Kubernetes

Why Kubernetes?

De-factor standard for orchestrating container deployments

- Automatic deployment
- Scaling & Load Balancing
- Management

Pod

- Pod contains and runs one or multiple containers
- Pods contain shared resources like volumes for all Pod containers
- Pod has a cluster-internal IP by default so containers inside a pod can communicate via localhost

For Pods to be managed for you, Deployment is needed.

Deployment

- Controls pods by defining which pods and containers to run and number of instances
- Deployments can be paused, deleted, or rolled back
- Deployments can be scaled dynamically and automatically

```yaml
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: second-app-deployment
  labels:
    group: example
spec:
  replicas: 1
  selector:
    matchLabels:
      app: second-app
      tier: backend
  template:
    metadata:
      labels:
        app: second-app
        tier: backend
    spec:
      containers:
        - name: second-node
          image: <REGISTRY>/<IMG_NAME>:<REVISION>
          imagePullPolicy: Always
          livenessProbe:
            httpGet:
              path: /
              port: 8080
```

```
            periodSeconds: 10
            initialDelaySeconds: 5
```

Service

- Service exposes and allows access from the cluster or from outside
- Service groups pods with a shared IP

Accessible IP and port can found with `minikube service <SERVICE_NAME>`.

```
# service.yaml
apiVersion: v1
kind: Service
metadata:
  name: backend
  labels:
    group: example
spec:
  selector:
    app: second-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: LoadBalancer
```

kubectl commands can be run against defined labels: `kubectl delete deployments,services -l group=example`.

Volumes

Volume's lifecycle depends on the Pod lifecycle because volumes are part of a pod. Volumes survive container restarts and removals but not pod restarts.

```
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: story-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: story
  template:
    metadata:
      labels:
        app: story
    spec:
      containers:
        - name: story
          image: <DOCKER_HUB>/<IMG_NAME>
          volumeMounts:
            - mountPath: /app/story
              name: story-volume
```

```
      volumes:
        - name: story-volume
          emptyDir: {} # creates an empty directory when pod starts and removed when pod is removed
---
apiVersion: v1
kind: Service
```

However, with the above approach, there will be data inconsistency issue when there are more than 1 pod (1 replica). hostPath can help solving this problem by allowing multiple pods to share one path.

```
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: story-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: story
  template:
    metadata:
      labels:
        app: story
    spec:
      containers:
        - name: story
          image: guykorean/kub-data-demo
          volumeMounts:
            - mountPath: /app/story
              name: story-volume
      volumes:
        - name: story-volume
          hostPath:
            path: /data
            type: DirectoryOrCreate
---
apiVersion: v1
```

However, `hostPath` partially works, meaning it works for `minikube` but not on multi-node environment, because it is a single node environment. Also, it is still a problem when a pod is removed or replaced. Therefore, sometimes there are cases where pod-independent and node-independent volumes are required. Beside the normal volumes, k8s offers `Persistent Volumes`.

Persistent Volumes

The main idea is that persistent volume (PV) is detached from the pod lifecycle. PV Claim can be defined in the node to request access to PVs. This will make managing configuration files and volumes easier for bigger projects.

```
host-pvc.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: host-pv
spec:
  capacity:
```

```
    storage: 1Gi
  volumeMode: Filesystem # options: Filesystem or Block
  storageClassName: standard
  accessModes:
    - ReadWriteOnce # ReadOnlyMany and ReadWriteMany not available in single node env
  hostPath: # only works in single node env
    path: /data
    type: DirectoryOrCreate
```

```
# host-pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: host-pvc
spec:
  volumeName: host-pv
  accessModes:
    - ReadWriteOnce
  storageClassName: standard
  resources:
    requests:
      storage: 1Gi # should be less than capacity defined in pv
```

```
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: story-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: story
  template:
    metadata:
      labels:
        app: story
    spec:
      containers:
        - name: story
          image: guykorean/kub-data-demo
          volumeMounts:
            - mountPath: /app/story
              name: story-volume
      volumes:
        - name: story-volume
          persistentVolumeClaim:
            claimName: host-pvc
---
apiVersion: v1
kind: Service
metadata:
  name: story-service
spec:
  selector:
    app: story
  type: LoadBalancer
```

```
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
```

Container Storage Interface (CSI) is a flexible type which allows user to attach any storage solution as long as integration is provided like AWS EFS.

Environment Variables

In node app, environment variable can be used like:

```
const filePath = path.join(__dirname, process.env.STORY_FOLDER, 'text.txt')
```

```
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: story-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: story
  template:
    metadata:
      labels:
        app: story
    spec:
      containers:
        - name: story
          image: guykorean/kub-data-demo:2
          env:
            - name: STORY_FOLDER
              value: story
          volumeMounts:
            - mountPath: /app/story
              name: story-volume
      volumes:
        - name: story-volume
          persistentVolumeClaim:
            claimName: host-pvc
---
apiVersion: v1
kind: Service
metadata:
  name: story-service
spec:
  selector:
    app: story
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
```

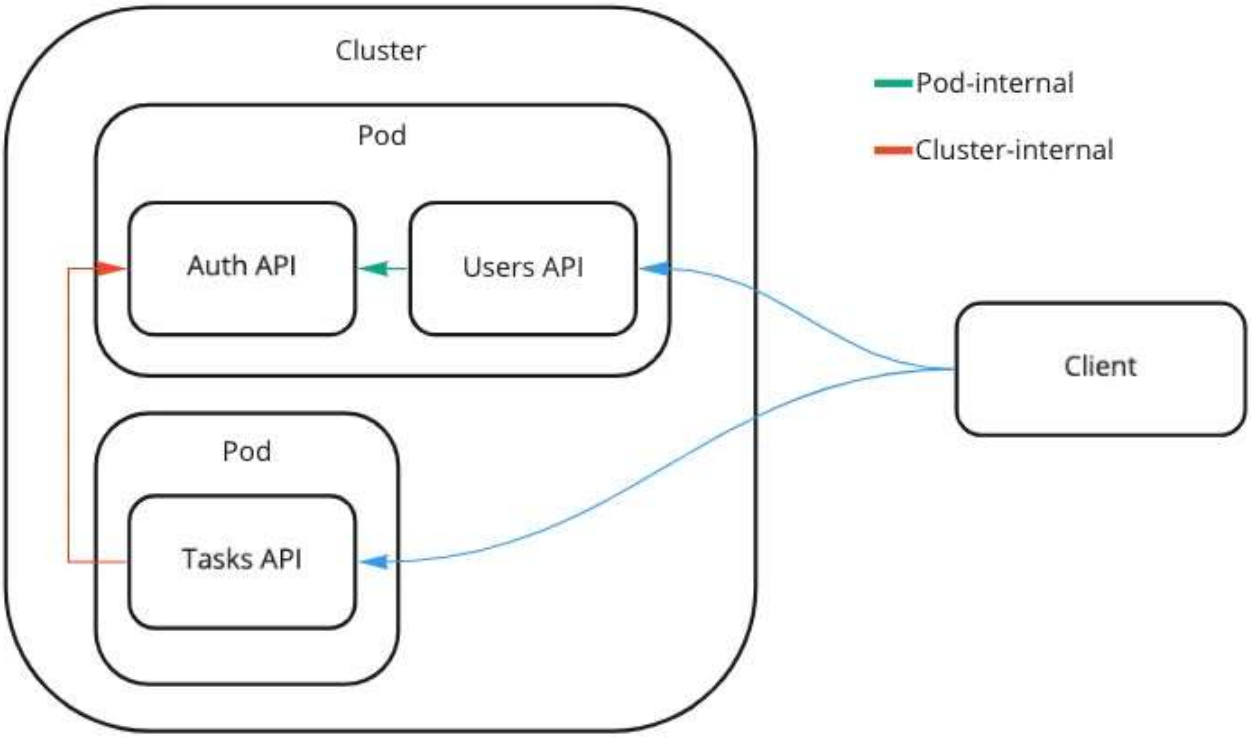
ConfigMap

```yaml
# environment.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: data-store-env
data:
  folder: 'story'
  # someKey: someValue
```

```yaml
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: story-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: story
  template:
    metadata:
      labels:
        app: story
    spec:
      containers:
        - name: story
          image: guykorean/kub-data-demo:2
          env:
            - name: STORY_FOLDER
              valueFrom:
                configMapKeyRef:
                  name: data-store-env
                  key: folder
          volumeMounts:
            - mountPath: /app/story
              name: story-volume
      volumes:
        - name: story-volume
          persistentVolumeClaim:
            claimName: host-pvc
---
apiVersion: v1
kind: Service
metadata:
  name: story-service
spec:
  selector:
    app: story
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
```

Networking

Pod-Internal Communication



Scenario:

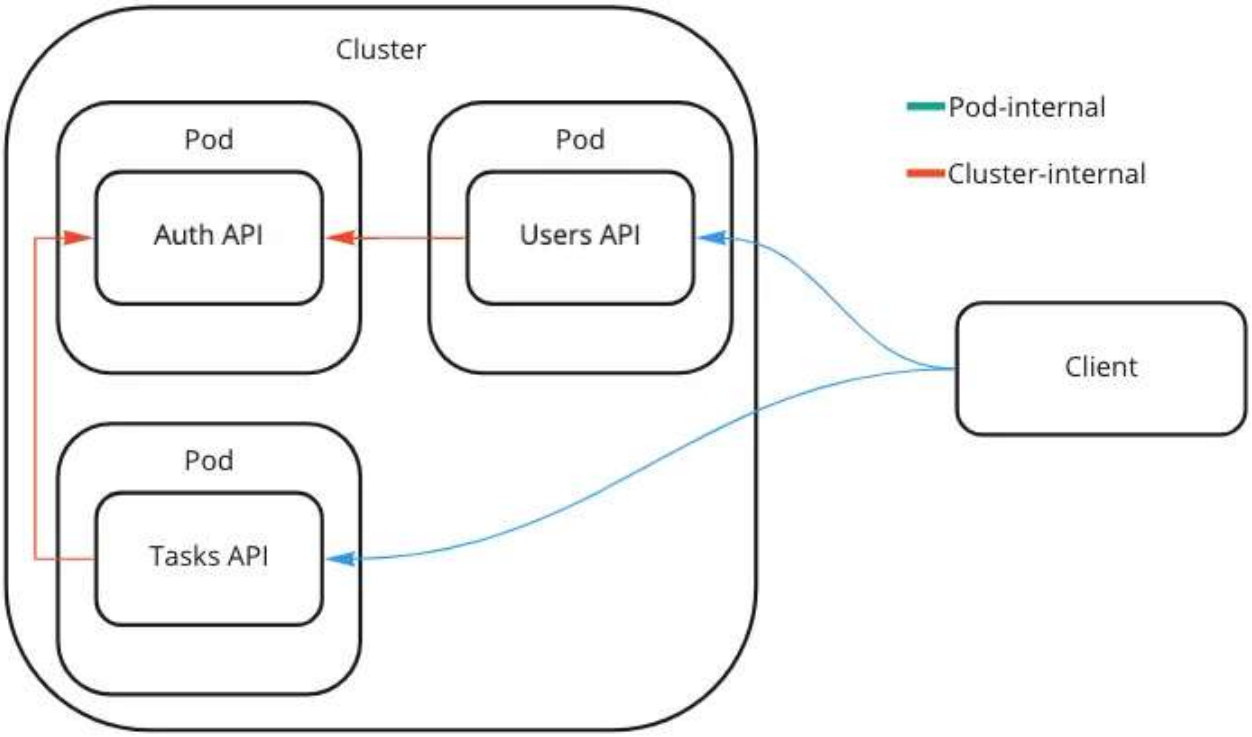
- Users API container runs on port 8080
- Auth API container runs on port 80
- Users and auth containers are running in the same pod
- Users API makes a request to auth API
- Users API needs to be exposed to the public

```
/* users backend */
axios.get(`http://${process.env.AUTH_ADDRESS}/some/path`)
```

```
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: users-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: users
  template:
    metadata:
      labels:
        app: users
    spec:
      containers:
        - name: users
          image: guykorean/kub-demo-users:latest
          env:
```

```
            - name: AUTH_ADDRESS
              value: localhost
        - name: auth
          image: guykorean/kub-demo-auth:latest
---
apiVersion: v1
kind: Service
metadata:
  name: users-service
spec:
  selector:
    app: users
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

Pod-to-Pod Communication



Scenario:

- `users API` container runs on port 8080
- `auth API` container runs on port 80
- `tasks API` container runs on port 8000
- `users`, `auth`, `tasks` each run in a separate pod
- `users API` and `tasks API` make requests to auth API
- `users API` and `tasks API` need to be exposed to the public

It should be noted that kubernetes automatically injects an environment variable `<SERVICE_NAME>_SERVICE_HOST` that can be used in the code (for example, `AUTH_SERVICE_SERVICE_HOST`). In addition to that kubernetes provides `CoreDNS` for service discovery within the cluster. In this example, `CoreDNS` is used.

```
# user-deployment.yaml
apiVersion: apps/v1
```

```yaml
kind: Deployment
metadata:
  name: users-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: users
  template:
    metadata:
      labels:
        app: users
    spec:
      containers:
        - name: users
          image: guykorean/kub-demo-users:latest
          env:
            - name: AUTH_ADDRESS
              value: auth-service
---
apiVersion: v1
kind: Service
metadata:
  name: users-service
spec:
  selector:
    app: users
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

```yaml
# auth-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: auth
  template:
    metadata:
      labels:
        app: auth
    spec:
      containers:
        - name: auth
          image: guykorean/kub-demo-auth:latest
---
apiVersion: v1
kind: Service
metadata:
  name: auth-service
spec:
  selector:
    app: auth
```

```
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

```
# tasks-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tasks-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: tasks
  template:
    metadata:
      labels:
        app: tasks
    spec:
      containers:
        - name: tasks
          image: guykorean/kub-demo-tasks:latest
          env:
            - name: AUTH_ADDRESS
              value: auth-service
            - name: TASKS_FOLDER
              value: tasks
---
apiVersion: v1
kind: Service
metadata:
  name: tasks-service
spec:
  selector:
    app: tasks
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 8000
      targetPort: 8000
```

Containerizing Frontend

Imagine there is another pod which hosts a React app in the cluster. This React app makes requests to `tasks API`.

First, make sure React app makes requests like `fetch('/api/tasks')`.

Second, define reverse proxy in `nginx.conf`. It is very nice that again DNS can be used to set up the reverse proxy. The port is defined as well because the `tasks API` container is exposed at port 8000. For the `location` syntax and how nginx behaves differently with the trailing slashes, refer to the reference at the very bottom.

```
# nginx.conf
server {
  listen 80;
```

```
  location /api/ {
    proxy_pass http://tasks-service.default:8000/;
  }

  location / {
    root /usr/share/nginx/html;
    index index.html index.htm;
    try_files $uri $uri/ /index.html =404;
  }

  include /etc/nginx/extra-conf.d/*.conf;
}
```

Third, define kubernetes yaml files

```
# frontend-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
        - name: frontend
          image: guykorean/kub-demo-frontend:latest
---
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  selector:
    app: frontend
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

References:

- https://www.udemy.com/course/docker-kubernetes-the-practical-guide/
- https://kubernetes.io/docs/tutorials/