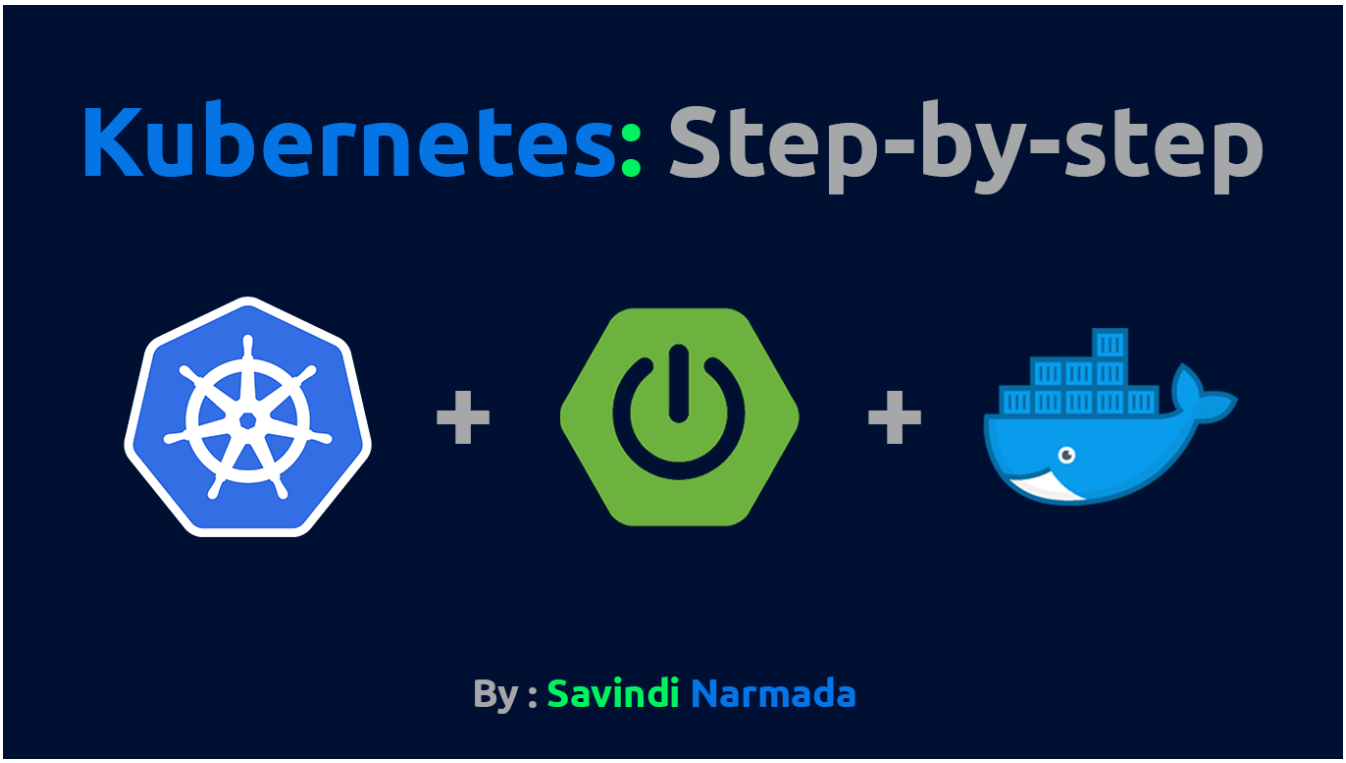
KUBERNETES: Step-by-Step Guide with Spring Boot, Docker & GKE

Narmada
May 4, 2019 · 13 min read



Hola again! Welcome back.

Today we are going to talk about another well-known technology in the world; Kubernetes. I am sure you have heard this many times. But do you really know what it is? If the answer is a big “no", go ahead and read this post. Today’s post will be a bit long one, but you will see yourself transform from “Zero” to “Hero” throughout it. 💪

Let me tell today’s outline first:

1. Containers
2. Container Orchestrator and its role

3. What is Kubernetes
4. Why Kubernetes
5. Architecture of Kubernetes
6. Essential Components of Kubernetes
7. Hands-on

Containers

A container is, in simple terms, means a virtual machine without its own Operating system. We can create these containers in far-away data centers and get our work done. Importance in it is, it ensures the maximum resource usage for our work, rather than wasting resources for the Operating system, etc.

In the usual production environment, there are hundreds and thousands of containers dedicated to various microservices. Therefore managing them is not a simple task. They need updates, version control, health checks, scaling and many more functionalities to be performed on them. This is where the need for a ***"Container Orchestrator"*** arises. If you want to learn more about containers, you can also check these best courses to learn Docker and Kubernetes for developers.

Container Orchestrator and its role

Think of a football match. If the coach does not tell players what are their places, they will stand wherever they want. If that is the case, will that team win the match? I guess not.

> So what coach does is, he tell the players where they need to stand and what they need to do.

Similarly, Kubernetes, or any other container orchestrator, tells the containers to where they should stay and how they should behave. In a nutshell, the container orchestrator is like the coach and the players are like the containers or the micro-services.

What is Kubernetes

Kubernetes was born in Google as one of their In-house technology for managing containers. It is written in Go-language.

In 2015, Kubernetes was first released to the public. Later on, Google handed it over to CNCF (Linux Foundation) to manage.

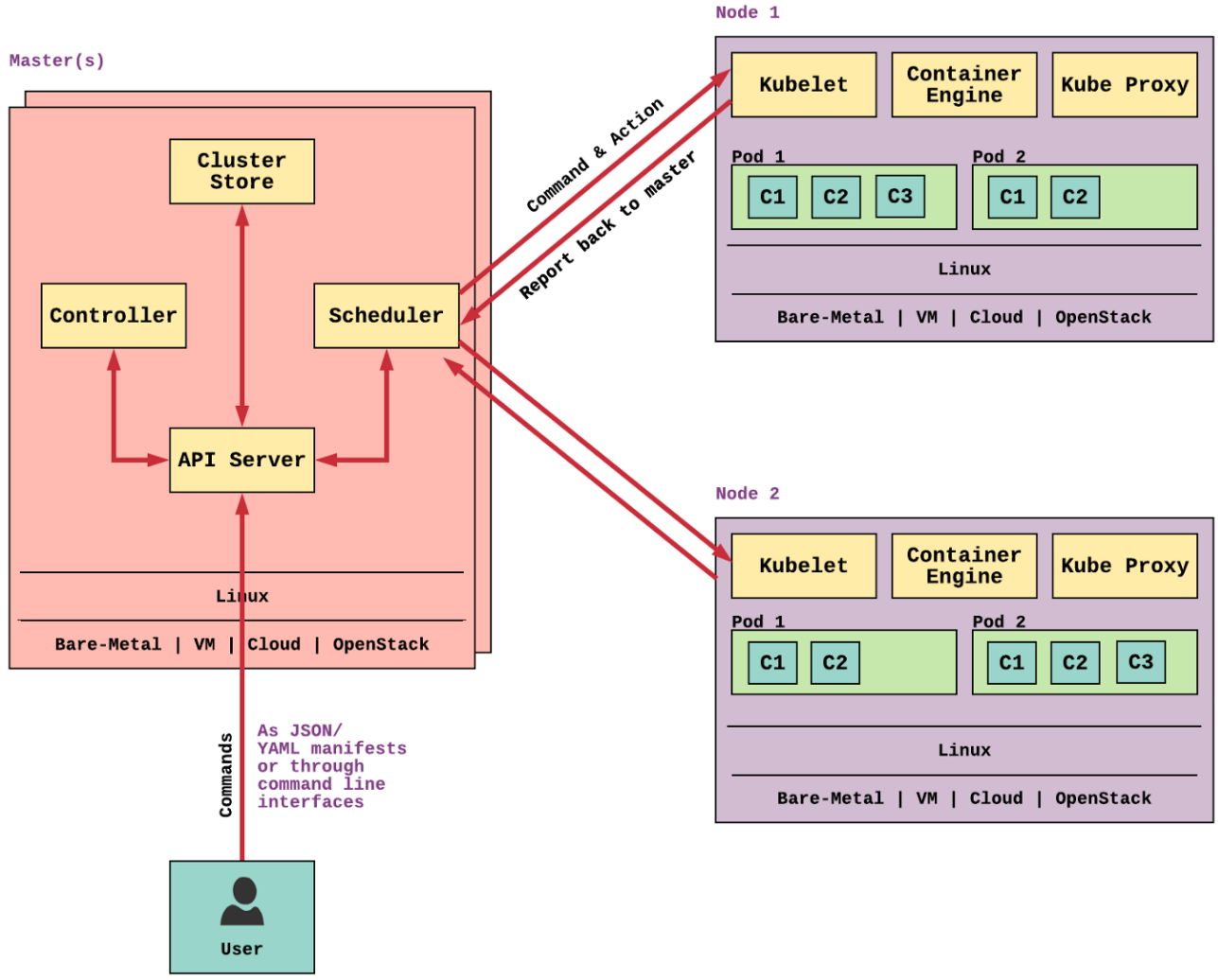
So currently Kubernetes is an open-source project under Apache 2.0 license. Sometimes in the industry, Kubernetes is also known as “K8s". Widely used term to phrase Kubernetes is ***“Container Orchestrator"***.

If you want to learn Kubernetes and Docker then, the **Docker and Kubernetes: The Complete Guide** is the best one to start with.

Why Kubernetes over other orchestrators

- **Very platform-agnostic**: i.e. Kubernetes can be used with bare-metal, virtual machines, cloud, Open stack, etc.
- Not only for container-based orchestration. You can also use it with normal clustering.
- Not tied with any other specific company or platform like Docker. Many companies support Kubernetes based clustering, including Google, Amazon Web Services, etc.
- Let target deployment

Architecture of Kubernetes



Kubernetes Architecture in one image

As you can see, Kubernetes also follows a Master-Slaves architecture (also known as Master-minions) let's talk about each of these components one by one in a very high-level manner.

Master

Master is the controlling element of the cluster. Some people call it the “Brain" of the cluster. It is the only endpoint that is open to the users of the cluster. For the purpose of fault-tolerance, one cluster may have multiple masters.

Master has 4 parts:

1. API server:

This is the front end that communicates with the user. It is a REST-based API that is designed to consume JSON inputs. As a default, it runs in port 443.

2. Scheduler:

Scheduler watches API server for new Pod requests. It communicates with Nodes to create new pods and to assign work to nodes while allocating resources or imposing constraints.

3. Cluster store:

Cluster store is a persistent storage holding cluster states and configuration details. It uses ETCD (open-source distributed key-value store) to store these data.

4. Controller:

Includes Node controller, Endpoint Controller, Namespace Controller, etc.

Nodes (Slaves/Minions)

Nodes are the workers. They are the ones that do all the "Work" assigned to the cluster. Inside a Node, there are 3main components, apart from the ***"Pods"*** (I will talk about Pods later on). Those 3 parts are;

1. Kubelet

Kublets do a lot of work inside a Node. They register the nodes with the cluster, watch for work assignments from the scheduler, instantiate new Pods, report back to the master, etc.

2. Container Engine

Container Engine is the responsible person for managing containers. It does all the image pulling, container stopping, starting, etc. Most widely used container engine is Docker. However, you can also use Rocket for this.

3. Kube Proxy

Kube Proxy is responsible for assigning IP addresses per pod. Each time a pod creates, a new IP address will be allocated for that pod. Kube Proxy also does the Loadbalancing work.

Apart from those mentioned components, Nodes have their own default pods like logging, health checking, DNS, etc. Each node expose 3 read-only endpoints through

(usually) localhost:10255. Those endpoints are,

- /specs
- /healthz
- /pods

Essential Components of Kubernetes

There are few main components of a Kubernetes Cluster architecture that anyone should know before going into working with Kubernetes. First one is a Pod:

Pods

> A pod is the atomic unit of deployment or scheduling in Kubernetes.

The Pod is a Ring-faced environment with its own Network stack and Kernal namespaces. It has containers inside. No pod can exist without a container. But there can be single-container pods or multi-container pods depending on the application we deploy.

For example, if you have a tightly coupled application with an API and a log, you can use one container for API and another for the log. But you can deploy both of them in the same Pod. However, industry best practice is to go with single-container architecture.

Another small thing to note about Pod is that they are ***"Mortal".*** Confused? Let me explain. A pod's life-cycle has 3 stages:

Pending → Running → Succeeded/Failed

This is similar to Born → Living → Dead. There will be no **Resurrection**; no re-birth. If a Pod died without completing his task, a new Pod will be created to replace the dead Pod. The most important thing is, this new pod's IP and all other factors will be different from the dead pod.

Deployment Controller

To manage the Pods, there are numerous controllers presented in Kubernetes. Such controller used for the purpose of deployment and declarative updates is known as Deployment Controller.

In the Deployment object (mostly used format is a YAML file. But in this tutorial, I use command line) we can describe our “Desired state” like what is the image needed to be deployed, what are the ports to expose, how many replicas to have, what are the labels needed to be added, etc. What Deployment Controller does is to check this desired state periodically and make changes in the cluster to make sure the desired state is achieved.

Service

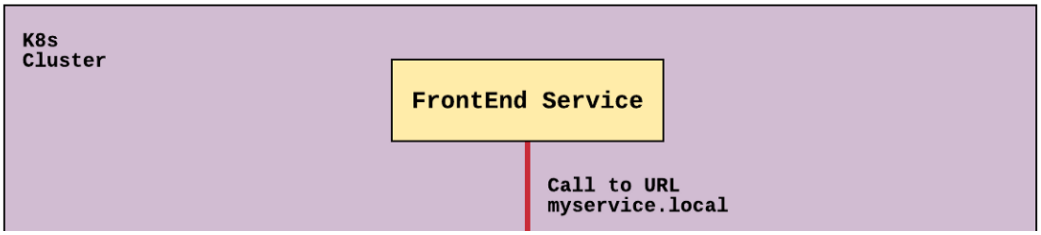
Another one component I am going to use in this tutorial is “Service”. Before telling what is a Service, I will describe why we need a service.

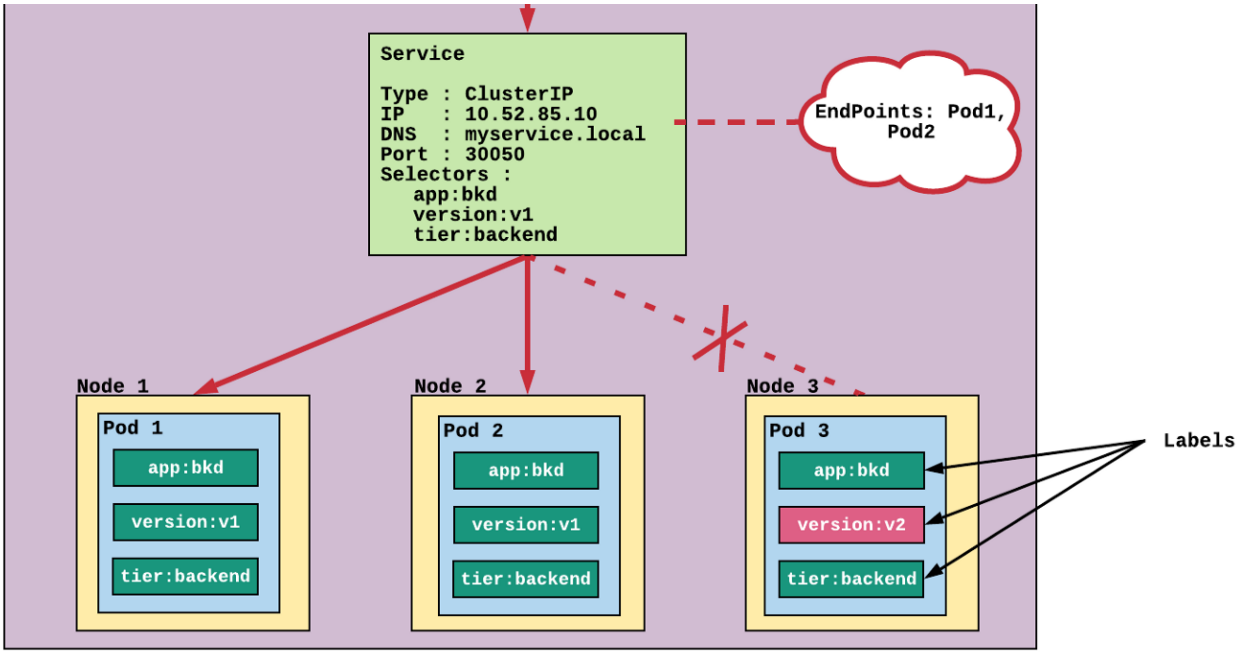
As I mentioned earlier, Pods are mortal. When a pod dies, a new one is born to take its place. It doesn’t have the same IP address as the dead one.

So think of a scenario where we have a system with both front end service and backend service. From the front end to call the backend, we need an IP or URL. Let's assume we used the pod IP of the backend service inside the frontend code. We face three issues:

1. We need to first deploy our backend and take its IP. Then we need to include it in the front end code before making the docker image. This order **must** be followed.
2. What if we want to scale our backend? We need to update the frontend again with the new pod IPs.
3. If the backend pod died, a new pod will be created. Then we need to change the front end code with the new pod IP and make the docker image again. We also have to swap the image in the frontend. This will become even more problematic if backend has several pods.

Too much work and complicated work. This is why we need a “Service”.





How Kubernetes Service works

Service has its own IP address and DNS which are stable. So the frontend is successfully decoupled from the backend services. Therefore, a Service is a High-level stable abstract point for multiple pods.

For the discovery of Pods, a service uses something called “labels”. Pods belong to a Service via labels. In the service initializing stage, we describe what labels the service should look for via “selector” flag. If the Service found a Pod with **all** the labels mentioned in the selector section, the Service will append its endpoint list and add the pod to the list. (Having extra labels than the mentioned, is acceptable. But should not miss any label mentioned.)

When a request comes to the Service, it uses a method like Round-Robbin, Random, etc. to select the request forwarding pod.

Use of Service object facilitates us with many advantages, like request forwarding to only healthy pods, load balancing, roll-back of versions, etc. But the most important advantage of a Service is successful decoupling of System components.

There are 5 types of Services available in Kuberntes which we can choose according to our purpose: (Source: Kubernetes.io, 2019)

1. **ClusterIP**: Exposes the service on a cluster-internal IP. Choosing this value makes the service only reachable from within the cluster. This is the default `ServiceType`.

2. **NodePort**: Exposes the service on each Node's IP at a static port (the NodePort). A ClusterIP service, to which the NodePort service will route, is automatically created. You'll be able to contact the NodePort service, from outside the cluster, by requesting `<NodeIP>:<NodePort>`.

3. **LoadBalancer**: Exposes the service externally using a cloud provider's load balancer. NodePort and ClusterIP services, to which the external load balancer will route, are automatically created.

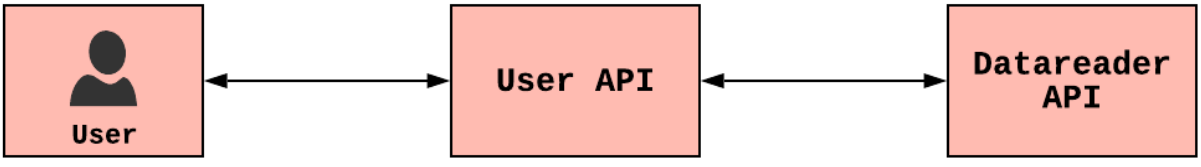
4. **ExternalName**: Maps the service to the contents of the `externalName` field (e.g. foo.bar.example.com), by returning a `CNAME` record with its value. No proxying of any kind is set up. This requires version 1.7 or higher of `kube-dns`

OK! Enough with theories. Let's get our hands dirty ☺. If you want to learn more check out Kubernetes for the Absolute Beginners — Hands-on course, its an absolute delight.

Hands-On with Springboot and Google Kubernetes Engine

Our small system can convert US State Codes to State names and vice versa. I need to make this tutorial to talk about communication between ***Pods.*** Therefore I will make the system have two APIs instead of one, even though we can do it with just one service.

All right! We are going to make 2 APIs; "***User API***" and "***Datareader API***".



Micro Service Architecture of Tutorial's Project

The User API which is given access by users have two end-points.

- [GET] request → /codeToState (with parameter “code”)
- [GET] request → /stateToCode (with parameter “state”)

The Datareader API which is accessed by the UserAPI also has two end-points.

- [GET] request → /readDataForCode
- [GET] request → /readDataForState

Let’s start with developing this locally using Spring boot.

1. Developing the Spring Boot API

I will be using IntelliJ Idea Ultimate to create my Spring Boot project.

In IntelliJ, Go to “New Project” and select Spring Initializer from the side-pane. Leave the service URL to default and select your JDK version and click next. In the next window also remember to select appropriate Java version. The rest you can change according to your preference and click next again.

In the next window, select web from side-pane and check the boxes in front of the Web and the Rest Repository dependencies. Click next again. Select the project location in the final window and click finish.

First, we will make the Datareader API with the project we just created. In the main class, write the following code.

```
package com.savindi.codestatebkend;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
@RestController
public class CodestatebkendApplication {
    final static String serverUrl1 = "https://gist.githubusercontent.com/PhantomGrin/a1e8ad/states_hash.json";

    final static String serverUrl2 = "https://gist.githubusercontent.com/PhantomGrin/a1e8ad/states_titlecase.json";

    public static void main(String[] args) {
        SpringApplication.run(CodestatebkendApplication.class, args);
    }

    public static String requestProcessedData(int urlid){
        String serverUrl = null;
        if(urlid == 1){
            serverUrl = serverUrl1;
```

```
        }else if (urlid == 2){
            serverUrl = serverUrl2;
        }else{
            return "ERROR";
        }

        RestTemplate request = new RestTemplate();
        String result = request.getForObject(serverUrl, String.class);
        System.out.print(serverUrl);
        return (result);
    }

    @GetMapping("/")
    public static String Hello(){
        return "HELLO IM DATA READER";
    }

    @GetMapping("/readDataForCode")
    public static String requestCodeData(){
        return requestProcessedData(1);
    }

    @GetMapping("/readDataForState")
    public static String requestStateData() {
        return requestProcessedData(2);
    }
}
```

Code for Datareader API (Find the code in my GitHub repo)

Since I wish to run the User API in Spring boot default port (8080), I will change this one's port to 9090 using "***application.properties***" file in the resource folder. In application.properties, write `server.port=9090` .

Then Execute Maven Goal (In the Maven Sidebar)→ clean install.

Now run the project and check end-points using software like Postman or using browser. If you are successful, you will receive two different JSON from calling two end-points.

Let's call the DataReader API using User API when a user triggers an endpoint of User API. Here is the code for it:

```
package com.savindi.codestate;

import org.json.JSONArray;
import org.json.JSONObject;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
@RestController
public class CodestateApplication {
    public static final String serverUrl = "http://localhost:9090";

    public static void main(String[] args) {
        SpringApplication.run(CodestateApplication.class, args);
    }

    public static String requestProcessedData(String url){
        RestTemplate request = new RestTemplate();
```

```
        String result = request.getForObject(url, String.class);
        System.out.print(url);
        return (result);
    }

    @GetMapping("/")
    public static String Hello(){
        return "I'M YOUR CONVERTOR";
    }

    @GetMapping("/codeToState")
    public static String CodeToState(@RequestParam("code") String code){
        String state = null;
        try {
            String response = requestProcessedData(serverUrl+"/readDataForCode");
            JSONObject jsonObject = new JSONObject(response);
            state = jsonObject.getString(code.toUpperCase());
        } catch (Exception e) {
            System.out.println("[ERROR] : [CUSTOM_LOG] : " + e);
        }

        if(state == null){
            state = "No Match Found";
        }
        return state;
    }

    @GetMapping("/stateToCode")
    public static String StateToCode(@RequestParam("state") String state){
        String value = "";
        try {
            String response = requestProcessedData(serverUrl+"/readDataForState");
            JSONArray jsonArray = new JSONArray(response);

            for(int n = 0; n < jsonArray.length(); n++)
            {
                JSONObject object = jsonArray.getJSONObject(n);
                String name = object.getString("name");

                if(state.equalsIgnoreCase(name)){
                    value = object.getString("abbreviation");
                    break;
                }
            }

        } catch (Exception e) {
            e.printStackTrace();
            System.out.println("[ERROR] : [CUSTOM_LOG] : " + e);
        }

        if(value == null){
            value = "No Match Found";
        }
        return value;
    }
```

Code for User API (Find the code in my GitHub repo)

Run both services simultaneously and check whether all endpoints are working.

> *Note: here the User API's serverUrl variable is localhost*

If all are working properly, you are good to go to the next step.

2. Converting Spring Boot API to Docker image

I will first dockerize my Datareader API.

Go to your project's root directory (in my case it's "codestatebkend") and open it in the terminal. Next, create a Dockerfile file using `touch Dockerfile`. Open that Dockerfile using a text editor or `nano` and enter the following:

```
# Start with a base image containing Java runtime (mine java 8)
FROM openjdk:8u212-jdk-slim

# Add Maintainer Info
LABEL maintainer="savindi.narmada@gmail.com"

# Add a volume pointing to /tmp
VOLUME /tmp

# Make port 8080 available to the world outside this container
EXPOSE 8080

# The application's jar file (when packaged)
ARG JAR_FILE=target/codestatebkend-0.0.1-SNAPSHOT.jar

# Add the application's jar to the container
ADD ${JAR_FILE} codestatebkend.jar

# Run the jar file
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-
jar","/codestatebkend.jar"]
```

As we have completed our Dockerfile, now we can move on to making the Docker Image. To avoid installing anything in my local machine and for the sake of ease, I am going to use Google shell from Dockerizing to Kubernetes deployment.

The easiest way to get your project into Google shell is through a Git repository. After you commit the code in the Git Repo, Go to Google cloud account and click the shell button in the right-side top corner. Then clone the Git repo inside the project shell using `git clone`.

Go to the cloned projects root directory and type, `./mvnw clean package` to make the jar file. Now all ready. First, set `PROJECT-ID` variable using:

```
export PROJECT_ID="$(gcloud config get-value project -q)"
```

Now, Let's build the Docker image with bellow code in the terminal. (don't forget the dot at the end)

```
docker build -t gcr.io/${PROJECT_ID}/codestatebkend .
```

Let's check whether we are successful:

```
docker images
```

If you see your image in the list, you are successful!

Now let's run this docker image to check whether everything is working fine.

```
docker run --rm -p 8080:8080
gcr.io/${PROJECT_ID}/codestatebkend:latest
```

Click on "Web preview" button in the right-top corner of the Shell window. You will see the message "Hello I'm DataReader"

Similarly, make the Docker image for the User API. Before that don't forget to change the target URL in the code (variable named `serverUrl` in my code) from localhost to down below:

```
http://dataservice.default.svc.cluster.local
```

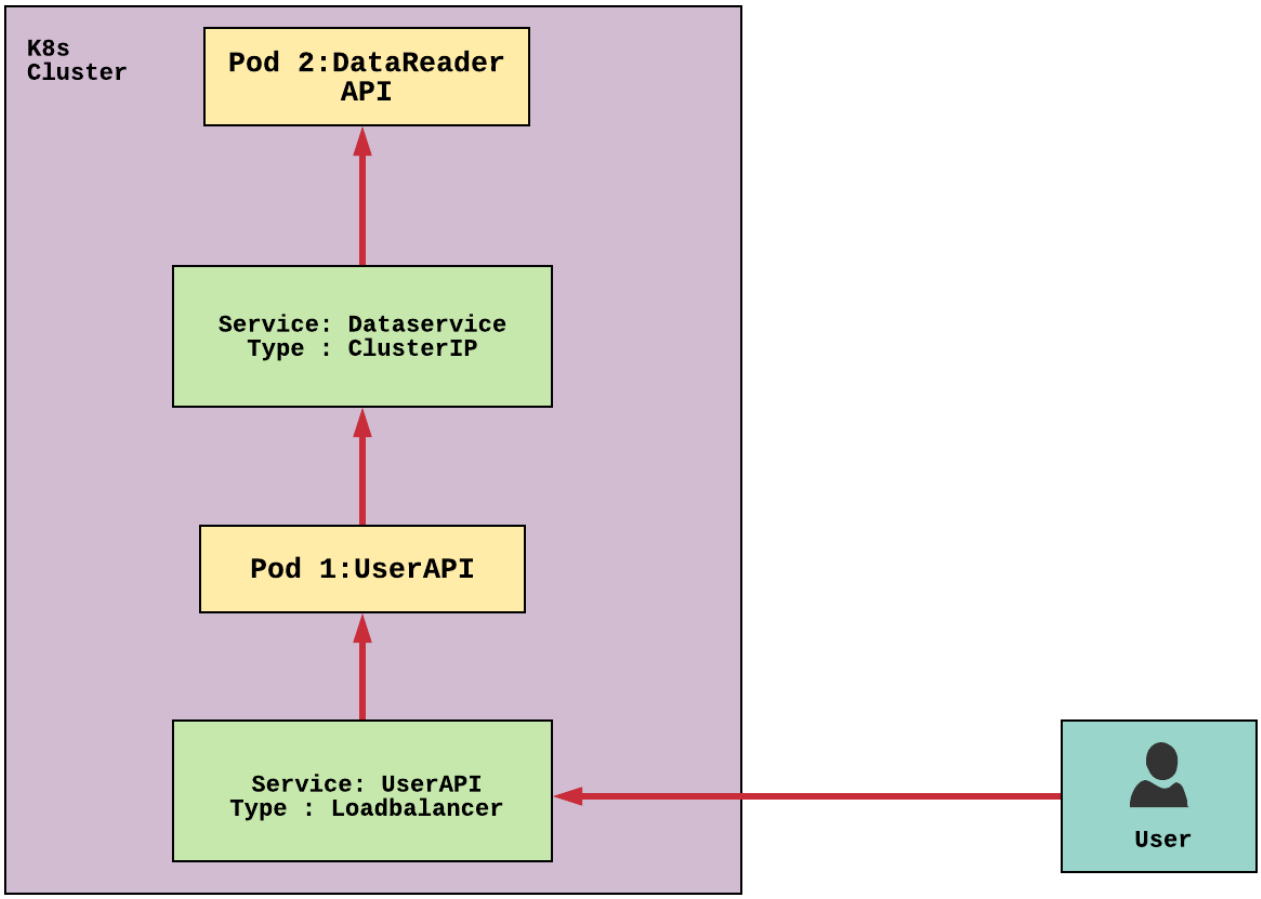
With the URL used as above, it is a **must** to name our ClusterIP Service as "dataservice".

3. Deploying the Docker image in Google Image Repository

```
docker push gcr.io/${PROJECT_ID}/codestatebkend:latest
```

4. Make a Cluster

Before making the cluster, I will describe you the architecture I am going to use in the deployment:



The architecture of the System being deployed

HEre according to the use case, I have chosen Loadbalancer Service and ClusterIP Service for our system.

Let's first create a cluster with 3 nodes.

```
gcloud container clusters create k8s-medium --num-nodes=3 --zone=us-
central1-b
```

5. Deploy the Datareader API

```
kubectl run dataserver --
image=gcr.io/${PROJECT_ID}/codestatebkend:latest --port 8080 --
labels="app=codestatebkend,tier=backend"
```

Let's check whether a pod is created with our deployment object

```
kubectl get pods
kubectl get deployment
```

```
savindi_wijenayaka@cloudshell:~ (k8s-medium)$ kubectl get pods
NAME                          READY     STATUS              RESTARTS   AGE
datareader-5db6d9dcb7-4dsmj   0/1       ContainerCreating   0          8s
savindi_wijenayaka@cloudshell:~ (k8s-medium)$ kubectl get deployment
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
datareader   1         1         1            1           59s
```

Newly created pod with Datareader API image and the newly created Deployment Object

6. Service Discovery

Expose your User API to the outside world using the `expose` command:

```
kubectl expose deployment userapi --type=LoadBalancer --port 80 --
target-port 8080
```

Use `kubectl get service` to get the list of Kubernetes services.

```
savindi_wijenayaka@cloudshell:~ (k8s-medium)$ kubectl get service
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)        AGE
kubernetes   ClusterIP      10.23.240.1     <none>           443/TCP        36m
userapi      LoadBalancer   10.23.245.145   104.154.103.27   80:30498/TCP   2m
```

newly created Service object

User the External IP mentioned here to access the service through a web browser. You can see the message “I’m your Converter” being displayed.

Now let’s connect the User API with the backend, Datareader API. For that, I am going to use a ClusterIP Service.

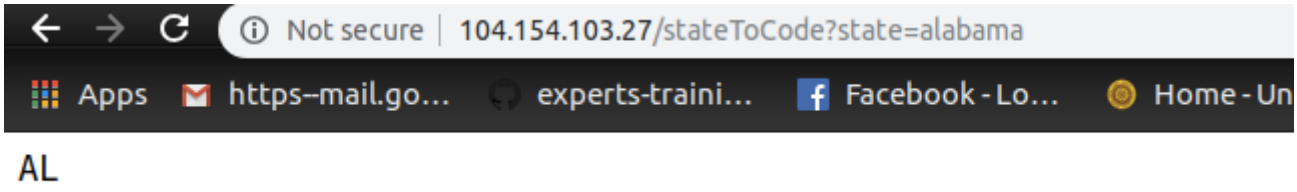
```
kubectl expose deployment dataservice --type=ClusterIP --port 80 --
target-port 8080 --selector="app=codestatebkend,tier=backend"
```

```
savindi_wijenayaka@cloudshell:~ (k8s-medium)$ kubectl get service
NAME          TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)        AGE
dataservice   ClusterIP      10.23.241.188   <none>           80/TCP         35m
kubernetes    ClusterIP      10.23.240.1     <none>           443/TCP        1h
userapi       LoadBalancer   10.23.245.145   104.154.103.27   80:30498/TCP   1h
```

Both services are up and running!

To check whether everything is working properly, call to an Endpoint of User API, using postman or browser

Ex: *http://104.154.103.27/stateToCode?state=alabama*



Success! Our System working as expected

Voila! You have a system with two tiers, fully running on a Kubernetes Cluster in Google Cloud!

6. Autoscaling capabilities

Something extra for you 😉

The above system I made with one running pod each for each API. But in a practical scenario, you need more than one Pod to manage the load. So Let me tell you how you can scale your application:

```
kubectl scale deployment userapi --replicas=2
```

Let's check whether we succeed, with `kubectl get pods` .

```
savindi_wijenayaka@cloudshell:~ (k8s-medium)$ kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
dataservice-64b49bb69f-lwt5f  1/1       Running   0          54m
userapi-6d6df9c77c-6s784      1/1       Running   0          1h
userapi-6d6df9c77c-qdb6x      1/1       Running   0          11s
```

Cluster with 2 UserAPI pods

7. Clean Up

```
gcloud container clusters delete [CLUSTER NAME]--zone=us-central1-b
```

Yes! It is simple as that □