DigitalOcean

Subscribe Share Contents

How To Deploy a PHP Application with Kubernetes on Ubuntu 16.04

5

Posted January 18, 2019 8.8k KUBERNETES PHP NGINX BLOCK STORAGE UBUNTU 16.04

By: Amitabh Dhiwal

The author selected the Open Internet/Free Speech to receive a donation as part of the Write for DOnations program.

Introduction

Kubernetes is an open source container orchestration system. It allows you to create, update, and scale containers without worrying about downtime.

To run a PHP application, Nginx acts as a proxy to PHP-FPM. Containerizing this setup in a single container can be a cumbersome process, but Kubernetes will help manage both services in separate containers. Using Kubernetes will allow you to keep your containers reusable and swappable, and you will not have to rebuild your container image every time there's a new version of Nginx or PHP.

In this tutorial, you will deploy a PHP 7 application on a Kubernetes cluster with Nginx and PHP-FPM running in separate containers. You will also learn how to keep your configuration files and application code outside the container image using DigitalOcean's Block Storage system. This approach will allow you to reuse the Nginx image for any application that needs a web/proxy server by passing a configuration volume, rather than rebuilding the image.

Prerequisites

- A basic understanding of Kubernetes objects. Check out our Introduction to Kubernetes article for more information.
- A Kubernetes cluster running on Ubuntu 16.04. You can set this up by following the How To Create a Kubernetes 1.10 Cluster Using Kubeadm on Ubuntu 16.04 tutorial.
- A DigitalOcean account and an API access token with read and write permissions to create our storage volume. If you don’t have your API access token, you can create it from here.
- Your application code hosted on a publicly accessible URL, such as Github.

Step 1 — Creating the PHP-FPM and Nginx Services

In this step, you will create the PHP-FPM and Nginx services. A service allows access to a set of pods from within the cluster. Services within a cluster can communicate directly through their names, without the need for IP addresses. The PHP-FPM service will allow access to the PHP-FPM pods, while the Nginx service will allow access to the Nginx pods.

Since Nginx pods will proxy the PHP-FPM pods, you will need to tell the service how to find them. Instead of using IP addresses, you will take advantage of Kubernetes' automatic service discovery to use human-readable names to route requests to the appropriate service.

To create the service, you will create an object definition file. Every Kubernetes object definition is a YAML file that contains at least the following items:

- `apiVersion` : The version of the Kubernetes API that the definition belongs to.
- `kind` : The Kubernetes object this file represents. For example, a `pod` or `service` .
- `metadata` : This contains the `name` of the object along with any `labels` that you may wish to apply to it.
- `spec` : This contains a specific configuration depending on the kind of object you are creating, such as the container image or the ports on which the container will be accessible from.

First you will create a directory to hold your Kubernetes object definitions.

SSH to your **master node** and create the `definitions` directory that will hold your Kubernetes object definitions.

```
$ mkdir definitions
```

Navigate to the newly created `definitions` directory:

```
$ cd definitions
```

Make your PHP-FPM service by creating a `php_service.yaml` file:

```
$ nano php_service.yaml
```

Set `kind` as `Service` to specify that this object is a service:

php_service.yaml

```
...
apiVersion: v1
kind: Service
```

Name the service `php` since it will provide access to PHP-FPM:

php_service.yaml

```
...
metadata:
  name: php
```

You will logically group different objects with labels. In this tutorial, you will use labels to group the objects into "tiers", such as frontend or backend. The PHP pods will run behind this service, so you will label it as `tier: backend`.

php_service.yaml

```
...
  labels:
    tier: backend
```

A service determines which pods to access by using `selector` labels. A pod that matches these labels will be serviced, independent of whether the pod was created before or after the service. You will add labels for your pods later in the tutorial.

Use the `tier: backend` label to assign the pod into the backend tier. You will also add the `app: php` label to specify that this pod runs PHP. Add these two labels after the `metadata` section.

php_service.yaml

```
...
spec:
  selector:
    app: php
    tier: backend
```

Next, specify the port used to access this service. You will use port `9000` in this tutorial. Add it to the `php_service.yaml` file under `spec`:

php_service.yaml

```
...
  ports:
    - protocol: TCP
      port: 9000
```

Your completed `php_service.yaml` file will look like this:

php_service.yaml

```
apiVersion: v1
kind: Service
```

```
metadata:
  name: php
  labels:
    tier: backend
spec:
  selector:
    app: php
    tier: backend
  ports:
  - protocol: TCP
    port: 9000
```

Hit `CTRL + o` to save the file, and then `CTRL + x` to exit `nano`.

Now that you've created the object definition for your service, to run the service you will use the `kubectl apply` command along with the `-f` argument and specify your `php_service.yaml` file.

Create your service:

```
$ kubectl apply -f php_service.yaml
```

This output confirms the service creation:

```
Output
service/php created
```

Verify that your service is running:

```
$ kubectl get svc
```

You will see your PHP-FPM service running:

```
Output
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
kubernetes    ClusterIP   10.96.0.1       <none>        443/TCP    10m
php           ClusterIP   10.100.59.238   <none>        9000/TCP   5m
```

There are various service types that Kubernetes supports. Your `php` service uses the default service type, `ClusterIP`. This service type assigns an internal IP and makes the service reachable only from within the cluster.

Now that the PHP-FPM service is ready, you will create the Nginx service. Create and open a new file called `nginx_service.yaml` with the editor:

```
$ nano nginx_service.yaml
```

This service will target Nginx pods, so you will name it `nginx`. You will also add a `tier: backend` label as it belongs in the backend tier:

nginx_service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    tier: backend
```

Similar to the `php` service, target the pods with the selector labels `app: nginx` and `tier: backend`. Make this service accessible on port 80, the default HTTP port.

nginx_service.yaml

```
...
spec:
  selector:
    app: nginx
    tier: backend
  ports:
  - protocol: TCP
    port: 80
```

The Nginx service will be publicly accessible to the internet from your Droplet's public IP address. `your_public_ip` can be found from your DigitalOcean Cloud Panel. Under `spec.externalIPs`, add:

nginx_service.yaml

```
...
spec:
  externalIPs:
  - your_public_ip
```

Your `nginx_service.yaml` file will look like this:

nginx_service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    tier: backend
spec:
  selector:
    app: nginx
    tier: backend
  ports:
  - protocol: TCP
    port: 80
  externalIPs:
  - your_public_ip
```

Save and close the file. Create the Nginx service:

```
$ kubectl apply -f nginx_service.yaml
```

You will see the following output when the service is running:

```
Output
service/nginx created
```

You can view all running services by executing:

```
$ kubectl get svc
```

You will see both the PHP-FPM and Nginx services listed in the output:

```
Output
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP    13m
nginx        ClusterIP   10.102.160.47   your_public_ip 80/TCP     50s
php          ClusterIP   10.100.59.238   <none>        9000/TCP   8m
```

Please note, if you want to delete a service you can run:

```
$ kubectl delete svc/service_name
```

Now that you've created your PHP-FPM and Nginx services, you will need to specify where to store your application code and configuration files.

Step 2 — Installing the DigitalOcean Storage Plug-In

Kubernetes provides different storage plug-ins that can create the storage space for your environment. In this step, you will install the DigitalOcean storage plug-in to create block storage on DigitalOcean. Once the installation is complete, it will add a storage class named `do-block-storage` that you will use to create your block storage.

You will first configure a Kubernetes Secret object to store your DigitalOcean API token. Secret objects are used to share sensitive information, like SSH keys and passwords, with other Kubernetes objects within the same namespace. Namespaces provide a way to logically separate your Kubernetes objects.

Open a file named `secret.yaml` with the editor:

```
$ nano secret.yaml
```

You will name your Secret object `digitalocean` and add it to the `kube-system namespace`. The `kube-system` namespace is the default namespace for Kubernetes' internal services and is also used by the DigitalOcean storage plug-in to launch various components.

secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: digitalocean
  namespace: kube-system
```

Instead of a `spec` key, a Secret uses a `data` or `stringData` key to hold the required information. The `data` parameter holds base64 encoded data that is automatically decoded when retrieved. The `stringData` parameter holds non-encoded data that is automatically encoded during creation or updates, and does not output the data when retrieving Secrets. You will use `stringData` in this tutorial for convenience.

Add the `access-token as stringData`:

secret.yaml

```
...
stringData:
  access-token: your-api-token
```

Save and exit the file.

Your `secret.yaml` file will look like this:

secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: digitalocean
  namespace: kube-system
stringData:
  access-token: your-api-token
```

Create the secret:

```
$ kubectl apply -f secret.yaml
```

You will see this output upon Secret creation:

```
Output
secret/digitalocean created
```

You can view the secret with the following command:

```
$ kubectl -n kube-system get secret digitalocean
```

The output will look similar to this:

```
Output
NAME           TYPE      DATA      AGE
digitalocean   Opaque    1         41s
```

The `Opaque` type means that this Secret is read-only, which is standard for `stringData` Secrets. You can read more about it on the Secret design spec. The `DATA` field shows the number of items stored in this Secret. In this case, it shows `1` because you have a single key stored.

Now that your Secret is in place, install the DigitalOcean block storage plug-in:

```
$ kubectl apply -f https://raw.githubusercontent.com/digitalocean/csi-digitalocean/master/deploy/kube
```

You will see output similar to the following:

```
Output
storageclass.storage.k8s.io/do-block-storage created
serviceaccount/csi-attacher created
clusterrole.rbac.authorization.k8s.io/external-attacher-runner created
clusterrolebinding.rbac.authorization.k8s.io/csi-attacher-role created
service/csi-attacher-doplug-in created
statefulset.apps/csi-attacher-doplug-in created
serviceaccount/csi-provisioner created
clusterrole.rbac.authorization.k8s.io/external-provisioner-runner created
clusterrolebinding.rbac.authorization.k8s.io/csi-provisioner-role created
service/csi-provisioner-doplug-in created
statefulset.apps/csi-provisioner-doplug-in created
serviceaccount/csi-doplug-in created
clusterrole.rbac.authorization.k8s.io/csi-doplug-in created
clusterrole.rbac.authorization.k8s.io/csi-doplug-in created
clusterrolebinding.rbac.authorization.k8s.io/csi-doplug-in created
daemonset.apps/csi-doplug-in created
```

Now that you have installed the DigitalOcean storage plug-in, you can create block storage to hold your application code and configuration files.

Step 3 — Creating the Persistent Volume

With your Secret in place and the block storage plug-in installed, you are now ready to create your *Persistent Volume*. A Persistent Volume, or PV, is block storage of a specified size that lives independently of a pod's life cycle. Using a Persistent Volume will allow you to manage or update your pods without worrying about losing your application code. A Persistent Volume is accessed by using a `PersistentVolumeClaim`, or PVC, which mounts the PV at the required path.

Open a file named `code_volume.yaml` with your editor:

```
$ nano code_volume.yaml
```

Name the PVC `code` by adding the following parameters and values to your file:

code_volume.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: code
```

The `spec` for a PVC contains the following items:

- `accessModes` which vary by the use case. These are:
  - `ReadWriteOnce` – mounts the volume as read-write by a single node
  - `ReadOnlyMany` – mounts the volume as read-only by many nodes
  - `ReadWriteMany` – mounts the volume as read-write by many nodes
- `resources` – the storage space that you require

DigitalOcean block storage is only mounted to a single node, so you will set the `accessModes` to `ReadWriteOnce`. This tutorial will guide you through adding a small amount of application code, so 1GB will be plenty in this use case. If you plan on storing a larger amount of code or data on the volume, you can modify the `storage` parameter to fit your requirements. You can increase the amount of storage after volume creation, but shrinking the disk is not supported.

code_volume.yaml

```
...
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Next, specify the storage class that Kubernetes will use to provision the volumes. You will use the `do-block-storage` class created by the DigitalOcean block storage plug-in.

code_volume.yaml

```
...
  storageClassName: do-block-storage
```

Your `code_volume.yaml` file will look like this:

code_volume.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: code
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: do-block-storage
```

Save and exit the file.

Create the `code` PersistentVolumeClaim using `kubectl`:

```
$ kubectl apply -f code_volume.yaml
```

The following output tells you that the object was successfully created, and you are ready to mount your 1GB PVC as a volume.

```
Output
persistentvolumeclaim/code created
```

To view available Persistent Volumes (PV):

```
$ kubectl get pv
```

You will see your PV listed:

```
Output
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS    CLAIM
pvc-ca4df10f-ab8c-11e8-b89d-12331aa95b13   1Gi        RWO            Delete           Bound     defau
```

The fields above are an overview of your configuration file, except for `Reclaim Policy` and `Status`. The `Reclaim Policy` defines what is done with the PV after the PVC accessing it is deleted. `Delete` removes the PV from Kubernetes as well as the DigitalOcean infrastructure. You can learn more about the `Reclaim Policy` and `Status` from the Kubernetes PV documentation.

You've successfully created a Persistent Volume using the DigitalOcean block storage plug-in. Now that your Persistent Volume is ready, you will create your pods using a Deployment.

Step 4 — Creating a PHP-FPM Deployment

In this step, you will learn how to use a Deployment to create your PHP-FPM pod. Deployments provide a uniform way to create, update, and manage pods by using ReplicaSets. If an update does not work as expected, a Deployment will automatically rollback its pods to a previous image.

The Deployment `spec.selector` key will list the labels of the pods it will manage. It will also use the `template` key to create the required pods.

This step will also introduce the use of Init Containers. *Init Containers* run one or more commands before the regular containers specified under the pod's `template` key. In this tutorial, your Init Container will fetch a sample `index.php` file from GitHub Gist using `wget`. These are the contents of the sample file:

index.php

```
<?php
echo phpinfo();
```

To create your Deployment, open a new file called `php_deployment.yaml` with your editor:

```
$ nano php_deployment.yaml
```

This Deployment will manage your PHP-FPM pods, so you will name the Deployment object `php`. The pods belong to the backend tier, so you will group the Deployment into this group by using the `tier: backend` label:

php_deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php
  labels:
    tier: backend
```

For the Deployment `spec`, you will specify how many copies of this pod to create by using the `replicas` parameter. The number of `replicas` will vary depending on your needs and available resources. You will create one replica in this tutorial:

php_deployment.yaml

```
...
spec:
  replicas: 1
```

This Deployment will manage pods that match the `app: php` and `tier: backend` labels. Under `selector` key add:

php_deployment.yaml

```
...
  selector:
    matchLabels:
      app: php
      tier: backend
```

Next, the Deployment `spec` requires the `template` for your pod's object definition. This template will define specifications to create the pod from. First, you will add the labels that were specified for the `php` service `selectors` and the Deployment's `matchLabels`. Add `app: php` and `tier: backend` under `template.metadata.labels`:

php_deployment.yaml

```
...
  template:
    metadata:
      labels:
        app: php
        tier: backend
```

A pod can have multiple containers and volumes, but each will need a name. You can selectively mount volumes to a container by specifying a mount path for each volume.

First, specify the volumes that your containers will access. You created a PVC named `code` to hold your application code, so name this volume `code` as well. Under `spec.template.spec.volumes`, add the following:

php_deployment.yaml

```
...
    spec:
      volumes:
      - name: code
        persistentVolumeClaim:
          claimName: code
```

Next, specify the container you want to run in this pod. You can find various images on the Docker store, but in this tutorial you will use the `php:7-fpm` image.

Under `spec.template.spec.containers`, add the following:

php_deployment.yaml

```
...
      containers:
      - name: php
        image: php:7-fpm
```

Next, you will mount the volumes that the container requires access to. This container will run your PHP code, so it will need access to the `code` volume. You will also use `mountPath` to specify `/code` as the mount point.

Under `spec.template.spec.containers.volumeMounts`, add:

php_deployment.yaml

```
...
        volumeMounts:
        - name: code
          mountPath: /code
```

Now that you have mounted your volume, you need to get your application code on the volume. You may have previously used FTP/SFTP or cloned the code over an SSH connection to accomplish this, but this step will show you how to copy the code using an Init Container.

Depending on the complexity of your setup process, you can either use a single `initContainer` to run a script that builds your application, or you can use one `initContainer` per command. Make sure that the volumes are mounted to the `initContainer`.

In this tutorial, you will use a single Init Container with `busybox` to download the code. `busybox` is a small image that contains the `wget` utility that you will use to accomplish this.

Under `spec.template.spec`, add your `initContainer` and specify the `busybox` image:

php_deployment.yaml

```
...
      initContainers:
      - name: install
        image: busybox
```

Your Init Container will need access to the `code` volume so that it can download the code in that location. Under `spec.template.spec.initContainers`, mount the volume `code` at the `/code` path:

php_deployment.yaml

```
...
        volumeMounts:
        - name: code
          mountPath: /code
```

Each Init Container needs to run a `command`. Your Init Container will use `wget` to download the code from Github into the `/code` working directory. The `-O` option gives the downloaded file a name, and you will name this file `index.php`.

> **Note:** Be sure to trust the code you're pulling. Before pulling it to your server, inspect the source code to ensure you are comfortable with what the code does.

Under the `install` container in `spec.template.spec.initContainers`, add these lines:

php_deployment.yaml

```
...
        command:
        - wget
        - "-O"
        - "/code/index.php"
        - https://raw.githubusercontent.com/do-community/php-kubernetes/master/index.php
```

Your completed `php_deployment.yaml` file will look like this:

php_deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php
  labels:
    tier: backend
spec:
  replicas: 1
  selector:
    matchLabels:
```

```
    matchLabels:
      app: php
      tier: backend
  template:
    metadata:
      labels:
        app: php
        tier: backend
    spec:
      volumes:
      - name: code
        persistentVolumeClaim:
          claimName: code
      containers:
      - name: php
        image: php:7-fpm
        volumeMounts:
        - name: code
          mountPath: /code
      initContainers:
      - name: install
        image: busybox
        volumeMounts:
        - name: code
          mountPath: /code
        command:
        - wget
        - "-O"
        - "/code/index.php"
        - https://raw.githubusercontent.com/do-community/php-kubernetes/master/index.php
```

Save the file and exit the editor.

Create the PHP-FPM Deployment with `kubectl`:

```
$  kubectl apply -f php_deployment.yaml
```

You will see the following output upon Deployment creation:

```
Output
deployment.apps/php created
```

To summarize, this Deployment will start by downloading the specified images. It will then request the `PersistentVolume` from your `PersistentVolumeClaim` and serially run your `initContainers`. Once complete, the containers will run and mount the `volumes` to the specified mount point. Once all of these steps are complete, your pod will be up and running.

You can view your Deployment by running:

```
$ kubectl get deployments
```

You will see the output:

```
Output
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
php       1         1         1            0           19s
```

This output can help you understand the current state of the Deployment. A `Deployment` is one of the controllers that maintains a desired state. The `template` you created specifies that the `DESIRED` state will have 1 `replicas` of the pod named `php`. The `CURRENT` field indicates how many replicas are running, so this should match the `DESIRED` state. You can read about the remaining fields in the Kubernetes Deployments documentation.

You can view the pods that this Deployment started with the following command:

```
$ kubectl get pods
```

The output of this command varies depending on how much time has passed since creating the Deployment. If you run it shortly after creation, the output will likely look like this:

```
Output
NAME                   READY     STATUS     RESTARTS   AGE
php-86d59fd666-bf8zd   0/1       Init:0/1   0          9s
```

The columns represent the following information:

- `Ready`: The number of `replicas` running this pod.
- `Status`: The status of the pod. `Init` indicates that the Init Containers are running. In this output, 0 out of 1 Init Containers have finished running.
- `Restarts`: How many times this process has restarted to start the pod. This number will increase if any of your Init Containers fail. The Deployment will restart it until it reaches a desired state.

Depending on the complexity of your startup scripts, it can take a couple of minutes for the status to change to `podInitializing`:

```
Output
NAME                   READY     STATUS            RESTARTS   AGE
php-86d59fd666-lkwgn   0/1       podInitializing   0          39s
```

This means the Init Containers have finished and the containers are initializing. If you run the command when all of the containers are running, you will see the pod status change to `Running`.

```
Output
NAME                     READY     STATUS           RESTARTS    AGE
php-86d59fd666-lkwgn    1/1       Running    0           1m
```

You now see that your pod is running successfully. If your pod doesn't start, you can debug with the following commands:

- View detailed information of a pod:

```
$ kubectl describe pods pod-name
```

- View logs generated by a pod:

```
$ kubectl logs pod-name
```

- View logs for a specific container in a pod:

```
$ kubectl logs pod-name container-name
```

Your application code is mounted and the PHP-FPM service is now ready to handle connections. You can now create your Nginx Deployment.

Step 5 — Creating the Nginx Deployment

In this step, you will use a *ConfigMap* to configure Nginx. A ConfigMap holds your configuration in a key-value format that you can reference in other Kubernetes object definitions. This approach will grant you the flexibility to reuse or swap the image with a different Nginx version if needed. Updating the ConfigMap will automatically replicate the changes to any pod mounting it.

Create a `nginx_configMap.yaml` file for your ConfigMap with your editor:

```
$ nano nginx_configMap.yaml
```

Name the ConfigMap `nginx-config` and group it into the `tier: backend` micro-service:

nginx_configMap.yaml

```
apiVersion: v1
kind: ConfigMap
```

```
metadata:
  name: nginx-config
  labels:
    tier: backend
```

Next, you will add the `data` for the ConfigMap. Name the key `config` and add the contents of your Nginx configuration file as the value. You can use the example Nginx configuration from this tutorial.

Because Kubernetes can route requests to the appropriate host for a service, you can enter the name of your PHP-FPM service in the `fastcgi_pass` parameter instead of its IP address. Add the following to your `nginx_configMap.yaml` file:

nginx_configMap.yaml

```
...
data:
  config : |
    server {
      index index.php index.html;
      error_log  /var/log/nginx/error.log;
      access_log /var/log/nginx/access.log;
      root ^/code^;

      location / {
          try_files $uri $uri/ /index.php?$query_string;
      }

      location ~ \.php$ {
          try_files $uri =404;
          fastcgi_split_path_info ^(.+\.php)(/.+)$;
          fastcgi_pass php:9000;
          fastcgi_index index.php;
          include fastcgi_params;
          fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
          fastcgi_param PATH_INFO $fastcgi_path_info;
        }
    }
```

Your `nginx_configMap.yaml` file will look like this:

nginx_configMap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
  labels:
    tier: backend
data:
  config : |
    server {
      index index.php index.html;
      error_log  /var/log/nginx/error.log;
      access_log /var/log/nginx/access.log;
      root /code;

      location / {
          try_files $uri $uri/ /index.php?$query_string;
      }

      location ~ \.php$ {
          try_files $uri =404;
          fastcgi_split_path_info ^(.+\.php)(/.+)$;
          fastcgi_pass php:9000;
          fastcgi_index index.php;
          include fastcgi_params;
          fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
          fastcgi_param PATH_INFO $fastcgi_path_info;
        }
    }
```

Save the file and exit the editor.

Create the ConfigMap:

```
$ kubectl apply -f nginx_configMap.yaml
```

You will see the following output:

```
Output
configmap/nginx-config created
```

You've finished creating your ConfigMap and can now build your Nginx Deployment.

Start by opening a new `nginx_deployment.yaml` file in the editor:

```
$ nano nginx_deployment.yaml
```

Name the Deployment `nginx` and add the label `tier: backend`:

nginx_deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    tier: backend
```

Specify that you want one `replicas` in the Deployment `spec`. This Deployment will manage pods with labels `app: nginx` and `tier: backend`. Add the following parameters and values:

nginx_deployment.yaml

```
...
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
      tier: backend
```

Next, add the pod `template`. You need to use the same labels that you added for the Deployment `selector.matchLabels`. Add the following:

nginx_deployment.yaml

```
...
  template:
    metadata:
      labels:
        app: nginx
        tier: backend
```

Give Nginx access to the `code` PVC that you created earlier. Under `spec.template.spec.volumes`, add:

nginx_deployment.yaml

```
...
    spec:
      volumes:
      - name: code
        persistentVolumeClaim:
          claimName: code
```

Pods can mount a ConfigMap as a volume. Specifying a file name and key will create a file with its value as the content. To use the ConfigMap, set `path` to name of the file that will hold the contents of the `key`. You want to create a file `site.conf` from the key `config`. Under `spec.template.spec.volumes`, add the following:

nginx_deployment.yaml

```
...
      - name: config
        configMap:
          name: nginx-config
          items:
          - key: config
            path: site.conf
```

> **Warning**: If a file is not specified, the contents of the key will replace the mountPath of the volume. This means that if a path is not explicitly specified, you will lose all content in the destination folder.

Next, you will specify the image to create your pod from. This tutorial will use the `nginx:1.7.9` image for stability, but you can find other Nginx images on the Docker store. Also, make Nginx available on the port 80. Under `spec.template.spec` add:

nginx_deployment.yaml

```
...
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Nginx and PHP-FPM need to access the file at the same path, so mount the `code` volume at `/code`:

nginx_deployment.yaml

```
...
        volumeMounts:
        - name: code
          mountPath: /code
```

The `nginx:1.7.9` image will automatically load any configuration files under the `/etc/nginx/conf.d` directory. Mounting the `config` volume in this directory will create the file `/etc/nginx/conf.d/site.conf`. Under `volumeMounts` add the following:

nginx_deployment.yaml

```
...
        - name: config
          mountPath: /etc/nginx/conf.d
```

Your `nginx_deployment.yaml` file will look like this:

nginx_deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    tier: backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
      tier: backend
  template:
    metadata:
      labels:
        app: nginx
        tier: backend
    spec:
      volumes:
      - name: code
        persistentVolumeClaim:
          claimName: code
      - name: config
        configMap:
          name: nginx-config
          items:
          - key: config
            path: site.conf
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
        volumeMounts:
        - name: code
          mountPath: /code
        - name: config
          mountPath: /etc/nginx/conf.d
```

Save the file and exit the editor.

Create the Nginx Deployment:

```
$ kubectl apply -f nginx_deployment.yaml
```

The following output indicates that your Deployment is now created:

```
Output
deployment.apps/nginx created
```

List your Deployments with this command:

```
$ kubectl get deployments
```

You will see the Nginx and PHP-FPM Deployments:

```
Output
NAME       DESIRED    CURRENT    UP-TO-DATE   AVAILABLE    AGE
nginx      1          1          1            0            16s
php        1          1          1            1            7m
```

List the pods managed by both of the Deployments:

```
$ kubectl get pods
```

You will see the pods that are running:

```
Output
NAME                     READY      STATUS     RESTARTS    AGE
nginx-7bf5476b6f-zppml   1/1        Running    0           32s
php-86d59fd666-lkwgn     1/1        Running    0           7m
```

Now that all of the Kubernetes objects are active, you can visit the Nginx service on your browser.

List the running services:

```
$ kubectl get services -o wide
```

Get the External IP for your Nginx service:

```
Output
NAME         TYPE         CLUSTER-IP      EXTERNAL-IP    PORT(S)    AGE       SELECTOR
kubernetes   ClusterIP    10.96.0.1       <none>         443/TCP    39m       <none>
nginx        ClusterIP    10.102.160.47   your_public_ip 80/TCP      27m       app=nginx,tier=backend
php          ClusterIP    10.100.59.238   <none>         9000/TCP   34m       app=php,tier=backend
```

On your browser, visit your server by typing in `http://your_public_ip`. You will see the output of `php_info()` and have confirmed that your Kubernetes services are up and running.

Conclusion

In this guide, you containerized the PHP-FPM and Nginx services so that you can manage them independently. This approach will not only improve the scalability of your project as you grow, but will also allow you to efficiently use resources as well. You also stored your application code on a volume so that you can easily update your services in the future.

By: Amitabh Dhiwal

♡ Upvote (5) Subscribe Share

Editor:
Haley Mills

Related Tutorials

An Introduction to Service Meshes

How To Ensure Code Quality with SonarQube on Ubuntu 18.04

How To Set Up a Private Docker Registry on Ubuntu 18.04

How To Set Up Multi-Node Deployments With Rancher 2.1, Kubernetes, and Docker Machine on Ubu...

How to Set Up an Nginx Ingress with Cert-Manager on DigitalOcean Kubernetes

0 Comments

Leave a comment...

Log In to Comment