North American Journal of Engineering and Research

Est. 2020

Deploy a Microservice Application using Docker Compose

Pallavi Priya Patharlagadda
Email:
pallavipriya527.p@gmail.com

Abstract

The deployment of microservices as Docker containers is examined in this paper. Kubernetes pods are used to deploy the containers after they have been orchestrated using Docker compose and Docker swarm. We go over how these technologies make microservices architectures scalable and easy to deploy. We illustrate the advantages of using these tools in contemporary software development with real-world examples and insights. By leveraging Docker for containerization and Docker Compose for the management of multi-container applications, our investigation demonstrates how these tools can be used to create microservice architectures that are resilient, scalable, and Flexible

Introduction

The evolution from monolithic applications to microservices reflects a shift in software architecture towards a more modular and scalable approach. Microservices are tiny, autonomous components that provide a single, clearly defined functionality. The microservices architecture gave developers the flexibility to write applications in their language of choice. On the other hand, containerization complemented this by packaging the dependencies so that the microservices could be deployed across any environment.

The concept of containers started with Linux containers. Linux Containers is an operating-system-level virtualization method for running multiple isolated Linux systems on a control host using a single Linux kernel. Docker is a tool that is used to automate the deployment of applications in containers so that applications can work efficiently in different environments in isolation. The isolation is achieved using Linux namespaces and Cgroups.

Docker Compose is a tool for defining and running multi-container applications. It allows you to define and manage multi-container applications in a single YAML file. This simplifies the complex task of orchestrating and coordinating various services, making it easier to manage and replicate your application environment.

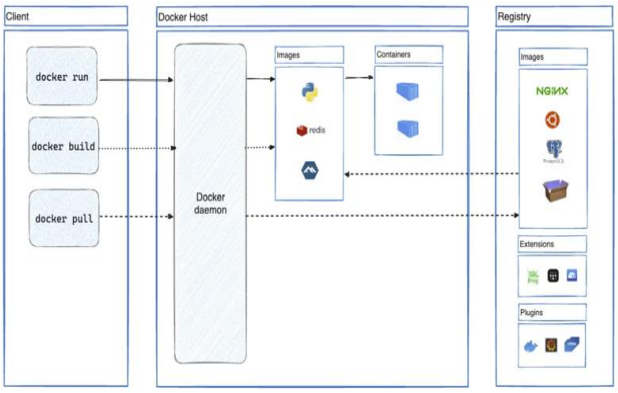
Architecture:

Docker:

Docker is a tool that provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security that containment provides let you run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application.

Docker provides a platform to manage the lifecycle of your containers. Develop your application and its supporting components using containers. Deploy the application into a production environment, either as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

Docker architecture:



The Docker client:

The Docker client (docker) is the frontend and primary way that Docker users interact with Docker. When docker commands such as docker run or docker build are executed, the client sends these commands to dockerd using the Docker API, which then performs the specified functionality. The Docker client can communicate with more than one daemon.

A. The Docker daemon:

The Docker daemon (dockerd) listens to Docker API requests and manages Docker objects such as images, containers,

networks, and volumes. A daemon can communicate with other daemons to manage Docker services.

The Docker registries:

The Docker registry is the place where Docker images are stored. Docker Hub is a public registry, and Docker looks for images on Docker Hub by default. The user can create a private registry. Once the Docker registry is configured, Docker tries to pull/push the required images from/to the configured registry.

Docker Objects:

Docker objects are images, containers, networks, volumes, plugins, and others.

There are a lot of images in the Docker Hub. The user can either pull an existing image from Dockerhub or create their own image by writing a Docker file. Docker file is nothing but a text file that takes a base image, installs the dependencies, copies the application, and executes the application. Docker build on the Dockerfile would create a Docker image. Docker run would create a container from the provided image.

Docker Desktop:

Docker Desktop provides the GUI for managing images, containers, and applications directly on your machine. This is very user friendly for the developers.

From the Dev Environments, we can create a sample project in the Docker Desktop. I will demonstrate this by using below Dockerfile.

Consider below Dockerfile:

```
FROM golang:1.21
WORKDIR /src
COPY <<EOF ./main.go
package main

import "fmt"

func main() {
  for {
        fmt.Println("hello, Docker")
      }
}
EOF
RUN go build -o /bin/hello./main.go
CMD ["/bin/hello"]
```

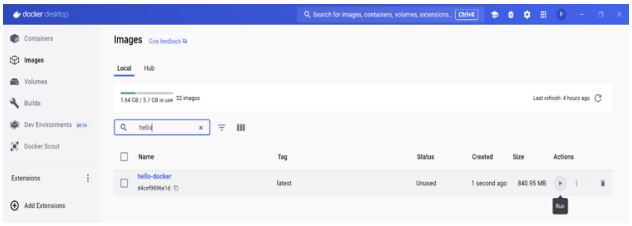
The Dockerfile performs the below actions.

1. Pulling the Docker base image golang:1.21 from Dockerhub repository.
2. Setting the current work directory on the container to /src
3. Copy main.go file to the current directory.
4. Build the main.go and save the generated binary as /bin/hello
5. Execute the binary on the container.

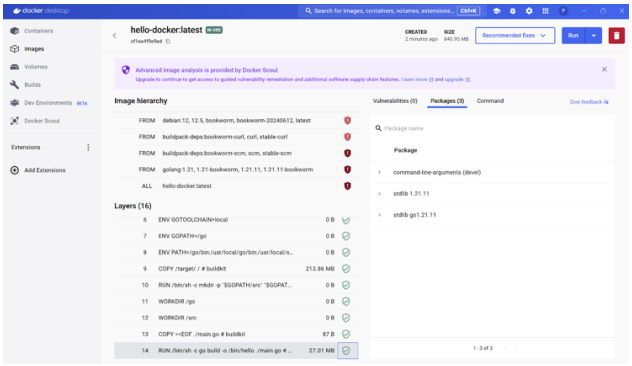
Build a Docker image using:

```
docker build -t hello-docker -f Dockerfile.
```

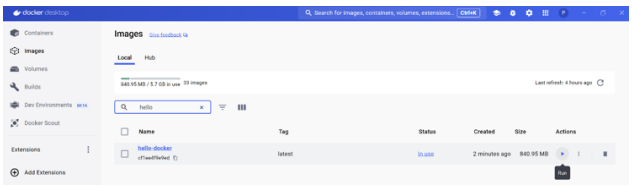
On successful Docker build, Docker image will be created. Let's check in the Docker Desktop.

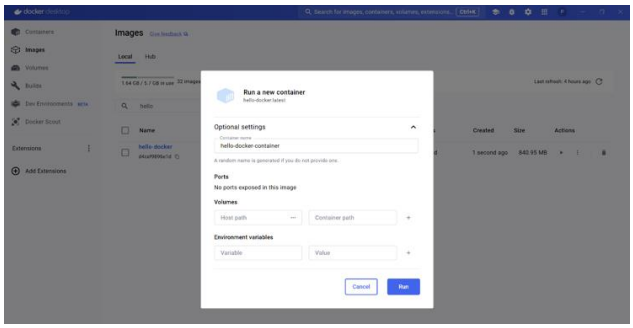


By clicking on the image, it would provide information on Layers, Image Hierarchy, etc.

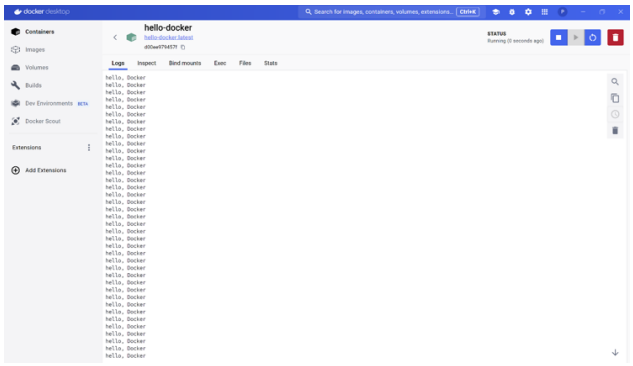


Let's navigate to the containers tab on the left panel and try to run the container from the image.





Docker Desktop allows us to provide arguments like container, volume, and environment variables. I just provided the container name. On running it, a new container will be created, which will take you to the logs page of the container.



Here we can see the other options, like inspecting the container, bind mounts, executing commands inside the container, filesystem, and the statistics that display the CPU, memory, disk, and I/O usage.

If your application is tiny and requires just one container, then a Docker container is sufficient. But if the application contains multiple docker containers, then it would be difficult to manage all these docker containers individually. In that case, Using Docker compose would make deployment easy.

Docker Compose:

Docker Compose is a tool for defining and running multi-container applications. Docker Compose uses a single comprehensible YAML configuration file to define services, networks, volumes, configs, secrets required by applications. This helps in creating and starting all the services from your configuration file with a single command. Below is the sequence of actions to be performed .rite Dockerfiles and build images.

.Create stacks (consisting of individual containers/services) using the Dockerfile images defined in docker-compose.yml.

.Deploy the entire application using docker-compose command

Docker compose have different top-level elements like services, networks, volumes, configs, secrets, etc. sample

Services:

A service is an abstract definition of a computing resource within an application which can be scaled or replaced independently from other components. Services are backed by a set of containers, run by the platform according to replication requirements and placement constraints. As services are backed by containers, they are defined by a Docker image and set of runtime arguments. All containers within a service are identically created with these arguments.

For services, each service may also include a build section, which defines how to create the Docker image for the service. Compose supports building docker images using this service definition. If not used, the build section is ignored, and the compose file is still considered valid.

***services**:*

frontend:

image: busybox

backend:

image: python

Networks:

Networks let services communicate with each other. Compose sets a default single network for the application. Each container for a service joins the default network, is reachable by other containers on that network, and is discoverable by the service's name.

The top-level networks element lets you configure named networks that can be reused across multiple services. To use a network across multiple services, an explicit grant should be provided for each service by using network attribute within the services top-level element.

services:

frontend:

```
  image: example/webapp
  networks:
   - front-net
   - back-net
networks:
 front-net:
 back-net:
```

Volumes:

Volumes are persistent data stores implemented by the container engine. Compose allows services to mount volumes, and configuration parameters to allocate them to infrastructure. The top-level volumes declaration lets you configure named volumes that can be reused across multiple services.

Like Networks, to use a volume across multiple services, one must explicitly grant each service access by using the volumes attribute within the services top-level element. The volumes attribute has additional syntax that provides more granular control.

```
services:
 backend:
  image: example/database
  volumes:
   - db-data:/etc/data

 backup:
  image: backup-service
  volumes:
   - db-data:/var/lib/backup/data
volumes:
 db-data:
```

Configs:

Configs let services adapt their behavior without rebuilding a Docker image. Configs are mounted as files in container's filesystem. The location of the mount point within the container defaults to /<config-name> in Linux containers and C:\<config-name> in Windows containers. Services can only access configs when explicitly granted by a configs attribute within the services top-level element. The top-level configs declaration defines or references configuration data that is granted to services in your Compose application. The source of the config is either a file or external. Below is a sample reference where config is a file.

```
configs:
 http_config:
  file: ./httpd.conf
```

<project_name>_http_config is created when the application is deployed, by registering the content of the httpd.conf as the configuration data.

Secrets:

Secrets are similar to Configs used for sensitive data, with specific constraint for this usage.

Services can only access secrets when explicitly granted by a secrets attribute within the services top-level element. The top-level secrets declaration defines or references sensitive data that is granted to the services in a Compose application. The source of the secret is either file or environment.

Below is an example.

```
secrets:
 server-certificate:
  file: ./server.cert
```

server-certificate secret is created as <project_name>_server-certificate when the application is deployed, by registering content of the server.cert as a platform secret.

Deploying an application using Docker compose:

I created a sample project where I have three containers

nginx as the frontend container.

A mysql container as DB container

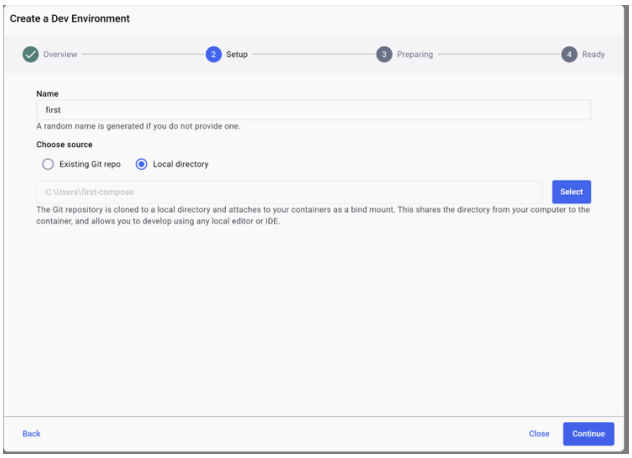
A Golang backend container that writes data to the database.

Once the data is written, I should be able to see the data in the browser and as well as curl output. Below is the sample compose file used.

```
services:
 backend:
  build:
   context: backend
   target: development
  secrets:
   - db-password
```

```
    depends_on:
      - db
  db:
    image: mariadb
    restart: always
    healthcheck:
      test: [ "CMD", "mysqladmin", "ping", "-h", "127.0.0.1", "--silent" ]
      interval: 3s
      retries: 5
      start_period: 30s
    secrets:
      - db-password
    volumes:
      - db-data:/var/lib/mysql
    environment:
      - MYSQL_DATABASE=example
      - MYSQL_ROOT_PASSWORD_FILE=/run/secrets/db-password
    expose:
      - 3306
  frontend:
    build: frontend
    ports:
      - 8080:80
    depends_on:
      - backend
volumes:
  db-data:
secrets:
  db-password:
    file: db/password.txt
```

From Docker Desktop, in the “Dev Environments” tab, we can provide a GitHub URL or a local directory. In my case, I provided my local folder.

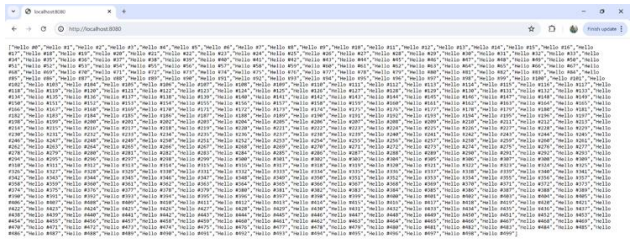


Docker Desktop would look for the compose file in the directory and start deploying the application.

```
Detecting main repo language...
2024/07/02 14:40:18 http2: server: error reading preface from client //./pipe/docker_engine: file has already been closed
#0 building with "default" instance using docker driver

#1 [backend internal] load build definition from Dockerfile
#1 transferring dockerfile: 344B done
#1 DONE 0.0s

#2 [backend internal] load metadata for docker.io/library/golang:1.16
#2 ...

#3 [backend auth] library/golang:pull token for registry-1.docker.io
#3 DONE 0.0s

#2 [backend internal] load metadata for docker.io/library/golang:1.16
#2 DONE 0.6s

#4 [backend internal] load .dockerignore
#4 transferring context: 2B done
#4 DONE 0.0s

#5 [backend build 1/4] FROM docker.io/library/golang:1.16@sha256:5f6a4662de3efc6d6bb812d02e9de3d8698eeea16b8eb7281f03e6f3e8383018e
#5 DONE 0.0s

#6 [backend internal] load build context
#6 transferring context: 1.92kB done
#6 DONE 0.0s

#7 [backend build 2/4] WORKDIR /go/src
#7 CACHED
```

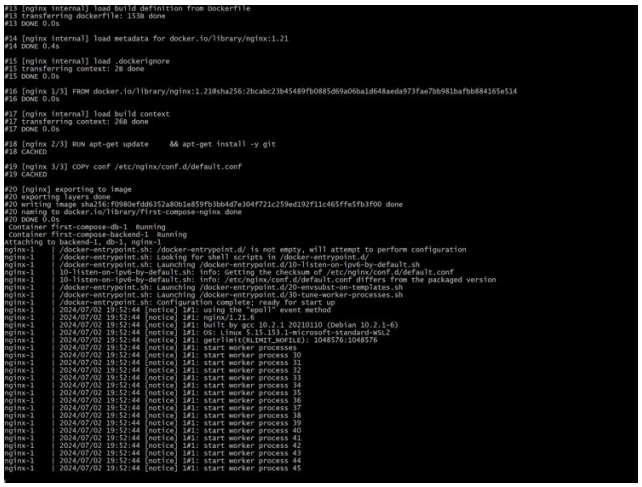
Once Deployed, we can see the project running.

We can go through the containers tab and see the logs of the container or exec to the container and issue a command, etc. To verify if the application is running, I gave the URL in the browser and was able to see the Hello messages written from Golang backend container to the database.



So, the application is up and running. We can also deploy the application from cli using the below command.

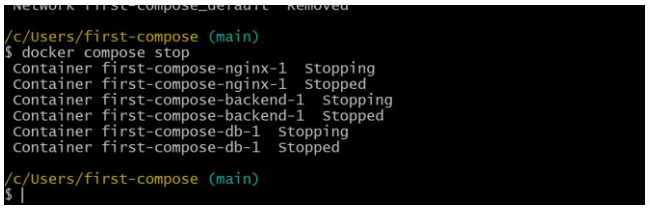
docker-compose -f docker-compose.yaml up



Curl on the other terminal would display the hello messages written.

Command to use to stop the application:

docker compose stop



Command to stop and remove the application containers:

docker compose down

Conclusion:

Docker and Docker compose are very helpful for deploying container applications. Docker Desktop is very useful for developers during initial deployment, as we can watch logs, inspect the container, execute commands inside the container, Monitor the CPU and RAM utilization, etc. Docker Desktop reduces the time spent on complex setups so you can focus on writing code. It takes care of port mappings, file system concerns, and other default settings.

References:

https://docs.docker.com/guides/docker-overview/#docker-architecture

https://docs.docker.com/compose/

https://docs.docker.com/desktop/