Dockerize Spring Applications

- Introduction to containers and Docker
- Present the Dockerfile and how to wrap a Spring Boot Jar into a Docker Image
- How to add environment variables to a Spring Docker image so that they will be propagated to the Spring Context
- Upload the image the a Docker registry
- Describe concepts of Continuous Integration / Continuous Deployment
- Practical example with hands-on lab

Introduction

Docker is a technology where developers or DevOps teams can build, deploy, and manage applications by using containers. Docker is an open source software so that everyone can run this on their own operating system, which should support virtualization and Docker for Mac/Windows/Linux.

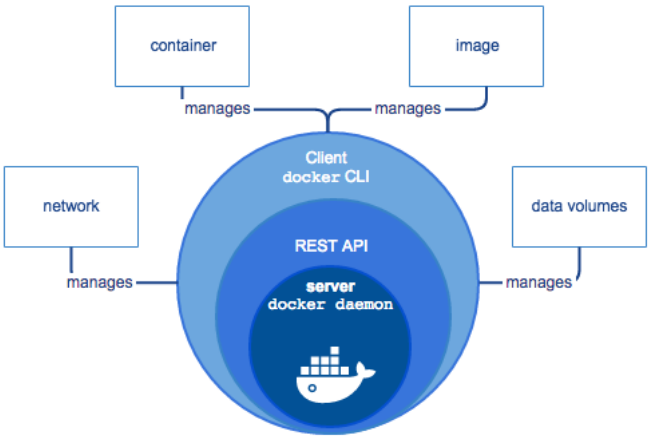
Docker also ships the ready images from one computer to another. Docker containers are sets of processes that are isolated from the rest of the processes in the host OS. Docker shares the kernel of the host operating system. On the other hand, VM is a technology that depends on the guest OS, which is completely separated from the host OS. Virtual machines communicate with the host OS through hypervisor. VM requires many hardware resources and processes. It's not a container-based technology.

> Docker moves up the abstractions of resources from hardware level to the operating system level. That's why application portability and infrastructure separation is easier.

Container-based technology takes lower resources of the host system. Most of the time, container-based technology uses the kernel of the host machine.

Basic Concepts

Docker Engine works as a client-server architecture. Docker daemon works as a server, which is the core part of Docker and is run on the host operating system. This service exposes some rest APIs that can be used by the client. A command line interface (CLI) client uses the services provided by Docker daemon with **Docker** command.



Docker Daemon

Important Terms

Image: Image is the executable application package file inside Docker. It contains everything needed to run an application, including libraries, config files, runtime, etc. It's a snapshot of a container.

Container: An instance of an image is called a container. When an image is executed and takes place in memory, then the instance of this image is called a container. It executes in a completely isolated environment.

> In OOP terms, Image is a class, and a container is an instance of this class — a runtime object.

Registry: A registry is a storage and content delivery system that stores Docker images. Docker Hub is a popular Docker registry. We can store different versions of images with different tag numbers to Docker Hub.

Dockerfile: It's a text file that contains all the commands to assemble an image.

> Dockerfile > (Build) > Image > (Run) > Container

```
FROM openjdk:latest

ARG JAR_FILE=target/cloud-stream-0.0.1-SNAPSHOT.jar
ADD ${JAR_FILE} app.jar

ENV JAVA_OPTS=""
ENV RABBIT_HOST="127.0.0.1"
ENV RABBIT_PORT=5672
ENV RABBIT_USER="guest"
ENV RABBIT_PASS="guest"

EXPOSE 8080
ENTRYPOINT exec java $JAVA_OPTS -Djava.security.egd=file:/dev/./urandom -jar app.jar
```

1. **FROM** – determines what base image we will be using in our docker image. This is helpful such as we don't need to create an image from scratch. We can then just create an image on top of a working image. In this case, we have used openjdk image that's already configured to have java installed. You can specify what version of openjdk you wanted to use by using the format openjdk:version eg. opendjk:8 for JDK 8. We also added -alpine which means that we will be using a minified version of Linux container making our image size smaller.

2. **VOLUME** – means that docker will create /tmp volume inside the container. This is useful if we will be using the file system in our application, for example, writing a log file. It's actually not needed in our simple spring boot application but it's good to have it ready in place.
3. **ARG** – creates a variable that you can use in your dockerfile.
4. **COPY** – accepts 2 arguments which are the source and destination files.
5. **ENTRYPOINT** – the command that will be run to start the application when running the container. Here in our example, we wanted to run java -jar /app.jar, thus, we added it in our ENTRYPOINT array.

```
##### IMPORTANT - Make sure that a repository is created before pushin

docker login -u "dragosn" -p "pass" docker.io

# to build image
docker build -t ${image-name}:latest . # where . is the path to the docker file
# tag the image
docker tag ${image-id} ${image-name}:latest

#create a network to be shared by the containers
docker network create ${network-name}
# run image
docker run -d --name ${name} -p 8080:8080 ${image-name}
# docker run -d --name spring-app -p 8080:8080 --network internal-container-network spring-app
# docker run -d --name spring-app -p 8080:8080 -e RABBIT_PORT=5673 -e RABBIT_HOST=rabbitmq --network internal-container-network
```

bash to build docker image

```
#!/bin/bash

set -o pipefail

IMAGE=...your image name...
VERSION=...the version...

docker build -t ${IMAGE}:${VERSION} . | tee build.log || exit 1
ID=$(tail -1 build.log | awk '{print $3;}')
docker tag $ID ${IMAGE}:latest

docker images | grep ${IMAGE}
```

Build docker image using maven

The design goals are:

- Don't do anything fancy. `Dockerfile`s are how you build Docker projects; that's what this plugin uses. They are mandatory.
- Make the Docker build process integrate with the Maven build process. If you bind the default phases, when you type `mvn package`, you get a Docker image. When you type `mvn deploy`, your image gets pushed.
- Make the goals remember what you are doing. You can type `mvn dockerfile:build` and later `mvn dockerfile:tag` and later `mvn dockerfile:push` without problems. This also eliminates the need for something like `mvn dockerfile:build -DalsoPush`; instead you can just say `mvn dockerfile:build dockerfile:push`.
- Integrate with the Maven build reactor. You can depend on the Docker image of one project in another project, and Maven will build the projects in the correct order. This is useful when you want to run integration tests involving multiple services.

```
<!-- DOCS: https://github.com/spotify/dockerfile-maven -->

<!-- Maven Docker plugin -->

<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>dockerfile-maven-plugin</artifactId>
  <version>1.4.0</version>
  <configuration>
    <!-- replace `{docker_id}` with your docker id-->
    <repository>{docker_id}/${project.artifactId}</repository>
    <tag>${project.version}</tag>
    <buildArgs>
      <JAR_FILE>target/${project.build.finalName}.jar</JAR_FILE>
    </buildArgs>
  </configuration>
  <executions>
    <execution>
      <id>default</id>
      <phase>install</phase>
      <goals>
        <goal>build</goal>
        <goal>push</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

```
# build the docker image
mvn package dockerfile:build

# push the docker image to the docker registry
mvn dockerfile:push

# OR
mvn dockerfile:build dockerfile:push
```

Continuous Integration / Continuous Deployment

> Modern software moves fast and demands more from developers than ever. Tools and concepts around CICD help developers deliver value faster and more transparently.

CI : Continuous Integration

The CI part of CICD can be summarized with: you want all parts of what goes into making your application go to the same place and run through the same processes with results published to an easy to access place.

The simplest example of continuous integration is something you might not have even thought of being significant: committing all your application code in a single repository! While that may seem like a no-brainer, having a single place where you “integrate” all your code is the foundation for extending other, more advanced practices.

Once you have all your code and changes going to the same place, you can run some processes on that repository every time something changes. This could include:

- Run automatic code quality scans on it and generate a report of how well your latest changes adhere to good coding practices
- Build the code and run any automated tests that you might have written to make sure your changes didn’t break any functionality
- Generate and publish a test coverage report to get an idea of how thorough your automated tests are

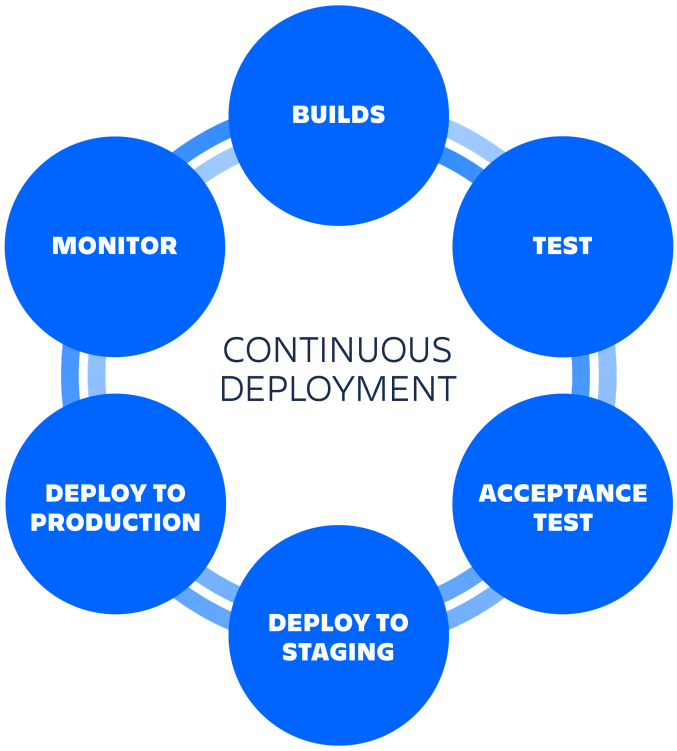
These simple additions (made easy with tooling that will be mentioned later) allows you, the developer, to focus on writing the code. Your central repository of code is there to receive your changes while your automated processes can build, test, and scan your code while providing reports.

CD : Continuous Deployment

Deploying code can be hard. If you've ever been jamming on building a project for a while, shifting your mindset to getting it ready to be deployed can be jarring.

One of the best things you can do to avoid this, much like other things in software, is to automate it! Make it so that your code gets automatically deployed to wherever you or your users can get to it.

There are many freely available tools to let you do this easily. One popular example is Travis CI, which integrates directly with Github. You can configure Travis to automatically run CI tasks like unit tests and push your code to a hosting platform like Heroku every time you push new changes to a branch.



Continuous Deployment

Continuous Deployment Best Practices

Once a continuous deployment pipeline is established, ongoing maintenance and participation is required from the engineering team to ensure its success. The following best practices and behaviors will ensure an engineering team is getting the most value out of a continuous deployment pipeline.

- **Test-driven development** Test-driven development is the practice of defining a behavior spec for new software features before development begins. Once the spec is defined developers will then write automated tests that match the spec. Finally, the actual deliverable code is written to satisfy the test cases and match the spec. This process ensures that all new code is covered with automated testing up front. The alternative to this is delivering the code first and then producing test coverage after. This leaves opportunity for gaps between the expected spec behavior and the produced code.
- **Single method of deployment** Once a continuous deployment pipeline is in place, it's critical that it is the only method of deployment. Developers should not be manually copying code to production or live editing things. Manual changes external to the CD pipeline will desync the deployment history, breaking the CD flow.
- **Containerization** Containerizing a software application ensures that it behaves the same across any machine it is deployed on. This eliminates a whole class of issues where software works on one machine but behaves differently on another. Containers can be integrated as part of the CD pipeline so that the code behaves the same on a developer's machine as it does during automated testing, and production deployment.