DM848: Microservice Programming

Sandboxed Interpreter for Interactive Code Execution Using Docker Virtualization

Anders Busch, anbus12@student.sdu.dk

June 25, 2016

Contents

1 Introduction

When learning a new programming language, it is often useful to have a environment where the learner can execute code, quick and easy without having to worry about breaking the execution environment or installing new software locally.

1.1 Application Goal

The goal of this project was to create a cloud server for executing JOLIE programs through the browser; and to embed the cloud server in a virtual environment provided by the virtualization library DOCKER to provide encapsulation and limit the resource consumption of the cloud server.

1.2 Problems

To achieve the application goal, we had to solve the following problems:

1. Create an DOCKER image for running JOLIE code in a DOCKER container. This image helps automating the process of creating new containers and is available on-line[1] for everyone interested in using DOCKER in conjunction with JOLIE.

2. Produce a service for handling the setup of the DOCKER container. As our execution environment uses JOLIE to communicate between services, we opted to create a service that handles the communication with the DOCKER CLI.

3. Make a service that can evaluate JOLIE within the DOCKER container. For our evaluator to work, we have decided to use a JOLIE service to evaluate sent JOLIE code.

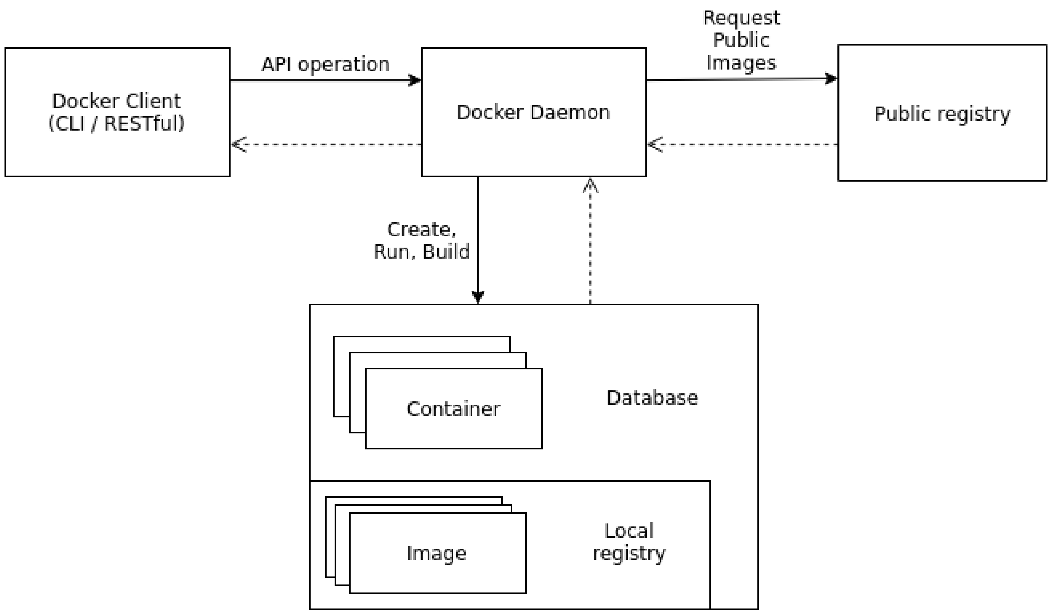
4. Assemble a front-end service for presenting and handling of the user interface. Here we opted for a simple JOLIE-based HTTP server that serves our user interface.

[1]https://hub.docker.com/r/ezbob/jolie/

2 Preliminaries

2.1 Docker Concepts

The DOCKER application is a lightweight virtualization technology that focuses on creating virtualization of applications in microservice manner.[1]



DOCKER application architecture

Architecture DOCKER uses a server-client architecture to communicate between the client, and the underlying server, the DOCKER daemon. The client, in this case, is a front-end program that exposes some interface to the user (Command Line Interface or a RESTful API), and uses this to communicate with the daemon (the server). The server is the main workhorse program that builds, executes and maintains the different DOCKER containers.

Containers A DOCKER container is the virtualization environment that contains a running application, and as such can be thought of an microservice or a instance of a class (using Object Oriented terminology) that the DOCKER daemon maintains. Container provides isolation for the running code and uses a layered file system called the union file system inside the container.[1] This union file system keeps track of changes much in the same way version control systems, such as GIT, does; namely by keeping track of writes and only write to the file system once an action has been committed. Containers are build from images.[2]

Images A DOCKER image, is an read-only minimal-size file that can be used to build new containers via the DOCKER daemon. These images make it possible to share different container-setup with different users, and can be created either using an existing DOCKER container or using a special file called a DOCKER file.

Registries Images can either be fetched from a local repository called the **local registry** or remotely from a repository called the **remote registry**. DOCKER provides a standard remote registry service[2] for fetching official images such as the UBUNTU linux image and user images such as JOLIE image we have created to be used in this application.

Volumes A volume is a way of creating a persistent data volume within a container. A volume is created separately from any containers, initialized with any container and can be mounted to one or more containers for sharing of persistent data.

Docker Engine The DOCKER daemon together with the front-end service is called the DOCKER engine. The engine communicates directly with the host operating system, on behave of the containers. This is different to most other virtual environments, where a hypervisor is employed support the virtual operating system running in the virtual environment.[1]

Encapsulation and Resource Configuration DOCKER uses two Linux kernel concepts to provide encapsulation and resource configuration for the containers; *control groups* and *namespaces*.

Control groups or *cgroups* enables the user to configure the resources of an container. This useful when one wants to limit for example the memory footprint or the number of CPU cores available for the container to use on the host machine.

Namespaces enables the encapsulation of a container as a process assigning some resources to it. Using namespaces limits the access from the container to the host system.

[2]https://hub.docker.com/

3 Technical Description

3.1 Application Overview

Our application uses JOLIE as the main choice of programming language, and is build around a number of microservices:

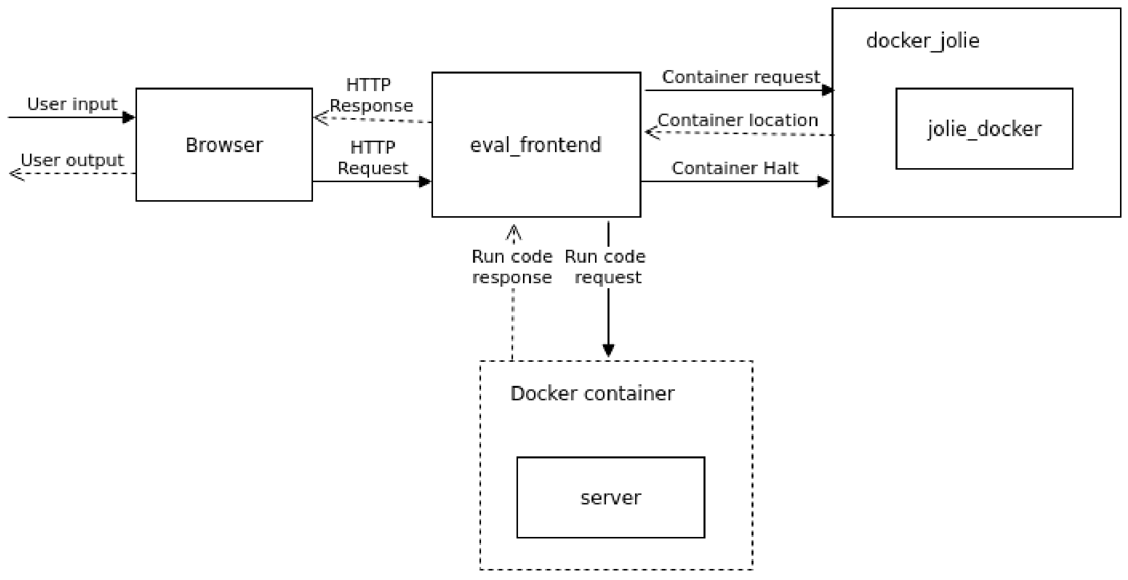
jolie_docker An embedded JAVA service that works as an interface for the DOCKER CLI. This service is responsible for creating containers and halting them through the DOCKER CLI.

docker_jolie This service is responsible for embedding the **jolie_docker** service and act as an container server for JOLIE clients.

eval_frontend A service that serves as the HTTP server front-end for the application.

server A JOLIE service, that serves as the cloud server embedded inside the DOCKER container.

The following diagram illustrates how the different services are connected:



Overview of our application

In the diagram, the dashed arrows indicates responses and the full arrows indicates request for some data.

3.2 Configuration of the Docker Container

When a service request a container, we pass some configuration parameters to DOCKER to limit the consumption of the host resources and to mount a certain portion of the host file system to hold the compiled JOLIE file created within the container. The following configuration parameters are parsed during the initialization phase:

–read-only makes the root file system read-only within the container.

-m 256m sets the main memory limit for the container to 256 mega bytes.

–cpu-shares 256 sets a relative weighted upper bound on the how much the container can use the CPUs. The maximum number of shares is 1024.

–cpuset-cpus 0,1 the container may only execute on CPU core 0 and 1.

–expose ⟨port⟩ exposes a single port (selected by the client), to enable the communication with the contained service.

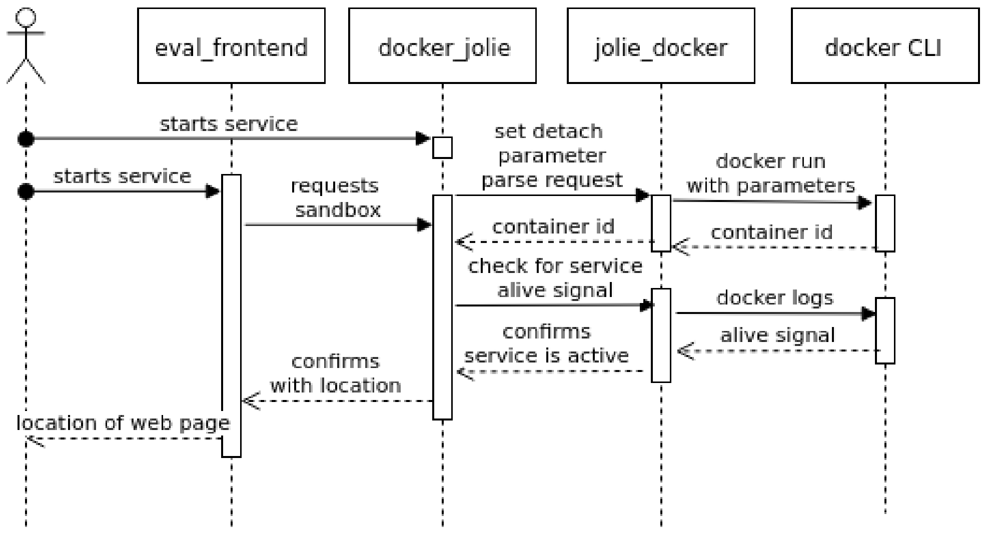
Furthermore, the JOLIE image uses an user called "jolie" with no root privileges and uses the home directory "/home/jolie" as the main workspace for execution of JOLIE programs.

3.3 Operations

Our application supports three main operations:

1. Initialization and creation of a container using a configuration that limits the container resources.
2. User-inputted code that was submitted using the browser editor will be executed by the contained service.
3. Shutdown of the application and removal of the created containers.

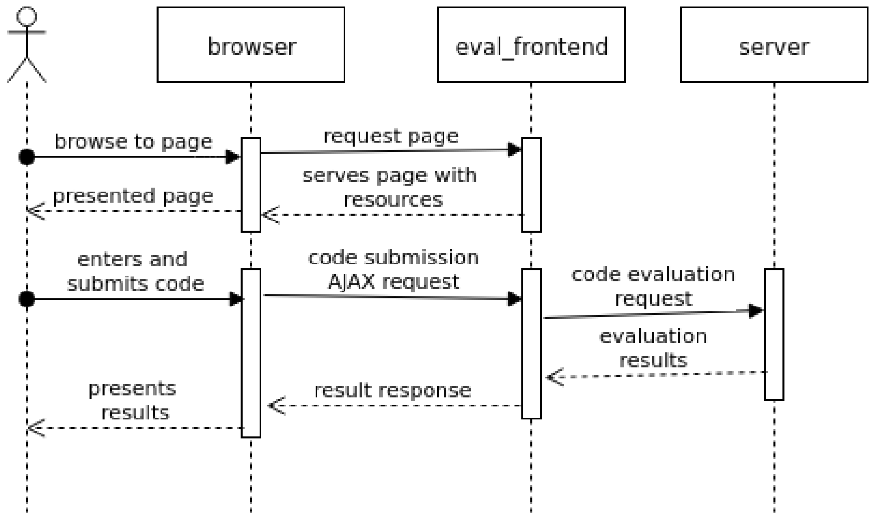
The initialization process is depicted in the following diagram:



The initialization process

To start the initialization process the user has to start the **docker_jolie** service first and then the **eval_frontend** service. Once this is done, the initialization phase begins by creating a new container which contains the service **server**. A synchronization phase is used between **jolie_docker** and **docker CLI**. This is done via examining the DOCKER logs for a pre-determined signal, emitted by **server**, to ensure that the contained service is up before proceeding.

The submission process is depicted in the following diagram:



The code submission phase

4 Related Work and Discussion

Although we partly achieved what we set out to create; there are some limitation in the current implementation that can be expanded upon in future revision of this execution tool, namely the support for inter-service communications in the context of the code executioner, better error explanation and dynamic linting[3] of the program code.

We would also have liked to generalize the **docker_jolie** service in such a way that it could handle the communication between the container and the JOLIE client in a way that would allow for remote access to the container.

[3]https://en.wikipedia.org/wiki/Lint_%28software%29

References

[1] Charles Anderson. Docker. *IEEE Software*, 32(3):102, 2015.

[2] Docker Inc. Docker.com: understanding docker, 2016.