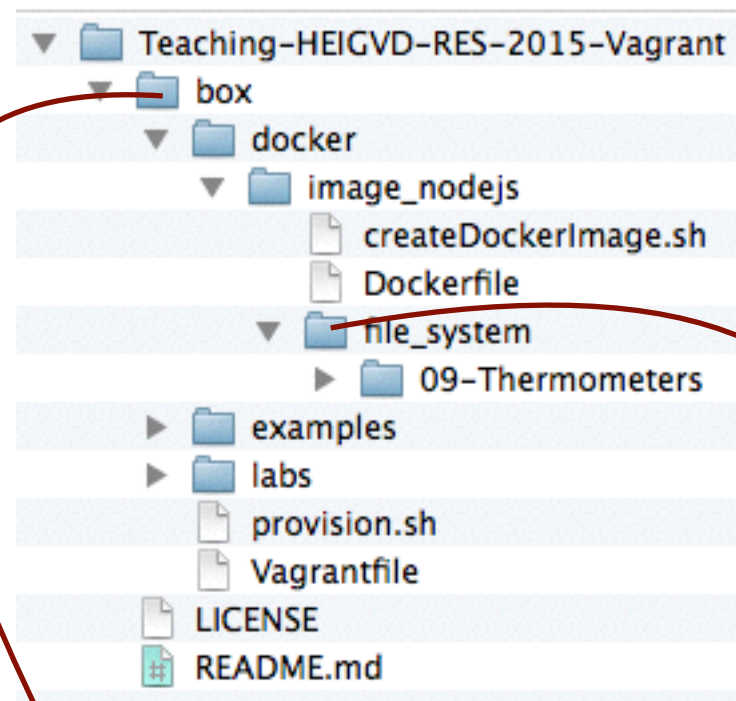
# With RUN, we can execute commands when we create the image. Here,
# we install the PM2 process manager

RUN npm install -g pm2@0.12.9
```

```
# Create an image from this Dockerfile
$ docker build -t heigvd/res-demo .

# Execute /bin/bash in a new container, created from the image
$ docker run -i -t heigvd/res-demo /bin/bash
```

shared folder

file system copy

Host (your laptop)

10.192.116.213

Vagrant box

/vagrant

192.168.42.42

172.17.42.1
docker0
bridge172.17.0.6
/opt/res
Docker
container172.17.0.2
/opt/res
Docker
container172.17.0.4
/opt/res
Docker
container172.17.0.1
/opt/res
Docker
container

Activity 1: use an existing Docker image (java server)

Use an existing Docker image

1. Find an image on **Docker Hub**
2. Understand that **the image packages an application** (read the **Dockerfile**).
Understand that this application is a TCP server written in Java.
3. Start a **container** (map the **container port** to a VM port)
4. **Interact with the server** running in the container.

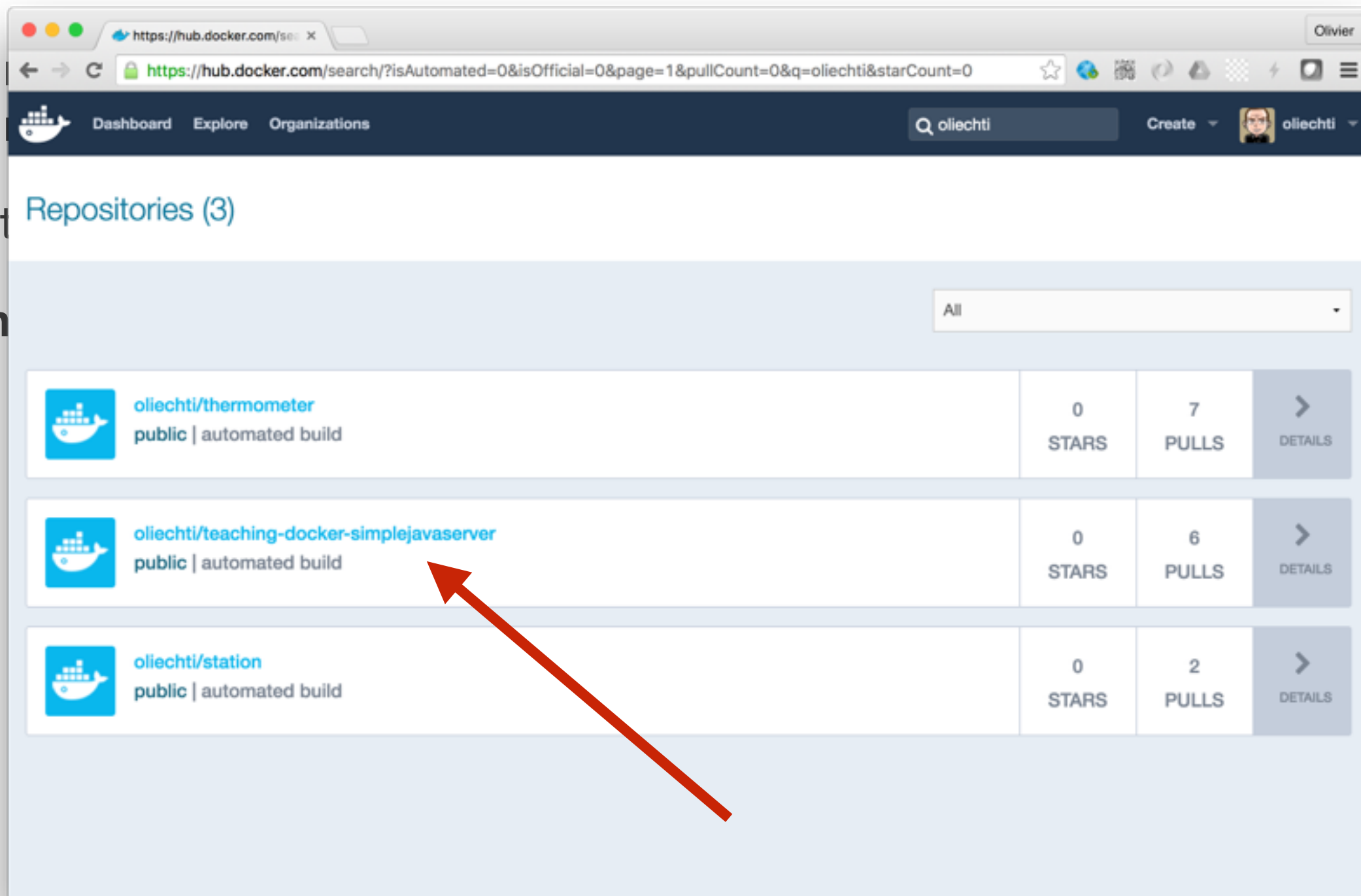
Use an existing Docker image

1. Find an image on **Docker Hub**

2. Use the image (e.g., `oliehti/thermometer` or `oliehti/teaching-docker-simplejavaserver`).

3. Start

4. In



Use an existing Docker image

1. Find an image on **Docker Hub**
2. Understand that **the image packages an application** (read the **Dockerfile**). Understand that this application is a TCP server written in Java.

```
FROM java:8
MAINTAINER Olivier Liechti <olivier.liechti@heig-vd.ch>

#
# When we build the image, we copy the executable jar in the image file system.
#
COPY StreamingTimeServer-1.0-SNAPSHOT-standalone.jar /opt/app/server.jar

#
# Our application will accept TCP connections on port 2205. Note that the EXPOSE statement
# does not make the container accessible via the host. For the container to really be accessible,
# we must either use the -p or the -P flag when using the docker run command. With -p, we can
# specify an explicit port mapping (and EXPOSE is not even required). With -P, we let Docker
# assign random ports for each EXPOSEd port. We can then use the docker port command to know the port
# numbers that have been selected.
#
EXPOSE 2205

#
# This is the command that is executed when the Docker container starts
#
CMD ["java", "-jar", "/opt/app/server.jar"]
```

Use an existing Docker image

1. Find an image on **Docker Hub**
2. Understand that **the image packages an application** (read the **Dockerfile**). Understand that this application is a TCP server written in Java.
3. Start a **container** (map the **container port** to a VM port)

```
# Start a container (more on this later)
$ docker run -p 2205:2205 oliechti/teaching-docker-simplejavaserver

# List running containers
$ docker ps

# List all containers
$ docker ps -a
```

Use an existing Docker image

1. Find an image on **Docker Hub**
2. Understand that **the image packages an application** (read the **Dockerfile**). Understand that this application is a TCP server written in Java.
3. Start a **container** (map the **container port** to a VM port)
4. **Interact with the server** running in the container.

```
$ telnet ??? 2205
```

host

vm

container

Activity 2: start several containers from 1 image

Multiple containers from 1 image

1. Understand that **we need multiple ports** on the VM.
2. Start container 1, with port mapping 1
3. Start container 2, with port mapping 2
4. Start container 3, with port mapping 3
5. Interact with the 3 servers

Multiple containers from 1 image

1. Understand that **we need multiple ports** on the VM.

host

vm

3205

container

2205

4205

container

2205

5205

container

2205



Multiple containers from 1 image

1. Understand that **we need multiple ports** on the VM.

host

vm

3205

container

2205

4205

container

2205

5205

container

2205

```
$ docker run -p 3205:2205 oliecti/teaching-docker-simplejavaserver  
$ docker run -p 4205:2205 oliecti/teaching-docker-simplejavaserver  
$ docker run -p 5205:2205 oliecti/teaching-docker-simplejavaserver
```



Overview

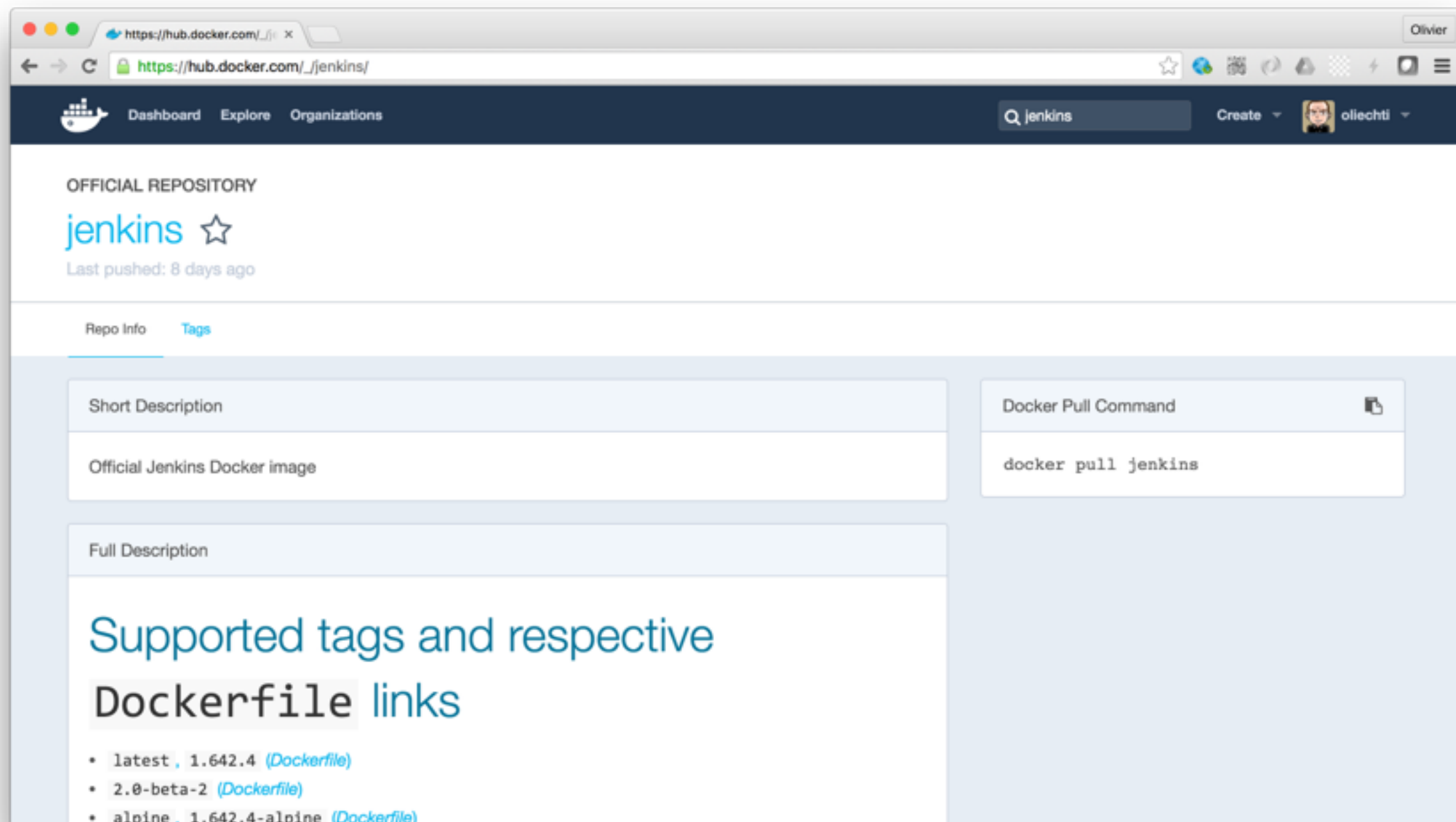
- **Jenkins** is one of the most popular continuous integration / continuous delivery servers.
- **There are many others.** Some are “cloud-first” services (e.g. Travis CI, CircleCI, etc.), others are “on-premise” products (e.g. GoCD, TeamCity, Bamboo, etc.).
- It is an **open source project** with a large community (lots of plugins) and the support of a commercial company (Cloudbees).
- It was initially created with a Java / maven focus but has a wider scope today.
- A **big transition** is underway, from being a continuous integration to becoming a full-fledged continuous delivery infrastructure. See the “pipeline” and “stage view” plugins released by Cloudbees.

First steps

- **The first thing that we want to do is to get a Jenkins server running on our machine and have a look at its web UI.**
- How do we do that?
 - We could make sure that we have a JVM on our machine, download the Jenkins binaries, follow the instructions, install it on our machine. We could then play with it. If we don't like it, there are certainly instructions for uninstalling it.
 - Or we can take advantage of Docker!

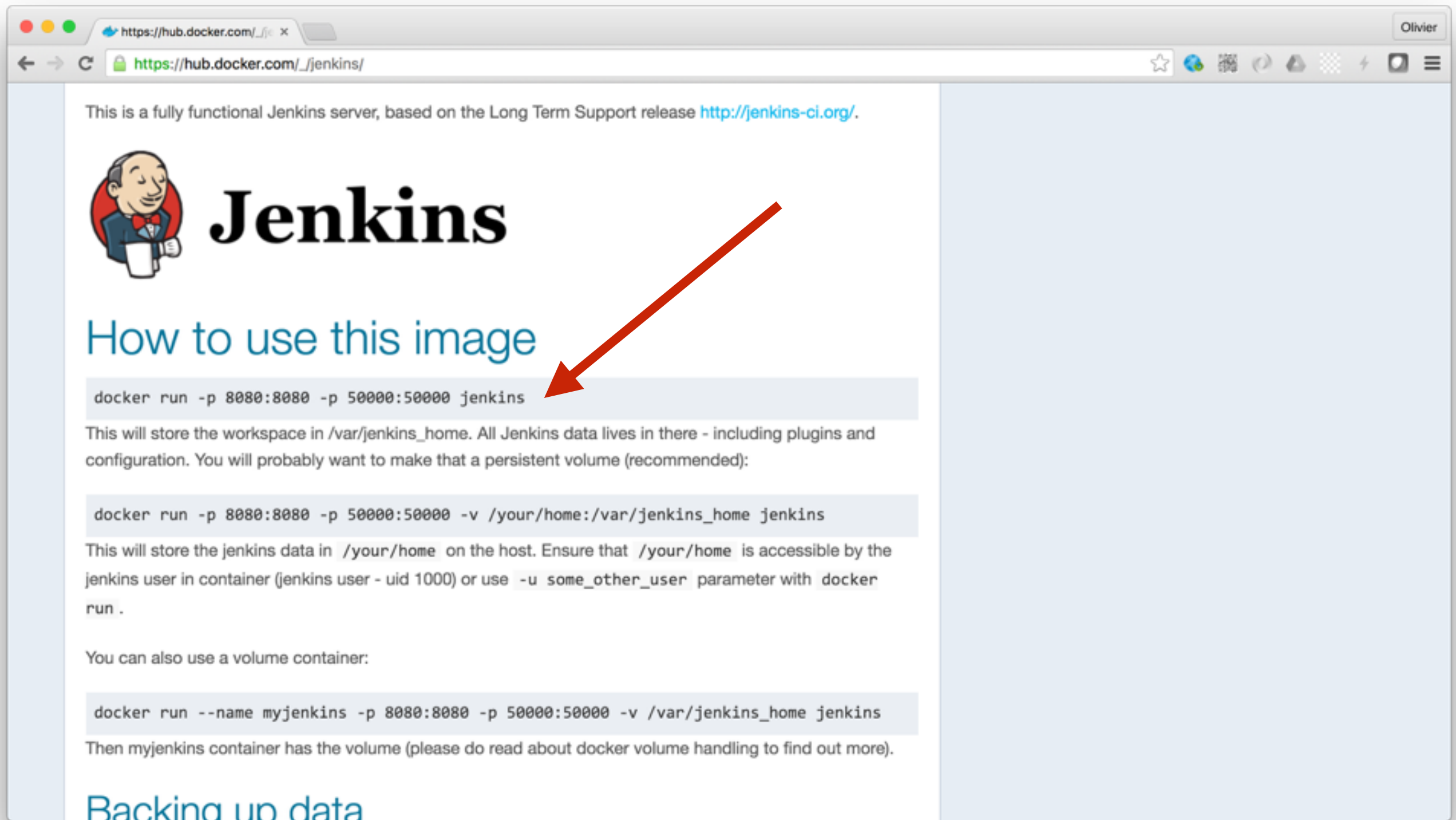
The official Jenkins Docker image

- There is a ready to use Docker image available on Docker Hub.




The official Jenkins Docker image

- It only takes one command to run it (notice the port mapping)!



The screenshot shows a web browser window displaying the Jenkins Docker image page on Docker Hub. The page title is "Jenkins" and the subtitle is "How to use this image". A red arrow points to the first command listed: `docker run -p 8080:8080 -p 50000:50000 jenkins`. Below this, there is a paragraph explaining that this command will store the workspace in `/var/jenkins_home`. Further down, another command is shown: `docker run -p 8080:8080 -p 50000:50000 -v /your/home:/var/jenkins_home jenkins`, with a paragraph explaining that this will store the Jenkins data in `/your/home` on the host. At the bottom, there is a section titled "Backing up data".

This is a fully functional Jenkins server, based on the Long Term Support release <http://jenkins-ci.org/>.



Jenkins

How to use this image

```
docker run -p 8080:8080 -p 50000:50000 jenkins
```

This will store the workspace in `/var/jenkins_home`. All Jenkins data lives in there - including plugins and configuration. You will probably want to make that a persistent volume (recommended):

```
docker run -p 8080:8080 -p 50000:50000 -v /your/home:/var/jenkins_home jenkins
```

This will store the Jenkins data in `/your/home` on the host. Ensure that `/your/home` is accessible by the Jenkins user in container (Jenkins user - uid 1000) or use `-u some_other_user` parameter with `docker run`.

You can also use a volume container:

```
docker run --name myjenkins -p 8080:8080 -p 50000:50000 -v /var/jenkins_home jenkins
```

Then myjenkins container has the volume (please do read about Docker volume handling to find out more).

Backing up data

The official Jenkins Docker image

- It only takes one command to run it (notice the port mapping)!

```
$ docker run -p 9080:8080 -p 50000:50000 jenkins
```

host port : container port

host

vm (192.168.42.42)

9080

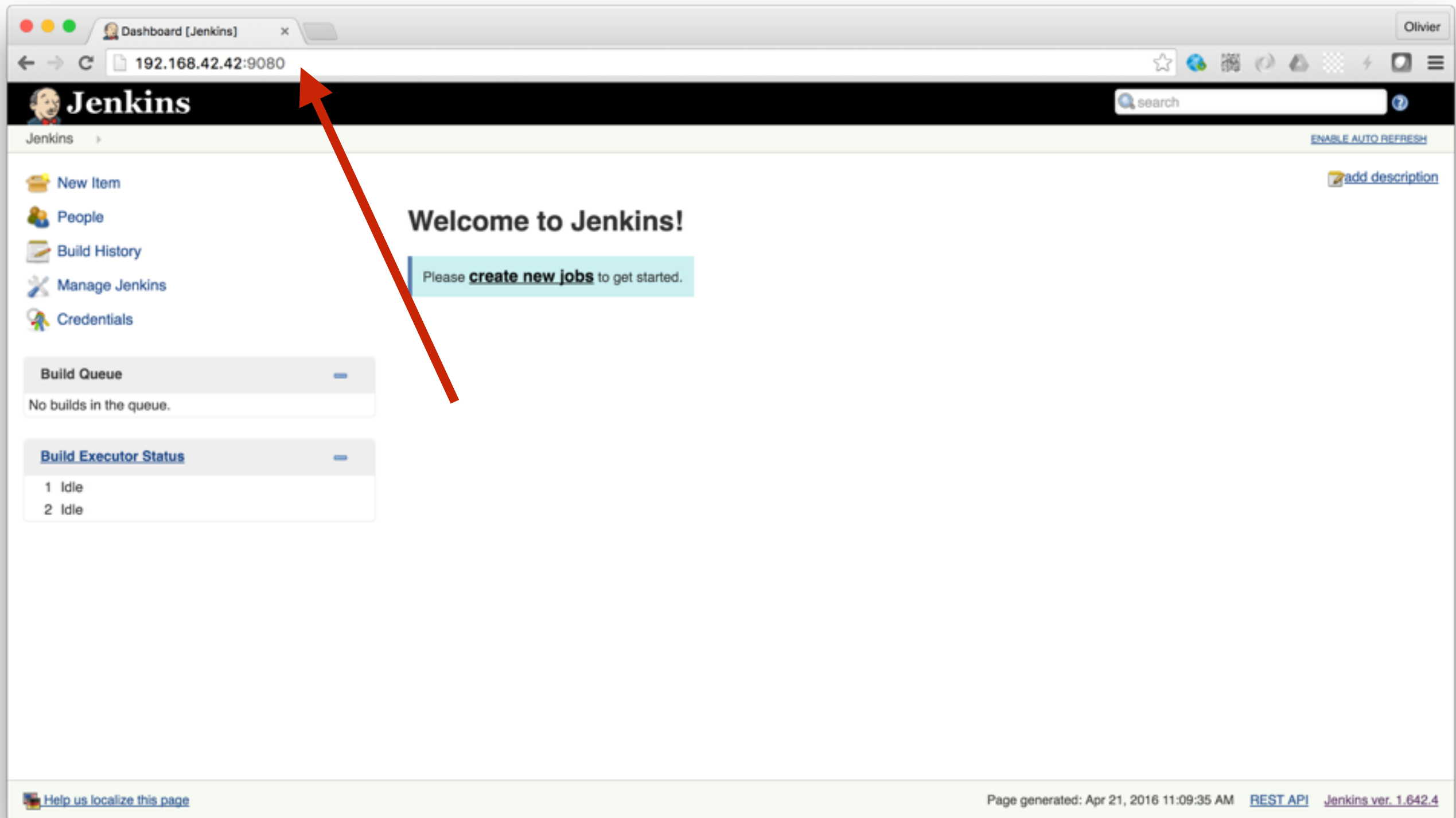
50000

8080

50000

jenkins container

The official Jenkins Docker image



Basic setup

- Out-of-the-box, this Jenkins version is not enough for our use case. **We need to configure the system** to be able to 1) fetch code from our **git** repo and 2) invoke maven.
- **Today, we will do this configuration manually.** Students who are fast are invited to look for ways to automate this process (for the others, we will show you how to do it another day).

Install the “Git plugin”

- The support for git is enabled via a plugin. To install it, you have to do the following:
 - In the left menu, select “Manage Jenkins”
 - In the center pane, select “Manage plugins”
 - Click on the “Available tabs”
 - Type “Git plugin” in the filter text box
 - Check the “Install” checkbox
 - Click on the “Install without restart” button
- During the installation, you can look at the console log in the docker container.

Install maven

- The support for git is enabled via a plugin. To install it, you have to do the following:
 - In the left menu, select “Manage Jenkins”
 - In the center pane, select “Configure system”
 - Scroll down to the “Maven” section and click on the “Add Maven” button
 - Enter “M3” in the name text field
 - Click on the “Save” button

We are now ready to build our app (1)

- In the Jenkins terminology, a “Job” is the equivalent to a pipeline.
- There are different job types. Today, we will use a simple one, specifically designed to build maven projects.
- **Create a job for your project:**
 - In the center pane, click on the “create new jobs” link
 - Type “Instruments” (or whatever) in the Item name field
 - Select the “Maven project” radio button
 - Click on the ok button

We are now ready to build our app (2)

- **Tell Jenkins where to find your sources:**
 - Scroll down to the Source Code Management section
 - Select the “Git” radio button
 - Enter the URL of your own GitHub repo (use the **HTTPS** protocol)
 - Click on the “Save” button

We are now ready to build our app (2)

- **Try to build your code and see it fail**
 - In the left menu, click on the “Build Now” button
 - You should see a red dot appear in the “Build History” widget. Click on the link and then, in the left menu, on the “Console Output” icon.
 - As you can see, the build has failed because Jenkins expected to find the pom.xml file at the root of repository. In our project, it is one level below, in the **Lab00App-build** directory.

We are now ready to build our app (2)

- **Fix the issue**

- In the top menu, click on the “Instruments” breadcrumb link.
- In the left menu, click on the “Configure” link.
- Scroll down to the Build section and enter “Lab00App-build/pom.xml” in the Root POM field.
- Click on the “Save” button.
- Hit the “Build Now” button.
- Go and check what happens in the Jenkins console. Since it is the first maven build, it takes a while because dependencies are downloaded.
- If your code is correct, it should compile and unit tests should pass.

We are now ready to build our app (2)

- **Validate the cycle:**
 - Do a modification in your code (e.g. to break a unit test on purpose).
 - Commit and push your modification to your GitHub repo.
 - Hit the “Build Now” button in Jenkins.
 - Watch the results.