

LIVE WEBINAR - Thursday, May 21st: Deploy and manage databases in the cloud with SkySQL

[Sign Up Now▶](#)

DZone > Cloud Zone > Dockerizing an Angular App via Nginx [Snippet]

Dockerizing an Angular App via Nginx [Snippet]

by Rishab Aggarwal · Jan. 03, 18 · Cloud Zone · Code Snippet

During one of my recent projects, my team faced the challenge of Dockerizing an Angular app, as we wanted the UI to be deployed in a separate container on AWS.

To understand this article, you should have a basic understanding of Docker.

Dockerization Tasks

- We will start by creating an Angular application via the CLI provided by Angular: <https://cli.angular.io/>
- Once the project is created, we need to run the command: "ng build --prod". This will create a dist folder that contains all the deployables.
- Further steps will be performed by a Dockerfile:
 - Install Nginx inside the Docker image using the FROM command.
 - Remove the default HTML files inside Nginx installed on Docker.
 - Copy contents of the dist folder to an HTML folder inside Nginx.
 - Start Docker

Actual Implementation

Go to the directory where the new application should be created, create the app, and then start the server with the help of a few commands.

Assumption: The Angular CLI is already installed. If not, install that first.

1. ng new angular-docker-app
2. cd angular-docker-app : You should see the following files inside your folder.

Name	Date modified	Type	Size
git	12/30/2017 8:34 PM	File folder	
node_modules	12/30/2017 8:31 PM	File folder	
src	12/30/2017 8:31 PM	File folder	
angular-cli.json	12/30/2017 8:31 PM	JSON File	2 KB
adbcconfig	12/30/2017 8:31 PM	EDITORCONFIG File	1 KB
gitignore	12/30/2017 8:31 PM	Text Document	1 KB
tsconfig.json	12/30/2017 8:31 PM	JavaScript File	1 KB
package.json	12/30/2017 8:31 PM	JSON File	2 KB
package-lock.json	12/30/2017 8:34 PM	JSON File	326 KB
protractor.conf.js	12/30/2017 8:31 PM	JavaScript File	1 KB
README.md	12/30/2017 8:31 PM	MD File	2 KB
tsconfig.json	12/30/2017 8:31 PM	JSON File	1 KB
tslint.json	12/30/2017 8:31 PM	JSON File	3 KB

3. ng serve

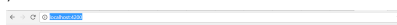
The application should be running on the localhost.

You can leave your ad blocker on and still support us

<http://localhost:4200/>

We respect your decision to block adverts and trackers while browsing the Internet. If you would like to support our content, though, you can choose to view a small number of premium adverts on our site by hitting the 'Support' button. These heavily vetted ads will not track you, and will fund our work.

Thank you for your support!



Welcome to app!

SUPPORT

No Thanks



Here are some links to help you start:

- [Tour of Heroes](#)
- [CLI Documentation](#)
- [Angular blog](#)

Begin Dockerization

Create a Dockerfile inside the project folder and add the following lines to it.

```
1 FROM nginx:1.13.3-alpine
2
3 ## Remove default nginx website
4 RUN rm -rf /usr/share/nginx/html/*
5
6 ## From 'builder' stage copy over the artifacts in dist folder to default nginx public folder
7 COPY /dist /usr/share/nginx/html
8
9 CMD ["nginx", "-g", "daemon off;"]
```

Create the Docker image and run it by using the following commands:

- `ng build --prod`
- `docker build -t angulardockertest .`
- `docker run -p 80:80 --name angulardockertest angulardockertest`

Now try to hit `http://localhost:80/`.

Stopping the Container

- Run the command `docker ps` to find the container id
- `docker stop <container Id>`
- `docker rm <container Id>`

You can download the source code here: <https://github.com/86rishab/angular-docker-app>.

Once Dockerization is complete, it can easily be pushed to the cloud.

Like This Article? Read More From DZone



DZone Article
Run 10,000 Docker Containers In Less Than 45 Minutes On 30 Rackspace Cloud Servers With 4GB Of Memory Each



DZone Article
Docker 3-Tier Java App Automation on Any Cloud



DZone Article
How I Switched My Blog From OVH to Google Container Engine



Free DZone Refcard
Hybrid Cloud vs. Multi-Cloud

TOPICS: ANGULAR, CLOUD, DOCKER, NGINX

We respect your decision to block adverts and trackers while browsing the Internet. If you would like to support our content, though, you can choose to view a small number of premium adverts on our site by hitting [this link](#). Your support is appreciated, and will fund our work.

Thank you for your support!
 DZone > Cloud Zone > Getting Started with Terraform

SUPPORT

No Thanks

Getting Started with Terraform

by Pradeep Bhadani · May 19, 20 · Cloud Zone · Tutorial

In this blog, you will learn to set up Terraform on your workstation in this step-by-step guide.

What is Terraform?

Terraform is an open-source tool allows us to build, change, and version our infrastructure in an easy and efficient way. It uses the declarative language HCL (HashiCorp Configuration Language) to define infrastructure as code.

Terraform Concepts

Let's quickly learn about some concepts in Terraform.

- **Providers:** Terraform Providers enable interaction with APIs and handles authentication of different IaaS services like Google Cloud Platform and Amazon Web Service, or SaaS services like CloudFlare. There are many Terraform-supported providers already available and a full list can be seen [here](#).
- **Resource:** Terraform Resource is a very important component. Each resource block describes the infrastructure object (e.g. VM instance, Storage buckets, DNS records, or Cloud NAT).
- **Modules:** Terraform modules are the collection of resources defined in a way that can be reused.
- **Data Sources:** Terraform Data Sources help to read infrastructure which is created using (or without using) Terraform.
- **State:** Terraform State stores information about the infrastructure created by Terraform code. It is used by Terraform to detect changes in the resources defined in the code.

The Terraform state is stored on the local machine by default in the name of terraform.tfstate but can be stored remotely on systems like Google Cloud Storage(GCS) and AWS S3.

Terraform Setup

Below steps are for Linux based system. For MAC, download the relevant package and the rest of the steps should be the same.

1. Download the latest terraform package from terraform.io/downloads.

```
export TF_VERSION=0.12.24
wget https://releases.hashicorp.com/terraform/${TF_VERSION}/terraform_${TF_VERSION}_linux_amd64.zip -O /tmp/terraform.zip
```

2. Unzip the terraform binary to a directory which is included in your system PATH .

```
sudo unzip /tmp/terraform.zip -d /usr/local/bin/
```

3. Reload your shell.

```
exec -l $SHELL
```

4. Verify installation.

terraform --help

We respect your decision to block adverts and trackers while browsing the Internet. If you would like to support our content, though, you can choose to view a small number of premium adverts on our site by hitting the 'Support' button. These heavily vetted ads will not track you, and will fund our work.

Terminal Recording!

```
contkio~$ asciinema export TF_VERSION=0.12.16
```

SUPPORT

No Thanks

```

cntekio~ asciinema$ wget https://releases.hashicorp.com/terraform/0.12.16/terraform_0.12.16_linux_amd64.zip -O /tmp/terraform.zip
--2019-12-01 17:46:23-- https://releases.hashicorp.com/terraform/0.12.16/terraform_0.12.16_linux_amd64.zip
Resolving releases.hashicorp.com (releases.hashicorp.com)... 151.101.129.183, 151.101.193.183, 151.101.1.183, ...
Connecting to releases.hashicorp.com (releases.hashicorp.com)[151.101.129.183]:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 16356886 (16M) [application/zip]
Saving to: '/tmp/terraform.zip'

/tmp/terraform.zip
100%[=====>] 15.60M 4.62MB/s in 3.4s

2019-12-01 17:46:26 (4.53 MB/s) - '/tmp/terraform.zip' saved [16356886/16356886]

cntekio~ asciinema$ sudo unzip /tmp/terraform.zip -d /usr/local/bin/
Archive: /tmp/terraform.zip
  inflating: /usr/local/bin/terraform
cntekio~ asciinema$


```

I hope this blog helps you to quickly get started with Terraform.

If you have feedback or questions, please reach out to me on LinkedIn or Twitter.

Originally published at pbhadani.com

Topics: DEVOPS, HASHICORP, INFRASTRUCTURE AS CODE, INSTALLATION, TERRAFORM, TERRAFORM TUTORIAL

Published at DZone with permission of Pradeep Bhadani . [See the original article here.](#) 

Opinions expressed by DZone contributors are their own.

DZone > Cloud Zone > Automate Spring Boot App Deployment With GitLab CI and Docker

Automate Spring Boot App Deployment With GitLab CI and Docker

by Jasmin Tankić · May 19, 20 · Cloud Zone · Tutorial

In this guide, we will walk through the process of the automated deployment of a Spring Boot application using GitLab CI.

Docker and Spring Boot is a very popular combination, and we will take advantage of GitLab CI and automatically build, push and run a Docker image on application server.

GitLab CI

The Gitlab CI/CD service is the part of GitLab that builds, tests, and places the latest changes in the desired environment whenever the developer pushes code to the GitLab repository.

Some of the main reasons why GitLab CI is a good choice:

1. It is easy to learn, use and scalable
2. Maintenance is easy
3. Integration of new jobs is easy
4. The CI is fully part of the repository
5. Good Docker integration
6. Container registry - basically your own private Docker hub
7. It's economically a good solution. Each month you have 2000 minutes of build time for free, which is more than enough for certain projects

Why GitLab CI Over Jenkins

This is for sure a wide and debatable topic, but in this article, we won't dive deeply into that. Both GitLab CI and Jenkins have pros and cons and they are for sure very powerful tools.

Let's try to point out use cases where GitLab might be wiser choice.

You can leave your ad blocker on and still support us
As it is previously mentioned, CI is fully part of GitLab repository, which means it's not needed to install it and maintain

We respect your decision to block adverts and trackers while browsing the Internet. If you would like to support our content, though, you can choose to view a small number of premium adverts on our site by hitting the 'Support' button. These heavily vetted ads will not track you, and will fund our work.

Jenkins can be overboard for small projects as you have to set up and configure everything by yourself. You also usually need a dedicated Jenkins server. and that is also extra maintenance. cost. and another thing to worry about.

What You'll Need

In order to successfully follow this guide, there are few prerequisites. If any help is needed related to these prerequisites I've included a link to the appropriate guide, so feel free to visit it if necessary.

1. You have Spring Boot project pushed on GitLab
2. You have Docker installed on application servers (guide)
3. You have container registry for Docker images (in this guide Docker hub will be used)
4. You have generated SSH RSA key on your servers (guide)

What You'll Build

You will basically create a **Dockerfile** and **.gitlab-ci.yml**, which will be used to automatically:

1. Build application Jar file for each successful deployment, which will be easily downloadable from GitLib
2. Build the Docker image
3. Push the image to the Docker repository
4. Run image on an application server

Basic Project Info

Spring Boot application for this guide is generated via Spring Initializr. Basically it is a Maven project built on Java 8 or 11. We will cover later how Java 8 and 11 affects the Docker image.

Dockerfile

Let's start with the Dockerfile.

```
Dockerfile

FROM maven:3.6.3-jdk-11-slim AS MAVEN_BUILD
#FROM maven:3.5.2-jdk-8-alpine AS MAVEN_BUILD FOR JAVA 8

ARG SPRING_ACTIVE_PROFILE

MAINTAINER Jasmin
COPY pom.xml /build/
COPY src /build/src/
WORKDIR /build/
RUN mvn clean install -Dspring.profiles.active=$SPRING_ACTIVE_PROFILE && mvn package -B -e -
Dspring.profiles.active=$SPRING_ACTIVE_PROFILE
FROM openjdk:11-slim
#FROM openjdk:8-alpine FOR JAVA 8
WORKDIR /app

COPY --from=MAVEN_BUILD /build/target/appdemo-*.jar /app/appdemo.jar
ENTRYPOINT ["java", "-jar", "appdemo.jar"]
```

There are few things nice to know related to this Dockerfile.

Java Version

Let's see what from Docker's point of view is different between Java 8 and 11. Long story short: it's the Docker image size and deployment time.

Docker images built on Java 8 will be noticeably smaller than ones on Java 11. That also means build and deploy times will be faster for Java 8 projects.

- Java 8 - Build time: ~ **4 min** with image size of ~**180 MB**
- Java 11 - Build time: ~ **14 min** with image size of ~**480 MB**

You can leave your ad blocker on and still support us

Note: These stats are related to plain Spring Boot projects that are used for the purpose of this guide; on a real application t

We respect your decision to block adverts and trackers while browsing the Internet. If you would like to support our site in a different way, you can choose to view a small number of premium adverts on our site by hitting the 'Support' button. These heavily vetted ads will not track you, and will fund our work.

Thank you for your support!

Docker Images

SUPPORT

No Thanks

As it is already seen in previous example, we have huge difference for app image size and build time just because of the Java version. The actual reason behind that is Docker images used in Dockerfile.

If we take another look at the Dockerfile, the real reason behind the large Java 11 image size is because there is no verified/tested Alpine version of open-jdk:11 image. Instead, we had to use a open-jdk:11-slim image which produces larger images.

If you are not familiar with the OpenJDK image versions, I suggest taking look at the official OpenJDK Docker documentation. There you can find an explanation for each OpenJDK version of image.

Additional Notes

ARG SPRING_ACTIVE_PROFILE is used so it's possible to build and package an application with correct environment-related properties.

Unfortunately, at the time when this guide is written, there is no clean way to use that variable in **ENTRYPOINT**, and in order to run an application on different environments, it's mandatory. Basically the **ENTRYPOINT** would look like this:

```
Java

1 ENTRYPOINT ["java", "-Dspring.profiles.active=development", "-jar", "appdemo.jar"]
```

And to make it dynamic, what you naturally would expect is to simply convert it to:

```
Java

1 ENTRYPOINT ["java", "-Dspring.profiles.active=$SPRING_ACTIVE_PROFILE", "-jar", "appdemo.jar"]
2
```

As already-mentioned, unfortunately, this is not possible, but fortunately this problem can be easily solved when running an image with the Docker. This will be covered in **.gitlab-ci.yml**.

gitlab-ci.yml

Before writing this file there are few things to prepare first. Basically what we want to achieve is that whenever code is pushed, an automatic deployment is started on the corresponding environment.

Create .env Files and Branches

We need first to create branches and .env files that contain environment-related variables. Each branch will actually represent the environment where our application is running.

We will deploy our application on three different environments: development, QA, and production. That means we need to create three "main" branches. We already have the master branch which represents the production environment, so simply create 2 more branches: development and QA.

Our dev, QA, and prod applications will run on different servers and they will have different Docker container tags, ports and SSH Keys. That means our **gitlab-ci.yml** file will need to be dynamic, let's solve that problem by creating .env file for each environment that we have.

```
.develop.env
.qa.env
.master.env
```

IMPORTANT: There is one simple rule when naming these files: they need to be named by the branch on GitLab, so a file name should be like this: **.\$BRANCH_NAME.env**

For example this is .develop.env file.

```
Java

1 export SPRING_ACTIVE_PROFILE='development'
2 export DOCKER_REPO='username/demo_app_dev'
```

3 export APP_NAME='demo_app_dev'

4 export PORT='8080'

5 export SERVER_IP='000.11.222.33'

6 export SERVER_SSH_KEY='key'

Important notes related to .env file:

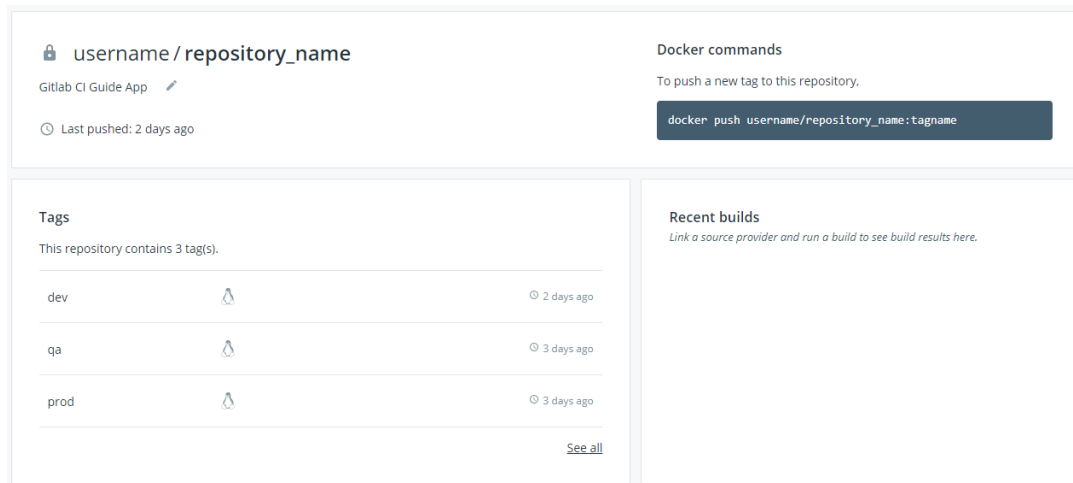
SUPPORT

No Thanks

SPRING_ACTIVE_PROFILE: self-explanatory, which Spring application properties we want to use.

DOCKER_REPO: This is a repository for the Docker image. Since we are using one repository we will use tags to distinguish between images used on different environments, so that means we will have 3 different tags: dev, qa and production.

And our Docker hub looks like this.



As you can see there is a repository with three different tags, and each tag (application version) is being updated whenever the code is pushed on a GitLab branch.

- **APP_NAME:** This property is very important because it allows us to name our running container. Based on it we will be able to stop and re-run latest Docker image on application server.
If you do not set this property, Docker will randomly give name to your container. That can be an issue because you won't be able to automatically stop running container on a clean way.
- **PORT:** This is the port where we want our Docker container to be run at.
- **SERVER_IP:** IP of the server where application is going to live. Usually each environment will be on different server.
- **SERVER_SSH_KEY:** This is SSH key that we already generated on each of our servers. It is a bit different than previous keys, because it does not contain hard-coded value, `$DEV_SSH_PRIVATE_KEY` is actually a variable that comes from GitLab repository. Below there will be more details about that.

Create GitLab Variables

The last step that needs to be done is creating GitLab variables. Here it's recommended to keep variables that you do not want everyone to see. We will keep here some credentials and ssh keys.

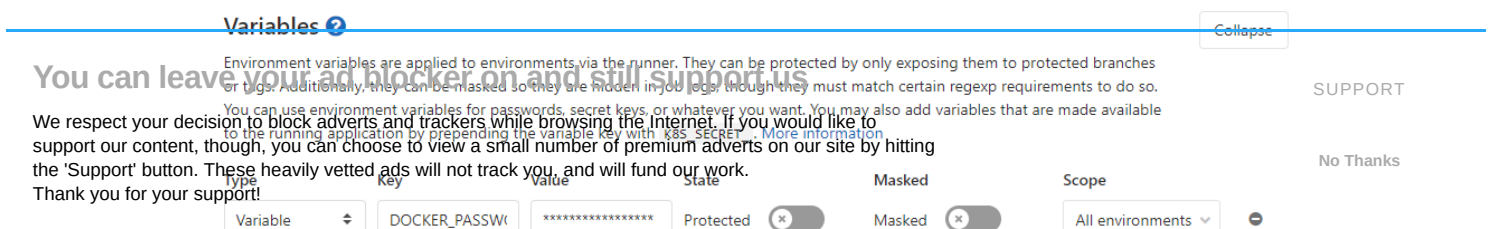
Open your GitLab repository and go to: **Settings -> CI/CD**. In the **Variables** section add new variables:

- **DOCKER_USER:** username for accessing Docker hub or any other container registry
- **DOCKER_PASSWORD:** password for accessing same container registry
- **\$ENV_SSH_PRIVATE_KEY:** SSH private key that you previously generated on your servers. Your key should look like this.

Important notes for SSH KEY:

- You need to copy full key value including:
-----BEGIN RSA PRIVATE KEY----- and -----END RSA PRIVATE KEY-----

At the end, your GitLab variables should look like this.



Variable	DOCKER_USER	*****	Protected	Masked	All environments
Variable	DEV_SSH_PRIVAT	*****	Protected	Masked	All environments
Variable	QA_SSH_PRIVATE	*****	Protected	Masked	All environments
Variable	PROD_SSH_PRIVAT	*****	Protected	Masked	All environments
Variable	Input variable key	Input variable	Protected	Masked	All environments

Create the gitlab-ci.yml File

And finally, let's create the file that will put together all this.

YAML

```

1 services:
2   - docker:19.03.7-dind
3
4 stages:
5   - build jar
6   - build and push docker image
7   - deploy
8
9 build:
10  image: maven:3.6.3-jdk-11-slim
11  stage: build jar
12  before_script:
13    - source ${CI_COMMIT_REF_NAME}.env
14  script:
15    - mvn clean install -Dspring.profiles.active=${SPRING_ACTIVE_PROFILE} && mvn package -B -e -
16  artifacts:
17    paths:
18      - target/*.jar
19
20 docker build:
21  image: docker:stable
22  stage: build and push docker image
23  before_script:
24    - source ${CI_COMMIT_REF_NAME}.env
25  script:
26    - docker build --build-arg SPRING_ACTIVE_PROFILE=${SPRING_ACTIVE_PROFILE} -t $DOCKER_REPO .
27    - docker login -u $DOCKER_USER -p $DOCKER_PASSWORD docker.io
28    - docker push $DOCKER_REPO
29
30 deploy:
31  image: ubuntu:latest
32  stage: deploy
33  before_script:
34    - 'which ssh-agent || ( apt-get update -y && apt-get install openssh-client -y )'
35    - eval $(ssh-agent -s)
36    - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -
37    - mkdir -p ~/.ssh
38    - chmod 700 ~/.ssh
39    - echo -e "Host *\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config
40    - source ${CI_COMMIT_REF_NAME}.env
41  script:
42    - ssh root@$SERVER "docker login -u $DOCKER_USER -p $DOCKER_PASSWORD docker.io; docker stop $APP_NAME; docker system
43  prune -a -f; docker pull $DOCKER_REPO; docker container run -d --name $APP_NAME -p $PORT:8080 -e
44  SPRING_PROFILES_ACTIVE=${SPRING_ACTIVE_PROFILE} $DOCKER_REPO; docker logout"

```

Let's explain what is happening here:

You can leave your ad blocker on and still support us

services:

- **docker:19.03.7-dind** to block adverts and trackers while browsing the Internet. If you would like to support our content, though, you can choose to view a small number of premium adverts on our site by hitting the 'Support' button. These heavily vetted ads will not track you, and will fund our work. This is actually a service that allows us to use Docker in Docker. Running Docker in Docker is generally not a good idea, but for this use case it's totally fine, since we will just build the image and push it to the repository. You can get more information about Docker in Docker here.

SUPPORT

No Thanks


```

1 - source ${CI_COMMIT_REF_NAME}.env
2 script:
3 - ssh root@$SERVER "docker stop $APP_NAME; docker system prune -a -f; docker pull $DOCKER_REPO; docker container run -d -
  -name $APP_NAME -p $PORT:8080 -e SPRING_PROFILES_ACTIVE=$SPRING_ACTIVE_PROFILE $DOCKER_REPO"

```

In this step, we are using the Ubuntu Docker image so we can ssh to our application server and run a few (Docker) commands. Part of the code in `before_script` is mostly taken from the official documentation, but, of course, we had to adjust to it a bit for our needs. In order to avoid making private key verified, this line of the code is added:

Shell

```
1 - echo -e "Host *\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config
```

But if you do not like it, you can follow this guide and verify your private key.

As you can see in the last stage of the script, we are executing a few Docker commands.

1. Stop the running Docker container by executing command: `docker stop $APP_NAME`. (This is the reason why we defined `APP_NAME` in our `.env` file)
2. Delete all the Docker images that are not running by executing `docker system prune -a -f`. This is actually not mandatory, but I wanted to remove all unused images on my server.
3. Pull the latest version of the Docker image (that was built and pushed in the previous stage).
4. Finally, run the Docker image with the following command:
`docker container run -d --name $APP_NAME -p $PORT:8080 -e SPRING_PROFILES_ACTIVE=$SPRING_ACTIVE_PROFILE`

I hope this guide will help you to better understand and implement GitLab CI in your projects. If you have any questions or suggestions feel free to contact me or leave a comment.

Like This Article? Read More From DZone

 related
article
thumbnail

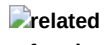
DZone Article
Build, Package, and Run Spring Boot Apps With Docker

 related
article
thumbnail

DZone Article
OpenStack Cloud Orchestration Pt. I: From Manual to Automated Deployment

 related
article
thumbnail

DZone Article
Optimizing a Spring Boot Application for Docker

 related
refcard
thumbnail

Free DZone Refcard
Hybrid Cloud vs. Multi-Cloud

Topics: AUTOMATION, CLOUD, DEPLOYMENT, DOCKER, DOCKER CONTAINERS, GITLAB CI, SPRING BOOT

Opinions expressed by DZone contributors are their own.

You can leave your ad blocker on and still support us

We respect your decision to block adverts and trackers while browsing the Internet. If you would like to support our content, though, you can choose to view a small number of premium adverts on our site by hitting the 'Support' button. These heavily vetted ads will not track you, and will fund our work. Thank you for your support!

SUPPORT

No Thanks