Docker Evolution Scenario

Benkler, Niko; Haßberg, Tobias; Heinrich, Robert

Institute for Program Structures and Data Organization

1 Docker

Docker is an open source software using container technology to isolate applications on a system.

An advantage of using this technology is the higher efficiency in the consumption of hardware resources by running independent containers on a single Linux Kernel and avoiding the overhead of maintaining virtual machines. Therefore, this technology provides higher efficiency compared to similar solutions using virtual machines. This leads to an in general lower demand of hardware resources. Also there is no need for updating the existing software installed on your local device apart of the Docker software. The container provides the others needed software in the right version.

Docker builds a so-called *Images* by using a Dockerfile. Starting from an empty server environment, Docker executes the commands of the Dockerfile to set up the docker container, the application environment and the application itself.

By offering to create Images, based on official supported ones, the used software within the containers will be already up to date when theres a new container created.

The docker environment is suitable for efficiently running various containers at a time. It is therefore predestined for Microservice-based applications, as the different Services can be deployed independently in a container.

2 Docker for CoCoME

As described in the CoCoME installation guide, CoCoME needs a bunch of software. This software needs to be kept up to date, no matter what kind of system it will be deployed. Thats why there is the idea of creating an platform independent runnable version of CoCoME, which fulfills the software requirements by itself.

3 Evolution Scenario

Within several steps, CoCoME became more complex. This includes further software which is needed to install and run CoCoME.

The main objective in this evolution scenario is to provide a platform independent version of CoCoME that does not require any preconditions like installing

or updating software. In this case, Docker is a suitable alternative.
By using Docker a version of CoCoME can be instantiated on any device without installing additional software.

4 Implementation Evolution

The main part of the performed work for this adaptive extension consisted in implementing the *Dockerfile*. It defines the way in which the technology stack will be extended. The installation is described in detail in section 5.
Briefly, we are using a Docker container that provides a full Linux distribution and deploys all needed software to accomplish the install of CoCoME as described in the official CoCoME installation guide[1].

5 Description

As shown in figure 1 the changes are affecting the technology stack by adding additional layers. More detailed, the given CoCoME Stack is moved into the Docker Deamon, which runs a Linux distribution. As mentioned, the original parts of the stack, like Glassfish and the Java Virtual Machine, are still a part of the stack.
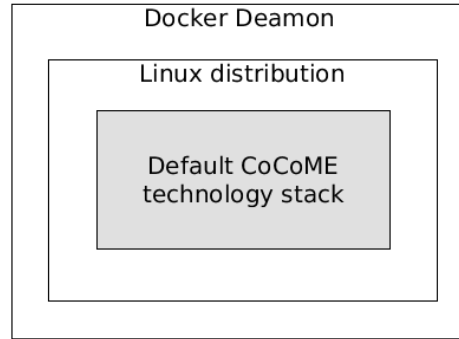
```
Docker Deamon

  Linux distribution

    Default CoCoME
    technology stack
```

Fig. 1. Extended technology stack CoCoME

The Dockerfile defines an environment based on the latest version of Ubuntu 16:04. Onto it there is installed Maven, Git and Java by using the Ubuntu package manager.

---

[1] https://github.com/cocome-community-case-study/
cocome-cloud-jee-platform-migration/blob/master/cocome-maven-project/
doc/Deployment%20Setup.md

Git has two purposes: On the one hand it is used to download the most recent version of CoCoME. On the other hand, it is used to download a prefabricated version of Glassfish that already includes domains and other adjustments required for CoCoME. Java is required by Glassfish and CoCoME as they need the Java Virtual Machine. Maven is needed to deploy the latest version of CoCoME onto the provided Glassfish servers.

6 Deployment

During the development, it was decided to implement and to provide two different versions. The first version always pulls the most recent CoCoME source code from GitHub, downloads the entire dependencies with maven, compiles and builds the project and finally, deploys CoCoME on the Glassfish servers. . As a consequence. creating and starting a Docker Container takes about one hour.

In contrast, the second version only pulls a prefabricated version of CoCoME from GitHub. Therefore, pulling the source code up to building the project is skipped. As a consequence, Maven does not have to be included in the technology stack. Solely, deploying CoCoME on the glassfish server is necessary.

This reduces the deployment time to a few minutes but has a disadvantage: The prefabricated version is updated manually. Therefore, it is sometime not the most recent version.

By providing both, a fast deploying version and a current version, the user can choose whats the best for its situation.
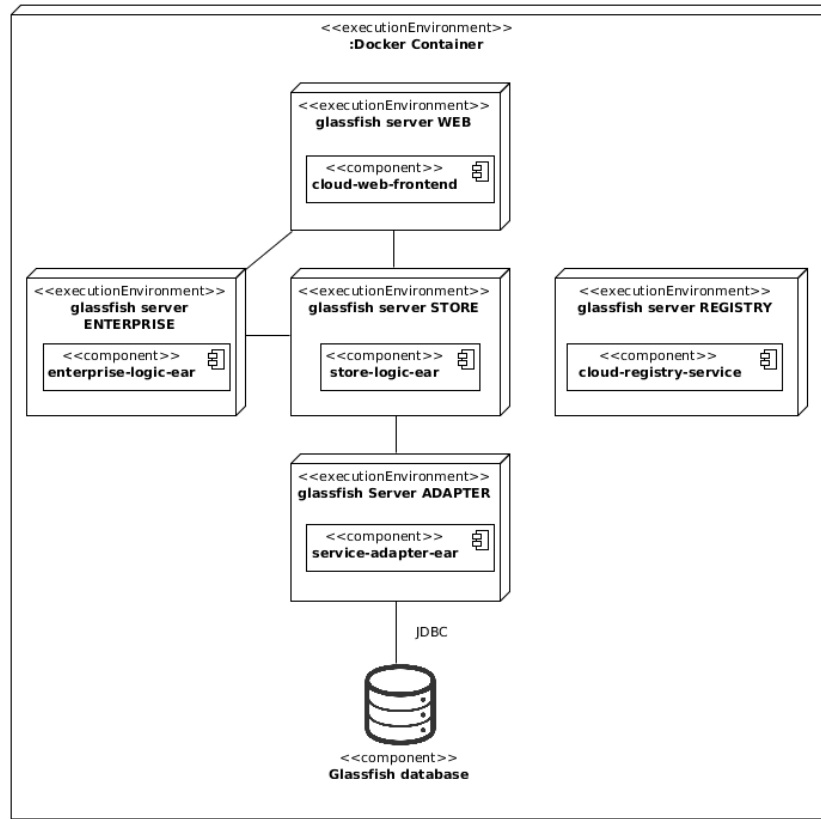
Fig. 2. Deployment diagram CoCoME

As shown above in figure 2 the docker Container contains five different Glass-fish servers. In particular they are called *WEB*, *ENTERPRISE*, *STORE*, *REG-ISTRY* and *ADAPTER* and correspond to the given by the CoCoME deply-oment setup. By default, Glassfish provides a Derby DB that is connected to the server Adapter using Java Database Conectivity (JDBS) interface.

As mentioned before, CoCoME is deployed inside the docker container on the same way it is usually deployed. This means the maven generated archive files *cloud-web-frontend*,*enterprise-logic-ear*,*store-logic-ear cloud-registry-sevice* and *service-adapter-ear* are deployed to the servers with the following assignment:

Server	Deployment file
WEB	cloud-web-frontend
ENTERPRISE	enterprise-logic-ear
STORE	store-logic-ear
REGISTRY	cloud-registry-service
ADAPTER	service-adapter-ear

Fig. 3. Assignment of archive files to Servers

3 demonstrates the assignment between the archive files and the servers as it is implemented and also recommended by the CoCoME deployment guide. This information is also represented in Fig. 2. As mentioned earlier, there are two versions of this Docker project. Both deploy the CoCoME main program with this assignment.

In addition, the fast version can be extends by the pickup shop[2]. This pickup-shop runs inside a separate container which is shown in figure 4.
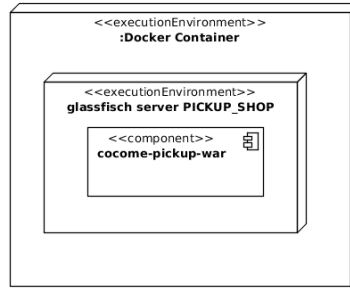

Fig. 4. Deployment diagram CoCoME Pickup Shop

As shown in figure 4, this container provides only one Glassfish server. To

Server	Deployment file
PICKUP_SHOP	cocome-pickup-war

Fig. 5. Assignment archive files to Servers

control the start of both containers, precisely the CoCoME and the Pick Up

---

[2] https://github.com/cocome-community-case-study/
cocome-cloud-jee-web-shop

Shop, another specific file is needed: the Docker Compose file. It ensures that the CoCoME Container is active, before the pickup-shop container is starting. This is necessary as the Pickup Shop requires a running instance of CoCoME to register itself.

Also CoCoME runs without the pickup-shop, the pickup-shop does not work without an running instance of CoCoME.

Both containers need to communicate with each other. By default, docker prohibits any outgoing and ingoing communication from an in a container. This is solved by opening specific ports through which the communication is possible. Which ports the containers can use is specified in the Docker Compose file as well.