Master Microservices with Spring Boot and Spring Cloud

Contents

Introduction to Microservices

Introduction to Microservices

- Small definition – small autonomous services that work together.

Challenges with Microservices

- Configuration Management
- Dynamic Scale up and down
- Visibility and Monitoring
- Fault Tolerance

Spring Cloud

- Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state).
- There are a wide range of projects under the umbrella of spring cloud.
- Spring Cloud provides solutions to above challenges.

Configuration Management

- Spring Cloud Config server
- We know that there would be multiple microservices, multiple environments for each of these microservices and multiple instances in many of those environments.
- This would mean that there is a lot of configuration for these microservices that the operations team needs to manage.
- Spring cloud config server provides an approach where you can store all configuration for all the different environment of all the microservices in a Git repository (Centralized location). And spring cloud config server can be used to expose that configuration to all the microservices.
- This helps us to keep the configuration in one place and that makes it very easy to maintain the configuration for all the MicroServices.

Dynamic Scale up and down

- Naming Server (Eureka)
    - All instances of all microservices will register with the naming server.
    - Naming Server has 2 features – service registration and service discovery.
- Spring Cloud LoadBalancer (New way of load balancing) Ribbon (Old way Client Side Load Balancing).
- Feign (Easier REST Clients)

Visibility (Logging) and Monitoring

- Zipkin distributed tracing
- Netflix API Gateway
- We would use Spring cloud Sleuth to assign ID to request across multiple components and we would use the Zipkin distributed tracing to trace a request across multiple components.
- One of the important things about microservices is these microservices have a lot of common features. For example logging, security, analytics and things like that you don't want to implement all these common features in every microservice.
- API Gateways provide great solutions to these kind of challenges. Netflix Zuul API gateway is an old way. The new way is to use Spring Cloud Gateway.

Fault Tolerence

- Hystrix (old way), Resilience4j (new way)
- We will also implement fault tolerance using hystrix.
- If a service is down, Hystrix help us to configure a default response.

Advantages of Microservices Architectures

- It enables you to **adapt new technology and processes** very easily. Each service can be implemented in different technologies.
- **Dynamic scaling** – you can scale up your applications and scale them down based on the Load. E.g. during festive seasons, you can scale up. During lockdown restriction, you can scale down.
- **Faster release cycles**.

Microservice Components – Standardizing Ports and URLs

- Since we might have many microservices, we develop a lot of components.
- Therefore it's very important to standardize the ports on which we would run these applications.
- E.g.
    - Ports: https://github.com/in28minutes/spring-microservices/tree/master/03.microservices#ports
    - URLs: https://github.com/in28minutes/spring-microservices/tree/master/03.microservices#urls

Microservices with Spring Cloud

- Refer Step by Step guide – https://github.com/in28minutes/spring-microservices-v2/blob/main/03.microservices/01-step-by-step-changes/microservices-v2-1.md

Using Feign REST Client for Service Invocation

- Using plain RestTemplate to perform service-t-service calls can be tedious, especially when we have multiple services.
- And that's where Spring Cloud provides us with a framework called Feign.
- Feign makes it really easy to call other Microservices.
- To use Feign, just add dependency spring-cloud-starter-openfeign
- Refer step 18 – https://github.com/in28minutes/spring-microservices-v2/blob/main/03.microservices/01-step-by-step-changes/microservices-v2-1.md#step-18
- Feign also helps us to do load balancing very easily.

Understand Naming Server and Setting up Eureka Naming Server

- Having Naming Server / Service Registry is a great solution since we can dynamically scale up or down.
- In a Microservice architecture, all the instances of all the Microservices would register with a Service Registry. And if a service wants to call another service, that caller service does a lookup in the service registry to find out the address of the other service. So that first service can call the other service using that address.
- Eureka is the famous naming server.
- Refer example – https://github.com/in28minutes/spring-microservices-v2/blob/main/03.microservices/01-step-by-step-changes/microservices-v2-1.md#eureka---step-19-to-21

Load Balancing with Eureka, Feign & Spring Cloud LoadBalancer

- When we add spring-cloud-starter-netflix-eureka-client to pom.xml, it's transitive dependency spring-cloud-starter-loadbalancer gets added in the classpath.
- spring-cloud-starter-loadbalancer is the load balancer framework that is used by feign to actually distribute the load among the multiple instances which are returned by Eureka.
- If you're using Eureka and Feign, then this client load balancing happens automatically.
- If you are add new instances and remove few existing instances of a service, you would see that typically within 15 to 30 seconds, all the changes are reflected and the load balancing will be done between all the available active instances at that particular point in time.

- Refer – https://github.com/in28minutes/spring-microservices-v2/blob/main/03.microservices/01-step-by-step-changes/microservices-v2-1.md#step-22

Setting up Spring Cloud API Gateway

- In a typical microservices architectures, there would be hundreds of microservices and these microservices have a lot of common features: authentication, authorization, logging, rate limiting, etc.
- These common features are typically implemented via API Gateways.
- In the older versions of Spring Cloud, the popular API gateway to use was **Zuul**. Since Zuul1 is no longer supported by Netflix, Spring Cloud has moved on and now the recommended option as an API gateway is **Spring Cloud Gateway**.
- Once API Gateway is setup, we can proxy through the API Gateway to other microservices using Eureka using the Cloud Gateway Discovery Locator feature.
- You can also customize the routes, add filters for routes, rewrite the routes with Spring Cloud Gateway.
- Refer – https://github.com/in28minutes/spring-microservices-v2/blob/main/03.microservices/01-step-by-step-changes/microservices-v2-1.md#spring-cloud-api-gateway---step-22-to-step-25
- Spring Cloud Gateway –
    - Simple, yet effective way to route to APIs
    - Provide cross cutting concerns: Security, Monitoring/metrics, etc.
    - Built on top of Spring WebFlux (Reactive Approach)
    - Features:
        - Match routes on any request attribute
        - Define Predicates and Filters
        - Integrates with Spring Cloud Discovery Client (Load Balancing)
        - Path Rewriting

Getting started with Circuit Breaker - Resilience4j

- Resilience4j is a lightweight, easy-to-use fault tolerance library inspired by Netflix Hystrix, but designed for Java 8 and functional programming.
- Official Website - https://resilience4j.readme.io/docs/getting-started
- Resilience4j provides solutions like below –
    - fallback response if a service is down
    - Implement a Circuit Breaker pattern to reduce load
    - Retry requests in case of temporary failures
    - Rate limiting (limiting ‘n’ number of requests per ‘m’ seconds)
    - Bulk head (max concurrent requests)

- Refer – https://github.com/in28minutes/spring-microservices-v2/blob/main/03.microservices/01-step-by-step-changes/microservices-v2-1.md#circuit-breaker---26-to-29

Distributed Tracing

- Microservices typically have complex call chains. So there are problems like debugging, tracing request across microservices when a request goes across microservices. That's where we go for a feature called Distributed Tracing.
- With Distributed Tracing, all the microservices involved would actually send all the information out to a single **Distributed Tracing Server** and this Server would store everything to a database (in-memory database or a real database).
- So, all the information that comes out from your microservices is stored by Distributed Tracing Server and the the Server would provide you an interface which would allow you to trace the requests across multiple microservices.
- E.g. **Zipkin** is one of the famous Distributed Tracing server.
- We would use Spring cloud Sleuth to assign ID to request across multiple components and we would use the Zipkin distributed tracing to trace a request across multiple components.
- Consider a scenario when the Distributed Tracing Server is down. The information from the microservices is lost. And to prevent that from happening what we can do is, we can have a queue in between like **Rabbit MQ**. All the microservices can send the information out to the Rabbit MQ queue and the Distributor Tracing Server can be picking up the information from the Rabbit MQ. This would make our Distributor Tracing more robust. And, this is the recommended architecture.
- Whenever we talk about sleuth or distributed tracing, we don't want to trace every request that goes between the microservices. What we want to do is to sample a percentage of the requests. We can do this setting a property in application.properties file.
- `spring.zipkin.baseUrl`, is the property you can configure with the Zipkin URL. By default. it's configured with http://localhost:9411. And if you are running Zipkin on the same URL as that of your microservices, it's able to easily find it. If you are running Zipkin on a different URL, you might also need to configure this as well.
- If you are using RabbitMQ to send information to zipkin, then we need to configure one more property `spring.zipkin.sender.type` to `rabbit`. By default it is `http`.
- Refer – https://github.com/in28minutes/spring-microservices-v2/blob/main/03.microservices/01-step-by-step-changes/microservices-v2-1.md#docker-section---connect-microservices-with-zipkin

Using Docker

- Refer – https://github.com/in28minutes/spring-microservices-v2/tree/main/04.docker

Dockerizing microservices

- Refer – https://github.com/in28minutes/spring-microservices-v2/tree/main/04.docker
- With Spring boot 2.3, it is very easy to create Docker containers.
- We just need to use the spring-boot-maven-plugin and add docker configuration for that microservice.

```
<plugin>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-maven-plugin</artifactId>
	<configuration>
		<image>
			<name>in28min/mmv2-
${project.artifactId}:${project.version}</name>
		</image>
		<pullPolicy>IF_NOT_PRESENT</pullPolicy>
	</configuration>
</plugin>
```

- To build the maven image, run

```
mvn spring-boot:build-image
```

Docker Compose

- To launch an application involving multiple microservice, we need to run each docker container for each microservice and manage their sequence as well. This is tedious in a typical microservices architecture.
- If we have multiple microservices (including naming service, API Gateway, zipkn, rabbitmq, etc.), launching up each one of those with all those docker commands is not going to be an easy thing and that's the reason why we go for Docker Compose.
- Docker Compose is a tool for defining and running multi-container Docker applications.
- You can simply configure a YAML file and with a single command (`docker-compose up`), you can launch up all the services which are defined inside the YAML file.
- Refer Step by Step docker-compose.yml file – https://github.com/in28minutes/spring-microservices-v2/tree/main/04.docker/backup
- Once docker-compose.yml file is ready, you can run it as

```
>docker-compose up
```

- **Note:** From inside the Docker container, when you try localhost:8761, it does not call localhost:8761 on your machine. This refers to the localhost inside the Docker container.

Kubernetes with Microservices using Docker, Spring Cloud

- Refer – https://github.com/in28minutes/spring-microservices-v2/tree/main/05.kubernetes

Container Orchestration

- Typical Features:
  - **Auto Scaling** - Scale containers based on demand
  - **Service Discovery** - Help microservices find one another
  - **Load Balancer** - Distribute load among multiple instances of a microservice
  - **Self Healing** - Do health checks and replace failing instances
  - **Zero Downtime Deployments** - Release new versions without downtime
- These are the features provided by container orchestration tools like kubernetes.
- Some of the container orchestration tool options are –
  - AWS Specific
    - AWS Elastic Container Service (ECS)
    - AWS Fargate : Serverless version of AWS ECS
  - Cloud Neutral - Kubernetes
    - AWS - Elastic Kubernetes Service (EKS)
    - Azure - Azure Kubernetes Service (AKS)
    - GCP - Google Kubernetes Engine (GKE)
    - EKS/AKS does not have a free tier!

Introduction

- Docker enables standardization of how you package and deploy your application, irrespective of the language or framework which is used to build the application or the platform where you would want to deploy the application to.
- With Kubernetes, you get all the features mentioned above. And you can manage 100s or 1000s of microservices in a declarative way.
- Kubernetes is cloud neutral. So, if you are deploying your applications to Kubernetes, you can run in any of the clouds.
- You need a Kubernetes cluster or a group of servers.
- Setting up Kubernetes is not easy. Creating and managing Kubernetes clusters can be really, really tough and that's where we use Google Kubernetes Engine (GKE).
- GKE is a managed service provided by GCP (Google Cloud Platform).
- So we don't really need to worry about creating a Kubernetes cluster and maintaining it; Google takes care of it for us.

Kubernetes Fun Facts

- Kubernetes' abbreviation is K8S.
- The logo of Kubernetes (Helmsman) represents somebody who is providing direction to a ship.
- Different cloud providers provide different services for Kubernetes.
    - Azure calls it Azure Kubernetes Service (AKS).
    - Amazon calls it Elastic Kubernetes Service (EKS)
    - Google calls it Google Kubernetes Engine (GKE).

Kubernetes command line

- `kubectl` is short form for Kube Controller. Kubectl is an awesome Kubernetes command to interact with the cluster.
- The interesting thing is, Kubectl would work with any Kubernetes cluster, irrespective of whether the cluster is in your local machine, whether it's in your data center, or it's in the cloud.
- Once you connect to the cluster, you can execute commands against any cluster using Kubectl.
- Kubectl can do a lot of powerful things for you like deploy a new application, increase the number of instances of an application, deploy a new version of the application, etc.
- The awesome thing is kubectl is already installed for us in the Google Cloud Shell.
- Some commands
    - To create a deployment
      `kubectl create deployment …….`
    - To expose deployed application to outside world
      `kubectl expose deployment …….`
- All commands for the course – https://github.com/in28minutes/spring-microservices-v2/tree/main/05.kubernetes#commands

Kubernetes Concepts

- Servers in the cloud are called **virtual servers**.
- The fact is that different cloud providers have different names for these virtual servers.
- **AWS** calls them EC2 (**Elastic Compute Cloud**).
  Azure calls them **virtual machines.**
  Google Cloud calls them **compute engines**.
  Kubernetes uses a very generic terminology and calls them **nodes**.
- Kubernetes can actually manage thousands of such nodes.
- To manage thousands of Kubernetes nodes, we have a few master nodes. Typically, you'll have one **master node**, but when you need high availability, you go for multiple master nodes.
- A **cluster** is nothing but a combination of nodes and the master node. The nodes that do the work are called **worker nodes or** simply **nodes**. The nodes that do the management work are called **master nodes**.
- Master nodes ensure that the nodes are available and are doing some useful work. So, at a high-level, a cluster contains nodes which are managed by a master node.
- We create and expose a simple deployment and the Kubernetes magically creates for us a pod, a replica set, a deployment, a service. (run `kubectl run events` command for details)
- Kubernetes uses single responsibility principle; one concept, one responsibility.
- When we execute the command `kubectl create deployment`, Kubernetes creates a **deployment, a replica set, and a pod.**
- When we execute the command `kubectl expose deployment`, Kubernetes creates a **service** for us.

Pods

- A **Pod** is the smallest deployable unit in Kubernetes.
- You might think, containers are the smallest deployable unit, but nope. In Kubernetes, the smallest deployment unit is actually the Pod.
- You cannot have a container in Kubernetes without a pod. Your container lives inside a pod.
- Command-
  ```
  >kubectl get pods
  >kubectl explain pods
  >kubectl describe pod <pod_name>
  ```
- Each pod has a unique IP address.
- A pod can actually contain multiple containers.
- All the containers which are present in a pod share resources.
- Within the same pod, the containers can talk to each other using localhost.

- A kubernetes **node** can contain multiple pods and each of these pods can contain multiple containers. These pods can be related to the same application or from different applications.
- To see details of a pod, run `kubectl describe pod`. You will see things associated with a pod like name, namespace, Annotations, node, labels, IP address, status and lot more.
- **Namespace** is a very important concept, it provides isolations for parts of the cluster, from other parts of the cluster. E.g. create namespace for each environment e.g. dev, qa.
- **Labels** are really important when we get to tying up a pod with a replica set or a service.
- **Annotations** are typically meta information about that specific pod like release id, build id, author name, etc.
- A pod provides a way to put your containers together. It gives them an IP address and also it provides a categorization for all these containers by associating them with labels.

Replica Sets

- Command-
  `>kubectl get replicasets`
- Replica sets ensure that a specific number of pods are running at all times.
- The replica set always keeps monitoring the pods and if there are lesser number of pods than what is needed, then it creates the pods.
- In practice, you would see that a replica set is always tied with a specific release version. So, a replica set V1 is responsible for making sure that, that specified number of instances of Version1 of the application are always running.

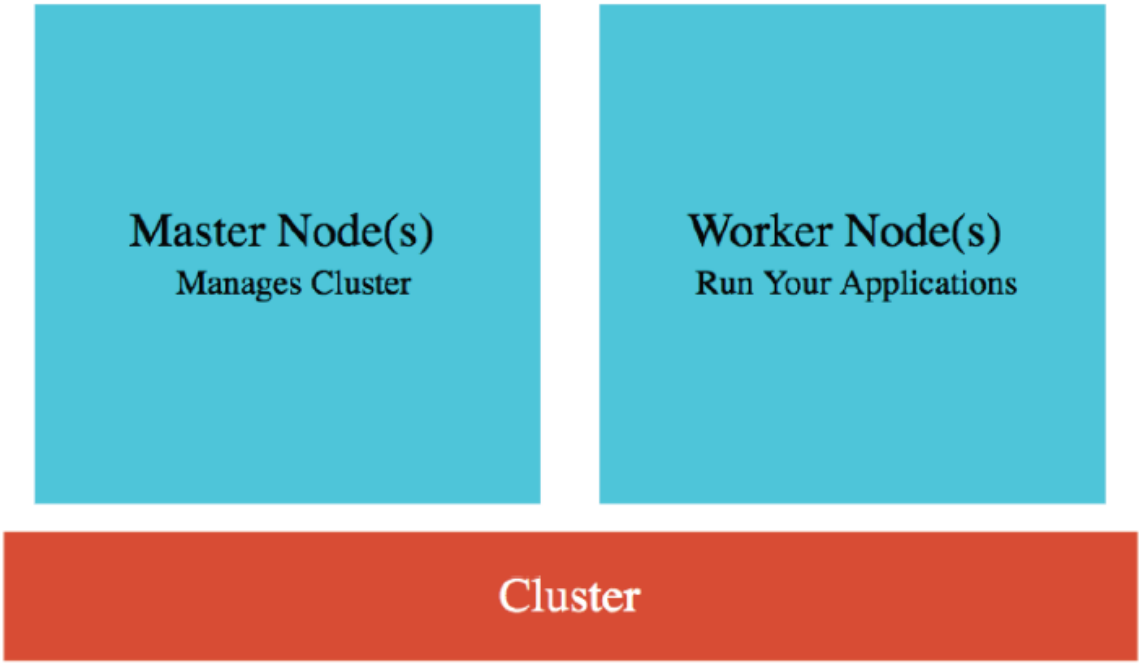
Deployment

- A deployment is very, very important to make sure that you are able to update new releases of applications without downtime.
- By default, the deployment uses **rolling updates strategy**.
- Let's say, I have five instances of V1 and I want to update to V2, the rolling update strategy, it updates one pod at a time. So, it launches up a new pod for V2, once it's up and running, it reduces the number of pods for V1. Next, it increases the number of pods for V2 and so on and so forth until the number of pods for V1 becomes zero and all the traffic goes then to V2.
- There are a variety of deployment strategies.

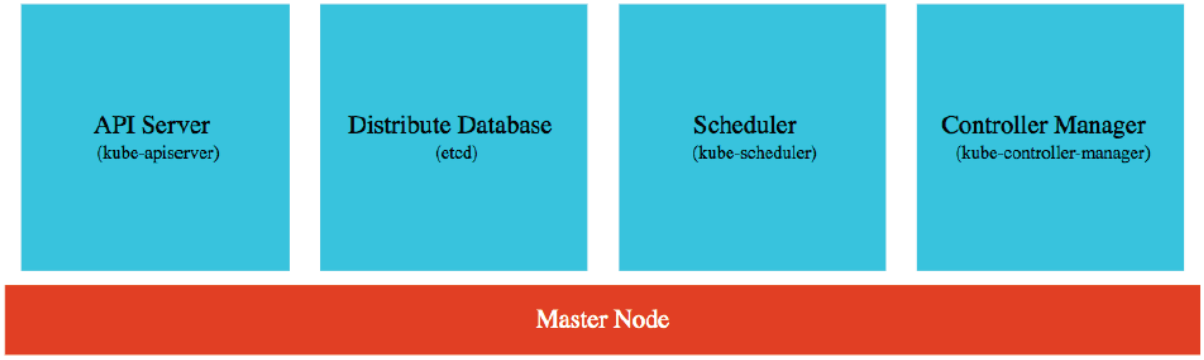
Services

- In the Kubernetes world, a pod is a throw away unit. Your pods might go down, new pods might come up. Whenever we do a new release, there might be several new pods which are created, and the old pods completely go away.
- Irrespective of all the changes happening with the pods, we don't want the consumer side of things to get affected. We don't want the user of the application to use a different URL as each pod gets a different IP address. That's the role of a service.
- The role of a service is to provide an always available external interface to the applications which are running inside the pods.
- A service basically allows your application to receive traffic through a permanent lifetime IP address.
- A service remains up and running. It provides a constant front-end interface, irrespective of whatever changes are happening to the back-end which has all the pods where your applications are running.
- Kubernetes provide excellent integration with different Cloud provider specific Load balancers.
- Command-
  `>kubectl get services`
- A **ClusterIP service** can only be accessed from inside the cluster. You will not be able to access this service from outside the cluster (No external IP address).

Kubernetes Architecture - Master Node and Nodes



Master Node



Distributed Database (etcd)

- The most important component that is running as part of your master node is something called **etcd**, that's the distributed database.
- All the configuration changes that we make, all the deployments that we create, all the scaling operations that we perform, all the details of those are stored in a distributed database. With those commands, we set the desired state for Kubernetes.
- So, all the Kubernetes resources like deployment, services, all the configuration that we make, is stored finally into this distributed database.
- Typically, we would recommend you to have about three to five replicas of this database so that the Kubernetes cluster state is not lost.

API Server (kube-apiserver)

- Kubectl or Cloud provider interface/console actions talks to Kubernetes cluster via this API Server.
- If I try to make a change through kubectl or if I try to make a change to the Google cloud console, the change is submitted to this API server and processed from here.

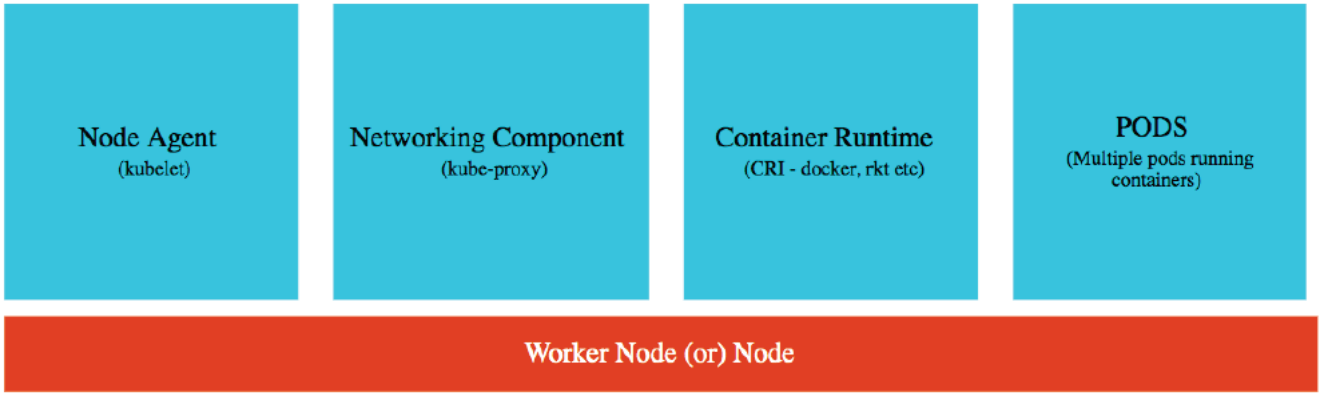
Scheduler

- The scheduler is responsible for scheduling the pods on to the nodes.
- In a Kubernetes cluster, you'll have several nodes and when we are creating a new pod, you need to decide which node the pod has to be scheduled on to.
- The decision might be based on how much memory is available, on how much CPU is available, are there any port conflicts, and a lot of such factors. So, scheduler considers all those factors and schedules the pods on to the appropriate nodes.

Controller Manager

- The controller manager manages the overall health of the cluster.
- Wherever we are executing kubectl commands, we are updating the desired state. The controller manager makes sure that the actual state of the Kubernetes cluster matches the desired state.

Worker Nodes (Nodes)



Pods

- The important thing about a master node is typically the user applications like our hello-world-rest-api will not be scheduled on to the master node.
- All the user applications would be running typically in pods inside the worker nodes or just the node.
- On a single Worker node, you might have several pods running.

Node Agent (Kubelet)

- The job of a kubelet is to make sure that it monitors what's happening on the (worker) node and communicates it back to the master node.
- So, if a pod goes down, it reports it to the controller manager.

Networking Component (kube-proxy)

- It helps you in exposing services around your nodes and your pods.

Container Runtime

- We would want to run containers inside our pods and these need the container runtime.
- The most frequently used container runtime is **docker**.
- You can use Kubernetes with any Open Container Interface (OCI) runtime specification implementations.

GCloud & Kubectl Install

- We can deploy our application using Cloud Shell from browser. However we can also deploy the application from our terminal or from our command prompt.
- And to be able to do that, we would need to install two simple tools. One is called **Gcloud**, which is basically a command-line interface to Google Cloud and the second one is **Kubectl**, which is basically the command-line interface for Kubernetes.
- To install Gcloud – https://cloud.google.com/sdk/docs/install
- To install Kubectl – https://kubernetes.io/docs/tasks/tools/

Kubernetes Centralized Configuration

- You can add env variables inside deployment yaml file (declarative).
- Kubernetes also provides a centralized configuration option via config map.
- Command –
  To Create –
  `kubectl create configmap <map-name> --from-literal=<key-name>=<value>`
  To See details –
  `kubectl get configmap <map-name> -o yaml`
  Extract configuration to .tml file –
  `kubectl get configmap <map-name> -o yaml >> envconfig.yml`
  Include this configuration inside your deployment.yml file and make envRef changes then apply it using kubectl apply.

Kubernetes Probes

- When we move from one release to another release, there is a little bit of downtime, almost 15 to 20 seconds of downtime.
- The way we can avoid this downtime is by using the **liveness** and the **readiness** probes which are provided by Kubernetes.
- You can configure them to help Kubernetes check the status of an application.
- Kubernetes uses probes to check the health of a microservice:
  - If **readiness** probe is not successful, no traffic is sent to that microservice.
  - If **liveness** probe is not successful, pod is restarted.
- Spring Boot Actuator (>=2.3) provides inbuilt readiness and liveness probes:
  - /health/readiness
  - /health/liveness
- So, we can use these two probes which are provided by Spring Boot Actuator and configure them to ensure that there is no downtime when we are deploying our applications.

- We do this configuration in the deployment.yml file for the container.
- Configuring a readiness and liveliness probe ensures that if the application is not ready to receive traffic, then it will not terminate the old version of it. It will terminate the old pods only when the new pods are ready to receive traffic. If, for some reason there is some problem with your application, Kubernetes can easily find it by using the readiness and the liveness probes.

Kubernetes Autoscaling

- We can manually scale the number of pods (using e.g. kubectl scale command or yml configuration). With manual scaling, you have to monitor the load and you have to scale it by yourself.
- However **autoscaling** means to increase/decrease the number of pods based on the load on your application.
- Command for autoscaling (Horizontal Pod Autoscaling (HPA)) –
  kubectl autoscale ….
- With this command, we can configure minimum number of containers that should always be running is one, maximum number of pods and also we can specify how Kubernetes decides when to scale it up (e.g. CPU %).

Notes

- Repository –
  - https://github.com/sameerbhilare/Microservices
  - https://github.com/in28minutes/spring-microservices
  - Latest: https://github.com/in28minutes/spring-microservices-v2
  - Step by Step Guide – https://github.com/in28minutes/spring-microservices-v2/blob/main/03.microservices/01-step-by-step-changes/microservices-v2-1.md
- Stop => SIGTERM => Stop gracefully
- Kill => SIGKILL => Stop forcefully