Technische Universität München
Fakultät für Informatik

Exercise Sheet 12: Continuous Deployment, Virtual machines and Containers

Welcome to the last exercise of Advanced Topics of Software Engineering! This week is about continuous deployment, virtual machines (VMs) and containers.

First, we will discuss Docker and Docker Compose, two powerful tools for virtualization via containers as introduced in the lecture. We show how to *containerize* a multi-tier application and orchestrate the containers in a scalable way. Second, we explain how to setup and connect to a VM in the Google Cloud Platform (GCP).

In the project exercises, you will deploy your ASEDelivery with Docker Compose to a VM in Amazon AWS or ngrok. As you will need to deploy your ASEDelivery in a *continuous* way by means of GitLab CI/CD, we further show how to automate this process.

In summary, this exercise sheet covers the following topics:

1. Docker: A virtualization tool to package software into containers
2. Orchestrating multi-container applications with Docker Compose
3. Continuous integration, containerization, and deployment with GitLab CI and AWS/ngrok

Practical Exercise

Exercise 1: Virtualization via Docker Containers

During a boring company meeting at work, you have prototyped a small meeting-time tracking application in Java and, with your good old ASE lecture in mind, have packaged it into a `.jar` file using Maven. Now, you are excited to show it to your coworkers, to increase the efficiency of company meetings, and therefore apply for access to a VM to deploy the application on. Unfortunately, your company is very old-fashioned and since the administrators of the company-internal VM infrastructure want to keep the VMs as clean as possible, you are not allowed to install a Java runtime on the VM.

After ranting about the infrastructure guys for a while, you notice that they have added your linux user account on the VM to the docker group, which means you can make full use of all Docker features. You start writing down the requirements for a Docker image for your application, which is a read-only template used to build containers:

- The JDK used to run the application must be of version 8[1]
- The application runs on port `8080` which should therefore be exposed in the container
- The `app.jar` file should be copied into the image at location `/app`, which is also considered the main working directory for any instruction in the `Dockerfile`[2]

[1] OpenJDK Official Docker Images: https://hub.docker.com/_/openjdk
[2] Dockerfile Reference: https://docs.docker.com/engine/reference/builder/#workdir

With the list of requirements, you see yourself confronted with the following questions:

1. How can you translate the requirements into a `Dockerfile`, which textually defines a Docker image?
2. How can you build the Docker image from the `Dockerfile` with an appropriate name and version tag, e.g. `time-tracking-app:1.0`?
3. How can you create and run a new container from the created local Docker image and bind its port 8080 to the host's port 80?
4. The meeting-time tracking application writes logs into the directory `/logs`, which is created on the start of the application. Since these logs will be stored only inside the container if it is run with Docker, you need to find a way to store them in your user's home directory the host system (i.e. `/~` on the VM) as well.[3]
5. Assuming you want to build the image on a different PC (e.g. your company laptop), how could you send this image to the company's Docker registry and then pull it onto the VM?

Exercise 2: Container Orchestration with Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. It uses YAML files to configure the different services, which can be components (e.g. frontend, database) of your application with one or more containers each. The `docker-compose` CLI utility allows users to run commands on multiple containers at once, for example, building images, scaling containers, running containers that were stopped, and more.

When you orchestrate multiple Docker containers with Docker Compose, you are implicitly creating a network for the containers in which they can communicate with each other by simply referencing another service by container name and port. Every service can have its own environment variables, dependencies to other services, volume mounts, port configuration, and can be scaled independently of other services.

To demonstrate the use of Docker Compose in a multi-tier application, we will revisit the project management Spring Boot application from exercise sheet 3. This application has three tiers in its simplest form without service discovery or API Gateway:

- A frontend service containing a web app that displays the list of projects that is fetched via HTTP from the backend
- A backend service which provides a REST API to query projects obtained from the SQL database
- A database service which runs a Mongo database server

The goal of this exercise is to **containerize** these tiers and **orchestrate the containers** with Docker Compose. For this you need to complete the following tasks:

1. Navigate into `exercise_12.zip/compose-app` in your local copy of the ASE repository. You should see two Spring Boot applications there, a Spring Boot `backend-service` and a React `frontend-service`.

   Unfortunately, running both of them via `mvn` or in an IDE will fail, as you need a running MongoDB database server on port 27017 with specific user credentials and databases to start the `backend-service`.
2. Containerize the services with Docker by providing a `Dockerfile` in the services' root directories. For the backend, your Docker image will most likely require a packaged Java application (as `.jar`), so you need to configure the `Dockerfile` such that it copy the `.jar` file into Docker, and execute this `.jar` when the Docker container runs. For the frontend service, you need to do a similar step. The difference is you need to copy the necessary files to build and run your React app in a Docker container.

[3] Docker CLI reference: https://docs.docker.com/engine/reference/run/#volume-shared-filesystems

3. Add a `docker-compose.yml` file to the root directory and add 3 services, namely `backend-service`, `frontend-service` and `db`. Set an appropriate build image for each service, the dependencies between them and expose relevant ports for each service.[4]

4. Update any URL used for the communication between the services to contain the service names instead of `localhost`. It is probably the best idea to use environment variables to keep URLs with `localhost` as default.

5. Configure the Mongo database server with `<ServerName>` as the container name. Create a database named `aseProject` and a database user `aseUser` with the password `ase`. You will need to figure out how to setup the mongo Docker container [5] on the `backend-service` as well as using environment variables.

6. To check if everything is working as expected, simply run `docker-compose up` and open a browser window at `localhost:8080`.

   You can run the containers in the background by providing the option `--detach` to the previous command and stop them by running `docker-compose down` in the same directory.

7. Until now, each service is run in a single container. Figure out how you can **scale** specific services to more than one container, which is one of the great advantages of containerized deployment. Check if multiple containers are spawned by observing the list of running containers with `sudo docker ps -a`.

Exercise 3: Continuous Integration and Containerization with GitLab CI

In the lecture, you learned about continuous integration (CI) and continuous deployment (CD), two crucial practices of modern software deployment. The continuous early feedback from CI tools allows developers to detect bugs early and track them down more easily. CD prescribes to regularly deploy software artifacts to production environments for user acceptance testing and to obtain customer feedback. Typically, the steps involved in CI and CD tasks are triggered and executed in pipelines when a developer submits changes to the code base.

You already got to know the tool GitLab CI on exercise sheet 9, which you will use again in this exercise for CI/CD pipeline configuration. In this exercise, you will not only execute tests (see exercise sheet 9) and package an application, but also publish the packaged artifacts as Docker images to a container image repository. These images will then be used in the next exercise, to deploy the application with Docker Compose to Amazon Web Service (AWS) or ngrok. The process and how it is divided between the exercises is illustrated in figure 1.

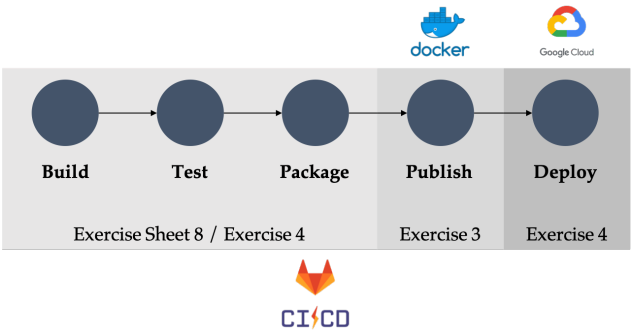

Figure 1: CI/CD Process with GitLab CI

The simple project management example application from exercise sheet 3 is reused, which contains two Spring Boot applications (i.e. `frontend-service` and `backend-service`) resulting in two images. Create a new GitLab repository, enable the **Pipelines** and **Container registry** feature for it in **Settings > General** and copy the two service directories from `exercise_12.zip/compose-app` including the `docker-compose.yml` from the previous exercise into the repository.

[4] Docker Compose Reference: https://docs.docker.com/compose/compose-file/
[5] "Docker MongoDB: https://hub.docker.com/_/mongo"

Note: If you do not have a working setup of a GitLab runner from exercise sheet 9, now is a good time to revisit the exercise sheet and set the runner up with correct Docker permissions (see https://docs.gitlab.com/ee/ci/docker/using_docker_build.html#runner-configuration). For compatibility, you can register a Gitlab Runner as a Docker service, so that all your team members can follow the same runner registration setup without depending on the OS. In this case, using a Docker socket binding (see https://docs.gitlab.com/ee/ci/docker/using_docker_build.html#use-docker-socket-binding) is a straightforward way to configure your Gitlab Runner.

Tip: You can use CI/CD variables for shared variables used across multiple jobs. The variable can be defined in Project Settings > CI/CD > Variables.

Your task is to write a `.gitlab-ci.yml` file, which contains CI pipeline configuration for the following four stages with respective jobs per service [6]:

1. **Build:** Runs `mvn clean compile` in each of the two services' directories to build them (2 jobs)
2. **Test**: Runs `mvn test` in each of the two services' directories to run their tests (2 jobs); be aware that the `backend-service` requires a Mongo database to run its tests, which you can simply link to the CI job with GitLab CI services and environment variables (e.g. `MONGO_URI`, see `application.yml`):
3. **Package**: Runs `mvn package` in each of the two services' directories to package them (without test execution!) into `.jar` artifacts, which you must store for subsequent jobs[7] (2 jobs)
4. **Publish**: Builds new Docker images for both services in their respective directories and pushes them to the GitLab container registry (2 jobs)[8]

Additionally, each of the jobs should only be executed if there was a change in the relevant source code directories[9], e.g. the jobs for `frontend-service` should only be run if there was a change in `frontend-service/**/*`.

After you are done, check if your pipeline works as expected by observing its execution in the **CI** / **CD** and the image creation in the **Packages > Container Registry** view.

After having published the Docker images of two services to the Docker container registry in the previous exercise, you are now ready to deploy your application to a cloud environment as instructed in the next exercise. If you would like to use ngrok, the last exercise will give you hint on how to deploy a globally accessible system from your local machine.

Exercise 4: Continuous Deployment to the Amazon Web Services (AWS) using GitLab CI and Docker Compose

You will use Docker Compose as an orchestration tool for containers of the services that you have packaged and published as Docker images.

There are numerous cloud providers of all sizes which offer infrastructure as a service, i.e. VMs of arbitrary configuration. In the following, we will walk you through the steps necessary to rent and obtain access to a VM on AWS. Then, you will use SSH to connect to the VM in your GitLab CI pipeline and start the orchestration of Docker containers required by the simple project management example application from the previous exercises.

Note: AWS gives a free-tier usage for one year, but it will ask you to pay 1 dollar for credit/debit card verification. A credit/debit card that enables only transactions within Germany may not be valid for verification. For the course project, **we are also opened to other cloud computing platforms aside from Amazon AWS**.

1. To get started, create an AWS account at https://portal.aws.amazon.com/billing/signup#/start and verify your credit/debit card information. For the server location, you can choose then nearest server in Frankfurt (Central Europe: eu-central-1).

[6] GitLab CI/CD Pipeline Configuration Reference: https://gitlab.lrz.de/help/ci/yaml/README

[7] GitLab CI Artifacts: https://gitlab.lrz.de/help/ci/yaml/README#artifacts

[8] Building Docker Images with GitLab CI: https://docs.gitlab.com/ee/ci/docker/using_docker_build.html and https://docs.gitlab.com/ee/ci/docker/using_docker_build.html#container-registry-examples

[9] GitLab CI Only/Except for Changes: https://docs.gitlab.com/ee/ci/yaml/#onlychangesexceptchanges

2. Navigate to the **AWS console** (https://console.aws.amazon.com/) and create a new EC2 instance by selecting **Launch a virtual machine** with EC2. If the payment validation process is not finished at this time, you may have to wait up to 24 hours to configure your instance.
3. On the next screen, select or search for an Ubuntu Server 20.04 LTS with an x86 CPU architecture Amazon Machine Image (AMI). Although we use this AMI for the exercise, you are free to choose your preferred OS image when deploying your ASEDelivery system.
4. For free tier usage, the maximum permitted configuration is **t3.micro** instance type with an 8 GiB storage size.
5. For security group, you need to configure the firewall traffic for inbound connection to allow requests from Gitlab and your team's computers.

   Add a new inbound rule with type SSH, and the source is *Anywhere-IPv4*. Add another inbound rule for the frontend component with type **Custom TCP**, **port 8080**, and the source to **Anywhere-IPv4**. Then save the rules.
6. Keep the rest of the VM instance settings to the default configuration and launch the VM.
7. AWS will ask you to create a key pair for the instance, You will be prompted to input a file path to save your SSH key pair to. If you already have an SSH key pair, you will need to choose a different name for the new key pair.
8. In the list of instances at https://eu-central-1.console.aws.amazon.com/ec2/v2/home?region=eu-central-1#Instances:, you should see the instance that you just created with its **External IP**, **Domain name** (Public IPv4 DNS). You can right-click on the instace and click **Connect**, a new page is open containing the IP and username that you will connect via **SSH**. Click **Connect** again to open the SSH terminal in the browser.

   Alternatively, you can run `ssh -i PATH_TO_PRIVATE_KEY USERNAME@VM_EXTERNAL_IP` on your local machine.
9. Install Docker and Docker Compose on the VM by following the steps at https://docs.docker.com/engine/install/ubuntu/#install-using-the-repository and https://docs.docker.com/compose/install/#install-compose-on-linux-systems.

You should now have a VM running in the AWS with Docker (Compose) installed, which you can access via SSH through a SSH key pair stored on your local machine.

The task for this exercise is to update the `.gitlab-ci.yml` file in your repository from exercise 2, to include a **single job** in the final **deploy** stage, which

i connects to the VM via SSH (you may want to use GitLab CI/CD environment variables[10]),

ii pulls the most recent version of the Docker images from the GitLab container registry (remember to log in with your Gitlab ID and Access Token).

iii restarts all containers via Docker Compose.

Hint on Container Registry Config: In the Container Registry of a project, you will see the commands needed for login, build, and push an image. These commands are necessary for you to push your backend and frontend images to the Registry, so that in AWS the images can be pulled by Docker, and deployed by Docker Compose.

For logging into Container Registry, you can create an Access Token instead of using your passwords. In your project, navigate to Settings > Access Tokens to create a new one with **read_registry** and **write_registry** scopes.

Check if everything is working as expected by opening a browser at
`http:``//VM_EXTERNAL_IP:FRONTEND_SERVICE_PORT/` after you are done with the deployment configuration (https will not work by default).

Depending on which `FRONTEND_SERVICE_PORT` you picked, you will have to create a custom firewall rule allowing specific IP ranges to access your service at a specific port other than 80 (default HTTP) or 443

[10] Using SSH keys with GitLab CI/CD: https://docs.gitlab.com/ee/ci/ssh_keys/

(default HTTPS) and add the rule to the VM. You can simply do so by configuring your inbound rules in AWS Security Groups https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/security-group-connection-tracking.html, e.g. if your `frontend-service` is exposed at port 8080, you need to create a rule that allows inbound traffic of the IP range `0.0.0.0/0` at ports `tcp:8080`.

Hint: You may create a new `docker-compose.prod.yml` file to run and orchestrate the containers on the VM, which uses the Docker image names from the GitLab container registry (which are most likely different to your local image names!).

Exercise 4: Continuous Deployment to a local machine using ngrok, GitLab CI and Docker Compose (Optional)

ngrok [11] is a tunneling tool that exposes your local web server to the Internet. ngrok combines with Gitlab enable you to demo the system using your own machine or an docker image as the running environment. Still, for a freemium ngrok account, you can only expose one <host>:<port> address, so you will need multiple ngrok accounts to run your microservice. Therefore, this exercise is optional and can be helpful when you develop monolithic system in other courses with Gitlab. We recommend you to deploy your system on a cloud computing platform. **Platforms other than Amazon AWS are still acceptable for the project deployment**.

Overall, deploying a system with ngrok is similar but simpler than in a cloud computing platform like AWS. In this exercise, you will setup your Gitlab CI/CD pipeline deployment to use Docker Compose for orchestrating the Docker containers of your services, and then expose the frontend service to the Internet using ngrok.

You can orchestrate the containers by using images from the Gitlab registry, or by building the images from the binaries and source code in your local machine. If your team has a branching strategy, you should carefully consider the side effects and configuration of each deployment method before applying them.

The recommended steps to setup an ngrok deployment:

1. Register an ngrok account at https://dashboard.ngrok.com/signup
2. Install ngrok in your machine, you can check the tab **Setup Installation** for the specific setup instruction for different OS. You can consider the following ways to run ngrok locally:
   - Run ngrok from your local shell to expose the frontend service's port using the ngrok executable.
   - Run ngrok as a Docker container, this will let you run ngrok in another Docker container that you use to deploy your system in Gitlab CI/CD pipeline. An example ngrok Docker container is wernight/ngrok at https://hub.docker.com/r/wernight/ngrok/.
3. Configure your ngrok **Authtoken** in the running environment using the following command:

   ```
   ./ngrok authtoken <your ngrok Authtoken>
   ```

   An ngrok.yml file should be created under **/.ngrok2** folder. If you run ngrok as a Docker container service, you can create a docker volume with a **/.ngrok2** folder inside to store the ngrok.yml file, then mount the volume into the ngrok Docker container so that it recognizes your ngrok account during deployment.
4. Add a **deploy** stage to your CI/CD pipeline from exercise 2 to:
   (a) Pull the most recent version of the Docker images of your services from the Gitlab Container Registry.
   (b) Use Docker Compose to orchestrate your services
   (c) Run ngrok to expose the frontend service, either by using a shell command or an ngrok Docker container.
5. In the Dashboard of your ngrok account page. Navigate to Endpoints > Status, and Refresh the webpage. You should see two http and https URLs which are the ngrok tunnels to your localhost. Accessing any of the URL in the browser should display a list of projects with their name and ID.

[11] ngrok: https://ngrok.com/product

At this stage, you have learned how to deploy your system and make it globally accessible to the world. Which is the last tutorial necessary for the accomplishment of your ASEDelivery project. We wish you good luck to your project development, and **do not hesitate to visit us during the consulting hours sessions if you have questions or would like to ask for feedback**.

We are looking forward to seeing your final deliverables.