18

7

39

```
      containers:
      - name: webapp1

        image: katacoda/docker-http-server:latest
        ports:
        - containerPort: 80
```

If you are familiar with Kubernetes, you can easily guess what this yaml says. It simply tells K8s to create a deployment which creates a pod, the pod runs the container image `katacoda/docker-http-server:latest`, it runs on port 80 inside the pod, so any request made to the pod at the port 80 should be received by this web-server. Let's deploy this with `kubectl`.

```
kubectl create -f web-server.yaml
```

If the cluster is properly setup, the deployment must be created and the pod should be running by now. Let's check.

```
kubectl get deployments
```

The output:

```
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
webapp1   1/1     1            1           15m
```

Now, let's see the pod. (I am using `-o wide` to see more information about the pod)

```
kubectl get pods -o wide
```

Output:

```
NAME                       READY   STATUS    RESTARTS   AGE   IP
webapp1-6b54fb89d9-ct7fk   1/1     Running   0          17m   10.4
```

Yes! We have the pod running for the deployment we created, Kubernetes has assigned an internal IP to the pod, which is `10.46.0.30`. We can use this IP anywhere inside our cluster to talk to the service. So, open the terminal inside the cluster (if minikube, run terminal directly, if you are using VMs, ssh into one of the VMs which is part of the cluster).
and make a `GET` to port 80 using curl. Make sure you replace the IP with the given IP in your cluster.

```
curl http://10.46.0.30
```

We see the response as shown below:

```
<h1>This request was processed by host: webapp1-6b54fb89d9-ct7fk</
```

This is the response returned by the web server. This means our set-up is correct and the web server is running.

Now it is time to setup another pod which makes a request to the web-server pod. To do this we use `byrnedo/alpine-curl` image and simply call `curl` command inside the pod by specifying the same IP. We will be creating a Job for this, since this is an one time activity. Let's create an YAML for this Job. The job simply makes a curl request to the IP we specified. `10.46.0.30`

is the IP of the server pod we created before. (`-s` is just to avoid printing unnecessary status and progress bar) (`client-job.yaml`)

```
apiVersion: batch/v1
kind: Job
metadata:
  name: client-job
spec:
  template:
    spec:
      containers:
      - name: client
        image: byrnedo/alpine-curl
        command: ["curl", "-s",  "http://10.46.0.30"]
      restartPolicy: Never
  backoffLimit: 4
```

Now we will deploy the job on K8s and see the result.

```
kubectl create -f client-job.yaml
```

Let's see the job

```
kubectl get jobs
```

Output:

```
NAME         COMPLETIONS   DURATION   AGE
client-job   1/1           2s         18m
```

The job is created and has terminated successfully. Now let's see the logs. Here `client-job-z6nql` is the pod created by the job `client-job` which we created in the previous step.

```
kubectl logs client-job-z6nql
```

So this will output the `curl` result.

```
<h1>This request was processed by host: webapp1-6b54fb89d9-ct7fk</
```

So that's it, our set-up is complete, now we explore various ways the communication between pods can be achieved, in fact this is one of the ways to communicate but it is very unreliable, we will see why.

1. Using Pod IPs directly

What we did till now was to communicate with web-server using it's internal IP directly. Whenever we create a pod in Kubernetes, it automatically assigns an internal IP to it. The IP will be picked up from `CIDR range` and will be assigned to the Pod. This IP will be available throughout the cluster and using this IP any pod can address our web-server. This is the simplest way to achieve communication, but it has some serious drawbacks.

1. The Pod IPs can change - In case the cluster got restarted, the Pod IPs can change sometimes, this might break your client or the requesting service.
2. You need to know the IP in-prior - Many K8s deployments are dynamic in nature, they are set-up and installed by CD tools, this makes it impossible to know the IP of the Pod in prior, because the Pod can get any IP when it is created.

2. Creating and using Services

Since Pods are non-permanent and dynamic in nature as discussed above, addressing them permanently becomes a problem. To mitigate issue Kubernetes came up with the concept of `Services`.

Service is an networking abstraction for a group of pods. In other words, a service maps a pod or a group of pods using a single name which never changes. Since the service assigns a constant name to a group of pods, we don't have to worry about Pod's IP anymore, this abstracts away the changing IP problem of pods. Secondly, since we create and assign service names, they can be used as a constant address for communication, K8s internal DNS takes care of mapping service name to Pod IPs.

In order to bring this into our set-up, we just have to create a `Service` resource for the web-server we created. Let's create the service definition with YAML. (`web-app-service.yaml`)

```
apiVersion: v1
kind: Service
metadata:
  name: web-app-service
spec:
  selector:
    app: webapp1
  ports:
    - protocol: TCP
      port: 80
```

The YAML file looks clean, `selector` is an important aspect to take care of. The selector is the one which tells where the service should map. We are targeting `webapp1` deployment by using `app` selector label. Let's deploy this service now.

```
kubectl create -f web-app-service.yaml
```

Now, let's see whether the service is created.

```
kubectl get svc
```

Output:

```
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S
kubernetes        ClusterIP   10.96.0.1       <none>        443/TC
web-app-service   ClusterIP   10.111.195.22   <none>        80/TCF
```

Yes, our service is running. The service got a `ClusterIP`, cluster IPs are static and are assigned only during the creation of service. Like Pod IPs, `ClusterIP` is available throughout the cluster for use, unlike Pod IP, cluster IP never changes, so now atleast we have a static destination for addressing permanently. But wait, it still didn't address another issue, how can we know this cluster IP prior, one way is to assign our own IP address (hardcoded), but it doesn't make any sense, it is the functionality of Kubernetes to assign IPs. Here are different ways we can mitigate this issue.

Using Environment variables

It can be tedious to know service cluster IP before or manually assign an IP address. But, Kubernetes has a solution for this problem. Whenever a Pod is created, kubernetes injects some environment variables into the pod's environment, these environment variables can be used by containers in the

pod to interact with the cluster. Fortunately, whenever you create a service,

the address of the service will be injected as an environment variable to all the Pods that run within the same namespace. If you exec into any of the pod and run `env` command, you will see all the variables that are exported by K8s.

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=client-job-bbwd6
KUBERNETES_SERVICE_HOST=10.96.0.1
KUBERNETES_SERVICE_PORT=443
WEB_APP_SERVICE_PORT_80_TCP_PORT=80
KUBERNETES_PORT_443_TCP_PORT=443
WEB_APP_SERVICE_PORT=tcp://10.111.195.22:80
WEB_APP_SERVICE_PORT_80_TCP_PROTO=tcp
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_PROTO=tcp
WEB_APP_SERVICE_SERVICE_HOST=10.111.195.22
WEB_APP_SERVICE_SERVICE_PORT=80
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
WEB_APP_SERVICE_PORT_80_TCP=tcp://10.111.195.22:80
WEB_APP_SERVICE_PORT_80_TCP_ADDR=10.111.195.22
HOME=/root
```

In the list we can see `WEB_APP_SERVICE_SERVICE_HOST` and `WEB_APP_SERVICE_SERVICE_PORT`, these are the host and port variables of the service `web-app-service` we created in one of the previous step. Any pod which runs in a namespace gets the `ClusterIP` and port details of all the services created within the same namespace. The kubernetes convention of these environment variables is as follows:

```
{{SERVICE_NAME}}_SERVICE_HOST   # For ClusterIP
{{SERVICE_NAME}}_SERVICE_PORT   # For port
```

All `-` in the service name are replaced by an underscore (`_`) , since Linux doesn't support `-` in variable names. Let's create a job to test this quickly:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: client-job
spec:
  template:
    spec:
      containers:
      - name: client
        image: byrnedo/alpine-curl
        command: ["/bin/sh", "-c", "curl -s http://${WEB_APP_SERVI
      restartPolicy: Never
  backoffLimit: 4
```

Instead of using Pod IPs or `ClusterIP` directly, we are using environment variables to dynamically infer the service IP and service port. Let's deploy this job. (`client-job-env.yaml`)

```
kubectl create client-job-env.yaml
```

The job should run without any errors if the service is mapped properly and we should see the response from web-server. Let's check. (`client-job-s7446` is the pod created by the job)

```
kubectl logs client-job-s7446
```

Output:

```
<h1>This request was processed by host: webapp1-6b54fb89d9-ct7fk</
```

Yes! The service is working properly as expected and we are able to address the service as desired.

Using Service names (Requires cluster DNS)

Another easier way is to use service name directly, if the port is already known. This is one of the simplest ways of addressing, but it requires cluster DNS to be set-up and working properly, most of the kubernetes deployment tools like `kubeadm` or `minikube` comes with `core-dns` installed. Also for core-dns to function correctly, you might require a CNI plugin like `flannel`, `cilium`, `weavenet` etc.

For example, if we create a service by name `web-app-service`, the URL `http://web-app-service` should be routed to the web-server pod properly. (on port `80` by default), any URL `http://web-app-service:xxxx` should be routed to the web-server pod at `xxxx` port. Kubernetes DNS takes care of name resolution. Let's redeploy the job by making this modification (`client-job-dns-1.yaml`)

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: client-job
spec:
  template:
    spec:
      containers:
      - name: client
        image: byrnedo/alpine-curl
        command: ["curl", "-s", "http://web-app-service"]
      restartPolicy: Never
  backoffLimit: 4
```

As you can see, we replaced the IP with name of service directly. This should work if cluster DNS is working properly. Let's check logs. (`client-job-mj5vr` is the pod created by the job)

```
kubectl logs client-job-mj5vr
```

Output:

```
<h1>This request was processed by host: webapp1-6b54fb89d9-ct7fk</
```

3. Communicating between services across namespaces

Till now all our deployments and jobs were in a single namespace. If the web-app and the client job are in different namespaces, we cannot communicate using environment variables, as Kubernetes doesn't inject variables from other namesapces. We cannot use just service names like `web-app-service` as they are valid only within the namespace. So, how do we communicate across namespaces? Let's see.

Using fully-qualified DNS names

Kubernetes has an answer for this problem as well. If we have cluster-aware DNS service like `CoreDNS` running, we can use fully qualified DNS names. starting from `cluster.local` Assume that our web-server is running in

namespace `test-namespace` and has a service `web-app-service` defined. We can address this using an URL shown below:

```
web-app-service.test-namespace.svc.cluster.local
```

Sounds tricky?? Here is the breakdown of the URL

1. `.cluster.local` : This is the root of our cluster DNS, every resource must be accessed from root.
2. `.svc` : This tells we are accessing a service resource.
3. `test-namespace` : This is the namespace where our `web-app-service` is defined.
4. `web-app-service`: This is our service name.

We can use URLs like `http://web-app-service.test-namespace.svc.cluster.local:[xxxx]` (`xxxx` is the Port, you can optionally ignore this if the service is mapping default http port `80`)

So the general format for addressing a service in another namespace is to use a fully qualified DNS name like the one shown above. It is always suitable to use URLs like this as they are universal and can be addressable anywhere throughout the cluster. Again here is the general format of the URL:

```
{{service_name}}.{{namespace}}.svc.cluster.local
```

So that's it! we have seen various possible ways to address and communicate between micorservices running on a Kubernetes cluster.

Thanks for spending your time reading this post. Please let me know your views and opinions in the comments section.

Discussion (3)

Subscribe

Add to the discussion

Peters Chikezie • Apr 16 '21

Hi Narasimha,
Thanks for this article.

i would like to know, can the service to service calls be done via HTTPS?
For example:
This

```
https://web-app-service

instead of

   http://web-app-service
```

1 like Reply

Narasimha Prasanna HN • Apr 30 '21

Yes, it should be possible, if the web-server serves on the default HTTPS port (443) and provides it's a certificate (For SSL/TLS) it can be self signed or purchased one. Even the port 443 is not mandatory, you can use `https://web-service-service:<custom-https-port>`. So the web-app-service must provide a full fledged HTTPS support, the client can then use https instead of http.

But most of the architectures don't implement HTTPs in the web-server itself, instead they use a reverse-proxy with HTTPS configured. It is called TLS termination. Only the reverse-proxy will be exposed to the public.

2 likes Reply

Peters Chikezie • May 12 '21

Thanks a lot for this.

Do you mind sharing any resource on this?

1 like Reply

Code of Conduct • Report abuse

Read next

Building an application with Go and SQLite
Jeremy Morgan - Apr 15

stop form spam
Geoffrey Callaghan - Apr 7

construct box plot of any data set using R.
maxwizard01 - Mar 29

How the TypeScript Readonly Type Works
Johnny Simpson - Apr 7