Software Engineering and Architecture
Continuous Delivery, *customizing Jenkins*

Olivier Liechti
HEIG-VD
olivier.liechti@heig-vd.ch

MSE | MASTER OF SCIENCE IN ENGINEERING

Quick recap (Docker & Jenkins)

What is Docker?

- Simply stated, Docker is a “lightweight” **virtualization technology**. It allows us to create “small” virtual machines (aka containers). On a machine, we can run “many” containers (many more than “standard VMs).
- Docker also illustrates the notion of “**Infrastructure as Code**”. It provides us with APIs, which we can use to completely automate the configuration, creation and management of computing nodes.
- Remember that the roles of “software engineer” and “IT Ops engineer” are blending (**DevOps**).
- Docker is becoming the **new way to package and deliver applications**. Instead of delivering an OS specific binary + an installation manual, we can deliver a ready-to-use Docker container.

Why is Docker useful for CI / CD?

- There are different use cases for Docker. In this course, we are interested to see how it can help us in the context of **continuous delivery**:
  - We have already seen a first (small) benefit: Docker makes it easy for us to use Jenkins (and other similar or complementary tools). It only takes one command (`docker run jenkins`) to have our CI server up and running.
  - What is more interesting is that Docker will allow us to **create on-the-fly, ephemeral and consistent environments** during the continuous delivery process.
  - In the past, teams had to create and **maintain** development, test, QA, pre-production, ..., production environments. This was error-prone, because over time the environments had a **tendency to diverge**.
  - With Docker, these environments can be re-created **automatically and quickly** based on version-controlled configuration files. We address the snowflake server issue.

Docker in a CD pipeline

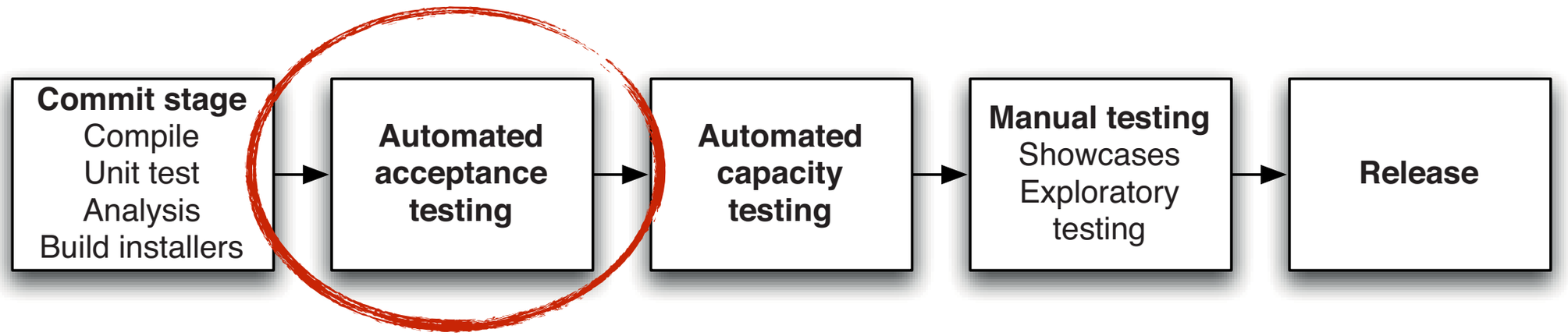
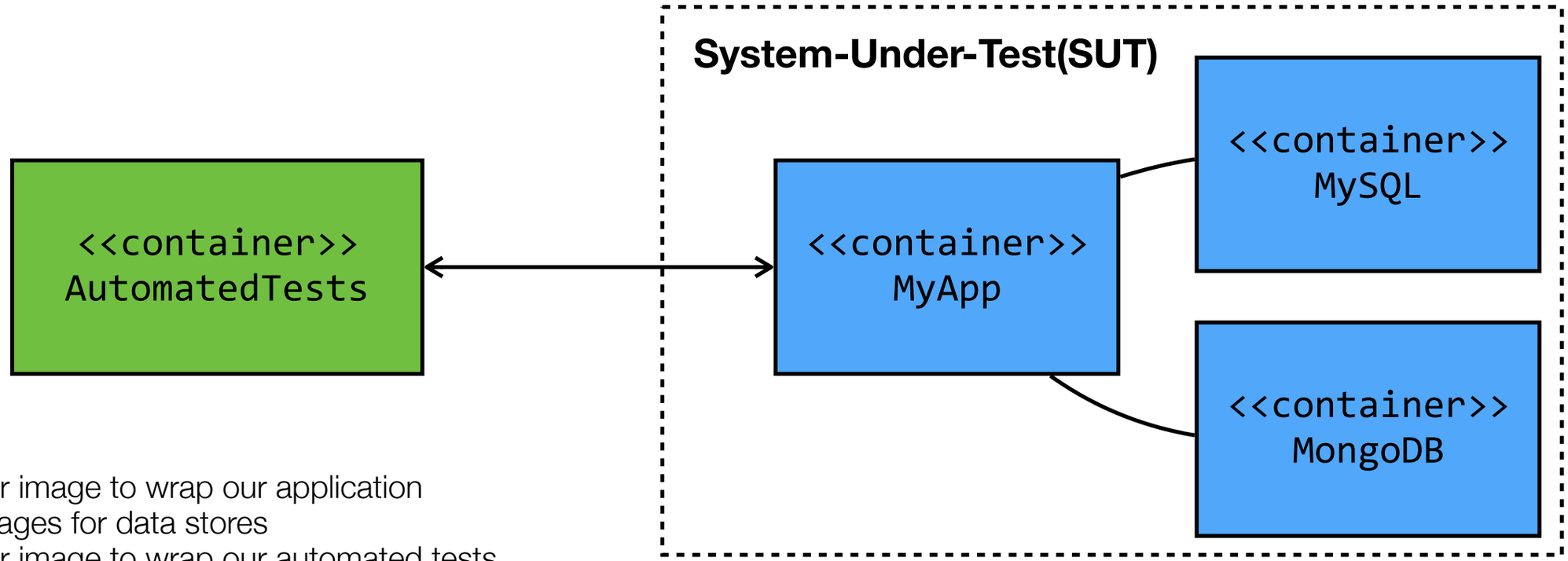


Figure 1.1 *The deployment pipeline*



1. We define a Docker image to wrap our application
2. We use existing images for data stores
3. We define a Docker image to wrap our automated tests
4. We use the `docker run` command in a script to start the SUT containers
5. We use the `docker run` command in a script to launch the automated tests
6. We configure Jenkins to invoke these two scripts

Docker in a CD pipeline

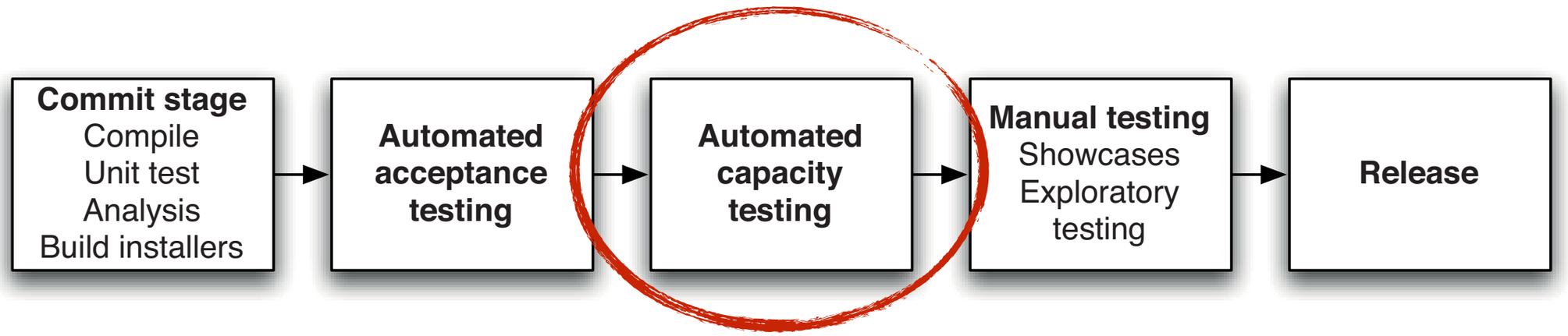
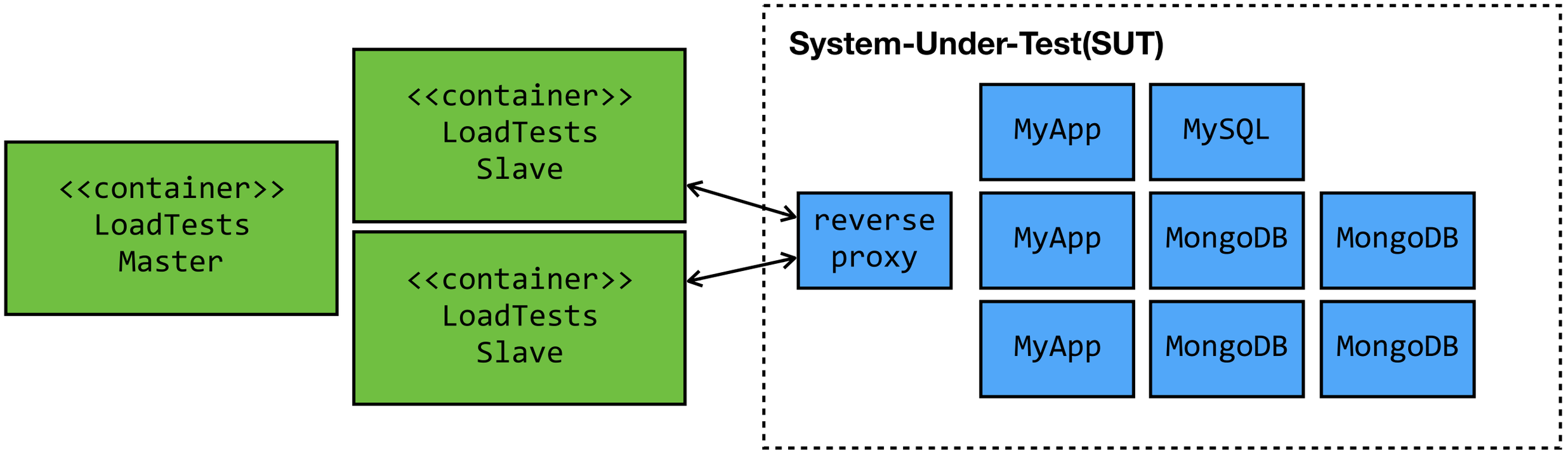


Figure 1.1 *The deployment pipeline*



1. With Docker, we can define and launch arbitrarily complex environments, which can be spread across multiple servers
2. We can therefore use it to perform non-functional tests on environments that are “similar” to the production environment

What is Jenkins?

- Jenkins is one of the tools that we can use to orchestrate the overall continuous delivery process (“chef d’orchestre”).
- Remember what we said:
  - Jenkins and maven share common functionality. **Their responsibility is to manage a process** (they delegate the hard work to plugins).
  - The difference is that Jenkins operates at the “**macro**” level (manages the entire delivery process, across technologies and environments), while maven operates at the “**micro**” level (manages the commit stage for a Java application component).
  - In other words, it is very common for Jenkins to tell maven to perform a given task. Maven is one of the tools that is managed by Jenkins.

Tools

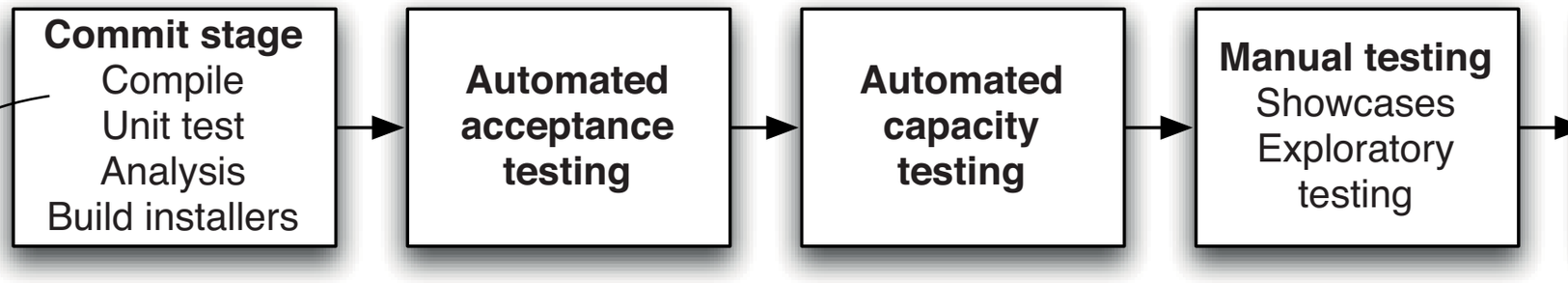


Figure 1.1 *The deployment pipeline*

Quick recap (practice)

What did we achieve?

- **We have created our first, very simple, build pipeline.** We cannot really talk about a “continuous delivery pipeline”, but should rather talk about a “continuous integration pipeline”.
- We have used the **official Docker image**, provided by Cloudbees.

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

- We then had to **perform 3 manual operations**, via the Jenkins Web UI:
  - Install the “git” plugin
  - Define a “maven tool installation”
  - Create a Job (and we have selected the “maven project” type) to build our “orchestra” Java project (and run unit tests).

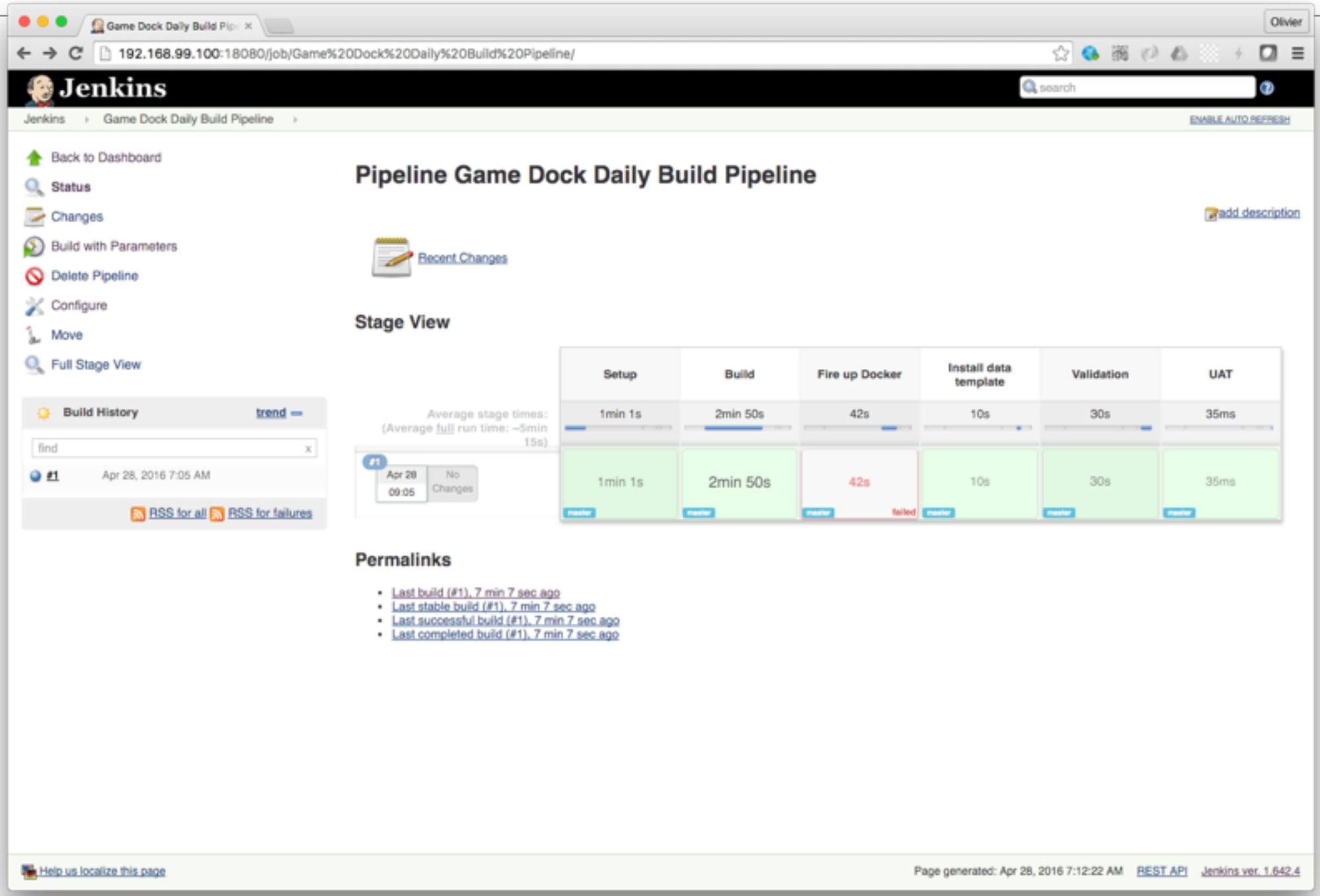
Next steps (practice)

What should we do next?

- We want to be able to:
  - **take a “fresh” machine** (typically a build server, but possibly also a developer’s laptop)
  - **type one command** that will do (as much as possible) of the work required to have a Jenkins server fully configured and ready to build our project.

- We also want to:
  - move from a “maven-centric” project type to a **more generic pipeline**
  - even if we like Java, we want to be able to implement some of our app components in **JavaScript**, **iOS**, etc. We want Jenkins to be able to deal with these.
  - we also want to **go beyond unit tests** and be able to write some of our tests with non-Java tools.

Demo


Game Dock Daily Build Pip
Olivier
192.168.99.100:18080/job/Game%20Dock%20Daily%20Build%20Pipeline/
Jenkins
search
Jenkins › Game Dock Daily Build Pipeline
ENABLE AUTO REFRESH
Back to Dashboard
Status
Changes
Build with Parameters
Delete Pipeline
Configure
Move
Full Stage View
Build History
trend
find
#1
Apr 28, 2016 7:05 AM
RSS for all
RSS for failures
Pipeline Game Dock Daily Build Pipeline
add description
Recent Changes
Stage View
Setup
Build
Fire up Docker
Install data template
Validation
UAT
Average stage times:
(Average full run time: ~5min 15s)
1min 1s
2min 50s
42s
10s
30s
35ms
#1
Apr 28
09:05
No Changes
1min 1s
2min 50s
42s
10s
30s
35ms
master
failed
Permalinks
Last build (#1), 7 min 7 sec ago
Last stable build (#1), 7 min 7 sec ago
Last successful build (#1), 7 min 7 sec ago
Last completed build (#1), 7 min 7 sec ago
Help us localize this page
Page generated: Apr 28, 2016 7:12:22 AM
REST API
Jenkins ver. 1.642.4

MSE | MASTER OF SCIENCE IN ENGINEERING

Bleeding Edge Alert

Jenkins, CloudBees & Pipelines

- Jenkins (formerly known as Hudson) was created as an extensible, open source "**continuous integration** server".
- It has captured a **significant market share** (first mover), but faces increasing competition from open source and commercial alternatives.
- **CloudBees** is a company that supports the Jenkins project (key committers are CloudBees employees). CloudBees sells products and services built on top of Jenkins.
- Continuous delivery is a natural evolution of continuous integration. It is a hot topic. **To stay relevant**, CloudBees is investing a lot to extend Jenkins with concepts that became popular with continuous delivery (e.g. pipelines).
- In other words, "pipelines" were initially not *explicit* features of Jenkins. Several pipeline plugins have been developed over the years. **And now, they are becoming a native and recommended feature**.

Jenkins 2.0 is here!

Published on 2016-04-26 by kohsuke

Tweet

Over the past 10 years, Jenkins has really grown to a de-facto standard tool that millions of people use to handle automation in software development and beyond. It is quite remarkable for a project that originally started as a hobby project under a different name. I'm very proud.

Around this time last year, we've celebrated 10 years, 1000 plugins, and 100K installations. That was a good time to retrospect, and we started thinking about the next 10 years of Jenkins and what's necessary to meet that challenge. This project has long been on a weekly "train" release model, so it was useful to step back and think about a big picture.

https://jenkins.io/blog/2016/04/26/jenkins-20-is-here/

First, one of the challenges our users are facing today is that the automation that happens between a commit and a production has significantly grown in its scope. Because of this, the clothing that used to fit (aka "freestyle project", which was the workhorse of Jenkins) no longer fits. We now need something that better fits today's use cases like "continuous delivery pipeline." This is why in 2.0 we've added the pipeline capability. This 2 year old effort allows you to describe your chain of automation in a textual form. This allows you to version control it, put it alongside your source tree, etc. It is also actually a domain specific language (DSL) of Groovy, so when your pipeline grows in complexity/sophistication, you can manage its complexity and keep it understandable far more easily.

Jenkins 2.0 and Docker

- There was already a **2.0-beta-2** official image available on Docker Hub, but it was not tagged as “latest”. When we ran docker run jenkins last week (without specifying a tag, we pulled the **1.642.2** image.
- Today, the image tagged “latest” is the **2.0** image. So, you have to realize that if you run the same run command as last week on a fresh machine, you will get a different distribution.
- When you do a `docker run`, however, you can always specify a tag. For instance, you can always to a `docker run jenkins:1.642.2`.

Some differences between Jenkins 1.x and 2.0

- **Default plugins**
  - With Jenkins 1.x, there are not many plugins installed by default. For instance, we have seen that even the git plugin had to be installed.
  - With Jenkins 2.0, the user has the choice to install “recommended” plugins and to have a set of “popular” tools automatically added to its base installation.
- **Pipelines**
  - CloudBees has developed a collection of pipeline-related plugins. The “pipeline” and the “stage view” plugins are the most important ones.
  - They can be added to a Jenkins 1.x installation, but they are now “first-class citizen” with Jenkins 2.0.
- **Wizard displayed at startup**
  - With the Jenkins 1.x image, you directly access the standard Jenkins UI (without authentication).
  - With Jenkins 2.x, you first see a security-related question and then a wizard to do the initial setup.

Docker Tip

- To build our pipeline, we will have to **customize the official Jenkins image**.
- So far, it is a **mysterious black box** for us. We don't know what is in the file system, what happen to files when we do actions in the Web UI, etc.
- We can use two Docker commands to help us explore the image:
- We can **override the default command** that is executed when a container is started from the image. We can also attach a terminal to the container (if you do that, have a look at `/usr/share/jenkins/`)

```
$ docker run -it jenkins /bin/bash
```

- Alternatively, we can connect to an already running container. This is very useful if we want to see how Jenkins files are added/modified after some UI actions (have a look at `/var/jenkins_home`).

```
$ docker exec -it /bin/bash JENKINS_CONTAINER_HASH
```

Automatically install plugins (1)

- We have seen that Jenkins can be extended with **plugins**.
- Some plugins have **dependencies** on other plugins.
- There is an easy way to customize the official Docker Jenkins image, by **giving a list of plugins** (in a text file), that must be downloaded and added on top of the default installation. Search for “Installing more tools” in https://hub.docker.com/_/jenkins/.
- However, at the moment, the **process does not manage transitive dependencies**. In other words, you cannot simply add the “pipeline” plugin in the text file. You have to also include all the dependencies (direct and indirect) of this plugin.
- This can be a tedious process. Here is a way to save some time.

Automatically install plugins (2)

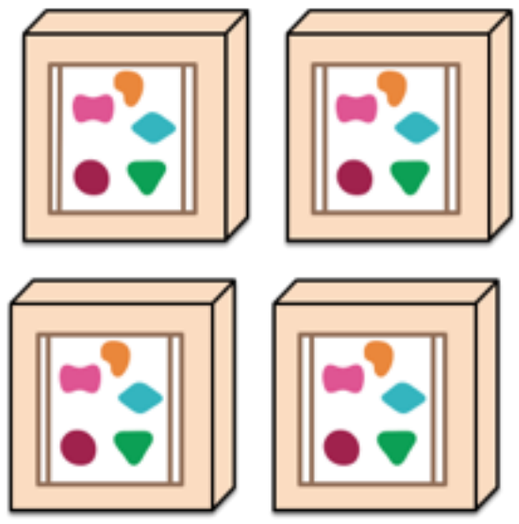
- **Step 1**
  - Start a new container, to get a fresh Jenkins installation.
- **Step 2**
  - Access the Web UI and install the desired plugins (just like last week).
    - This time, add the “Pipeline” and the “Pipeline: Stage View Plugin” plugins.

Automatically install plugins (2)

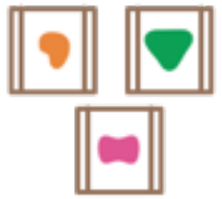
- **Step 3**
  - We need to get a list of all installed plugins (in a format that makes it easy for us to prepare a plugins.txt file)
  - This is not easy to find in the documentation, but this thread on stackoverflow gives you some hints: http://stackoverflow.com/questions/9815273/how-to-get-a-list-of-installed-jenkins-plugins-with-name-and-version-pair. Here is what I do:
    - Log into the running container (with `docker exec`)
    - Go to `/var/jenkins_home/war/WEB-INF/`
    - Run this command: `java -jar jenkins-cli.jar -s http://localhost:8080/ list-plugins`
- **Step 4**
  - Create your own Docker image and follow the instructions provided on Docker Hub (i.e. create a plugins.txt file)

A monolithic application puts all its functionality into a single process...

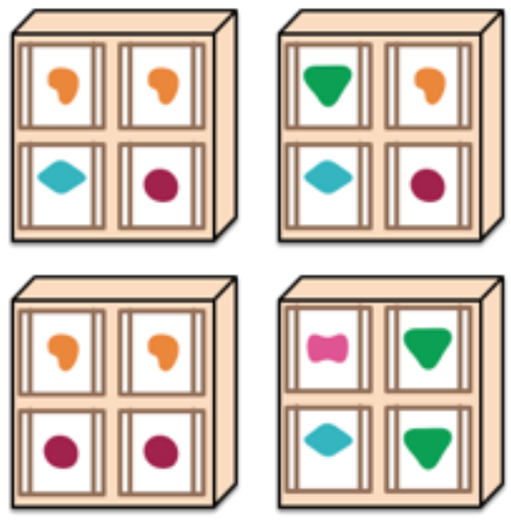
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.



http://martinfowler.com/articles/microservices.html

Micro Services Project

Project overview

- We will do a small project to have a **context for lectures** and **experiments**
- The project will provide an opportunity to apply both:
  - **software architecture approaches** (in particular, we will talk about micro-services and why everybody is talking about them these days...).
  - **software engineering methods** (we will focus on continuous delivery, quality management and validation).
- When we are done, we will have built a **distributed application** based on the micro-services architectural style. We will also have built an infrastructure that allows us to **manage the evolution of our distributed application** in a “continuous delivery” fashion.
- We do not have a lot of time, so the functionality of the application will be very basic. Yet, we will implement and orchestrate several micro-services.

Project overview

- We have created a GitHub repository for the project.
- Fork and clone it.
- https://github.com/wasadigi/Teaching-MSE-SEA-2016-MicroServices

Project: up to you

- You may now proceed with the next step of the project.
- The objective is to **create a Docker image**, based on the official Jenkins image, which performs initial configuration.
- Once again, the goal is that by typing a single command (docker run mse-sea-jenkins), one can launch a Docker container that hosts our fully functional CD pipeline.
- We will build this image iteratively and it will be complete only at the end of the semester.
- You will do this by editing the `./docker-images/sea-jenkins/Dockerfile` and using the `docker build` command.
- The first goal is to add the **Pipeline** and **Pipeline: Stage View Plugin** plugins in the official image.
- Next time, we will describe what the Pipeline plugin actually does and how we can use it instead of the “maven project job” that we have used last week.