Service Accounts and Network Policies

Service Accounts

- Humans authenticate through User Accounts.
- Service Accounts are needed when processes inside of containers need to access the K8s API server.

```
kind: ServiceAccount
apiVersion: v1
metadata:
name: my-service-account
```

Using service accounts

```
apiVersion: v1
kind: Pod
metadata:
  name: use-service-account-pod
spec:
# Set the service account containers in this pod
# use when they make requests to the API server
serviceAccountName: my-service-account
containers:
  - name: container-service-account
    image: nginx:latest
```

- Service accounts are on a namespace level

Network Policies

Networking model

- Containers on the same pod can talk to each other via localhost.
- Pods on the same node communicate via a pod's IP address.
- Pods on different nodes communicate via a pod's IP address.

Advantages

- Allows multi-container design patterns.
- No port coordination between pods is needed.
- Pods on different nodes can easily communicate with each other (unlike Docker networking).
- If a container within a pod dies and restarts, the IP address that other pods or services use to access it is still valid.
- Pods don't need to be concerned about which node a pod they are accessing is running on.
- **Using services is preferred to direct pod access**.

Network policies

- By default, every pod can communicate directly with every other pod via an IP address.
- However, you might want to restrict how groups of pods can communicate with each other.

Network policies (cont.)

- Network Policies let you group pods together using labels, and define rules between these groups.
- These rules are defined for:
  - *Ingress*: incoming connections to the group of pods
  - *Egress:* outgoing connections made by pods in the group.
- These policies are able to restrict network connections to specific IP ranges and port numbers.

Example

- Run a sample web app.

```
kubectl run hello-web --labels app=hello \
--image=gcr.io/google-samples/hello-app:1.0 \
--port 8080 --expose
```

Example (cont.)

```
#hello-allow-from-foo.yaml

kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: hello-allow-from-foo
spec:
  policyTypes:
  - Ingress
  podSelector:
    matchLabels:
      app: hello
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: foo
```

```
kubectl apply -f hello-allow-from-foo.yaml
```

Example (cont.)

- Now let's validate the ingress policy.
- First, get into a shell of a `foo` pod.

  ```
  kubectl run -l app=foo --image=alpine --restart=Never --rm -i -t test-1
  ```

- Make a request to `hello`.

  ```
  wget -qO- --timeout=2 http://hello-web:8080
  ```

Example (cont.)

- Check that you indeed can't connect to other app.
  `kubectl run -l app=other --image=alpine --restart=Never --rm -i -t test-1`
- `wget -qO- --timeout=2 http://hello-web:8080`

Example (cont.)

```yaml
# - Now let's try the opposite direction
# foo-allow-to-hello.yaml
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: foo-allow-to-hello
spec:
  policyTypes:
  - Egress
  podSelector:
    matchLabels:
      app: foo
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: hello
  - ports:
    - port: 53
      protocol: TCP
    - port: 53
      protocol: UDP
```

Example (cont.)

- Apply changes

  `kubectl apply -f foo-allow-to-hello.yaml`

- Create and expose a new application

```
kubectl run hello-web-2 --labels app=hello-2 \
  --image=gcr.io/google-samples/hello-app:1.0 \
  --port 8080 --expose
```

Example (cont.)

- Run a temporary pod with the `app=foo` label and get a shell inside the container

  `kubectl run -l app=foo --image=alpine --rm -i -t --restart=Never test-3`
- Check whether the pod can establish connections to:
  - `hello-web:8080`
  - `hello-web-2:8080`
  - `http://www.example.com`