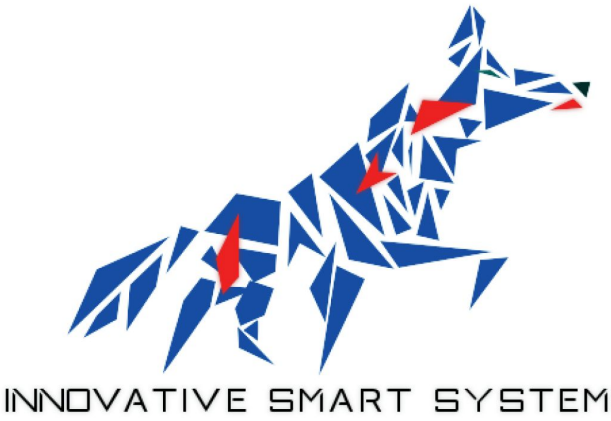
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Adaptability: Cloud and Autonomic Management

INSA Toulouse - 5th year Innovative Smart Systems

Middleware and service



Promo 2019

Julien Chouvet, Jean Baptiste Laffosse and Mathieu Raynaud

Table of contents

Lab 1: Introduction to Cloud Hypervisors

Theoretical part
(objectives 1 to 3)

1. Difference between the main virtualisation hosts (VM et CT)

First we can explain what are Virtual Machine and a Container.

The Virtual Machine (VM) is an operating system (OS) or application environment that is installed on a software which imitates dedicated hardware: mouse, memory, CPU, or network cards. The user has the same experience on the virtual machine as they would have on dedicated hardware[1].

The container is a standard unit of software that regroups code and all its dependencies. Containers include everything to run an application: code, runtime, system tools, system libraries and settings. They always run the same, regardless of the infrastructure, they isolate software from their environment and ensure that they work well when they are moved from one server environment to another[2].

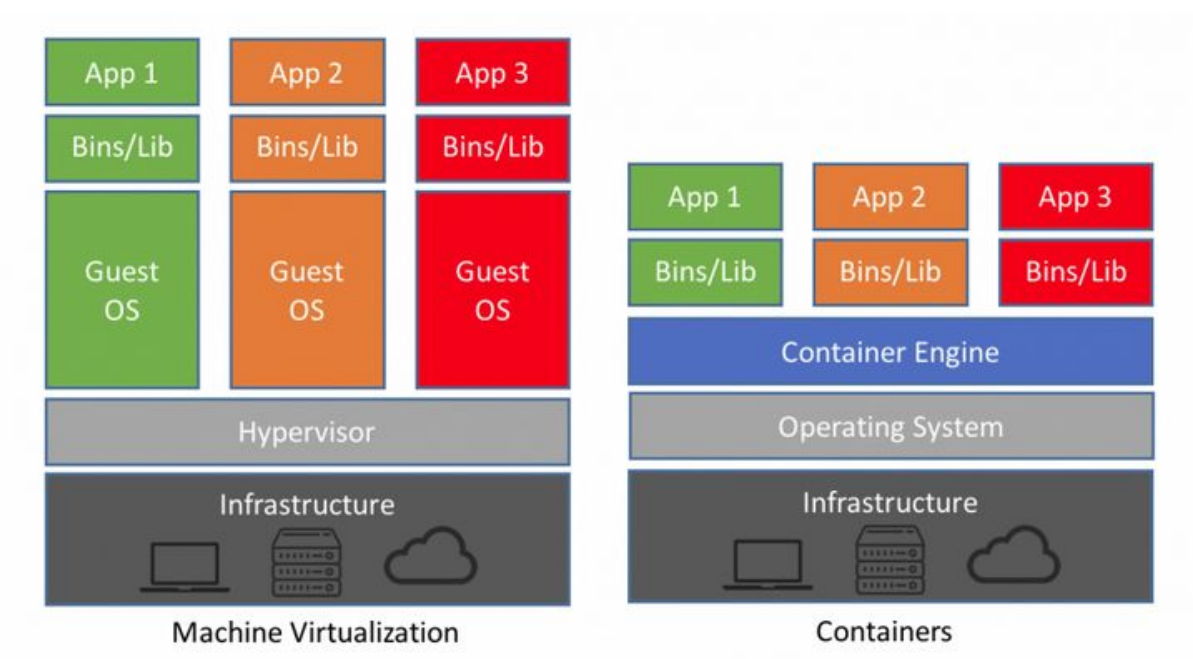


Figure 1: Comparison between Virtual Machine and Containers

[1] Virtual Machine: https://searchservervirtualization.techtarget.com/definition/virtual-machine
[2] Container: https://www.docker.com/resources/what-container

One of the main difference between VM and Containers is that for VM, the hypervisor virtualizes the hardware of the host so that the virtual OS appears to have its own processor, memory and other hardware resources. The containers run multiple virtual instance of the same OS but over a single hardware (the one of the host). This means according to the virtualization cost, VM needs a lot of resources for the memory size and the CPU.

From an application developer's view, VM has its drawbacks. In matter of facts, they add overhead in memory, storage footprint and CPU usage. As it turns out, this issue adds complexity to all the stages of a software development lifecycle, this ranges from development and test to the production, and includes disaster recovery. So, containers can be created much faster than VM instances[3].

Then, containers allow the developers to upgrade the application in a much easier way than for a VM, they also allow to follow all the different versions, and compare them. Containers also isolate the processes of one container with another one and the underlying infrastructure as well. Thus any upgrade or changes in one container do not affect another container.

From an application developer's view, container is an immature technology, it is relatively new in the market, this means that the number of resources available is limited along the developers.

Moreover, one of the main problem with the containers is the security. Containers have access to some critical namespaces, and a lot of important Linux kernel or Windows subsystems are outside the container. So, if an user or application has a superuser privileges within the container, the underlying operating system could, in theory, be cracked. Container should not be run with root or administrator privileges. That is why, during the configuration of the container, you need to pay attention at a lot of potential gate for hackers and the level of access the container. For example, you can set the file system in read-only, like this, the container processes are forced to write only to container-specific file[4].

On the other hand, VM have also their own security defaults.

From an infrastructure administrator point of view, because all the containers share a common operating system, this reduce the management overhead and only a single operating system needs to deal with the bug fixes, patches and so on[5]. However, if the number of containers running an application is too high, the complexity increase dramatically, and the management of too many containers can be challenging for the operating system.

[3] Development phase: https://dzone.com/articles/container-technologies-overview
[4] Security problem: https://www.itworld.com/article/2915530/virtualization/containers-vs-virtual-machines-how-to-tell-which-is-the-right-choice-for-your-enterprise.html
[5] Management of Container: https://blog.netapp.com/blogs/containers-vs-vms/

    - virtualization cost, taking into consideration memory size and CPU,
- Usage of CPU, memory and network for a given application,
- **Security for the application, regarding access right and resources sharing,**
- **Performances regarding response time,**
- Tooling for the continuous integration support

https://pubs.vmware.com/vsphere-4-esx-vcenter/index.jsp?topic=/com.vmware.vsphere.server_configclassic.doc_40/esx_server_config/security_for_esx_systems/c_security_and_virtual_machines.html

This approach also severely limits the portability of applications between public clouds, private clouds, and traditional data centers.

2. Differences between the existing CT types

Linux LxC:

- OS-level virtualization
- Each container is assign a unique process ID. Each container can run a single process.
- Containers get a private IP address and virtual ethernet interface connecting to a linux bridge on the host.
- Allows creation and running of multiple isolated Linux Virtual Environment (VE) to run applications.
- Allows limitation and prioritization of resources like CPU or memory.
- Tooling: API, continuous integration.

Docker:

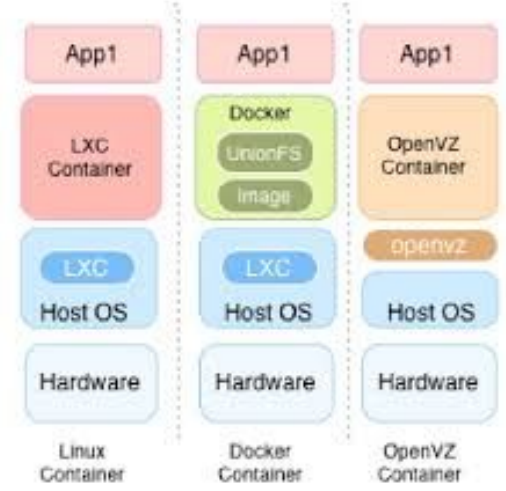
- Relies on the host operating system.
- Each container is assign a unique process ID.
- Containers get a private IP address and virtual ethernet interface connecting to a linux bridge on the host.
- Cannot run multiple processes in a single container.
- Packages application and all its dependencies in a virtual container.
- Tooling: API, continuous integration.

OpenVZ:

- OS-level virtualization
- Each container has its own PID.
- Uses net namespace, has its own virtual network device and its own IP address.
- Allows creation and running of multiple isolated Linux Virtual Environment (VE) to run applications.
- Requires a special kernel to be installed.
- Access to physical hardware from inside a container is disabled by default

(good security feature but need to configure manually if you want to use an hardware component).

- Tooling: API.

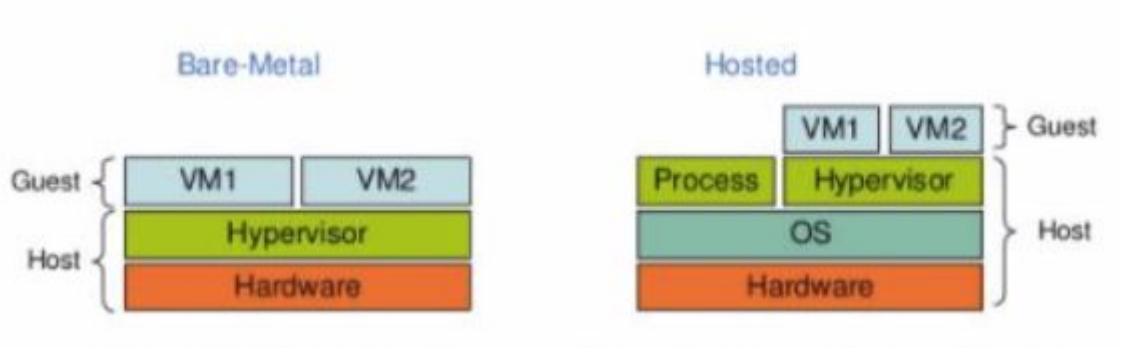


Comparison between LxC, Docker and OpenVZ

3. Differences between Type 1 & Type 2 for hypervisors' architectures

There is two types of hypervisors' architectures:

1. **Bare-metal (or Hardware level)** in which the hypervisor run directly on the system hardware. It controls the hardware and manage guest operating systems.
2. **Hosted (or OS level)** in which the hypervisor run on a host operating system like other process. VirtualBox uses this type of hypervisor architecture as well as Proxmox which uses KVM (*Kernel-based Virtual Machine*) hypervisor.



Practical part
(objectives 4 to 7)

1. Creating a VirtualBox VM (in NAT mode), and setting up the network to enable two-way communication with the outside

After creating the VM and applying the given configurations in the subject we obtain the ip address of the VM, which is: **10.0.2.15**
The IP address of the host system is: **10.1.5.46**

Second part: Testing connectivity

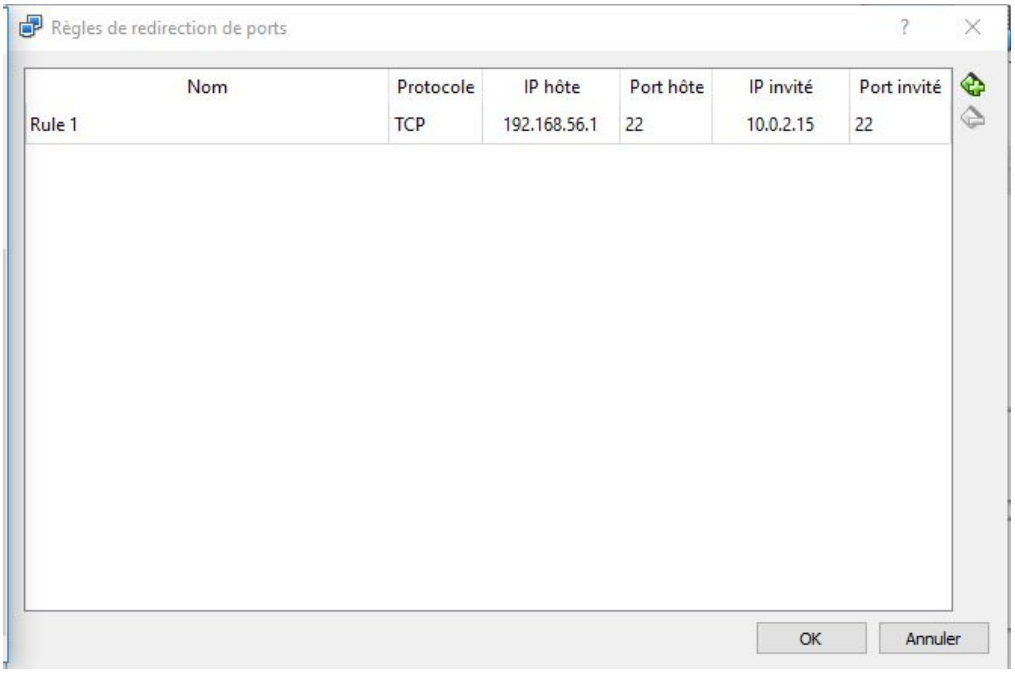
- **Check the connectivity from the VM to the outside. What do you observe?**
  We check the connectivity with the outside by making a *ping 8.8.8.8*. This one works ! This shows that the VM can reach the outside. In fact, the hypervisor (VirtualBox) act as a NAT, allowing our ping to transit to the LAN (outside).

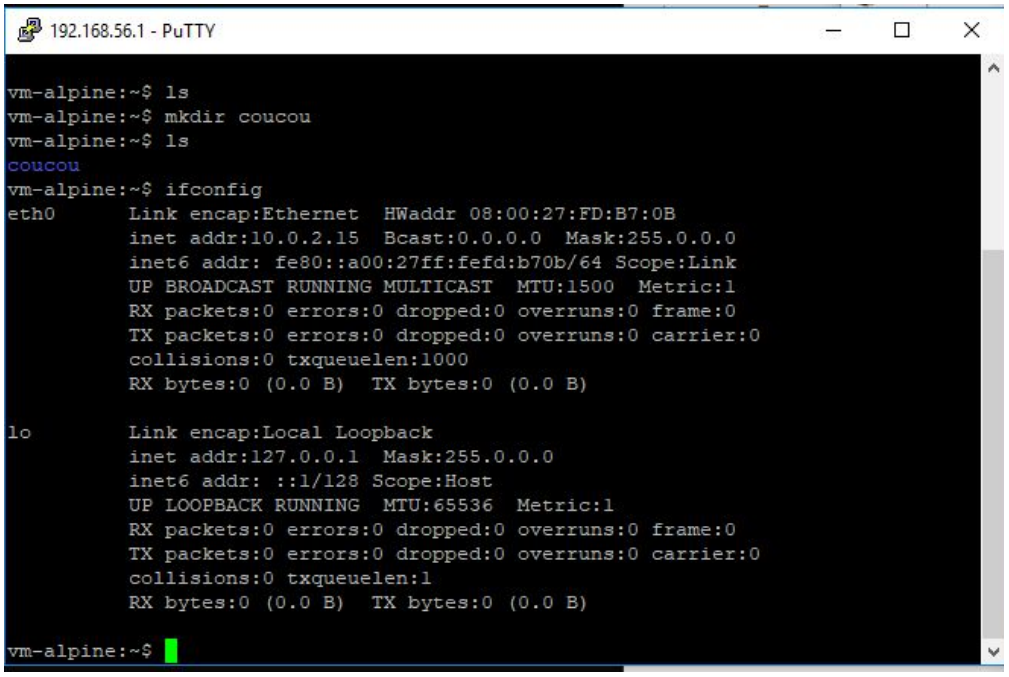
- **Check the connectivity from your neighbour's host to your hosted VM. What do you observe?**
  It is impossible for our neighbour's host to reach our VM. This is because our VM is on it's own (virtual) network which is unreachable from the outside.

- **Check the connectivity from your host to the hosted VM. What do you observe?**
  If we make *ping 10.0.2.15* this doesn't work. This is for the same reason as the previous question.
  If we make *ping 192.168.56.1* which is the IP address of "VirtualBox Host-Only Network", of course it works because this is the address of the virtual network created by VirtualBox for our VM.

Third part: Set up the "missing" connectivity

In this part we are going to add a connectivity from the outside to our VM. To do that, we use the *port-forwarding* method to add the *SSH* (port 22) connectivity.
Once the configuration is done, we try with *Putty*, on the host machine, to connect by *SSH* to our VM.
As shown in the following pictures, the *SSH* works:





2. Create a Proxmox CT (in Bridge and DHCP mode), Manually snapshot and restore this CT, Manually migrate this CT from the host server to another

Second part: Connectivity test

Lab 2: Paravirtualization over Cloud Hypervisors

After the first lab where we acquired the skills necessary to the creation and configuration of the Proxmox containers, we learnt how create snapshots, restore and migrate containers manually via the Proxmox Web interface. We will now see in the second lab, how to do this automatically thanks to the Proxmox API.

We use for this the provided project in the subject, we can find all java files in the following Github repository:

https://github.com/MathieuRaynaud/Proxmox-App-5ISS

First PART: Retrieve containers information
(objectives 1 to 4)

Using the Proxmox API to communicate with Proxmox server

A HTTP REST client is required to be able to interact with the Proxmox API, for this we used a RESTclient java library available in the given project in the package:

org.ctlv.proxmox.api

First, we fill in the Constants file, all the information about our logs, two servers name and the host name.

Thanks to the Proxmox API, we can:

- To login to the Proxmox platform thanks to our INSA login and password.
- List all the servers on the platform.
- Get the server information according to his name.
- Create, start, stop, delete or migrate a container.
- List all the containers.
- Get one container information.

List and display the containers information

We can now implement our methods to obtain all the server and containers information in the function *Monitor.java* and *Analyzer.java*.

We can go through all the servers, to obtain only the containers in our two servers. Then, we can fill two LXC lists with the containers. The LXC corresponds to the container, we can access for example to the container name, or other useful information.

```
// Récupérer les données sur les serveurs
ArrayList<LXC> LXCList = new ArrayList<LXC>();
ArrayList<LXC> LXCList2 = new ArrayList<LXC>();

try {
    for (int i=1; i<=10; i++) {
        String srv ="srv-px"+i;

        if (srv.equals(Constants.SERVER1)) {
            List<LXC> cts = api.getCTs(srv);
            System.out.println("Creation LXC List");
            for (LXC lxc : cts) {
                LXCList.add(lxc);
            }
        }
        if(srv.equals(Constants.SERVER2)) {
            List<LXC> cts = api.getCTs(srv);
            System.out.println("Creation LXC List");
            for (LXC lxc : cts) {
                LXCList2.add(lxc);
            }
        }


    }
} catch (Exception e) {
    e.printStackTrace();
}
```

We create a HashMap with inside the containers in our both servers. We use a HashMap as is easier to manipulate later in our code, we can access to the containers with the server name as a key.

```
System.out.println("Creation HasMap");
HashMap<String, ArrayList<LXC>> ctMap = new HashMap<String, ArrayList<LXC>>();
ctMap.put(Constants.SERVER1, LXCList);
ctMap.put(Constants.SERVER2, LXCList2);
```

Then, we call the *analyse()* method within the *Analyzer* class to get the information we want about the containers.

```
// Lancer l'analyse
System.out.println("Start analyze");
this.analyzer.analyze(ctMap);
```

Finally, inside the *analyse()* method, we can from the HashMap retrieve the container information and display them:

```
//Display Container information
for (Entry<String, ArrayList<LXC>> e : myCTsPerServer.entrySet()) {
    analyzeServer(e.getKey());

    for(LXC lxc : e.getValue()) {
        if (lxc.getVmid().substring(0, 2).equals("35")) {
            System.out.println("\tContainer " + lxc.getName() +":");
            System.out.println("\t\t"+"CPU usage: " + lxc.getCpu()*100 + "%");
            System.out.println("\t\t"+"Disk usage: " + lxc.getDisk()*100/lxc.getMaxdisk()+ "%");
            System.out.println("\t\t"+"RAM usage: " + lxc.getMem()*100/lxc.getMaxmem()+ "%");
            System.out.println("");

        }
    }
}
```

Second PART: Manage the Proxmox resources

(related to objective 5)

We share two servers with others groups, so we need to share the resources with them and we are allowed to use only 16% of the RAM of the hosting server for your containers. In this part, we will first generate containers and start and then after the limit is reached we will migrate or delete the containers.

Create Containers

We want to create containers randomly on the two servers, 66% on the server 1 and 33% on the server 2. The creation of containers is also periodical driven by a statistic distribution, we use an exponential law for this.

In order to create containers, we need first to check if we do not exceed 16% of the memory on the server. To do this, we create our own function *getMemoryOfServer()* to get the memory we actually used.

```
//Obtain the memory used by the server (%)
public long getMemoryOfServer(String serverName) {
    long mem = -1;

    try {
        Node server = api.getNode(serverName);
        mem = server.getMemory_used()*100/server.getMemory_total();
    } catch (LoginException | JSONException | IOException e) {
        e.printStackTrace();
    }

    return mem;
}
```

This function allow us to obtain the percentage of the used memory according to the server name.

Then, we check if the memory used on the server is lower than 16% of the total memory, if yes we can create others containers:

```
while (true) {

    // Calculer la quantité de RAM utilisée par mes CTs sur chaque serveur
    long memOnServer1 = analyzer.getMemoryOfServer(Constants.SERVER1);
    long memOnServer2 = analyzer.getMemoryOfServer(Constants.SERVER2);

    // Mémoire autorisée sur chaque serveur
    float memRatioOnServer1 = api.getNode(Constants.SERVER1).getMemory_total()*16/100;
    float memRatioOnServer2 = api.getNode(Constants.SERVER2).getMemory_total()*16/100;

    if (memOnServer1 < memRatioOnServer1 && memOnServer2 < memRatioOnServer2) {

        // choisir un serveur aléatoirement avec les ratios spécifiés 66% vs 33%
        String serverName;
        if (rndServer.nextFloat() < Constants.CT_CREATION_RATIO_ON_SERVER1)
            serverName = Constants.SERVER1;
        else
            serverName = Constants.SERVER2;

        //regarder si l'ID n'est pas déjà pris
        ArrayList<String> existing_ids = analyzer.getIds();
        while (existing_ids.contains(vmid.toString()))
            vmid++;

        // créer un contenaire sur ce serveur
        System.out.println("Creating a container on server " + serverName +"...");
        api.createCT(serverName, vmid.toString(), "ct-tpiss-virt-C5-ct"+indice, Constants.RAM_SIZE[1]);
        System.out.println("Created!");

        while (api.getCT(serverName, vmid.toString()).getStatus().equals("stopped")) {
            try {
                System.out.println("Starting container ct-tpiss-virt-C5-ct"+indice);
                api.startCT(serverName, vmid.toString());
                System.out.println("Started!");
            } catch (Exception ignore ) {}
        }

        indice++;

        // planifier la prochaine création
        int timeToWait = getNextEventExponential(lambda); // par exemple une loi expo d'une moyenne de 30sec

        // attendre jusqu'au prochain événement
        Thread.sleep(500 * timeToWait);
    }
    else {
        System.out.println("Servers are loaded, waiting ...");
        Thread.sleep(Constants.GENERATION_WAIT_TIME* 1000);
    }
```

We can spread the creation of the containers on the two servers. When we create the container, we need after to start them. To do this we choose to watch the status of the container and wait until the container status goes from “stopped” to another status.

When we create containers, we have to make sure the Id we will use is not taken, so we create a function to see all the container Ids in the server we have chosen:

```java
//Obtain all the container Ids
public ArrayList<String> getIds(){
    ArrayList<String> Ids = new ArrayList<String>();
    try {
        for (String serverName : api.getNodes()) {
            for(LXC lxc : api.getCTs(serverName)) {
                Ids.add(lxc.getVmid());
            }
        }
    } catch (LoginException | JSONException | IOException e) {
        e.printStackTrace();
    }
    return Ids;
}
```

Manage Containers

In the following part, we will manage the resources of the two servers. In this *analyze()* function inside the *Analyzer* class, we are doing two different actions: first the migrate part and and then the delete part.

- For the migrate part: we migrate one container from one server to another only when the RAM used by the container is upper than 8% of the total RAM of the server.

- For the delete part: if the total RAM used in the server exceed 12%, we can delete the youngest container inside the server (it is our choice). To find the youngest container inside a server, we can look the *getUpTime()* value in order to see which container is awake for the smaller time, we can take the minimum of all the *getUpTime()* value to delete the youngest container.

```java
public void analyze(HashMap<String, ArrayList<LXC>> myCTsPerServer)  {

    // Mémoire autorisée sur chaque serveur

    try {
        long MaxMemCTonServer1 = api.getNode(Constants.SERVER1).getMemory_total()*8/100;
        long MaxMemCTonServer2 = api.getNode(Constants.SERVER2).getMemory_total()*8/100;


        // Calculer la quantité de RAM utilisée par mes CTs sur chaque serveur
        for (Entry<String, ArrayList<LXC>> e : myCTsPerServer.entrySet()) {
            analyzeServer(e.getKey());

            for(LXC lxc : e.getValue()) {
                if (lxc.getVmid().substring(0, 2).equals("35")) {
                    System.out.println("\tContainer " + lxc.getName() +":");
                    System.out.println("\t\t"+"CPU usage: " + lxc.getCpu()*100 + "%");
                    System.out.println("\t\t"+"Disk usage: " + lxc.getDisk()*100/lxc.getMaxdisk()+ "%");
                    System.out.println("\t\t"+"RAM usage: " + lxc.getMem()*100/lxc.getMaxmem()+ "%");
                    System.out.println("");
                    long ramusedCT = lxc.getMem()*100/api.getNode(e.getKey()).getMemory_total();

                    if(e.getKey().equals(Constants.SERVER1)) {
                        if(ramusedCT >= MaxMemCTonServer1) {
                            api.migrateCT(Constants.SERVER1, lxc.getVmid(), Constants.SERVER2);
                            System.out.println("CT " + lxc.getVmid() + " migrated to the server " + Constants.SERVER2);
                        }
                    }
                    else {
                        if(ramusedCT >= MaxMemCTonServer2) {
                            api.migrateCT(Constants.SERVER2, lxc.getVmid(), Constants.SERVER1);
                            System.out.println("CT " + lxc.getVmid() + " migrated to the server " + Constants.SERVER1);
                        }
                    }
                }
            }
            long load = api.getNode(e.getKey()).getMemory_used()*100/api.getNode(e.getKey()).getMemory_total();
            if(load > 12/100) {
                ArrayList<String> existing_ids = getIds();
                long min = (long) Integer.MAX_VALUE;
                LXC toDelete = null;
                for (LXC lxc : e.getValue()) {
                    if (lxc.getVmid().substring(0, 2).equals("35")) {
                        if (lxc.getUptime() < min) {
                            toDelete = lxc;
                            min = lxc.getUptime();
                        }
                    }
                }

                if (toDelete != null) {
                    while (api.getCT(e.getKey(), toDelete.getVmid()).getStatus().equals("running")) {
                        try {
                            System.out.println("Stopping container " + toDelete.getName());
                            api.stopCT(e.getKey(), toDelete.getVmid());
                            System.out.println("Stopped!");
                        } catch (Exception ignore ) {}
                    }
                }
            }
        }

    } catch (LoginException | JSONException | IOException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    }
```

Lab 3: Provisioning End-user Application in Cloud Platforms

Objectives of the lab

The objective of this lab is to understand the way to use service-based applications in cloud computing.
To reach this objective, we studied the several provisioning process phases and implemented them to provision either a sample HelloWorld code on Google Cloud Platform or the resources generator and manager we developed during lab 2.

Here are the 5 objectives covered by the lab:

- Objective 1: Read and assimilate the several phases that make up cloud end-user applications provisioning process.
- Objective 2: Implement such a process using sample JEE application on Google Cloud Platform, Cloud Foundry and Jelastic. Various approaches and tools will be used for each one of the previously mentioned PaaS (e.g. through a plugin, a CLI and the Web).
- Objective 3: Learn how to build a more complex cloud application, and the application architecture concepts/principles you will need to succeed.
- Objective 4: Implement the lessons learned from Objective 3 to design and develop a simple MVC application with database access, on a PaaS.
- Objective 5: Design and develop your own embryonic PaaS using LXC containers on top of the Proxmox infrastructure and migrate your Resources Generator and Manager applications over there.

1. Read and assimilate the several phases and steps that make up cloud end-user applications provisioning process

Provisioning cloud resources process consists of three main phases. These phases are the following ones:

1. Phase 1: Development

It involves (non-exhaustive list, not all sub-phases are mandatory, some of them might be concurrent):

- Modeling the application (e.g. designing the required service / component-based topology)
- Developing the sources (e.g. writing code and scripts, referring to remote components/modules published over repositories, importing required libs)
- Compiling the sources
- Testing and debugging

- Building the artifacts (a.k.a executables, archives, deployables)
- Packaging in appropriate wrappers if necessary (e.g. service containers, standalone frameworks)
- Modeling the management plan
- Settling the SLAs
- Specifying the deployment descriptor (a.k.a manifest)

2. Phase 2: Deployment

It involves:

- Allocating the required cloud resources
- Uploading the executables over these resources
- Activating the application according to the specified plans/SLAs (including the automatic binding and activation of the required services – such as storage, logging, monitoring and so on.

3. Phase 3: Management

It involves:

- Executing the application
- Performing the appropriate/required management operations that aim at optimizing its performance and reducing its operation cost. SLAs violations, from end-user point of view, would trigger the execution of a specific management operation. Management operations are specified through the management plans and might be implemented as workflows. Scaling up/down or migrating an application component are among the examples of management operations.

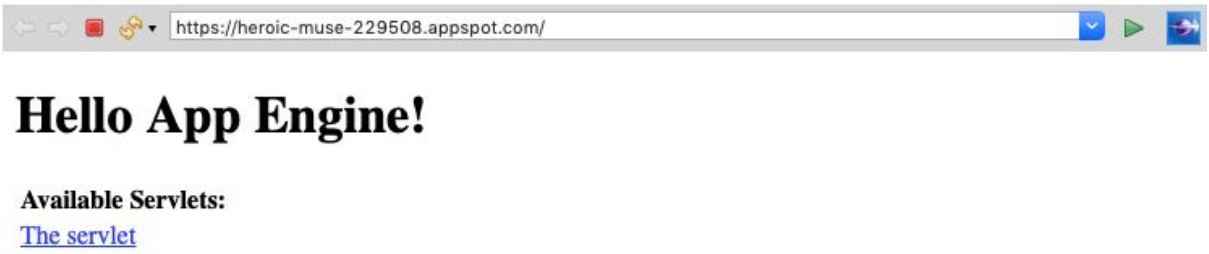
2. Implement part of the provisioning process using sample JEE application on Google Cloud Platform, Cloud Foundry and Jelastic

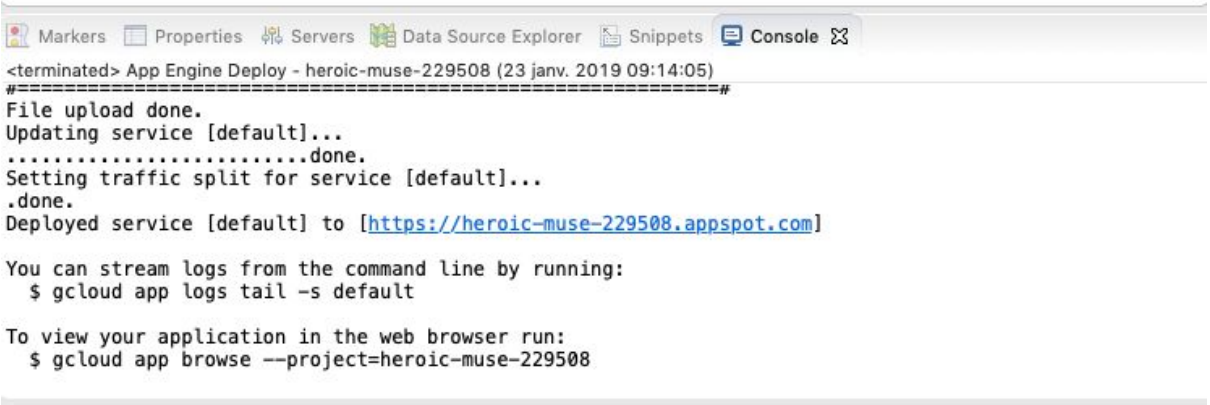
2.1 - Provisioning HelloWorld servlet on Google Cloud Platform

To provision a servlet on Google Cloud Platform, we first downloaded the Google Cloud Tools for Eclipse plugin in our Eclipse J2EE IDE.

Then, we wanted to create a new Google Cloud Application. Consequently, we first created the application in the Google Cloud Platform (https://console.cloud.google.com/home/dashboard). Then, we created a “Google App Engine Standard Java Project” in Eclipse.

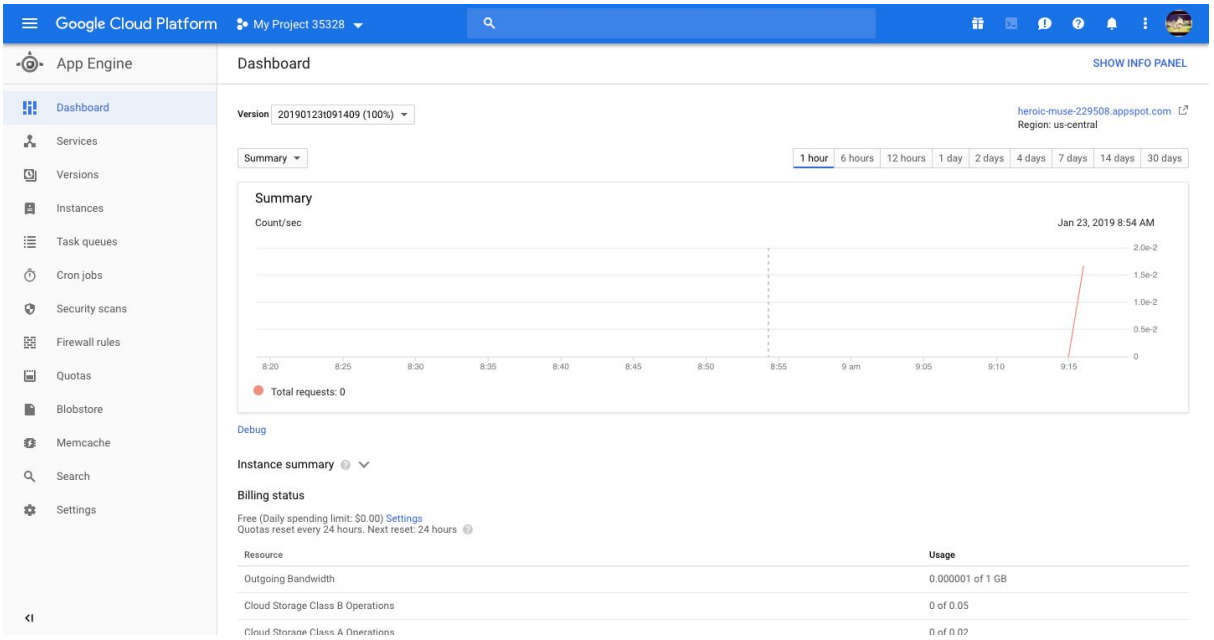
Once this was done, we ran the Eclipse project as a Google App Engine, and the following page has been deployed at https://heroic-muse-229508.appspot.com/ :





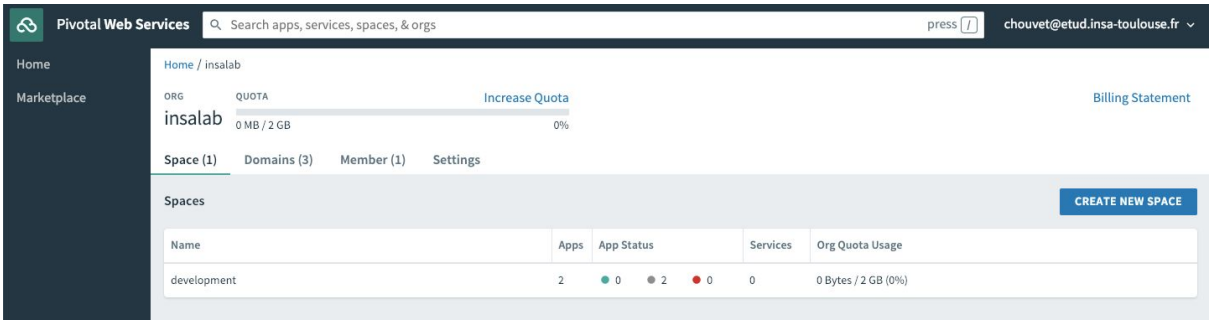
When we clicked on “The servlet”, a page displayed “Hello App Engine!”.

Once our application was deployed, we were able to manage it through the following administration dashboard:



2.2 - Provisioning HelloWorld Servlet on Cloud Foundry

After creating an account, we were able to create an **org** called *insalab* that contains a space called *development*. Here is a view of our Pivotal web service.



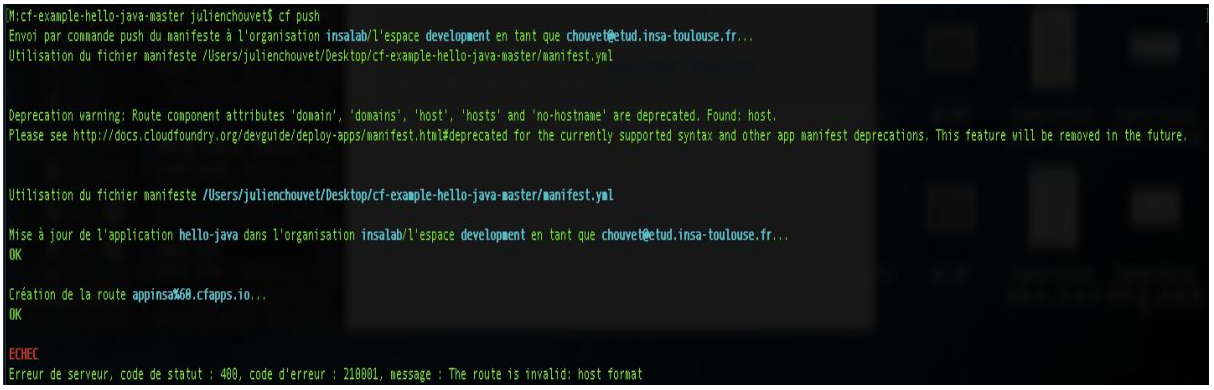
Pivotal web service

Then, we downloaded the *Cloud Foundry CLI* to be able to manage our apps. Finally, thanks to its CLI we were able, in command line, to push our HelloWorld app on cloud Foundry. Here is the *manifest.yml* that we modified according to the subject:

```
1 ---
2 applications:
3 - name: hello-java
4   memory: 1024M
5   instances: 2
6   host: appinsa`
7   path: target/hello-java.war
8
```

Manifest.yml to create our app on the cloud thanks to the “cf push” command

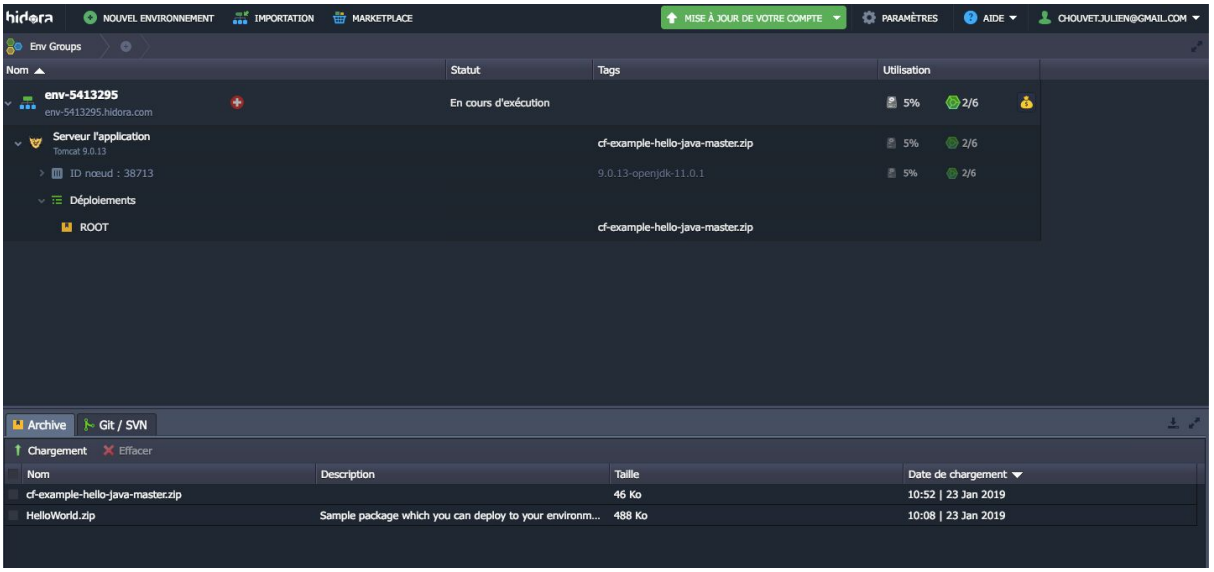
Unfortunately, we add a problem when pushing the application. Hence we could not deploy our HelloWorld app.



Error when pushing our HelloWorld application

2.3 - Provisioning HelloWorld Servlet on Jelastic

In this part we tried to deploy our application on Jelastic.
After creating an account we were able to create an environment to host the HelloWorld servlet. Then, we uploaded the HelloWorld artifact on the Jelastic and finally we deployed it under the previously created environment.



Jelastic environment and deployment of the application artifact

Unfortunately, we had a problem when we tried to execute application.

Error while executing the deployed application