Kubernetes Workshop

Part 4: Kubernetes Advanced

What will we do for the next hour?

Learn the more advanced features of Kubernetes

- High availability
- Liveness probes / Readiness probes (health checks)
- Resource Quotas
- Automatic scaling
- Node selectors
- Static pods / daemon sets
- Volumes / Secrets
- Ingress
- Debugging
- Namespaces
- Identity / Authorization

Kubernetes Labels and Selectors

- Most Kubernetes components support the use of Labels which allow you to quickly filter them.

- Kubernetes Selectors can be used to connect different Kubernetes components (like the example of Service and Pods).

```
# to filter a resource for a label -l
$ kubectl get pods -l environment=production
```

Example labels:

- `"release" : "stable"`, `"release" : "canary"`
- `"environment" : "dev"`, `"environment" : "qa"`, `"environment" : "production"`
- `"tier" : "frontend"`, `"tier" : "backend"`, `"tier" : "cache"`
- `"partition" : "customerA"`, `"partition" : "customerB"`
- `"track" : "daily"`, `"track" : "weekly"`

Health checks

- You can use health checks in your application to inform Kubernetes

- Kubernetes supports the use of different health checks
  - Readiness probe
  - Liveness probe

- A Readiness probe makes the pod unavailable if the health check fails

- A Liveness probe restarts the pod if the health check fails
  - You should use this if the application cannot recover from an unhealthy state.

Using health checks

- Readiness probes and Liveness probes share a similar API.
- A Pod will not serve traffic until the Readiness probe is successful. (initialDelaySeconds)
- You can see why a Readiness probe or Liveness probe fails with **`kubectl describe pods ...`**
- By correctly using Readiness probes and Liveness probes you can get a self healing cluster.

```
$ cat nginxv1.deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
        readinessProbe:
          httpGet:
            path: /index.html
            port: 80
          initialDelaySeconds: 30
          timeoutSeconds: 1
```

Exercise: Deploy the sample application while using labels and health checks

Exercise

- There is a webapp folder in ~/Desktop/Kubernetes Workshop/webapp
- The docker container is already available in your cluster with: gcr.io/aegal-kubernetes-workshop/webapp
- Deploy this application, while using labels and health checks.

Next up: Special Deployments

- Node selectors
- Daemon sets

Kubernetes Node selectors

- Labels can also be assigned to Nodes, and with the use of Node selectors, you can tell a pod on which nodes to run.
  - For example, only certain services are allowed to run on node X and Y. For security or resource reasons.

```
# to assign labels to nodes (marking node as highCPU group):
$ kubectl label nodes <node-name> group=highcpu
```

```
$ cat nginxv1.deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
      nodeSelector:
        group: highcpu
```

Kubernetes Daemon sets

- Daemon sets can be used if you want all nodes in your cluster to run a copy of a Pod.

- Daemon sets can also be used in combination with node selectors to only run on some nodes in your cluster.

- Some uses for Daemon sets:
    - Running a cluster storage daemon on each node (glusterd, ceph).
    - Running a logs collection daemon on each node (fluentd, logstash).
    - Running a node monitoring daemon on each node

Exercise: Convert your Nginx pod to a Daemon set

Exercise

- Convert your Nginx deployment to a Daemon set.
- Make use of node selectors so that Nginx only runs on 2/3 nodes.

```
# to assign labels to nodes (marking node as highCPU group):
$ kubectl label nodes <node-name> group=highcpu
```

```
$ cat nginxv1.deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
      nodeSelector:
        group: highcpu
```

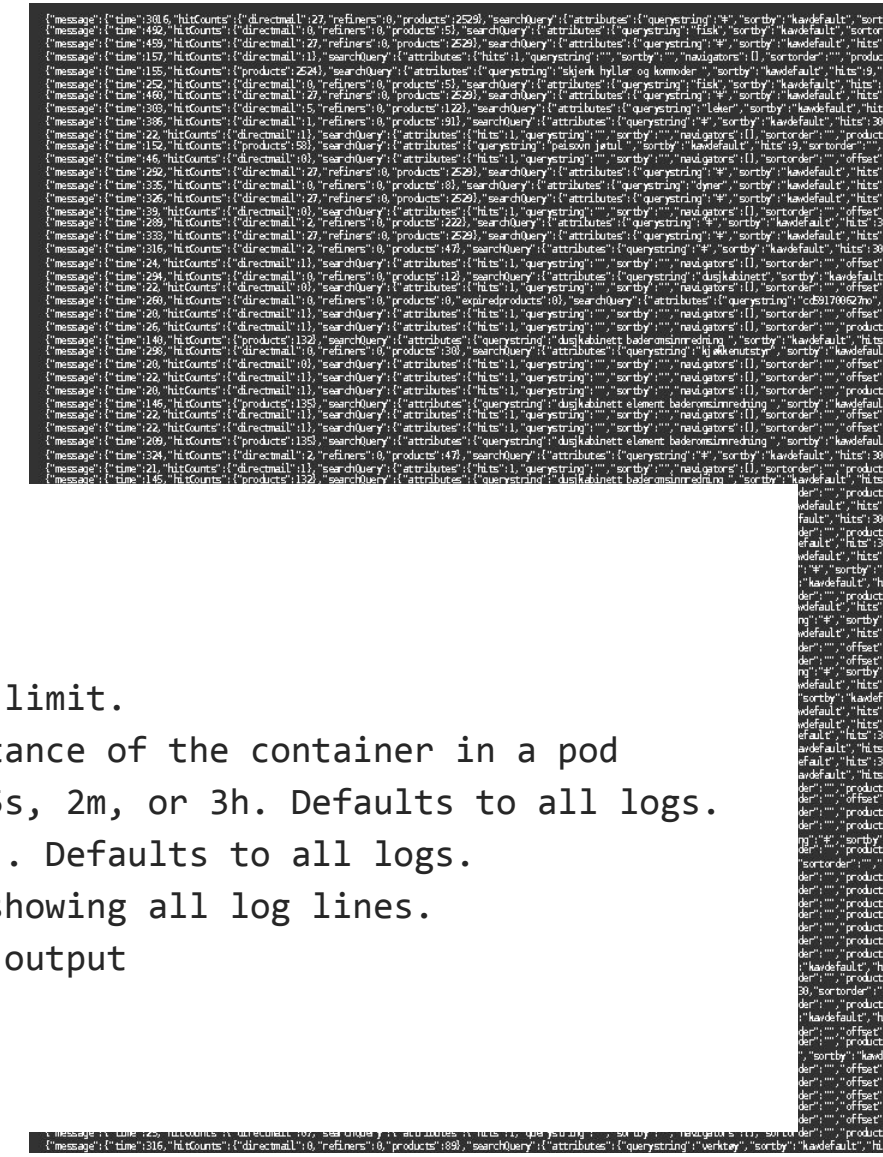
Next up: Debugging your cluster

- Logging
- Exec
- Monitoring

Logging

To see the logs of a container:

```
$ kubectl logs <podname>
```



```
$ kubectl logs -h
  -c, --container="": Print the logs of this container
  -f, --follow[=false]: Specify if the logs should be streamed.
      --limit-bytes=0: Maximum bytes of logs to return. Defaults to no limit.
  -p, --previous[=false]: If true, print the logs for the previous instance of the container in a pod
      --since=0: Only return logs newer than a relative duration like 5s, 2m, or 3h. Defaults to all logs.
      --since-time="": Only return logs after a specific date (RFC3339). Defaults to all logs.
      --tail=-1: Lines of recent log file to display. Defaults to -1, showing all log lines.
      --timestamps[=false]: Include timestamps on each line in the log output
```

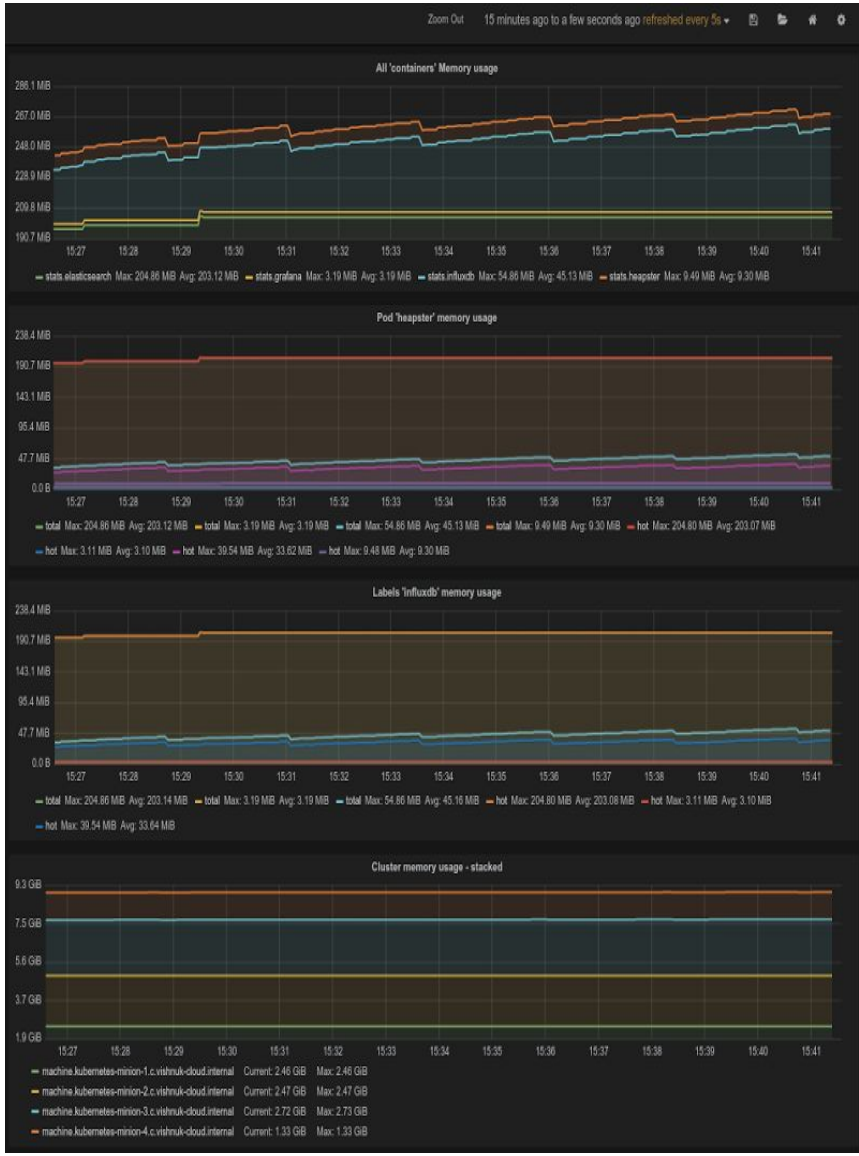
Exec

Just like **`docker exec`**, you can also do **`kubectl exec`**

```
$ kubectl exec <args> <podname> <command>
```

Monitoring

- Understanding how your cluster behaves is crucial
- Kubernetes supports monitoring on different levels: containers, pods, services, and whole clusters
- Your Kubernetes cluster already contains Heapster to collect the metrics.



Exercise: Add monitoring to your cluster

Execute the following commands:

```
cd ~/Desktop/kubernetes-workshop/examples/
$ kubectl create -f monitoring
$ kubectl get pods --all-namespaces

$ kubectl cluster-info
# find the line that says: monitoring-grafana

$ kubectl config view
# and use that username (admin) + password
```

Next up: Volumes

- Default volumes
- Secrets
- Persistent volumes

Kubernetes volumes

- Just like in Docker, you can assign volumes to Pods.

- But Kubernetes supports different types of Volumes:
  - emptyDir
  - **hostPath** (similar to Docker volumes)
  - gcePersistentDisk, awsElasticBlockStore, azureFileVolume
  - nfs, iscsi, flocker, glusterfs, rbd
  - gitRepo
  - **secret**
  - **persistentVolumeClaim**

```
$ cat nginxv1.deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
      volumeMounts:
      - mountPath: /test-pd
       name: test-volume
    volumes:
    - hostPath:
      name: test-volume
      path: /export
```

Kubernetes Secrets

- Secrets should be used when holding sensitive information such as passwords, OAuth tokens, and ssh keys.
- Secrets can be uploaded to Kubernetes, and only administrators can access them (those who have access to ETCD).
- Secrets can be created from file:

```
$ kubectl create secret generic db-user-pass --from-file=password.txt
```

- Or from definition:

```
apiVersion: v1
kind: Secret
metadata:
  name: db-user-pass
type: Opaque
data:
  password: MWYyZDFlMmU2N2RmCg==
```

```
$ cat nginxv1.deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
      volumeMounts:
      - mountPath: /ssh-keys/
       name: ssh-keys
    volumes:
    - secret:
      secretName: ssh-keys
```

Persistent volumes

- Instead of telling the pods which volume type they need, you can encapsulate the type into a persistent volume object:
  - gcePersistentDisk, awsElasticBlockStore, azureFileVolume, nfs, iscsi, glusterfs, rbd

- Other Pods can then do a Persistent Volume claim to request storage from the Persistent Volume capacity.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  nfs:
    path: /tmp
    server: 172.17.0.2
```

★ Persistent volumes are usable right now, but it is still in development.

Next up: Accessing pods

- Port exposing
- Ingress

Port exposing

To quickly access a pod, you can use port forwarding in Kubernetes

```
$ kubectl port-forward POD [LOCAL_PORT:]REMOTE_PORT [...[LOCAL_PORT_N:]REMOTE_PORT_N]
```

Ingress

- An Ingress is a collection of rules that allow inbound connections to reach the cluster services. (a.k.a. reverse proxy)

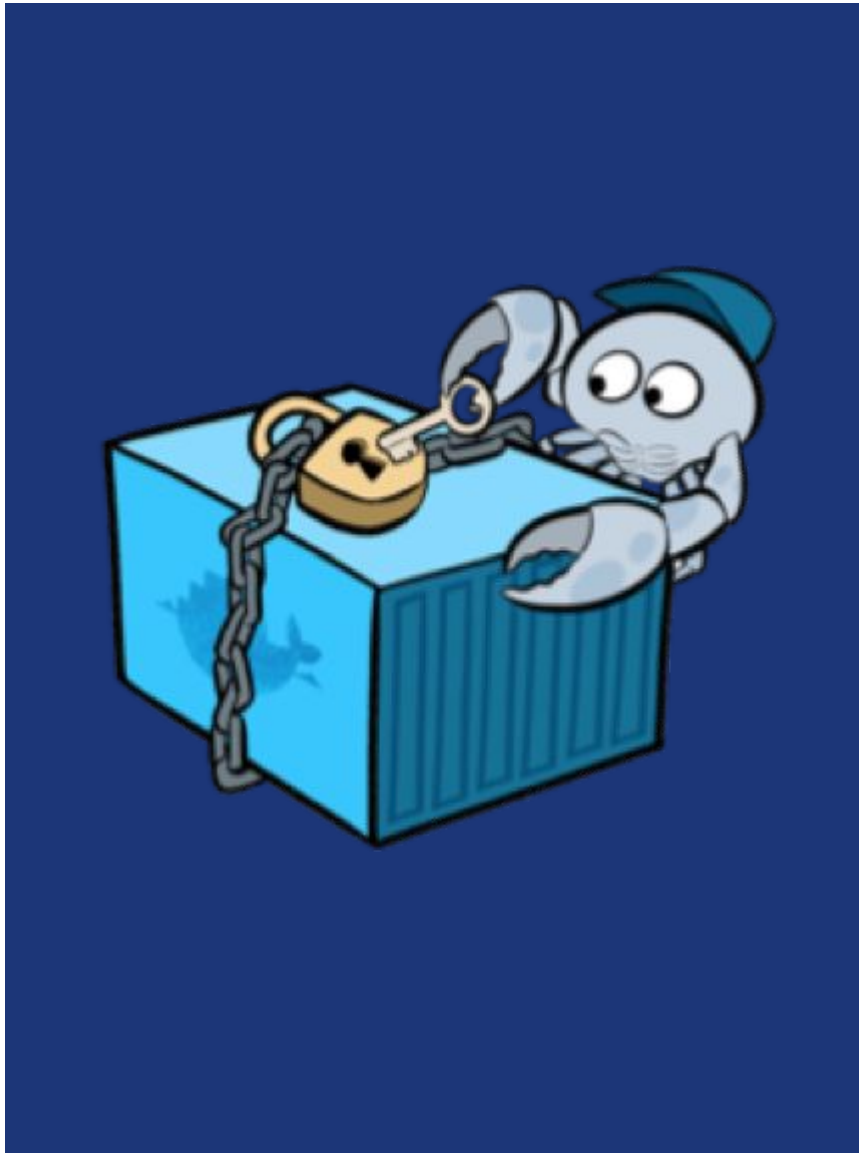
- It can be configured to give services:
  - externally-reachable urls
  - load balance traffic
  - terminate SSL
  - offer name based virtual hosting
  - etc

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: s1
          servicePort: 80
      - path: /bar
        backend:
          serviceName: s2
          servicePort: 80
```

Security

Kubernetes supports a number of security configurations, that can make the use of Docker more secure:

- RunAsNonRoot - By default, Docker runs its containers as root, this option prevents that.
- SeLinux options - Allows SeLinux options to enhance the security.
- RunAsUser - Forces the container to run as a certain user.
- Capabilities - Allow fine grained control to extend or limit the capabilities of a container.



Administrating your cluster

You can use the following Kubernetes features, for more control over your cluster

- Resource quota - limits the use of CPU and RAM for a Pod.
- Horizontal Pod Auto scaling - Allows automatic scaling based on CPU usage.
- Namespaces - Allows you to separate Kubernetes pods / users.
- ServiceAccounts - Allows access control and authorization for Kubectl.

Final Exercise: Deploy an ELK cluster to Kubernetes

Other K8S features

- Auto scaling.
- Federations (since 1.5)
- PetSets (Statefull sets)
- Dynamic storage