Index

Application Information - *Solution2*

In this DemoApp, we will deploy a web application using Kubernetes. We will be using Docker as our container runtime. The application in our Assignment has three layers:

Compared to Solutions1 we are bringing in below high-end solutions to achieve for more Scalability, Performance and Security

- **Database (MySQL server)   * Using stateful set to Run a Replicated Stateful Application**
- **Back end (Java Spring Boot application) * using Load balancer with HTTPS listener**
- **Front end (Angular app) * Using Ingress Controller, Ingress and Application Load Balancer and External DNS**

Kubernetes Background for this Assignment:

1. We will deploy all of these components on a Kubernetes cluster.
2. We will configure the cluster by creating Kubernetes objects. These Kubernetes objects will contain the desired state of our deployment. Once these objects are persisted into the cluster state store, the internal architecture of Kubernetes will take necessary steps to ensure that the abstract state in the cluster state store is the same as the physical state of the cluster.
3. We will use kubectl to create the objects.
4. We will use the declarative approach in this example. For each object, we will first prepare a manifest file, a yaml file containing all the information related to the object. Then we will execute the kubectl command, kubectl apply -f <FILE_NAME>to persist the object in the cluster state store.

Application Background:

The application we are going to deploy has front end implemented with Angular Framework, and back end implemented with Spring Boot Framework, we will build executables with build tools (Angular CLI and Maven in this case), backend database is MySql

Application Functional Background:

This application is simple Tasks List/Todo List creator, you will be accessing the frontend to create the list where backend will save the record to database and also helps to get the records from database.

Migration Steps:

- **Application Deployment process for this Assignment:**

Step 1. Containerize the application

Containerize the application and upload image to container image registry, We will create container images by building docker images using Dockerfile for both frontend and backend used in this assignment are shown below.

I. Navigate to frontend Source Code path where your Dockerfile exists and run below commands:

   a. builds the image

      docker build -t todoapp1:latest .

   b. tag the image name with your repository and tag

      docker tag todoapp1:latest kirank2005/todoapp1:latest

   c. push the image with the above tag using below command (make sure you are authenticated with Docker Hub using docker login)

      docker push kirank2005/todoapp1:latest

II. Navigate to backend Source Code path where your Dockerfile exists and run below commands:

   a. builds the image

      docker build -t todoserver4:latest .

   b. tag the image name with your repository and tag

      docker tag todoserver4:latest kirank2005/todoserver4:latest

   c. push the image with the above tag using below command (make sure you are authenticated with Docker Hub using docker login)

      docker push kirank2005/todoserver4:latest

Step 2. Database pre-requisite configuration setup

Part of this step - we will create one configMap and two secrets. The configMap will contain non-sensitive information about the database setup, like the location where the database is hosted and the name of the database.

Execute the below Kubectl commands-

I. **kubectl apply -f 01CrerateDemoNamespace.yaml**
II. **kubectl apply -f 02db-credentials-secret.yaml**
III. **kubectl apply -f 03db-root-credentials-secret.yaml**
IV. **kubectl apply -f 04mysql-configmap.yaml**

Step 3. Configure PVC, SQL Service, and statefulset for database

Our next step would be to create the services and Replicated stateful application using a StatefulSet controller for replicated MySQL database. The example topology has a single primary server and multiple replicas, using asynchronous row-based replication.

Execute the below Kubectl commands-

I. **kubectl apply -f 05mysql-pv-service-statefulset.yaml**

Step 4. Configure service and deployment for backend

Part of this step we set up our back-end application deployment. Below is the yaml file which creates the required Kubernetes objects.

We are using Service of type LoadBalancer, which exposes the back-end instances. Loadbalancer type provides an External-IP(External load balancer DNS) , through which one could access the back-end services externally.

we will create the Deployment object configured to contain two replicas of the back-end instance. And then injected the required environment variables from the configMaps and secrets we have created earlier. This deployment will use the image **kirank2005/todoserver4:latest** which we created in step one.

Note: For security reasons you can deploy service with https by applying below yaml, by creating Public ACM Corticate using instructions from https://docs.aws.amazon.com/acm/latest/userguide/gs-acm-request-public.html , **06middlelayer-deployment.yaml**, annotation service.beta.kubernetes.io/aws-load-balancer-ssl-cert: arn:aws:acm:{region}:{user id}:certificate/{id} then apply the yaml.

Execute the below Kubectl commands-

I. **kubectl apply -f 06middlelayer-deployment.yaml**

Step 5. Front-end configuration setup

Front end expects the value of External-IP of back end, generated in the above step, to be passed in the form of the environment variable SERVER_URI. We will now create a config map to store this information related to the back-end setup.

ConfigMap storing information related to back-end server URI

We will use this configMap to inject SERVER_URI value when configuring the deployment of front end, in the next step.

Execute the below Kubectl commands-

NOTE : first you need to update 07frontend-backend-configmap.yaml using (I and II)

I. **kubectl get svc | grep <backend service name>**
kubectl get svc | grep to-do-app-backend

II. **copy the External Loadbalancer DNS Name and update the 07frontend-backend-configmap.yaml**

III. **kubectl apply -f 07frontend-backend-configmap.yaml**

Step 6. Configure service and deployment for frontend

we set up our front-end application deployment. Below is the yaml file which creates the required Kubernetes objects.

we will be creating Service of type **NodePort**, then we will be deploying **ingress controller** then we deploy ingress resource, based on ingress definition ALB ingress controller will create Application Load Balancer through which one could access the front-end services externally. This deployment will use the image **kirank2005/todoapp1:latest,** which we created in step one. After this, we injected the environment variable SERVER_URI from configMap, which we created in the above setup.

Note1: Complete the pre-requisites for Ingress Controller configuration from https://docs.aws.amazon.com/eks/latest/userguide/aws-load-balancer-controller.html then deploy **08IngressController.yaml**

Note2: the below **09frontend-Ingress.yaml** configuration contains Application Load Balancer configuration for HTTPS you need to create Public ACM Corticate using instructions from https://docs.aws.amazon.com/acm/latest/userguide/gs-acm-request-public.html , update **09frontend-Ingress.yaml**, annotation alb.ingress.kubernetes.io/certificate-arn: arn:aws:acm:{region}:{user id}:certificate/{id} then apply the yaml.

Execute the below Kubectl commands-

I. **kubectl apply -f 09frontendservice.yaml**
II. **kubectl apply -f 08IngressController.yaml**
III. **kubectl apply -f 09frontend-Ingress.yaml**

That's it. Our **Assignment** application is now completely deployed. After this, the front end of the application should be accessible using front-end service External-IP from any browser. The Angular app will call the back end through HTTP and back end will communicate with MySQL database.

All the steps from here help to provide additional high availability and security featres for our cluster and application.

Step 7. Setting up ExternalDNS for Services on AWS

ExternalDNS makes Kubernetes resources discoverable via public DNS servers. Like KubeDNS, it retrieves a list of resources (Services, Ingresses, etc.) from the Kubernetes API to determine a desired list of DNS records. Unlike KubeDNS, however, it's not a DNS server itself, but merely configures other DNS providers accordingly—e.g. AWS Route 53 or Google Cloud DNS.

Note 1: Complete the pre-requisites from https://github.com/kubernetes-sigs/external-dns/blob/master/docs/tutorials/aws.md

Then deploy external dns using

I. **kubectl apply -f 15ExternalDNS.yaml**

Step 8. Configure Horizontal Pod Autoscaler for frontend and backend deployments for high availability

the Kubernetes Horizontal Pod Autoscaler automatically scales the number of pods in a deployment, replication controller, or replica set based on that resource's CPU utilization. This can help your applications scale out to meet increased demand or scale in when resources are not needed, thus freeing up your nodes for other applications. When you set a target CPU utilization percentage, the Horizontal Pod Autoscaler scales your application in or out to try to meet that target.

Execute the below Kubectl commands-

I. **Kubectl apply -f 10HPA-backend.yaml**
II. **Kubectl apply -f 11HPA-frontend.yaml**

Step 9. Deploy ClusterAutoScaler

The Kubernetes Cluster Autoscaler automatically adjusts the number of nodes in your cluster when pods fail or are rescheduled onto other nodes. That is, the AWS Cloud Provider implementation within the Kubernetes Cluster Autoscaler controls the .DesiredReplicas field of Amazon EC2 Auto Scaling groups. The Cluster Autoscaler is typically installed as a Deployment in your cluster. It uses leader election to ensure high availability, but scaling is one done by a single replica at a time.

Execute the below Kubectl commands-

I. **Complete the pre-requisites from - https://docs.aws.amazon.com/eks/latest/userguide/cluster-autoscaler.html**
II. **Update the 14ClusterAutoScaler.yaml with cluster name**
   - --node-group-auto-discovery=asg:tag=k8s.io/cluster-autoscaler/enabled,k8s.io/cluster-autoscaler/<YOUR CLUSTER NAME>
III. **Kubectl apply -f 14ClusterAutoScaler.yaml**

Step 10. Deploy PodSecurityPolicy

A Pod Security Policy is a cluster-level resource that controls security sensitive aspects of the pod specification. The PodSecurityPolicy objects define a set of conditions that a pod must run with in order to be accepted into the system, as well as defaults for the related fields. They allow an administrator to control users and service account access.

Execute the below Kubectl commands-

I. **Kubectl apply -f 13PodSecurityPolicy.yaml**

Note: Pod security policy control is implemented as an optional (but recommended) admission controller. PodSecurityPolicies are enforced by enabling the admission controller, but doing so without authorizing any policies will prevent any pods from being created in the cluster.

When a PodSecurityPolicy resource is created, it does nothing. In order to use it, the requesting user or target pod's service account must be authorized to use the policy, by allowing the use verb on the policy. (I have added the example role and role binding in **14PodSecuritypolicyRoleandRolebinding.yaml),** please update the role and role binding details and apply

Execute the below Kubectl commands-

II. **Kubectl apply -f 14PodSecuritypolicyRoleandRolebinding.yaml**

For additional information please follow the reference - https://kubernetes.io/docs/concepts/policy/pod-security-policy/

Step 11. Disruption Budget for your Application

limits the number of concurrent disruptions that your application experiences, allowing for higher availability while permitting the cluster administrator to manage the clusters nodes

Execute the below Kubectl commands-

I. Kubectl apply -f 09PodDistruptionBudgets.yaml

Step 12. Network Policies

using Kubernetes NetworkPolicies for particular applications in your cluster. NetworkPolicies are an application-centric construct which allow you to specify how a pod is allowed to communicate with various network "entities" (we use the word "entity" here to avoid overloading the more common terms such as "endpoints" and "services", which have specific Kubernetes connotations) over the network.

Execute the below Kubectl commands-

II. Kubectl apply -f 12NetwrokPolocies.yaml

Note: i have created a sample example policy but that can be improvised using documentation

https://kubernetes.io/docs/concepts/services-networking/network-policies/

Testing:

- **How to access the application?**

Once you deploy below solution, you can open browser and simply use the frontend service load balancer DNS name or if you configured external DNS, it will create Domain using Route 53 or any other DNS solution you can use the domain name.