**Infra Optimization.**

Project 1 **Description**

**Background of the problem statement:**

A popular payment application, EasyPay where users add money to their wallet accounts, faces an issue in its payment success rate. The timeout that occurs with the connectivity of the database has been the reason for the issue. While troubleshooting, it is found that the database server has several downtime instances at irregular intervals. This situation compels the company to create their own infrastructure that runs in high-availability mode. Given that online shopping experiences continue to evolve as per customer expectations, the developers are driven to make their app more reliable, fast, and secure for improving the performance of the current system.

**Solution**

Create a DevOps infrastructure for an e-commerce application to run on **HA** (high availability) node.

**High-Level Details**

The application will run on Amazon’s Elastic Kubernetes Service (EKS). EKS is a robust commercial solution that is reliable and easily scales. Key to the service is the deployment of a database, running inside a Pod, that can easily be restarted for any reason. The data in the database must be persistent, i.e., not lost if the server goes down. So, in this solution the database pod uses a volume tied to the hardware of a particular node. Since we separate the backend hardware solution from the front-end, the vendor who employs this solution can decide how much to invest in network, storage, and processing for the database server.

**Details**

In order to get this working, we need to install the AWS CLI, EKS, and Ansible on a Mac, PC, or Linux system. (In my particular case I setup a Debian VM running in VirtualBox on my Macbook Pro.) With these tools in place we are ready to launch the Kubernetes cluster on AWS.

EKS requires an AWS account, which has been previously setup. Next we launch the EKS cluster using the EKS CLI, eksctl, as follows:

eksctl create cluster \

--name evansjr \

--region us-west-2 \

--nodes 3 \

--nodes-max 5 \

--with-oidc \

--ssh-access \

--managed

(On average it takes Amazon about twenty minutes to setup the cluster.)

Once the Kubernetes cluster is up and running, the nodes are provisioned with Ansible. This requires that we be able to SSH into the nodes, which EKS allows using PKI certificates. All the ansible scripts are available at the github repository for this Project: **http://github.com/evansjr2000/Capstone.git**, and some examples are given in the appendix. After the K8s cluster was up and running I used ansible to configure the nodes to run my favorite editor (EMACS) and setup the system to run NFS on the database node, and NFS client on the remaining nodes. Here I use a ping command with ansible to see that all my nodes are up and running:

evansjr@MyDeb2:~/ctools/myansible/pre-bootstrap$ ./ping.sh

k8snode01 | SUCCESS => {

"changed": false,

"ping": "pong"

}

k8snode02 | SUCCESS => {

"changed": false,

"ping": "pong"

}

k8smaster01 | SUCCESS => {

"changed": false,

"ping": "pong"

And here I login to the master node, and show that NFS is configured and running;

$ ssh ec2-user@$master

Warning: No xauth data; using fake authentication data for X11 forwarding.

Last login: Tue Jul 6 19:32:28 2021 from 096-035-036-192.res.spectrum.com

\_\_| \_\_|\_ )

\_| ( / Amazon Linux 2 AMI

\_\_\_|\\_\_\_|\_\_\_|

https://aws.amazon.com/amazon-linux-2/

3 package(s) needed for security, out of 12 available

Run "sudo yum update" to apply all updates.

[ec2-user@master01 ~]$ exportfs

exportfs: could not open /var/lib/nfs/.etab.lock for locking: errno 13 (Permission denied)

[ec2-user@master01 ~]$ sudo exportfs

/nfs\_share\_dir 192.168.57.176

/nfs\_share\_dir 192.168.95.52

[ec2-user@master01 ~]$

**Setting up a Persistent Volume**

The database will reside on a specific node of the cluster. To prevent the database pod running on a node not containing the database storage we will label the node.

kubectllabel node$(kubectl getnodes -ojsonpath=’{.items[0].metadata.name}’) kiamol=ch05

Here is the specification for the persistent volume, which will attach to the node with the correct label, as defined above.

apiVersion: v1 : PersistentVolume metadata:

name: pv01 spec:

capacity:

storage: 50Gi

accessModes:

* ReadWriteOnce local:

path: /nfs\_share\_dir nodeAffinity:

required:

nodeSelectorTerms:

* matchExpressions:
* key: kiamol operator: In values:
* ch05

And here is the manifest for the Persistent Volume Claim.

apiVersion: v1 kind: PersistentVolumeClaim metadata: name: postgres-pvc spec:

accessModes:

* ReadWriteOnce resources:

requests:

storage: 40Mi storageClassName: ""

The access mode of **ReadWriteOnce** limits access to this volume to one pod, in this case the database pod. As previously mentioned the database pod will run on a specified node of the cluster.

Here are the manifests for the database service and deployment;

apiVersion: v1 kind: Service metadata:

name: todo-db spec:

ports:

- port: 5432 targetPort: 5432

selector:

app: todo-db

type: ClusterIP

--apiVersion: apps/v1 kind: Deployment metadata:

name: todo-db spec:

selector:

matchLabels:

app: todo-db template:

metadata:

labels: app: todo-db spec:

containers:

* name: db image: postgres:11.6-alpine env:
* name: POSTGRES\_PASSWORD\_FILE value: /secrets/postgres\_password volumeMounts:
* name: secret mountPath: "/secrets"
* name: data mountPath: /var/lib/postgresql/data

volumes:

* name: secret secret:

secretName: todo-db-secret defaultMode: 0400 items:

* key: POSTGRES\_PASSWORD path: postgres\_password
* name: data persistentVolumeClaim: claimName: postgres-pvc

To access the database we use a web front-end. This particular example uses a LoadBalancer manifest and we allow it to scale, using a Horizontal Pod Autoscaler (HPA) manifest as follows:

apiVersion: autoscaling/v1 kind: HorizontalPodAutoscaler metadata: name: todo-web labels: kiamol: ch19 spec:

scaleTargetRef: apiVersion: apps/v1 kind: Deployment name: todo-web minReplicas: 1 maxReplicas: 5

targetCPUUtilizationPercentage: 10

The target CPUUtilizationPercentage is set artificially low above in order to demonstrate autoscaling when the from end web server undergoes a large load.

The following is the manifest for the web service:

apiVersion: v1 kind: Service metadata: name: todo-web spec:

ports:

- port: 8081 targetPort: 80

selector:

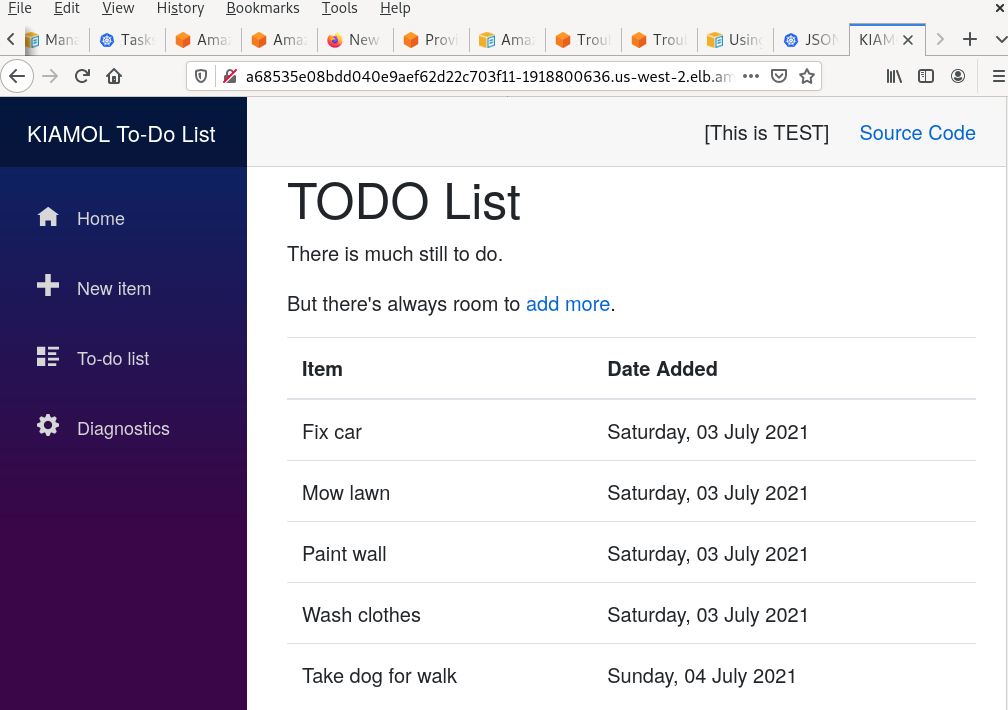
app: todo-web

type: LoadBalancer

The manifest for the deployment is very large and can be found in the appendix.

**Testing**

1. To verify that the database robustly restarts after dying (or being deleted) and that the backend data is preserved (persistent) we show what is in the database before deleting the service, then as the service will automatically start another deployment of the database, we show that the data is still there.



**I**

**$ kubectl get all**

**NAME READY STATUS RESTARTS AGE**

**pod/todo-db-795b8996bc-6854n 1/1 Running 0 3d4h**

**pod/todo-web-7dcbdb6586-lmc62 1/1 Running 0 2d9h**

**NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE**

**service/kubernetes ClusterIP 10.100.0.1 <none> 443/TCP 6d7h**

**service/todo-db ClusterIP 10.100.203.42 <none> 5432/TCP 3d4h**

**service/todo-web LoadBalancer 10.100.29.67 a68535e08bdd040e9aef62d22c703f11-1918800636.us-west-2.elb.amazonaws.com 8081:30580/TCP 3d4h**

**NAME READY UP-TO-DATE AVAILABLE AGE**

**deployment.apps/todo-db 1/1 1 1 3d4h**

**deployment.apps/todo-web 1/1 1 1 3d4h**

**NAME DESIRED CURRENT READY AGE**

**replicaset.apps/todo-db-795b8996bc 1 1 1 3d4h**

**replicaset.apps/todo-web-7dcbdb6586 1 1 1 2d9h**

**NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE**

**horizontalpodautoscaler.autoscaling/todo-web Deployment/todo-web 0%/10% 1 5 1 2d10h**

**Now we kill the database pod:**

**evansjr@MyDeb2(180)$ kubectl get pods**

**NAME READY STATUS RESTARTS AGE**

**todo-db-795b8996bc-6854n 1/1 Running 0 3d4h**

**todo-web-7dcbdb6586-lmc62 1/1 Running 0 2d9h**

**evansjr@MyDeb2(181)$ kubectl delete pods todo-db-795b8996bc-6854n**

**pod "todo-db-795b8996bc-6854n" deleted**

**kubectl get pods**

**evansjr@MyDeb2(182)$ kubectl get pods**

**NAME READY STATUS RESTARTS AGE**

**todo-db-795b8996bc-l25zl 1/1 Running 0 37s**

**todo-web-7dcbdb6586-lmc62 1/1 Running 0 2d9h**

**As you can see above. the database pod is deleted, but since it runs as**

**part of a service it is restarted immediately.**

**To verify the data is till there we can run a curl command on the URL:**

**curl** [**http://a68535e08bdd040e9aef62d22c703f11-1918800636.us-west-2.elb.amazonaws.com:8081/list**](http://a68535e08bdd040e9aef62d22c703f11-1918800636.us-west-2.elb.amazonaws.com:8081/list)

**And showing part of the output, we see that the data is still there.**

**<tbody>**

**<tr>**

**<td>Fix car</td>**

**<td>Saturday, 03 July 2021</td>**

**</tr>**

**<tr>**

**<td>Mow lawn</td>**

**<td>Saturday, 03 July 2021</td>**

**</tr>**

**<tr>**

**<td>Paint wall</td>**

**<td>Saturday, 03 July 2021</td>**

**</tr>**

**<tr>**

**<td>Wash clothes</td>**

**<td>Saturday, 03 July 2021</td>**

**</tr>**

**<tr>**

**<td>Take dog for walk</td>**

**<td>Sunday, 04 July 2021</td>**

**</tr>**

**</tbody>**

**Now to show that the application scales. I will call the curl command**

**several thousand times, and we get:**

**NAME READY UP-TO-DATE AVAILABLE AGE**

**deployment.apps/todo-db 1/1 1 1 3d5h**

**deployment.apps/todo-web 3/3 3 3 3d4h**

**NAME DESIRED CURRENT READY AGE**

**replicaset.apps/todo-db-795b8996bc 1 1 1 3d5h**

**replicaset.apps/todo-web-7dcbdb6586 3 3 3 2d10h**

**NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE**

**horizontalpodautoscaler.autoscaling/todo-web Deployment/todo-web 11%/10% 1 5 2 2d10h**

**Which as you can see is autoscaling.**

\ll{Accessing the K8s cluster}

To access the AWS EKS cluster, a user with proper permissions must be

setup from the AWS console using the IAM service.

\ll{Conclusion}

Kubernetes is ideal for setting up reliable services, as it takes care of all the networking and cluster management for you. It is designed for setting up robust reliable services.

**Appendix**

$ kubectl get nodes -o wide

NAME STATUS ROLES AGE

VERSION INTERNAL-IP EXTERNAL-IP OS-IMAGE KERNEL-VERSION

CONTAINER-RUNTIME ip-192-168-1-152.us-west-2.compute.internal Ready <none> 2d2h v1.19.6-eks-49a6c0 192.168.1.152 34.216.36.123 Amazon Linux 2 5.4.117-58.216.amzn2.x86\_64 docker://19.3.13

ip-192-168-57-176.us-west-2.compute.internal Ready <none> 2d2h v1.19.6-eks-49a6c0 192.168.57.176 35.80.10.134 Amazon Linux 2 5.4.117-58.216.amzn2.x86\_64 docker://19.3.13

ip-192-168-95-52.us-west-2.compute.internal Ready <none> 2d2h v1.19.6-eks-49a6c0 192.168.95.52 34.221.188.50 Amazon Linux 2 5.4.117-58.216.amzn2.x86\_64 docker://19.3.13

Using the above information I create a hosts file for Ansible to access the Kubernetes nodes:

$ cat hosts [k8s-nodes]

k8smaster01 ansible\_host=34.216.36.123 ansible\_user=ec2user

k8snode01 ansible\_host=35.80.10.134 ansible\_user=ec2-user

k8snode02 ansible\_host=34.221.188.50 ansible\_user=ec2user

[all:vars]

# Debian uses python3 instead of python2, # we need to set the interpreter for ansible ansible\_python\_interpreter=/usr/bin/python

[kube-master] k8smaster01

[kube-node] k8snode01 k8snode02

Here is the kubernetes manifest for the deployment of the web service:

apiVersion: apps/v1 kind: Deployment metadata: name: todo-web spec:

selector: matchLabels: app: todo-web template:

metadata:

labels:

app: todo-web spec:

containers:

* name: web image: kiamol/ch04-todo-list resources:

limits:

cpu: 250m

requests:

cpu: 125m

env:

* name: ASPNETCORE\_ENVIRONMENT value: Test volumeMounts:
* name: config mountPath: "/app/config" readOnly: true
* name: secret mountPath: "/app/secrets" readOnly: true

volumes:

* name: config configMap: name: todo-web-config items:
* key: config.json path: config.json
* name: secret secret:

secretName: todo-web-secret defaultMode: 0400 items:

* key: secrets.json path: secrets.json