Deployment

Systems Engineering Laboratory 1

1 Introduction

Deploying multi-component applications is not trivial, since we should take care on shared data, communication and the right environments. Doing this deployment for each release, even more for each test execution requires a systematic approach to ensure the consistent environment and comparable executions. Doing integration tests requires frequent deployment in a reproducible way.

2 Deployment techniques

During this lab we focus on the multi-component software deployments. Such components should be deployed on the same or different physical or virtual machine or container environment, while the communication between the components should be ensured.

There are three main approaches:

- Installing the components on the same or different hosts directly.
- Packaging the components into application container images and running the containers one one or multiple hosts.
- Packaging the components into virtual machine images and running these virtual machines.

During the deployment and also testing phases the reproducible environment is very important. We should ensure that the components are executed in the same environment (software stack) in the development and production environment and during the tests. Deploying a consistent environment is easier said than done. Even if we use configuration management tools like Chef and Puppet, there are always OS updates and other things that change between hosts and environments.

When installing components directly to the hosts, the consistency and reproducible requirement cannot be ensured. Using a systematic method for building container images or virtual machine images provides stable, well-defined environments and reproducible executions.

Using a virtualization method and building an image provides consistent environments. The virtualization method can be categorized based on how it mimics hardware to a guest operating system and emulates guest operating environment. Primarily, there are two main types of virtualization:

1. **Emulation and paravirtualization** is based on some type of hypervisor which is responsible for translating guest OS kernel code to software instructions or directly executes it on the bare-metal hardware. A virtual machine provides a computing environment with dedicated resources, requests for CPU, memory, hard disk, network and other hardware resources that are managed by a virtualization layer. A virtual machine has its own OS and full software stack.
2. **Container-based virtualization**, also known as operating system-level virtualization, enables multiple isolated executions within a single operating system kernel. It has the best possible performance and density, while featuring dynamic resource management. The isolated virtual execution environment provided by this type of virtualization is called a *container* and can be viewed as a well-defined group of processes.

Starting from a base image (virtual machine or container image) the environment can be built up and saved as a new image in a consistent way.

There are many technologies that provide automatic virtual machine image construction (e.g. vagrant, packer) on different platforms (e.g. Amazon AWS, VMware, Xen). For container-based solutions there is the Docker that is the de facto standard. During this lab we will rely on these lightweight Docker-based container solutions.

3 Docker basics

Docker provides the ability to package and run an application in a loosely isolated environment called a container. It provides tooling and a platform to manage the lifecycle of your containers:

- Encapsulate your applications (and supporting components) into Docker containers.
- Distribute and ship those containers to your teams for further development and testing.
- Deploy those applications to your production environment, whether it is in a local data center or the cloud.

The core concepts of Docker are *images* and *containers*. In the following we highlight the most important parts of the Docker overview.

A *Docker image* is a read-only template with instructions for creating a Docker container. For example, an image might contain an Ubuntu operating system with an Apache web server and your web application installed. You can build or update images from scratch or download and use images created by others. An image may be based on, or may extend, one or more other images. A Docker image is described in text file called a Dockerfile, which has a simple, well-defined syntax.

A *Docker container* is a runnable instance of a Docker image. You can run, start, stop, move, or delete a container using Docker API or CLI commands. When you run a container, you can provide configuration metadata such as networking information or environment variables.

An image is defined in a `Dockerfile`. Every image starts from a base image, e.g. from `ubuntu`, a base Ubuntu image. The Docker image is built from the base image using a simple, descriptive set of steps we call instructions, which are stored in a `Dockerfile` (e.g. `RUN` a command, `ADD` a file).

Images come to life with the `docker run` command, which creates a container by adding a read-write layer on top of the image. This combination of read-only layers topped with a read-write layer is known as a union file system. Changes exist only within an individual container instance. When a container is deleted, any changes are lost unless steps are taken to preserve them.

Instead of copying the tutorials available on the internet, we collected them here and propose the required sections to read. In order to prepare for the lab, we advise you to read the following additional material:

- Docker overview
- Working with Docker Containers
- Build your own image
- Dockerfile reference (Read the Usage, Format, FROM, RUN, ADD, CMD, ENTRYPOINT sections)
- Docker run reference (Read only the Detached vs. foreground, CMD (default command or options), ENV (environment variables), VOLUME (shared filesystems) and EXPOSE (incoming ports) sections)

4 Building infrastructure based on containers: Docker Compose

Starting a multi-component system requires multiple `docker run` executions with a proper parameters, but this task should not do manually every time, because there is the Docker compose tool to overtake it. Docker compose is a tool for defining and running multi-container Docker applications. With Compose, you use a Compose file to configure your application's services. Then, using a single command, you create and start all the services from your configuration.

Using the `docker-compose.yml` file you can define the services (containers to start). Each service is based on a Docker image or is built from a `Dockerfile`. Additionally you can define the previously mentioned parameters (like ports, volumes, environment variables) using that YAML configuration file and finally you can fire up all the services with just one command: `docker-compose up`.

For more details, please read the following guides:

- Get started with Docker Compose
- Compose file reference (image, environment, volumes, ports sections)