Docker Swarm vs Kubernetes: A Practical Comparison

Link: https://betterstack.com/community/guides/scaling-docker/docker-swarm-kubernetes/

DockerDocker SwarmKubernetes

Donald Le

Updated on December 22, 2023

Contents

In the world of container orchestration, two prominent platforms have emerged as leaders: Docker Swarm and Kubernete. As organizations increasingly adopt containerization to achieve scalability, resilience, and efficient resource utilization, choosing between these two solutions becomes crucial.

This article will compare both platforms, exploring their features, strengths, and trade-offs. By understanding their differences and use cases, you will gain valuable insights to select the ideal container orchestration tool for your specific needs.

We will examine various aspects, including installation and setup, container compatibility, scalability, high availability, networking, ease of use, ecosystem, and security. Through practical hands-on example, you will get a taste of how each platform tackles common challenges in deploying and managing containerized applications.

Whether you are new to container orchestration or seeking to transition from one platform to another, this article aims to provide comprehensive knowledge to make an educated choice. By the end, you be well-equipped to decide which option aligns best with your requirements and goals.

Without any further ado, let's learn what Docker Swarm is first.

What is Docker Swarm?

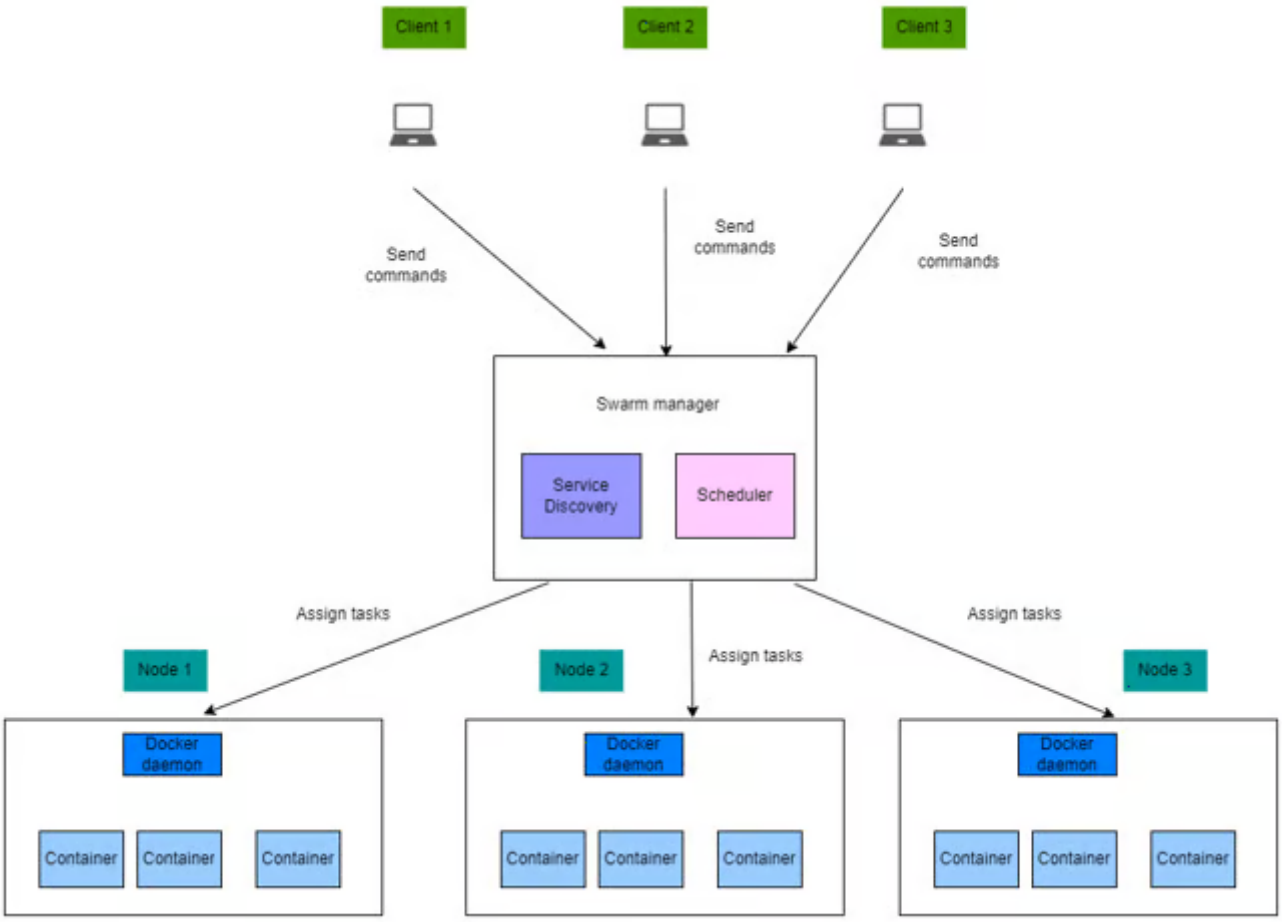
Docker Swarm is a container orchestration tool for managing and deploying applications using a containerization approach. It allows you to create a cluster of Docker nodes (known as a Swarm) and deploy services to the cluster. Docker Swarm is lightweight and straightforward, allowing you to containerize and deploy your software project in minutes.

To manage and deploy applications efficiently, Docker Swarm offers the following capabilities:

- **Cluster Management**: You can create a cluster of nodes to manage the application containers. A node can be a manager or a worker (or both).
- **Service Abstraction**: Docker Swarm introduces the "Service" component, which lets you set the network configurations, resource limits, and number of container replicas. By managing the services, Docker Swarm ensures the application's desired state is maintained,
- **Load Balancing**: Swarm offers built-in load balancing, which allows services inside the cluster to interact without the need for you to define the load balancer file manually.
- **Rolling Back**: Rolling back to a specific service to its previous version is fully supported in case of a failed deployment.
- **Fault Tolerance**: Docker Swarm automatically checks for failures in nodes and containers to generate new ones to allow users to keep using the application without noticing any problems.
- **Scalability**: With Docker Swarm, you have the flexibility to adjust the number of replicas for your containers easily. This allows you to scale your application up or down based on the changing demands of your workload.

Docker Swarm architecture

Below is the diagram representing Docker Swarm Architecture:



In summary, the Swarm cluster consists of several vital components that work together to manage and deploy containers efficiently:

1. Manager nodes are responsible for receiving client commands and assigning tasks to worker nodes. Inside each manager node, we have the Service Discovery and Scheduler component. The former is responsible for collecting information on the worker nodes, while the latter finds suitable worker nodes to assign tasks to.
2. Worker nodes are solely responsible for executing the tasks assigned by a manager node.

In the above diagram, the deployment steps typically work like this:

- The client machines send commands to a manager node, let's say to deploy a new PostgreSQL database to the Swarm cluster.
- The Swarm manager finds the appropriate worker node to assign tasks for creating the new deployment.
- Finally, the worker node will create new containers for the PostgreSQL database and manage the containers' states.

Now that you've understood what Docker Swarm is and how it works let's learn about Kubernetes.

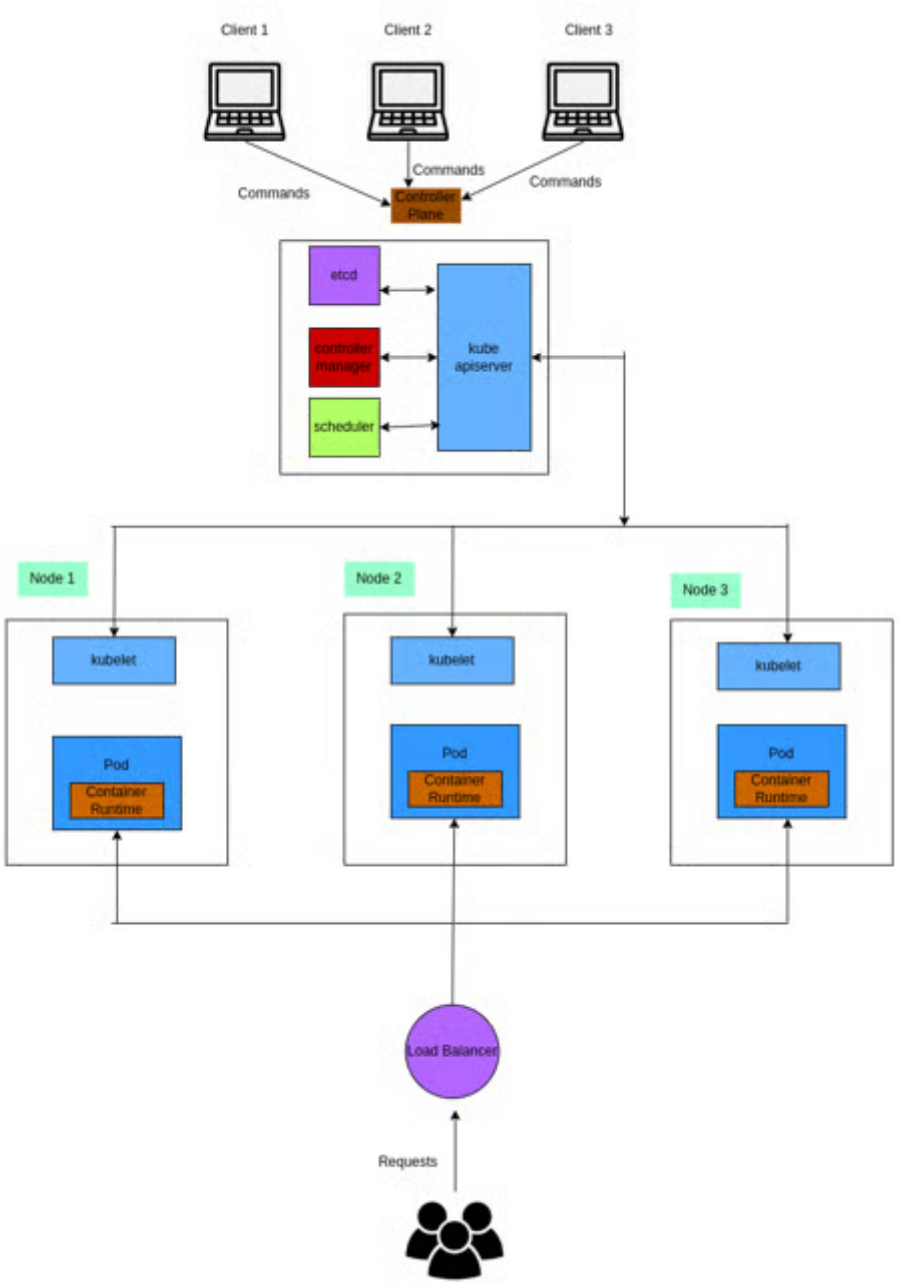
What is Kubernetes?

Kubernetes is a container orchestration tool for managing and deploying containerized applications. It supports a wide variety of container runtimes, including Docker, Containerd, Mirantis, and others, but the most commonly used runtime is Docker. Kubernetes is feature-rich and highly configurable, which makes it ideal for deploying complex applications, but it requires a steep learning curve.

Some of the key features of Kubernetes are listed below:

- **Container Orchestration with Pods**: A pod is the smallest and simplest deployment unit. Kubernetes allows you to deploy applications by creating and managing one or more containers in pods.
- **Service Discovery**: Kubernetes allows containers to interact with each other easily within the same cluster.
- **Load Balancing**: Kubernetes' load balancer allows access to a group of pods via an external network. Clients from outside the Kubernetes cluster can access the pods running inside the Kubernetes cluster via the balancer's external IP. If any of the pods in the group goes down, client requests will automatically be forwarded to other pods, allowing the deployed applications to be highly available.
- **Automatic Scaling**: You can scale-in or scale-out your application by adjusting number of deployed containers based on resource utilization or custom-defined metrics.
- **Self-Healing**: If one of your pods goes down, Kubernetes will try to recreate that pod automatically so that normal operation is restored.
- **Persistent Storage**: It provides support for various storage options, allowing you to persist data beyond the lifecycle of individual pods.
- **Configuration and Secrets Management**: Kubernetes allows you to store and manage configuration data and secrets securely, decoupling sensitive information from the application code.
- **Rolling back**: If there's a problem with your deployment, you can easily transition to a previous version of the application.
- **Resource Management**: With Kubernetes, you can flexibly set the resource constraints when deploying your application.
- **Extensibility**: Kubernetes has a vast ecosystem and can be extended with numerous plugins and tools for added functionality. You can also interact with Kubernetes components using Kubernetes APIs to deploy your application programmatically.

Kubernetes architecture



The above diagram illustrates the Kubernetes architecture. It contains the following components:

- **Control Plane**: This is a set of components responsible for managing the operation of a Kubernetes cluster. The main components of the Kubernetes control plane are:
  - Kube API Server: Provides an interface to interact with the Kubernetes cluster. It exposes the Kubernetes API so that clients from outside the cluster can communicate and control it.
  - Etcd: It is a distributed key-value store for storing configuration data and the desired state of the cluster.
  - Scheduler: It decides where to place newly created pods within the cluster based on resource requirements, node availability, and other defined rules. It also continuously monitors the cluster's resources to balance the workload across nodes.
  - Kube Controller Manager: It is responsible for managing and maintaining the desired state of various Kubernetes components such as pods, services, or volumes. It

continuously checks the state of Kubernetes components, whether they match expectations or not. If they don't, the controller-manager will take action to restore order to the problematic components.

- **Kubernetes node**: The Kubernetes node (which could be a physical or virtual machine) is responsible for executing the tasks assigned by the control plane. Here are the main components and concepts related to Kubernetes' nodes:
  - Kubelet: The Kubelet is an agent that runs on each node and manages the containers and their lifecycle. It communicates with the control plane, receives instructions about which containers to run, and ensures that the containers are in the desired state. The Kubelet also monitors the health of the containers and reports their status back to the control plane.
  - Pod: Kubernetes pod is the smallest component of Kubernetes cluster. It represents a single instance of a running process in the cluster. Inside a pod you have the container runtime, which is responsible for pulling container images from a registry, creating container instances, and managing their execution.

In the Kubernetes architecture diagram above, the execution steps work like this:

- The client from outside the Kubernetes cluster will interact with the cluster by executing `kubectl` command to the Kubernetes controller plane.
- The controller plane will assign deployment tasks to the Kubernetes Node.
- The Kubernetes node will create new pods with container runtime for the container execution.
- To allow users outside the cluster to interact with the deployed app, the client sends requests to the Kubernetes control plane to create a load balancer.
- Finally, users can interact with the deployed application via the load balancer IP Address.

Now that you understand what Kubernetes is, let's continue to the next section to learn the differences between Docker Swarm and Kubernetes.

Comparing Docker Swarm and Kubernetes

Although Docker Swarm and Kubernetes offer the capabilities to orchestrate containers, they are fundamentally different and serve different use cases.

Criteria	Docker Swarm	Kubernetes
Installation and setup	Easy to set up using the `docker` command	Complicated to manually set up the Kubernetes cluster
Types of containers they support	Only works with Docker containers	Supports Containerd, Docker, CRI-O, and others

Criteria	Docker Swarm	Kubernetes
High Availability	Provides basic configuration to support high availability	Offers feature-rich support for high availability
Popularity	Popular	Popular
Networking	Support basic networking features	Advanced features support for networking
GUI support	Yes	Yes
Learning curve	Easy to get started	Requires a steeper learning curve
Complexity	Simple and lightweight	Complicated and offer a lot of features
Load balancing	Automatic load-balancing	Manual load-balancing
CLIs	Do not need to install other CLI	Need to install other CLIs such as `kubectl`
Scalability	Does not support automatic scaling	Supports automatic scaling
Security	Only supports TLS	Supports RBAC, SSL/TLS, and secret management

Setup and installation

Regarding installation and setup, Docker Swarm is easier to set up than Kubernetes. Assuming Docker is already installed, you only need to run `docker swarm init` to create the cluster, then attach a node to the cluster using `docker swarm join`.

The steps for setting up a Kubernetes cluster are not as straightforward without using cloud platforms. However, there are some tools ease the process such as Kubeadm, or Kubespray. For example, to create a new cluster using Kubeadm you need to:

- Set up the Docker runtime on all the nodes.
- Install Kubeadm on all the nodes.
- Initiate Kubernetes Control Plane on the master node.
- Install the network plugin.
- Join the worker node to the master node.
- Using `kubectl get node` to check whether all the nodes are added to the Kubernetes cluster.

Types of containers they support

Docker Swarm only supports Docker containers, but Kubernetes supports any container runtime that implements its Container Runtime Interface, including Docker, Containerd, CRI-O, Mirantis, and

others, giving you many options to choose the one that best fits your use cases.

High availability

Docker Swarm provides built-in high availability for services. It automatically replicates services across multiple nodes in the Swarm cluster, ensuring that containers are distributed and balanced for fault tolerance.

On the other hand, Kubernetes provides a comprehensive set of features to achieve high availability, such as advanced scheduling to define pod placement based on node availability or resource utilization. Factors such as node health, resource constraints, and workload demands are also considered when distributing pods across the cluster.

Popularity

Both Docker Swarm and Kubernetes are popular in the industry and have been battle-tested for a wide variety of use cases. Docker Swarm is often used where simplicity and fast deployment are the primary considerations, while Kubernetes is well-suited for complex applications that demand advanced features and capabilities. It excels in scenarios where high availability, scalability, and flexibility are crucial requirements.

Networking

Both Docker Swarm and Kubernetes offer networking support. Docker Swarm provides simpler built-in overlay networking capabilities, while Kubernetes supports a more extensive and flexible networking model through its wide range of plugins and advanced networking features.

GUI support

Both Docker Swarm and Kubernetes have GUI tools to help you easily interact with them. With Docker Swarm, you have Swarmpit or Shipyard. For Kubernetes, you have Kubernetes Lens, Kube-dashboard, or Octant.

Learning curve

Docker Swarm is easy to learn and start with since it is lightweight and provides a simple approach to managing Docker containers. Kubernetes, however, has a much steeper learning curve since it offers more features and flexibility for deploying and managing containers.

Complexity

Docker Swarm is designed to be lightweight and easy to use while Kubernetes is designed to have a lot of features and support to deploy complex applications. Thus, Kubernetes is much more complicated to operate compared to Docker Swarm.

Load Balancing

Docker Swarm offers an automatic load balancing mechanism, ensuring seamless communication and interaction between containers within the Docker Swarm cluster. The load balancing functionality is built-in, requiring minimal configuration.

In contrast, Kubernetes provides a more customizable approach to load balancing. It allows you to define and configure load balancers based on your specific requirements. While this requires some manual setup, it grants you greater control over the load balancing configuration within the Kubernetes cluster.

Command-line tools

When working with Docker Swarm, there is no need to install an additional command line tool specifically for cluster management. The standard `docker` command is sufficient to create and interact with the Swarm cluster.

However, Kubernetes requires the installation of the `kubectl` command line tool to work with the cluster. `kubectl` is the primary way to interact with a Kubernetes cluster and it provides extensive functionality for managing deployments, services, pods, and other resources within the Kubernetes environment.

Scalability

Kubernetes allows you to automatically scale in or scale out the containers based on resource utilization or other factors. Docker Swarm, on the other hand, does not support auto-scaling ability.

Security

Docker Swarm and Kubernetes were built with security in mind, and they both provide several features and mechanisms to deploy applications safely. However, Kubernetes supports more authentication and authorization mechanisms such as Role-Based Access Control (RBAC), secure access layers (SSL), TLS protocol, and secret management.

Now that you have a clearer understanding of the distinctions between Docker Swarm and Kubernetes, we will now focus on deploying a sample Go application to both platforms. This hands-on experience will provide you with practical insights into the deployment process, showcasing the unique features and considerations of each platform.

Prerequisites

Before following through with the rest of this tutorial, you need to meet the following requirements:

1. A ready-to-use Linux machine, such as an Ubuntu version 22.04 server.
2. A recent version of Go installed on your machine.
3. A ready-to-use Git command line for cloning the sample application from its GitHub repo.
4. A recent version of Docker installed and accessible without using `sudo`.
5. A free DockerHub account for storing and sharing Docker images.
6. An already installed PostgreSQL instance.

Setting up the demo application

To fully understand how Docker Swarm and Kubernetes work and how they differ, you will deploy a demo application using both tools. This application is a blog service that allows users to create, update, retrieve, and delete blog posts. It is built using the Go programming language, and its data is stored in PostgreSQL.

However, you don't need to be familiar with Go or PostgreSQL to follow through. If you prefer, you can use an application stack that you're more familiar with and just follow the deployment steps to practice using Docker Swarm and Kubernetes.

To test out the application's functionality, let's try to run it directly on the local machine before deploying it through Docker Swarm and Kubernetes.

1. Cloning the GitHub repository

In this step, you will clone the this GitHub repository to your machine using the commands below:

```
git clone https://github.com/betterstack-community/docker-swarm-kubernetes
```

```
cd betterstack-swarm-kubernetes
```

The structure of the directory should look like this:

```
├── config
├── Dockerfile
├── go.mod
├── go.sum
├── handler
├── kubernetes
├── main.go
└── swarm
```

- The `swarm` and `kubernetes` directories, along with the `Dockerfile` file are for deploying the app to Docker Swarm and Kubernetes platform. You will temporarily skip them for now.
- The rest of the directories and files constitute the Go application.

2. Creating a PostgreSQL database in the local environment

Assuming PostgreSQL is installed on your machine, you can go ahead and access the `psql` command-line using the default `postgres` user by running the following command:

```
sudo -u postgres psql
```

In the `psql` interface, create a new user named "blog_user":

```
CREATE USER blog_user;
```

Run the following command to list all existing users in the PostgreSQL database. You should see the `blog_user` user there.

```
\du
```

Output

```
                                List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
```

```
 blog_user      |                                                    | {}
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
```

Next, change the password of the `blog_user` user to "blog_password":

```
ALTER ROLE blog_user WITH PASSWORD 'blog_password';
```

Afterward, create a new `blog_db` database by running the following command:

```
CREATE DATABASE blog_db;
```

Finally, quit the current session using `\q`, and then log in again to `blog_db` database using `blog_user` user. Enter the password for the blog user if prompted:

```
psql -U blog_user -d blog_db -h localhost
```

At this stage, you can create a new `posts` table within the `blog_db` database by running the following command:

```
CREATE TABLE IF NOT EXISTS posts (
   ID         SERIAL NOT NULL PRIMARY KEY,
   BODY       TEXT   NOT NULL,
   CREATED_AT TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   UPDATED_AT TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Now that you've successfully set up the database let's go ahead and run the application in the local environment.

3. Starting the application

To bring up the app in the local environment, you need to install the required dependencies first:

```
go mod tidy
```

Then run the following commands to set the necessary environment variables for the Go application to access the PostgreSQL database:

```
export POSTGRES_PASSWORD=blog_password
```

```
export POSTGRES_DB=blog_db
```

```
export POSTGRES_USER=blog_user
```

```
export POSTGRES_HOST=localhost
```

Afterward, run the following command to start the Go application:

```
go run main.go
```

Output

```
You are connected to the database
```

Once the application is up and running open a separate terminal and run the following command to create a new blog entry:

```
curl --location 'http://localhost:8081/blog' \
    --header 'Content-Type: application/json' \
    --data '{
        "body":"this is a great blog"
    }'
```

Then run the following command to get the entry with an id of `1`:

```
curl --location 'http://localhost:8081/blog/1'
```

You should see the result below:

Output

```
{"id":1,"body":"this is a great blog","created_at":"2023-05-
13T15:03:40.461732Z","updated_at":"2023-05-13T15:03:40.461732Z"}
```

At this point, you've successfully set up the app in the local environment and interacted with its APIs to confirm that it's in working order. Let's now move on to the next section where you will deploy the application using Docker Swarm.

Deploying the application with Docker Swarm

Since you no longer need the PostgreSQL database service to run, you can stop it by running the following command:

```
sudo service postgresql stop
```

You can also stop the current application by pressing `CTRL+C` in the terminal.

1. Create a Docker Swarm cluster

In a Docker Swarm cluster, you will have one or more manager nodes to distribute tasks to one or more worker nodes. To simplify this demonstration, you will only create the manger node responsible for deploying the services and handling the application workload.

To create a Docker Swarm cluster, run the following command:

```
docker swarm init
```

You should see the following output confirming that the current node is a manager and relevant instructions to add a worker node to the cluster:

Output

```
Swarm initialized: current node (rpuk92y8wypwqwnv5kqzk5fik) is now a manager.


To add a worker to this swarm, run the following command:
```

```
    docker swarm join --token <token> <ip_address>

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the
instructions.
```

Since we won't be adding a worker node to the Swarm cluster, let's move on to the next section and push the Docker image to DockerHub. If you'd like to see a full demonstration of running a highly available Docker Swarm setup in production, please see the linked article.

2. Building the application Docker image

Inside the application directory, you have a `Dockerfile` which has the following contents:

Dockerfile

```
FROM golang:1.20

WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY main.go ./
COPY config/db.go ./config/db.go
COPY handler/handlers.go ./handler/handlers.go
RUN CGO_ENABLED=0 GOOS=linux go build -o /out/main ./
EXPOSE 8081

# Run
ENTRYPOINT ["/out/main"]
```

These commands inside the `Dockerfile` instructs Docker to pull the Go version 1.20 image, then copy the files from our project directory into the Docker image and build an application executable named `main` in the /out directory. It also specifies that the Docker container will listen on port 8081 and that the `/out/main` binary is executed when the container is started.

To build the application image, run the following command. Remember to replace the `<username>` placeholder with your DockerHub username:

```
docker build -t <username>/blog .
```

You should observe the following output at the end of the process:

Output

```
. . .
Successfully built 222ca8bc81b8
Successfully tagged <username>/blog:latest
```

Now that you have successfully built the application image, log into your DockerHub account from the current session so that you can push the image to your account:

```
docker login
```

Output

```
WARNING! Your password will be stored unencrypted in /home/<user>/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

Once login is successful, run the following command to push the image to DockerHub:

```
docker push <username>/blog
```

Now that you've successfully pushed the image to your DockerHub account, let's move on to the next section and deploy the application to the Swarm cluster.

3. Deploying the application to the Swarm cluster

In this section, you will deploy the Go application and the PostgreSQL database to the Swarm cluster. Within the `swarm` directory, you have the two following files:

- `initdb.sql` is for initializing the database state by creating a new table named "posts" when deploying the PostgreSQL database to the Swarm cluster
- `docker-compose.yml` is for creating the PostgreSQL database and the application services to the Swarm cluster

Open the `docker-compose.yml` file in your text editor and update the `<username>` placeholder to your Docker username:

swarm/docker-compose.yml

```
# Use postgres/example user/password credentials
version: '3.6'

services:
  db:
    image: postgres
    environment:
      - POSTGRES_DB=blog_db
      - POSTGRES_USER=blog_user
      - POSTGRES_PASSWORD=blog_password
    ports:
      - '5432:5432'
    volumes:
      - ./initdb.sql:/docker-entrypoint-initdb.d/initdb.sql
    networks:
      - blog-network

  blog:
```

image: <username>/blog

```


    environment:
      - POSTGRES_DB=blog_db
      - POSTGRES_USER=blog_user
      - POSTGRES_PASSWORD=blog_password
      - POSTGRES_HOST=db:5432
    ports:
      - '8081:8081'
    volumes:
      - .:/app
    networks:
      - blog-network

networks:
  blog-network:
    name: blog-network
```

This file instructs Docker Swarm to create three services called `db`, `blog`, and `networks`:

- The `db` service is for creating a PostgreSQL database using the specified configuration. Its state is initialized using the `initdb.sql` file, and it uses the `blog-network` network.
- The `blog` service creates the blog application using the Docker image you previously pushed to DockerHub. You also need to provide environment variables for this service so that the application can access the PostgreSQL database. This service will also use the `blog-network` network allowing interaction with the `db` service through the host URL: `db:5432`.
- The `networks` service creates a network named `blog-network` so the two services can communicate via this network.

To deploy these three services to the running Swarm cluster, run the following commands in turn:

```
cd swarm
```

```
docker stack deploy --compose-file docker-compose.yml blogapp
```

Output

```
Creating network blog-network
Creating service blogapp_db
Creating service blogapp_blog
```

Afterward, run the following command to verify that the services are running:

```
docker stack services blogapp
```

You should see the following results:

Output

```
ID             NAME           MODE         REPLICAS   IMAGE                  PORTS
uptqkx630zpy   blogapp_blog   replicated   1/1        username/blog:latest   *:8081->8081/tcp
jgb19lcx32y6   blogapp_db     replicated   1/1        postgres:latest        *:5432->5432/tcp
```

You should also be able to interact with the application endpoints now. Run the following command to create a new blog post:

```
curl --location 'http://localhost:8081/blog' \
    --header 'Content-Type: application/json' \
    --data '{
        "body":"this is a great blog"
    }'
```

Then run the command below to retrieve the newly created post:


```
curl --location 'http://localhost:8081/blog/1'
```

You should observe the same output as before:

Output

```
{"id":1,"body":"this is a great blog","created_at":"2023-05-
13T08:56:26.140815Z","updated_at":"2023-05-13T08:56:26.140815Z"}
```

Now that you've successfully deployed the blog application using Docker Swarm. Let's move on to the next section and learn how to deploy the application using Kubernetes.

Deploying the application with Kubernetes

To simplify the setup for the Kubernetes cluster, you will use the minikube tool which allows you to create an run a local Kubernetes cluster on your machine.

Start by following these instructions to install the `minikube` binary on your machine, then launch the minikube service by running the following command:


```
minikube start
```

Afterwards, run the following command to check whether the minikube service is up and running:


```
minikube kubectl -- get node
```

You should see the output below:

Output

```
NAME       STATUS   ROLES           AGE    VERSION
minikube   Ready    control-plane   188d   v1.25.3
```

For ease of use, you should alias the `minukube kubectl --` command to `kubectl` as follows:


```
alias kubectl="minikube kubectl --"
```

Now that the minikube service is up and running, you will can proceed with deploying the Go application and PostgreSQL database to the Kubernetes cluster. Let's start by deploying the PostgreSQL database in the next section.

1. Deploying the PostgreSQL database to the Kubernetes cluster

To deploy the PostgreSQL database to the Kubernetes cluster, you need to:

- Create a Kubernetes Persistent Volume (Kubernetes PV) so that you don't lose the application data when the Kubernetes cluster is down.
- Create the Kubernetes Persistent Volume Claim (Kubernetes PVC) to request storage inside the Kubernetes PV.
- Create the Kubernetes Config Map with initializing SQL query to create a table named `posts` when deploying the PostgreSQL database.
- Create the Kubernetes Deployment to deploy the PostgreSQL database to the Kubernetes cluster.
- Create the Kubernetes Service, which allows the blog application to access to the PostgreSQL database.

From the current terminal, navigate to the `kubernetes` directory:


```
cd ../kubernetes
```

Next, create a `postgres/docker-pg-vol/data` directory inside the `kubernetes` directory to store the persistent data:


```
mkdir postgres/docker-pg-vol/data -p
```

Update the `path` inside the `hostPath` block in the `postgres-volume.yml` file with the absolute path to the persistent data directory. The path should look like this: `/home/<username>/betterstack-swarm-kubernetes/kubernetes/postgres/docker-pg-vol/data`

```
nano postgres-volume.yml
```

kubernetes/postgres-volume.yml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: postgresql-claim0
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
```

path: "/home/<username>/betterstack-swarm-kubernetes/kubernetes/postgres/docker-pg-vol/data"

```

```

Afterward, run the following command to create the Kubernetes Persistent Volume. If you do not specify the `--namespace` value, Kubernetes will automatically use the `default` namespace.

```
kubectl apply -f postgres-volume.yml
```

Output

```
persistentvolume/postgresql-claim0 created
```

Then run the following command to create the Persistent Volume Claim.

```
kubectl apply -f postgres-pvc.yml
```

Output

```
persistentvolumeclaim/postgresql-claim0 created
```

To create the configuration map for initializing the SQL query, run the following command:

```
kubectl apply -f postgres-initdb-config.yml
```

Output

```
configmap/postgresql-initdb-config created
```

Next, run the following command to deploy the PostgreSQL database:

```
kubectl apply -f postgres-deployment.yml
```

Output

```
deployment.apps/postgresql created
```

Finally, create the Kubernetes service for the PostgreSQL database:

```
kubectl apply -f postgres-service.yml
```

Output

```
service/postgresql created
```

At this point, the PostgreSQL instance should be up and running. Go ahead and run the command below to confirm:

```
kubectl get pod
```

You should see the following result:

Output

```
NAME                              READY   STATUS    RESTARTS        AGE
postgresql-655746b9f8-n4dcf       1/1     Running   0               2m36s
```

Run the following command to check for the PostgreSQL service.

```
kubectl get service
```

You should see the following output:

Output

```
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP    14d
postgresql   ClusterIP   10.98.157.228   <none>        5432/TCP   2s
```

Now that you've successfully deployed the PostgreSQL database to the Kubernetes cluster, let's deploy the blog application next.

2. Deploying the application to the Kubernetes cluster

Before you can deploy your application using Kubernetes, you need to create a Docker secret so that Kubernetes can access your DockerHub account to pull the application image. Note that you need to replace the username, password, and email below with your DockerHub account information:

```
kubectl create secret docker-registry dockerhub-secret \
  --docker-server=docker.io \
  --docker-username=<username> \
  --docker-password=<password> \
  --docker-email=<email>
```

Output

```
secret/dockerhub-secret created
```

Once the secret is created, open the `deployment.yml` file in your text editor and edit it as follows:

```
nano kubernetes/deployment.yml
```

kubernetes/deployment.yml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: blogapp
spec:
  replicas: 2
  selector:
    matchLabels:
      name: blogapp
  template:
    metadata:
      labels:
        name: blogapp
    spec:
      containers:
        - name: application
```

image: <username>/blog

```
          imagePullPolicy: Always
          envFrom:
            - secretRef:
                name: dockerhub-secret
          env:
            - name: POSTGRES_DB
              value: blog_db
            - name: POSTGRES_USER
              value: blog_user
            - name: POSTGRES_PASSWORD
              value: blog_password
            - name: POSTGRES_HOST
              value: postgresql:5432
          ports:
            - containerPort: 8081
```

Save and close the file, then run the following command to deploy the blog app:

```
kubectl apply -f deployment.yml
```

Output

```
deployment.apps/blogapp created
```

Then run the following command to create the Kubernetes service for the blog app, so that you can access the app from outside the Kubernetes cluster.


```
kubectl apply -f service.yml
```

Output

```
service/blogapp-service created
```

Run the following command to get the running services inside the Kubernetes cluster.


```
kubectl get service
```

You should see the following result:


```
NAME              TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
blogapp-service   LoadBalancer   10.97.173.87   <pending>     8081:31983/TCP   5s
kubernetes        ClusterIP      10.96.0.1      <none>        443/TCP          14d
postgresql        ClusterIP      10.96.38.7     <none>        5432/TCP         8m1s
```

Notice that the `EXTERNAL-IP` column for the `blogapp-service` is pending. This is because you are using a custom Kubernetes cluster (minikube) that does not have an integrated load balancer. To workaround this issue, create a minikube tunnel in a new terminal through the command below:


```
minikube tunnel
```

Then run the `kubectl get service` command again. You should see the updated result with a proper `EXTERNAL-IP` value:

Output

```
NAME              TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)          AGE
blogapp-service   LoadBalancer   10.97.173.87   10.97.173.87   8081:31983/TCP   4m40s
kubernetes        ClusterIP      10.96.0.1      <none>         443/TCP          14d
postgresql        ClusterIP      10.96.38.7     <none>         5432/TCP         12m
```

Now that you've deployed the blog application to the Kubernetes cluster, you can access it through its external IP which in the above example is `10.97.173.87`.

```
curl --location 'http://<external_ip>:8081/blog' \
    --header 'Content-Type: application/json' \
    --data '{
        "body":"this is a great blog"
    }'
```

Then run the following command to get the blog content with id `1`.

```
curl --location 'http://<external_ip>:8081/blog/1'
```

You should see the output as:

Output

```
{"id":1,"body":"this is a great blog","created_at":"2023-05-
13T08:56:26.140815Z","updated_at":"2023-05-13T08:56:26.140815Z"}
```

And that's how you deploy the blog application to the Kubernetes cluster.

Comparing the deployment process between Docker Swarm and Kubernetes, we observe distinct trade-offs. Docker Swarm offers a simpler deployment experience, but it lacks built-in support for persistent data storage. In the event of a Docker Swarm cluster failure or recreation, there is a risk of losing the application data.

In contrast, Kubernetes introduces a more intricate deployment process but it offers the advantage of creating persistent data storage, ensuring data durability even during cluster disruptions or recreation. Additionally, Kubernetes provides flexibility in exposing the blog app service as per your requirements by leveraging the Kubernetes Service definition file.

Use cases of Kubernetes vs. Docker Swarm

Docker Swarm and Kubernetes, despite being container orchestration platforms, possess distinct characteristics that cater to different use cases.

Docker Swarm is well-suited for:

- Beginners in containerization who seek to learn application deployment using containerization techniques.
- Small to medium-sized applications that require straightforward deployment and management.
- Users familiar with Docker and prefer a Docker-centric approach to application deployment. Docker Swarm seamlessly integrates with the Docker ecosystem.
- Applications with a relatively stable user base or predictable traffic patterns. Docker Swarm, although lacking advanced automatic scaling features compared to Kubernetes, can adequately handle such scenarios.

Kubernetes should be considered when:

- You have a comprehensive understanding of Kubernetes components and have acquired proficiency in working with the platform. Kubernetes has a steeper learning curve but offers a comprehensive solution for deploying and managing containerized applications.
- Your application is complex and demands extensive customization during deployment. Kubernetes excels in managing large-scale, intricate applications, providing advanced management, scalability, and customization capabilities.
- Fine-grained control and customization options are crucial for your deployment.
- Automatic scaling is necessary due to your application's growing user base.

In summary, Docker Swarm suits simpler deployments and those who prefer a Docker-centric approach, while Kubernetes caters to complex applications, fine-grained control, and automatic scaling needs. Carefully assess your requirements and familiarity with the platforms to determine the most suitable choice for your specific use case.

Final thoughts

Throughout this article, you've gained insights into the distinctions between Docker Swarm and Kubernetes, and you've also had practical experience deploying a sample Go application using both platforms.

To further explore technical articles on cloud-native and containerization technologies, we invite you to visit our BetterStack community guides. There, you can find additional resources and in-depth information to enhance your understanding of these technologies.

Thanks for reading!

Revision #1
Created 22 March 2026 18:12:43 by Administrador
Updated 22 March 2026 18:17:19 by Administrador