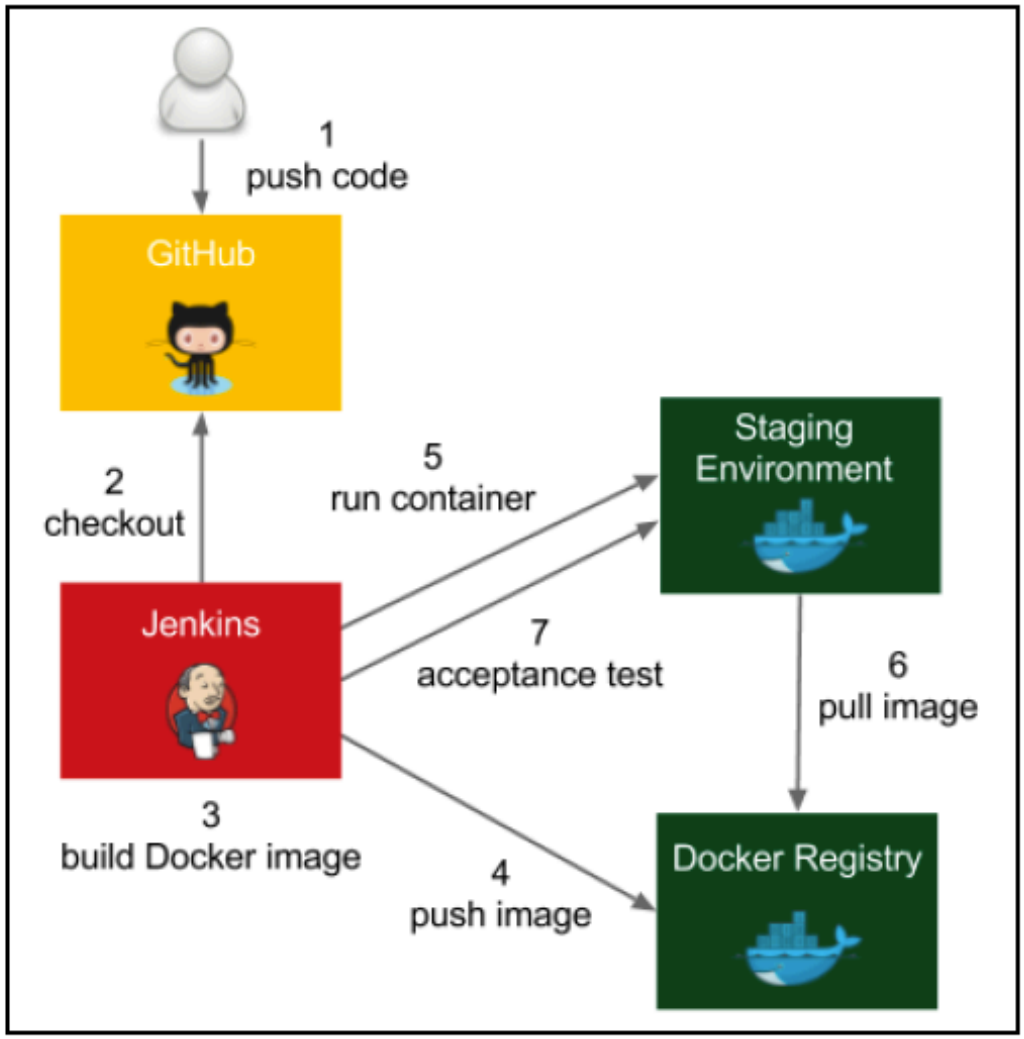
# Automated Acceptance Testing

Raúl Estrada

Octubre 2020

Let's look at the figure that presents the process we will use:



The process goes as follows:

1. The developer pushes a code change to GitHub.
2. Jenkins detects the change, triggers the build, and checks out the current code.
3. Jenkins executes the commit phase and builds the Docker image.
4. Jenkins pushes the image to Docker registry.
5. Jenkins runs the Docker container in the staging environment.
6. Staging the Docker host needs to pull the image from the Docker registry.
7. Jenkins runs the acceptance test suite against the application running in the staging environment.

For the sake of simplicity, we will run the Docker container locally (and not on a separate staging server). In order to run it remotely, we need to use the `-H` option or to configure the `DOCKER_HOST` environment variable. We will cover this part in the next chapter.

Let's continue the pipeline we started in the previous chapter and add three more stages:

- `Docker build`
- `Docker push`
- `Acceptance test`

Keep in mind that you need to have the Docker tool installed on the Jenkins executor (agent slave or master, in the case of slave-less configuration) so that it is able to build Docker images.

# The Docker build stage

We would like to run the calculator project as a Docker container, so we need to create Dockerfile and add the "`Docker build`" stage to Jenkinsfile.

# Adding Dockerfile

Let's create Dockerfile in the root directory of the calculator project:

```
FROM frolvlad/alpine-oraclejdk8:slim
COPY build/libs/calculator-0.0.1-SNAPSHOT.jar app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Dockerfile uses a base image that contains JDK 8 (`frolvlad/alpine-oraclejdk8:slim`). It also copies the application JAR (created by Gradle) and runs it. Let's check if the application builds and runs:

```
$ ./gradlew build
$ docker build -t calculator .
$ docker run -p 8080:8080 --name calculator calculator
```

Using the preceding commands, we've built the application, built the Docker image, and run the Docker container. After a while, we should be able to open the browser to `http://localhost:8080/sum?a=1&b=2` and see `3` as a result.

We can stop the container and push the Dockerfile to the GitHub repository:

```
$ git add Dockerfile
$ git commit -m "Add Dockerfile"
$ git push
```

## Adding the Docker build to the pipeline

The last step we need is adding the `"Docker build"` stage to Jenkinsfile. Usually, the JAR packaging is also declared as a separate `Package` stage:

```
stage("Package") {
      steps {
            sh "./gradlew build"
      }
}

stage("Docker build") {
      steps {
            sh "docker build -t leszko/calculator ."
      }
}
```

Note that we used the Docker registry name in the image tag. There is no need to have the image tagged twice `calculator` and `leszko/calculator`.

When we commit and push Jenkinsfile, the pipeline build should start automatically and we should see all boxes green. This means that the Docker image has been built successfully.


There is also a Gradle plugin for Docker that allows executing the Docker operations within Gradle scripts. You can see an example at: `https://spring.io/guides/gs/spring-boot-docker/`.

# The Docker push stage

https://spring.io/guides/gs/spring-boot-docker/

When the image is ready, we can store it in the registry. The `Docker push` stage is very simple. It's enough to add the following code to Jenkinsfile:

```
stage("Docker push") {
     steps {
          sh "docker push leszko/calculator"
     }
}
```

If Docker registry has the access restricted, then first we need to log in using the `docker login` command. Needless to say, the credentials must be well secured, for example, using a dedicated credential store as described on the official Docker page: `https://docs.docker.com/engine/reference/commandline/login/#credentials-store`.

As always, pushing changes to the GitHub repository triggers Jenkins to start the build and, after a while, we should have the image automatically stored in the registry.

# Adding a staging deployment to the pipeline

Let's add a stage to run the `calculator` container:

```
stage("Deploy to staging") {
     steps {
          sh "docker run -d --rm -p 8765:8080 --name calculator
leszko/calculator"
     }
}
```

After running this stage, the `calculator` container is running as a daemon, publishing its port as `8765` and being removed automatically when stopped.

## Adding an acceptance test to the pipeline

Acceptance testing usually requires running a dedicated black-box test suite that checks the behavior of the system. We will cover it in the *Writing acceptance tests* section. At the moment, for the sake of simplicity, let's perform acceptance testing simply by calling the web service endpoint with the `curl` tool and checking the result using the `test` command.

In the root directory of the project, let's create the `acceptance_test.sh` file:

```
#!/bin/bash
test $(curl localhost:8765/sum?a=1\&b=2) -eq 3
```

We call the `sum` endpoint with parameters `a=1` and `b=2` and expect to receive `3` in response.

Then, the `Acceptance test` stage can look as follows:

```
stage("Acceptance test") {
     steps {
          sleep 60
          sh "./acceptance_test.sh"
     }
}
```

## Adding a cleaning stage environment

As the last step of acceptance testing, we can add the staging environment cleanup. The best place to do this is in the `post` section, to make sure it executes even in case of failure:

```
post {
     always {
          sh "docker stop calculator"
     }
}
```

This statement makes sure that the `calculator` container is no longer running on the Docker host.

# Defining docker-compose.yml

The `docker-compose.yml` file is used to define the configuration for containers, their relations, and runtime properties.

In other words, when Dockerfile specifies how to create a single Docker image, then `docker-compose.yml` specifies how to set up the entire environment out of Docker images.

Let's start with an example and imagine that our calculator project uses the Redis server for caching. In this case, we need an environment with two containers, `calculator` and `redis`. In a new directory, let's create the `docker-compose.yml` file.

```
version: "3"
services:
     calculator:
          image: calculator:latest
          ports:
               - 8080
     redis:
          image: redis:latest
```

The command started two containers, `calculator` and `redis` in the background (`-d` option). We can check that the containers are running:

```
$ docker-compose ps
 Name                      Command               State           Ports
-------------------------------------------------------------------------------
project_calculator_1   java -jar app.jar        Up      0.0.0.0:8080->8080/tcp
project_redis_1        docker-entrypoint.sh redis ... Up 6379/tcp
```

The container names are prefixed with the project name `project`, which is taken from the name of the directory in which the `docker-compose.yml` file is placed. We could specify the project name manually using the `-p <project_name>` option. Since Docker Compose is run on top of Docker, we can also use the `docker` command to confirm that the containers are running:

```
$ docker ps
CONTAINER ID  IMAGE              COMMAND                  PORTS
360518e46bd3  calculator:latest "java -jar app.jar"
0.0.0.0:8080->8080/tcp
2268b9f1e14b  redis:latest       "docker-entrypoint..."  6379/tcp
```

# Scaling services

Docker Compose provides the functionality to automatically create multiple instances of the same container. We can either specify the `replicas: <number>` parameter inside `docker-compose.yml` or use the `docker-compose scale` command.

As an example, let's run the environment again and replicate the `calculator` container:

```
$ docker-compose up -d
$ docker-compose scale calculator=5
```

We can check which containers are running:

```
$ docker-compose ps
 Name                        Command              State Ports
-------------------------------------------------------------------------
calculator_calculator_1   java -jar app.jar   Up   0.0.0.0:32777->8080/tcp
calculator_calculator_2   java -jar app.jar   Up   0.0.0.0:32778->8080/tcp
calculator_calculator_3   java -jar app.jar   Up   0.0.0.0:32779->8080/tcp
calculator_calculator_4   java -jar app.jar   Up   0.0.0.0:32781->8080/tcp
calculator_calculator_5   java -jar app.jar   Up   0.0.0.0:32780->8080/tcp
calculator_redis_1        docker-entrypoint.sh redis ... Up 6379/tcp
```

## Changing the staging deployment stage

Let's change the `Deploy to staging` stage to use Docker Compose:

```
stage("Deploy to staging") {
    steps {
        sh "docker-compose up -d"
    }
}
```

We must change the clean up in exactly the same way:

```
post {
    always {
        sh "docker-compose down"
    }
}
```

## Creating a Dockerfile for acceptance test

We will start by creating a separate Dockerfile for acceptance testing. Let's create a new directory `acceptance` in the calculator project and a Dockerfile inside:

```
FROM ubuntu:trusty
RUN apt-get update && \
    apt-get install -yq curl
COPY test.sh .
CMD ["bash", "test.sh"]
```

It creates an image that runs the acceptance test.

## Creating an acceptance test script

The last missing part is the test script. In the same directory, let's create the `test.sh` file that represents the acceptance test:

```
#!/bin/bash
sleep 60
test $(curl calculator:8080/sum?a=1\&b=2) -eq 3
```

It's very similar to the previous acceptance test script, the only difference is that we can address the calculator service by the `calculator` hostname and that the port number is always `8080`. Also, in this case, we wait inside the script, not in the Jenkinsfile.

## Running the acceptance test

We can run the test locally using the Docker Compose command from the root project directory:

```
$ docker-compose -f docker-compose.yml -f acceptance/docker-compose-
acceptance.yml -p acceptance up -d --build
```

The command uses two Docker Compose configurations to run the `acceptance` project. One of the started containers should be called `acceptance_test_1` and be interested in its result. We can check its logs with the following command:

```
$ docker logs acceptance_test_1
 %   Total %   Received % Xferd Average Speed Time
100 1     100 1         0 0     1       0     0:00:01
```

The log shows that the `curl` command has been successfully called. If we want to check whether the test succeeded or failed, we can check the exit code of the container:

```
$ docker wait acceptance_test_1
0
```

The `0` exit code means that the test succeeded. Any code other than `0` would mean that the test failed. After the test is done, we should, as always, tear down the environment:

```
$ docker-compose -f docker-compose.yml -f acceptance/docker-compose-
acceptance.yml -p acceptance down
```

## Changing the acceptance test stage

As the last step, we can add the acceptance test execution to the pipeline. Let's replace the last three stages in Jenkinsfile with one new **Acceptance test** stage:

```
stage("Acceptance test") {
    steps {
        sh "docker-compose -f docker-compose.yml
                    -f acceptance/docker-compose-acceptance.yml build test"
        sh "docker-compose -f docker-compose.yml
                    -f acceptance/docker-compose-acceptance.yml
                    -p acceptance up -d"
        sh 'test $(docker wait acceptance_test_1) -eq 0'
    }
}
```

This time, we first build the `test` service. There is no need to build the `calculator` image; it's already done by the previous stages. In the end, we should clean up the environment:

```
post {
    always {
        sh "docker-compose -f docker-compose.yml
                    -f acceptance/docker-compose-acceptance.yml
                    -p acceptance down"
    }
}
```

After adding this to Jenkinsfile, we're done with the second method. We can test this by pushing all the changes to GitHub.