Accessing K8S pods

This is some note, nothing serious.

Exposing services to external clients

Few ways to make a service accessible externally.

- Port-forwarding
- `NodePort` Service Type
- Service Object (`LoadBalancer` Service Type)
- Create `Ingress` Resource (Radically Different Mechanism)

Forwarding a Local Network Port to a Port in The Pod

When you want to talk to a specific pod without going through a service (for debugging or other reasons), Kubernetes allows you to configure port forwarding to the pod.

This is done through the `kubectl port-forward` command. The following command will forward your machine's local port `8888` to port `8080` of your e.g `kubia-manual` pod.

Example:

```
$ kubectl port-forward kubia-manual 8888:8080
```

Output:

```
... Forwarding from 127.0.0.1:8888 -> 8080
... Forwarding from [::1]:8888 -> 8080
```

Connecting to The Pod Through the Port Forwarder

In a different terminal, you can now use curl to send an HTTP request to your pod through the `kubectl port-forward` proxy running on `localhost:8888`.

Example:

```
$ curl localhost:8888
```

Output:

```
You've hit kubia-manual
```

Service Object

Each pod gets its own IP address, but this address is internal to the cluster and isn't accessible from outside of it. To make the pod accessible from the outside, you'll expose it through a Service object. You'll create a special service of type `LoadBalancer`, because if you create a regular service (a `ClusterIP` service), like the pod, it would also only be accessible from inside the cluster. By creating a `LoadBalancer` type service, an external load balancer will be created and you can connect to the pod through the load balancer's public IP.

Creating a Service Object

To create the service, you'll tell Kubernetes to expose the ReplicationController you created:

Using YAML file

Manifest:

```
apiVersion: v1
kind: Service
metadata:
  name: kubia
spec:
  ports:
  - port: 80 # service's port
    targetPort: 8080 # the forward-to port by service
  selector: # all pods labeled `kubia` will follow/select this service
    app: kubia
```

Apply the service:

```
$ kubectl create -f kubia-srv.yaml
```

Using kubectl CLI options

Template:

```
$ kubectl expose rc <rep-controller-name> --type=LoadBalancer --name <lb-name>
```

Expose:

```
$ kubectl expose rc kubia --type=LoadBalancer --name kubia-http
```

Output:

```
service "kubia-http" exposed
```

Remotely Executing Commands in Running Containers:

- You'll also need to obtain the cluster IP of your service (using `kubectl get svc`, for example)

```
$ kubectl exec kubia-7nog1 -- curl -s http://10.111.249.153
```

Output:

```
You've hit kubia-gzwli
```

Session Affinity on the Service

If you execute the same command a few more times, you should hit a different pod with every invocation, because the service proxy normally forwards each connection to a randomly selected backing pod, even if the connections are coming from the same client.

If, on the other hand, you want all requests made by a certain client to be redirected to the same pod every time, you can set the service's `sessionAffinity` property to `ClientIP` (instead of `None`, which is the default), as shown in the following listing.

Service with `ClientIP` Session Affinity Manifest

```
apiVersion: v1
kind: Service
spec:
  sessionAffinity: ClientIP
  ...
```

- Kubernetes supports only two types of service session affinity: `None` and `ClientIP`.
- Kubernetes services don't operate at the HTTP level. Services deal with TCP and UDP packets and don't care about the payload they carry. Because cookies are a construct of the HTTP protocol, services don't know about them, which explains why session affinity cannot be based on cookies.

Exposing Multiple Ports in the Same Service

Manifest

```
apiVersion: v1
kind: Service
metadata:
  name: kubia
spec:
  ports:
  - name: http
    port: 80
    targetPort: 8080
  - name: https
    port: 443
    targetPort: 8443
  selector:
    app: kubia
```

Using Named Ports

You can give a name to each pod's port and refer to it by name in the service spec.

Specifying port names in a pod definition Manifest:

```
kind: Pod
spec:
  containers:
  - name: kubia
    ports:
    - name: http
      containerPort: 8080
    - name: https
      containerPort: 8443
```

Referring to named ports in a service Manifest:

```
apiVersion: v1
kind: Service
spec:
  ports:
  - name: http
    port: 80
    targetPort: http
  - name: https
    port: 443
    targetPort: https
```

Connecting to services living outside the cluster

Instead of having the service redirect connections to pods in the cluster, you want it to redirect to external IP(s) and port(s).

This allows you to take advantage of both service load balancing and service discovery. Client pods running in the cluster can connect to the external service like they connect to internal services.

Service Endpoints

Services don't link to pods directly. Instead, a resource sits in between—the Endpoints resource. You may have already noticed endpoints if you used the `kubectl describe` command on your service.

Full details of a service:

```
$ kubectl describe svc kubia
```

Output:

```
Name:              kubia
Namespace:         default
Labels:            <none>
Selector:          app=kubia
Type:              ClusterIP
IP:                10.111.249.153
Port:              <unset> 80/TCP
Endpoints:         10.108.1.4:8080,10.108.2.5:8080,10.108.2.6:8080
Session            Affinity: None
No events.
```

An Endpoints resource (yes, plural) is a list of IP addresses and ports exposing a service. The Endpoints resource is like any other Kubernetes resource, so you can display its basic info with kubectl get.

```
$ kubectl get endpoints kubia
```

Output:

```
NAME     ENDPOINTS                                            AGE
kubia    10.108.1.4:8080,10.108.2.5:8080,10.108.2.6:8080      1h
```

Manually Configuring Service Endpoints

- having the service's endpoints decoupled from the service allows them to be configured and updated manually.
- If you create a service without a pod selector, Kubernetes won't even create the Endpoints resource
  - after all, without a selector, it can't know which pods to include in the service
- To create a service with manually managed endpoints, you need to create both a Service and an Endpoints resource

A service without a pod selector: `external-service.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: external-service # must match the endpoints name
spec:
  ports:
  - port: 80
```

- Endpoints are a separate resource and not an attribute of a service
- Because you created the service without a selector, the corresponding Endpoints resource hasn't been created automatically

A manually created Endpoints resource: `external-service-endpoints.yaml`

```
apiVersion: v1
kind: Endpoints
metadata:
  name: external-service # must match the service name
subsets:
  - addresses:
    - ip: 11.11.11.11
    - ip: 22.22.22.22
    ports:
    - port: 80 # target port of endpoints
```

Exposing services to external clients

download