Automation of Microservices Application Deployment Made Easy By Rundeck and Kubernetes

1st Harika Rajavaram
International Institute of Information Technology, Bangalore
Bangalore, India
rajavaram.harika@iiitb.org

2nd Vineet Rajula
International Institute of Information Technology, Bangalore
Bangalore, India
vineet.reddy@iiitb.org

3rd B. Thangaraju
International Institute of Information Technology, Bangalore
Bangalore, India
b.thangaraju@iiitb.ac.in

***Abstract*—Today, more than 2.6 million apps are managed on Google Play Store [1]. To be able to compete with this type of huge volume of applications, a powerful, easy to scale and a flexible application is required to sustain in the market and overpower the competitors. Microservices architecture and continuous deployment practices tackle the above challenges and help to improve the productivity of the organizations. This paper explores the different procedures of deploying microservices in CI/CD pipeline using Rundeck, Docker containers, Docker-Compose and Kubernetes. Moreover, it weighs the pros and cons of these orchestration methods, explaining suitable use cases for each one of them. It concludes by discussing that Kubernetes is the most efficient way to deploy microservices to achieve high availability and scalability.**

***Index Terms*—microservices, automation, rundeck, kubernetes, containers, deployment**

I. Introduction

Microservices are small independent, lightweight modules of an application that are integrated together to work as a whole. Many E-commerce and Media-Streaming giants like Netflix and Amazon have moved to microservices architecture in the past decade [2]. Eighty percent of the application development on cloud platforms will be using microservices by 2021, as predicted by The International Data Corporation [3]. Along with the microservices architecture, the organizations need to have stable releases of the application and the time from development to deployment has to be minimized. This is possible by using the continuous deployment practices. Automating the process of deployment into production is called Continuous Deployment (CD).

This work focuses on the various approaches to continuous deployment of any generic web application as microservices and it discusses the ease and efficiency of each of the methods. The architecture diagram of the procedure explained in this paper for a generic web application as microservices is summarized in Fig. 1.

The CI/ CD pipeline (Continuous Integration/ Continuous Deployment) for the same application is elaborated in Fig. 2.

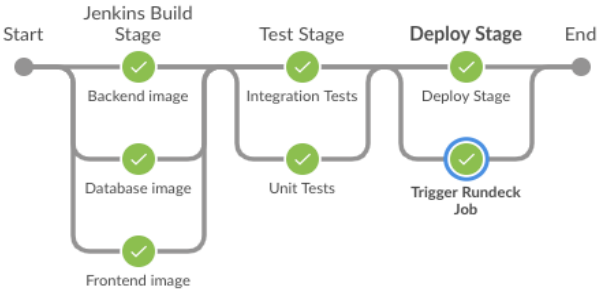

Fig. 2. CI/ CD Jenkins Pipeline

The different tools used in different phases of the pipeline are mentioned in Table 1 and similar approach was reported by V. Singh and S. K. Peddoju [4].

TABLE I
Components and Tools in Pipeline

Pipeline Component	Tool
Version Control System	Git
Source Code Manager	GitHub
Build Server	Jenkins
Container Repository	Docker Hub
Deployment Server	Rundeck
Container Orchestration	Docker-Compose, Kubernetes

Jenkins, a widely used CI tool, uses the Git Plugin to integrate with GitHub. With every commit in the GitHub repository, Jenkins triggers the build stage which creates an artifact. Here, the artifacts are docker images which are created by building Dockerfiles. There would be multiple

978-1-7281-2472-8/19/$31.00 © 2019 IEEE

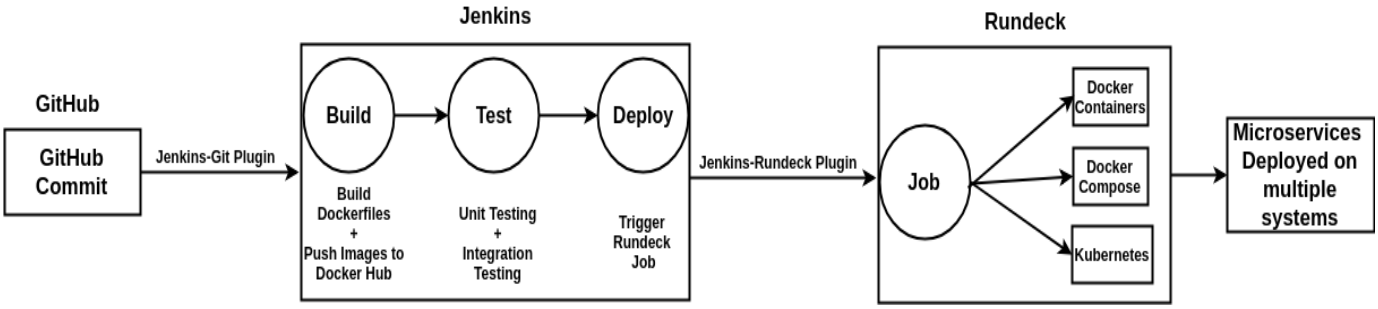

Fig. 1. Architecture Diagram Of Procedure

docker images, one for each of the microservices such as front-end, back-end, and database. Docker Registry (a.k.a called Docker Hub) is used to push the docker images so that they can be pulled in multiple servers to run the application. All these steps of dockerizing the application can be executed as a shell script or the Docker Plugin in Jenkins can be used.

Next, in the test stage, unit testing and integration testing are done on the artifact. The tools used for testing are application specific like selenium, JUnit, etc. Then comes the deploy stage where the application is deployed on multiple servers with different configurations. This pipeline uses a CD tool named Rundeck. The Rundeck Job deploys the application onto multiple servers which is triggered from the deploy stage in Jenkins using the Rundeck Plugin. Our main goal is to decide on the most optimal way of continuous deployment using Rundeck. This is explored in detail in the following sections.

II. Deploying Using Docker Containers

Since we have pushed the images to Docker Hub, we need to pull the images and run the containers. We also need to link all the microservices together such that they can communicate amongst themselves to run the application. Previously, *–link* argument in the docker run command was used to create a dependency between the containers. In the version 1.9 of docker, *–link* has been deprecated. Currently, User-Defined Bridge networks are created to link the containers together [5]. But, it becomes difficult to manage these container instances when the number of servers grows. Every time it is required to create a network and link the containers to the network, which is a tedious process. The Intricacies of Docker Networking were explained in detail by Harika and Shreyak [6]. To manage and organize the containers, docker provides a mechanism to connect containers with each other using a single *yaml* file and by running it with docker-compose.

III. Deploying Using Docker Compose

Docker-compose is a tool that enables configuration management of Docker containers [7]. Docker-compose helps in deploying and linking multiple containers as microservices. *$ docker-compose up* command deploys the microservices application. Production environment is configured as infrastructure as code (IaC) and does not require any further manual configurations. It supports portability as a single file can set up the application and get it running without giving multiple complex docker commands. The *docker-compose.yaml* file should be pushed to the servers where the application is deployed. The Fig. 3 shows the code snippet of how database (db) service, running the database server, is linked with web service running the back-end application.

```
version: '1'
services:
  db:
    image: <<docker-hub-username/db-image-name>>
  web:
    image: <<docker-hub-username/app-image-name>>
    command: <<command to run the server>>
    volumes:
      - .:/app
    ports:
      - "host port : app server port inside container"
    depends_on:
      - db
```

Fig. 3. Code snippet for docker-compose.yaml

Rundeck pulls docker-compose.yaml from the GitHub repository and executes the command *$ docker-compose up*. The job to get docker-compose file and run it is shown in the Fig. 4. After executing the job, each microservice would be running on a different container and the application would be up and running.

Docker compose simplifies the process of connecting containers but comes with drawbacks as well. The containers with all the microservices, by default, are deployed on the same system so that they can communicate amongst themselves. This is suitable for applications which do not get a large number of requests. As the traffic grows on the application, it becomes necessary to run the application on multiple systems. Docker compose does not provide a mechanism to deploy containers on multiple hosts. In addition to this, to update the application using docker-compose, we have to stop the containers using *$ docker-compose down* and then start it again with the updated docker images. This results in a significant amount of downtime of the application.

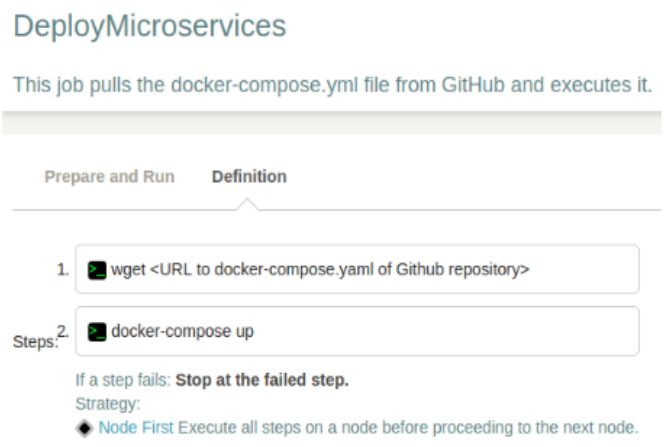

Fig. 4. Rundeck job to deploy an app using docker-compose

IV. Deploying Using Kubernetes

Kubernetes is a container orchestration tool which is open source. Integration of Kubernetes with Rundeck is done using a Rundeck plugin which enables Kubernetes to deploy the microservices onto the Kubernetes cluster. Kubernetes deployment and service is created for each microservice as shown in Fig. 5.

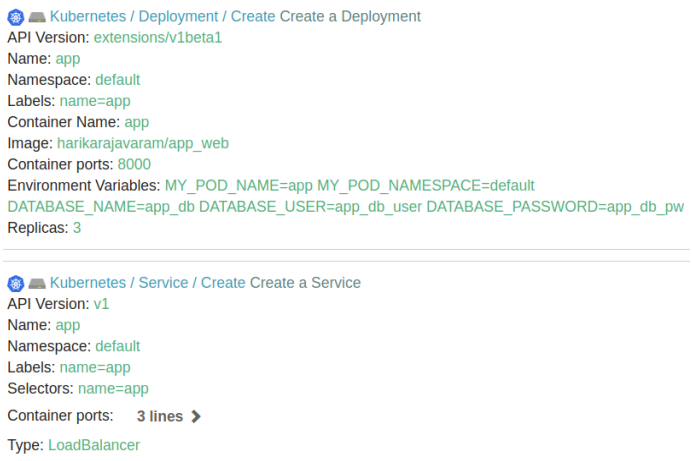

Fig. 5. Rundeck job to deploy an app using Kubernetes

The command line arguments for Kubernetes command can be configured in Rundeck or they can be passed as Jenkins variables from Jenkins pipeline job once the new artifact is formed.

Kubernetes supports microservices architecture seamlessly. Moreover, Kubernetes can be easily integrated into the DevOps pipeline. Kubernetes comes with self-healing, that is, whenever an application instance is down, it is immediately discovered and it creates a new instance in its place. This makes sure that all the microservices are up and running, resulting in high availability. It also makes it very easy to update the application image, without bringing the app down, unlike docker-compose. New pods with the new image are created while still redirecting the traffic to the previous version images. Once the new pods are up, slowly the traffic is redirected to the new version pods and the old pods are killed at the end. Also, unlike in docker-compose, Kubernetes supports scaling of the number of pods (replicas of the application) on the fly. Also, Kubernetes allows microservices to be deployed on multiple hosts by creating a virtual network amongst the pods deployed on all the nodes and every pod can reach every other pod through this network. This eases the deployment of microservices using Kubernetes [8].

It is found that Kubernetes, with all these features and with a full-fledged integration plugin with Rundeck, appears to be the best practical approach to microservices orchestration as per the implementation work done in this paper.

V. Conclusion

Companies with huge user base roll-out frequent updates and require integration of new features. Microservices architecture is very crucial in such cases to make the applications scalable and flexible [4]. We implemented a CI/CD pipeline and explained the different ways in which microservices can be continuously deployed. It analysed the pros and cons of each of the procedures and when each of the methods could be used. It has been established that Kubernetes for container orchestration is the most suitable method to deploy the microservices when the application demands high availability and scalability.

References

[1] AppBrain. Number of android applications, 2017.
[2] P. D. Francesco. Architecting microservices. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 224–229, April 2017.
[3] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert. Microservices. *IEEE Software*, 35(3):96–100, May 2018.
[4] V. Singh and S. K. Peddoju. Container-based microservice architecture for cloud applications. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 847–852, May 2017.
[5] Docker. Legacy container links from docker official documentation at https://docs.docker.com/network/links.
[6] R. Harika, S. Upadhyay, and B. Thangaraju. Intricacies of docker networking. volume 06, pages 39–44, June 2018.
[7] K. Klinbua and W. Vatanawood. Translating tosca into docker-compose yaml file using antlr. In *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 145–148, Nov 2017.
[8] Kubernetes Official Documentation at https://kubernetes.io/docs.