Earthly Technologies

DOCKER
FUNDAMENTALS

Contents

Introduction to Docker Fundamentals

Docker is a tool that's reshaped how developers create, deploy, and run applications by using containers. This introduction explains Docker's basics, its application in development, and its importance.

What is Docker?

Docker provides a way to package an application and its dependencies in a container — a standardized unit of software. This packaging ensures that the application works in any environment, addressing the "it works on my machine" headache by offering consistency across development, testing, and production settings.

Docker vs. Virtual Machines

Understanding Docker's benefits starts with comparing it to virtual machines (VMs). VMs virtualize hardware, requiring a full operating system for each VM. Docker containers, on the other hand, share the host system's kernel, start faster, and use fewer resources than VMs. This efficiency makes Docker containers an attractive option for running multiple applications on a single system.

Use Cases and Advantages

Docker is flexible and useful for anything from small projects to large, complex applications. Its key advantages include:

- **Configuration Simplicity**: Docker reduces compatibility problems by standardizing environments.
- **Code Pipeline Management**: It ensures that applications run as expected in different environments.
- **Application Isolation**: Multiple applications can run on the same server without affecting each other, optimizing resource use.
- **Microservices Architecture**: Docker is ideal for microservices, allowing each service to be deployed independently.

Importance of Docker in Development

Docker's role in development and deployment workflows stems from:

- **Efficiency and Speed**: Containers are lightweight and portable, enabling fast launches and scalability.
- **Consistency Across Environments**: Docker containers ensure that software behaves the same way in every environment.
- **Rapid Deployment and Scaling**: Docker fits well with CI/CD practices, allowing for quick updates and easy scaling.
- **Productivity**: By handling complex setups, Docker lets developers focus more on coding.

Navigating Through This eBook

This eBook covers Docker's essentials, progressing from basic to more advanced topics. It starts with Docker volumes, networking, and secrets management, then moves into logging, multistage builds, debugging, and practical use cases like integration with Visual Studio Code and GitHub Actions. This guide aims to provide a solid understanding of Docker, focusing on practical applications and streamlined explanations.

As you proceed, you'll gain a clearer view of how Docker can improve your development processes, making your applications more portable, scalable, and efficient. This journey will equip you with the knowledge to leverage Docker effectively in your projects.

Understanding Docker Volumes

Docker is a common containerization solution that offers a user-friendly interface. It allows you to deploy your application as a lightweight process set rather than a complete virtual machine.

Docker images are like a snapshot of a container's file system and contain both your application and its dependencies. When you run it, you recreate the container's state. You don't have to be concerned about setting up your environment because running an image recreates everything for you and is isolated from your operating system and other running containers.

The Docker interface is simple and users can easily create and implement applications into their containers or carry out version management, copy, share, and modify, just like managing ordinary code.

However, containers often need to use data beyond their container or share data between containers. While it may be tempting to rely on the host file system, a better solution is to work with persistent data in a container, namely Docker volumes.

A Docker volume is an independent file system entirely managed by Docker and exists as a normal file or directory on the host, where data is persisted.

In this guide, you'll learn how volumes work with Docker, what they do, and what the best practices are for keeping them secure and effective.

What Are Docker Volumes

The purpose of using Docker volumes is to persist data outside the container so it can be backed up or shared.

Docker volumes are dependent on Docker's file system and are the preferred method of persisting data for Docker containers and services. When a container is started, Docker loads the read-only image layer, adds a read-write layer on top of the image stack, and mounts volumes onto the container filesystem.

Why Docker Volumes?

If you are using Docker for development, you must be familiar with the `-v` or `--volume` flag that lets you mount your local files into the container. For instance, you can mount your local

`./target` onto the `/usr/share/nginx/html` directory container or an nginx container to visualize your html files.

```
echo "<h1>Hello from Host</h1>" > ./target/index.html
docker run -it --rm --name nginx -p 8080:80 -v "$(pwd)"/target:/usr/share/nginx/html nginx
```

Navigate to `http://localhost:8080/` and you should see "Hello from Host".

This is called a bind mount and is commonly used by developers. But, if you are using Docker Desktop on Windows or MacOS bind, mounts have significant performance issues. As a result, using volumes may be the best alternative for holding state between container runs.

Unlike bind mount, where you can mount any directory from your host, volumes are stored in a single location (most likely /var/lib/docker/volumes/ on unix systems) and greatly facilitates managing data (backup, restore, and migration). Docker volumes can safely be shared between several running containers.

You can also save data to a remote server or in cloud Docker volumes with alternative volume drivers like `sshfs`.

In addition, Docker enables you to manage volume with the command line `docker volume`, making their management simple.

Creating and Managing Docker Volumes

In this section, you'll learn how to create a Docker volume implicitly and explicitly and then declare it from a Docker file. Then you'll learn how to view a data volume, mount it to a container, and configure it using `docker-compose`.

Create a Docker Volume Implicitly

The easiest way to create and use a volume is with `docker run` and the `-v` or `--volume` flag. This flag takes three arguments separated by `::`

```
-v <source>:<destination>:<options>
```

If the "source" is a path that was used in the previous example, Docker will use a mount bind. If the "source" is a name, then Docker tries to find this volume or creates one if one cannot be found. Below, the previous example has been updated to use a volume instead of a mount bind:

```
docker run -it --rm --name nginx -p 8080:80 -v demo-earthly:/usr/share/nginx/html nginx
```

You can check to make sure the container was properly created with `docker volume ls` which lists all existing volumes.

> Note that the volume in question is not empty. If a volume is completely empty, the container's content is copied to the volume.

You can check the status of your volumes on Linux. This gives you a chance to see where volumes are stored:

```
ls /var/lib/docker/volumes/target/_data/demo-earthly
```

```
root@bounty:/home/box/docker-helloworld# docker volume ls
DRIVER    VOLUME NAME
local     8efcdf2aa7ad911206ad3dc795b87f009e46a2019054821d2f2c04652acec920
local     91c29beb0ff5c6df8739c10a6a077a2b4125ef9f082ac48d84bac952e5712caa
local     161209b30cfcfbb990f02ca7a5d6750f81ce9c00685736bc6e4e3d5911f53a5d
local     aa3925b28a8d5bcf1d087141fb8353dd91f80bfda127f39fcd98aefa7b515681
local     b3b4a2dfc079a9613d8298aa5f7ece0712db250b1dab3223b2d3be8a40954a66
local     b222f5f9c4959749a0ff36b1d052cbbd3d8d0f87fc32d675fac0c45b23cab1ff
local     demo-earthly
local     demo-volume
```

Figure 1: `docker volume ls`

On Mac and Windows it's a bit more tricky. In order to keep things simple, you can mount the volume on an ubuntu container and use `ls` to see the content of your volume:

```
docker run -it --rm -v demo-earthly:/opt/demo-earthly ubuntu ls /opt/demo-earthly
```

Create a Docker Volume Explicitly

Alternatively you can use the `docker volume create` command to explicitly create a data volume. This command gives you the option to choose and configure the volume driver. The implicit creation of volumes always uses the `local` driver with default settings.

```
docker volume create --name demo-earthly
```

Declare a Docker Volume from Dockerfile

Volumes can be declared in your Dockerfile using the `VOLUME` statement. This statement declares that a specific path of the container must be mounted to a Docker volume. When you run the container, Docker will create an anonymous volume (volume with a unique id as the name) and mount it to the specified path.

```
FROM nginx:latest

RUN echo "<h1>Hello from Volume</h1>" > /usr/share/nginx/html/index.html
VOLUME /usr/share/nginx/html
```

Lets build and run your new image:

```
docker build -t demo-earthly .
docker run -p 8080:80  demo-earthly
```

You can now validate that nginx serves your message at `http://localhost:8080/`.

More importantly, an anonymous Docker volume has been created, and every time you start a new container, another volume is created with the content of `/usr/share/nginx/html`.

```
docker volume ls
```

From the above example, a volume directory `data` with the text file `test` containing "Hello from Volume" is created.

View a Data Volume

To manage your data, sometimes you need to list data volumes from the command line as a point of reference, which is faster than repeatedly checking the configuration files. You can use the `docker volume ls` command to view a list of data volumes.

```
root@bounty:/home/box/docker-helloworld# docker volume ls
DRIVER    VOLUME NAME
local     8efcdf2aa7ad911206ad3dc795b87f009e46a2019054821d2f2c04652acec920
local     91c29beb0ff5c6df8739c10a6a077a2b4125ef9f082ac48d84bac952e5712caa
local     161209b30cfcfbb990f02ca7a5d6750f81ce9c00685736bc6e4e3d5911f53a5d
local     aa3925b28a8d5bcf1d087141fb8353dd91f80bfda127f39fcd98aefa7b515681
local     b3b4a2dfc079a9613d8298aa5f7ece0712db250b1dab3223b2d3be8a40954a66
local     b222f5f9c4959749a0ff36b1d052cbbd3d8d0f87fc32d675fac0c45b23cab1ff
local     demo-earthly
local     demo-volume
```

Figure 2: `docker volume ls`

Use the `docker volume inspect` command to view the data volume details.

```
root@bounty:/home/box/docker-helloworld# docker volume inspect demo-earthly
[
    {
        "CreatedAt": "2021-11-13T09:30:03Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/demo-earthly/_data",
        "Name": "demo-earthly",
        "Options": {},
        "Scope": "local"
    }
]
```

Figure 3: `docker volume inspect`

Mount a Volume to a Container

As you have seen through the various examples `-v` and `--volume` are the most common way to mount a volume to a container using the syntax:

`-v <name>:<destination>:<options>`

One notable option is `ro` which means that the volume will be mounted as read-only:

```
docker run -it -v demo-volume:/data:ro ubuntu
```

Try to write into the folder/data to validate that the volume is in read-only mode:

```
echo "test" > /data/test
```

An alternative to `-v` is to add the `-mount` option to the `docker run` command. `--mount` is the more verbose counterpart of `-v`.

To launch a container and mount a data volume to it, follow this syntax:

```
docker run --mount source=[volume_name],destination=[path_in_container] [docker_image]
```

Replace `[path in container]` with the path to attach the Docker volume `[volume_name]` in the container.

For example, run the following command to start an Ubuntu container and mount the data volume to it.

```
docker run -it --name=example --mount source=demo-volume,destination=/data ubuntu
```

Remember if the volume doesn't exist Docker will create it for you.

List the contents of the container to see if the volume is mounted successfully. You should find the Docker volume name defined in the above data syntax.

Figure 4: Container content

Configure a Volume Using `docker-compose`

Although there are many ways to create a volume, it's more convenient to use the `docker-compose` command to easily share data between multiple containers.

The use of the `volume` property in compose files is very similar to `-v` and `--volume`. That being said, to perform a bind mount (mount a directory from your local machine), you can use a relative path unlike `-v` with the command `docker run` that requires an absolute path.

```
version: "3.2"
services:
  web:
    image: nginx:latest
    ports:
      - 8080:80
    volumes:
      - ./target:/usr/share/nginx/html
```

The containers and hosts in the above configuration use `volumes` in the `services` definition (`web`) to mount `./target` from the host to `/usr/share/nginx/html` of the container. As with the first example, if you navigate to `http://localhost:8080/` you should read "Hello from Host".

With `docker-compose`, volumes must be declared at the same level as `services`. Then you can refer to them by their name.

```
version: "3.2"
services:
  web:
    image: nginx:latest
    ports:
      - 8080:80
    volumes:
      - html_files:/usr/share/nginx/html
```

```
  web1:
    image: nginx:latest
    ports:
      - 8081:80
    volumes:
      - html_files:/usr/share/nginx/html

volumes:
  html_files:
```

In this example, you declared a volume named `html_files` and used it in both `web` and `web1` service. Multiple containers can mount the same volume.

Running `docker-compose up` will create a volume named `<project_name>_html_files` if it doesn't already exist . Then run `docker volume ls` to list the two volumes created, starting with the project name.

You can also manage container outside of you docker-compose file, but you still need to declare them under `volumes` and set the property `external: true`.

```
version: "3.2"
services:
  web:
    image: nginx:latest
    ports:
      - 8080:80
    volumes:
      - html_files:/usr/share/nginx/html

volumes:
  html_files:
    external: true
```

If you don't have `html_files`, you can use `docker volume create html_files` to create it. When you add `external`, Docker will find out if the volume exists; but if it doesn't, an error will be reported.

Copy Files Between Containers from a Shared Volume

Let's look at how Docker volumes enable file sharing across containers.

In this example, use the volume and container we previously defined and execute the following commands:

```
docker create volume demo-earthly
```

```
docker run -it --name=another-example --mount source=demo-volume,destination=/data ubuntu
```

Navigate to the data volume directory and create a file using the command `touch demo.txt`. Exit the container, then launch a new container `another-example-two` with the same data volume:

```
docker run -it --name=another-example-two --mount source=demo-volume,destination=/data ubuntu
```

The `demo.txt` file you created in the preceding container should list `another-example` in the output.

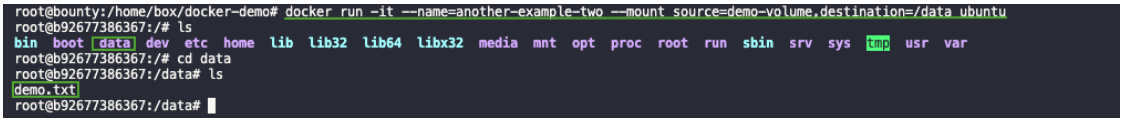


Figure 5: Copying files

Docker Volume Best Practices

Now that you've learned how to implement Docker volumes, it's important to keep in mind a few best practices:

- Always mount volumes as read-only if you only need to read from them.
- Always set the permissions and ownership on a volume.
- Always use environment variables for the host path or volume name in a production environment.

Conclusion

Often, you want your containers to use or persist data beyond the scope of the container's lifetime. You can use volumes to solve this problem by working with Docker to mount, create, and share volumes between containers.

In this guide, you looked at how volumes work with Docker, what they do, and where volumes are the preferred solution.

Understanding Docker Networking

Docker is the de facto model for building and running containers at scale in most enterprise organizations today. At a very high level, Docker is a combination of CLI and a daemon process that solves common software problems like installing, publishing, removing, and managing containers. It's perfect for microservices, where you have many services handling a typical business functionality; Docker makes the packaging easier, enabling you to encapsulate those services in containers.

Once the application is inside a container, it's easier to scale and even runs on different cloud platforms, like AWS, GCP, and Azure. In this article, let's focus on the networking aspect of Docker.

What Is a Docker Network?

Networking is about communication among processes, and Docker's networking is no different. Docker networking is primarily used to establish communication between Docker containers and the outside world via the host machine where the Docker daemon is running.

Docker supports different types of networks, each fit for certain use cases. We'll be exploring the network drivers supported by Docker in general, along with some coding examples.

Docker networking differs from virtual machine (VM) or physical machine networking in a few ways:

1. Virtual machines are more flexible in some ways as they can support configurations like NAT and host networking. Docker typically uses a bridge network, and while it can support host networking, that option is only available on Linux.
2. When using Docker containers, network isolation is achieved using a network namespace, not an entirely separate networking stack.
3. You can run hundreds of containers on a single-node Docker host, so it's required that the host can support networking at this scale. VMs usually don't run into these network limits as they typically run fewer processes per VM.

What Are Docker Network Drivers?

Docker handles communication between containers by creating a default bridge network, so you often don't have to deal with networking and can instead focus on creating and running containers.

This default bridge network works in most cases, but it's not the only option you have.

Docker allows you to create three different types of network drivers out-of-the-box: bridge, host, and none. However, they may not fit every use case, so we'll also explore user-defined networks such as `overlay` and `macvlan`. Let's take a closer look at each one.

The Bridge Driver

This is the default. Whenever you start Docker, a bridge network gets created and all newly started containers will connect automatically to the default bridge network.

You can use this whenever you want your containers running in isolation to connect and communicate with each other. Since containers run in isolation, the bridge network solves the port conflict problem. Containers running in the same bridge network can communicate with each other, and Docker uses iptables on the host machine to prevent access outside of the bridge.

Let's look at some examples of how a bridge network driver works.

1. Check the available network by running the `docker network ls` command
2. Start two busybox containers named `busybox1` and `busybox2` in detached mode by passing the `-dit` flag.

```
$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
5077a7b25ae6   bridge    bridge    local
7e25f334b07f   host      host      local
475e50be0fe0   none      null      local
```

```
docker run -dit --name busybox1 busybox /bin/sh
docker run -dit --name busybox2 busybox /bin/sh
```

3. Run the `docker ps` command to verify that containers are up and running.

```
$ docker ps
CONTAINER ID   IMAGE     COMMAND     CREATED          STATUS          PORTS     NAMES
9e6464e82c4c   busybox   "/bin/sh"   5 seconds ago    Up 5 seconds              busybox2
7fea14032748   busybox   "/bin/sh"   26 seconds ago   Up 26 seconds             busybox1
```

4. Verify that the containers are attached to the bridge network.

```
$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "5077a7b25ae67abd46cff0fde160303476c8a9e2e1d52ad01ba2b4bf04acc0e0",
        "Created": "2021-03-05T03:25:58.232446444Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
```

```
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "7fea140327487b57c3cf31d7502cfaf701e4ea4314621f0c726309e396105885": {
                "Name": "busybox1",
                "EndpointID": "05f216032784786c3315e30b3d54d50a25c1efc7d2030dc664716dda38056326",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            },
            "9e6464e82c4ca647b9fb60a85ca25e71370330982ea497d51c1238d073148f63": {
                "Name": "busybox2",
                "EndpointID": "3dcc24e927246c44a2063b5be30b5f5e1787dcd4d53864c6ff2bb3c561519115",
                "MacAddress": "02:42:ac:11:00:03",
                "IPv4Address": "172.17.0.3/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
```


5. Under the container's key, you can observe that two containers (`busybox1` and `busybox2`) are listed with information about IP addresses. Since containers are running in the background, attach to the `busybox1` container and try to ping to `busybox2` with its IP address.

```
$ docker attach busybox1
```

```
/ # whoami
root
/ # hostname -i
172.17.0.2
/ # ping 172.17.0.3
PING 172.17.0.3 (172.17.0.3): 56 data bytes
64 bytes from 172.17.0.3: seq=0 ttl=64 time=2.083 ms
64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.144 ms
/ # ping busybox2
ping: bad address 'busybox2'
```

6. Observe that the ping works by passing the IP address of `busybox2` but fails when the container name is passed instead.

The downside with the bridge driver is that it's not recommended for production; the containers communicate via IP address instead of automatic service discovery to resolve an IP address to the container name. Every time you run a container, a different IP address gets assigned to it. It may work well for local development or CI/CD, but it's definitely not a sustainable approach for applications running in production.

Another reason not to use it in production is that it will allow unrelated containers to communicate with each other, which could be a security risk. I'll cover how you can create custom bridge networks later.

The Host Driver

As the name suggests, host drivers use the networking provided by the host machine. And it removes network isolation between the container and the host machine where Docker is running. For example, If you run a container that binds to port 80 and uses host networking, the container's application is available on port 80 on the host's IP address. You can use the host network if you don't want to rely on Docker's networking but instead rely on the host machine networking.

One limitation with the host driver is that it doesn't work on Docker desktop: you need a Linux host to use it. This article focuses on Docker desktop, but I'll show you the commands required to work with the Linux host.

The following command will start an Nginx image and listen to port 80 on the host machine:

```
docker run --rm -d --network host --name my_nginx nginx
```

You can access Nginx by hitting the `http://localhost:80/ url`.

The downside with the host network is that you can't run multiple containers on the same host having the same port. Ports are shared by all containers on the host machine network.

The None Driver

The none network driver does not attach containers to any network. Containers do not access the external network or communicate with other containers. You can use it when you want to disable the networking on a container.

The Overlay Driver

The Overlay driver is for multi-host network communication, as with Docker Swarm or Kubernetes. It allows containers across the host to communicate with each other without worrying about the setup. Think of an overlay network as a distributed virtualized network that's built on top of an existing computer network.

To create an overlay network for Docker Swarm services, use the following command:

```
docker network create -d overlay my-overlay-network
```

To create an overlay network so that standalone containers can communicate with each other, use this command:

```
docker network create -d overlay --attachable my-attachable-overlay
```

The Macvlan Driver

This driver connects Docker containers directly to the physical host network. As per the Docker documentation:

> "Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the `macvlan` driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack."

Macvlan networks are best for legacy applications that need to be modernized by containerizing them and running them on the cloud because they need to be attached to a physical network for performance reasons. A macvlan network is also not supported on Docker desktop for macOS.

Basic Docker Networking Commands

To see which commands list, create, connect, disconnect, inspect, or remove a Docker network, use the `docker network help` command.

```
$ docker network help

Usage:  docker network COMMAND

Manage networks

Commands:
  connect     Connect a container to a network
  create      Create a network
  disconnect  Disconnect a container from a network
  inspect     Display detailed information on one or more networks
  ls          List networks
  prune       Remove all unused networks
  rm          Remove one or more networks
```

Let's run through some examples of each command, starting with `docker network connect`.

Connecting a Container to a Network

Let's try to connect a container to the `mynetwork` we have created. First, we would need a running container that can connect to `mynetwork`.

```
$ docker run -it ubuntu bash
root@0f8d7a833f42:/#
```

Now we have an Ubuntu Linux image and started a login shell as root inside it. We have the container running in interactive mode with the help of the `-it` flags.

```
$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED          STATUS          PORTS     NAMES
0f8d7a833f42   ubuntu    "bash"    9 seconds ago    Up 7 seconds              wizardly_greider
```

Run the `docker network connect 0f8d7a833f42` command to connect the container named `wizardly_greider` with `mynetwork`. To verify that this container is connected to `mynetwork`, use the `docker inspect` command.


```
"mynetwork": {
                    "IPAMConfig": {},
                    "Links": null,
                    "Aliases": [
                        "0f8d7a833f42"
                    ],
                    "NetworkID": "97a158252c995d3632560852c62bd140984769c8714b1f990c8133a5c8ae65d
                    "EndpointID": "db21f395ca781523c115706b11063ebe879cf5ef246c24fd128fe621a582ca
                    "Gateway": "172.20.0.1",
                    "IPAddress": "172.20.0.2",
                    "IPPrefixLen": 16,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "02:42:ac:14:00:02",
                    "DriverOpts": {}
                }
```


Creating a Network

You can use `docker network create mynetwork` to create a Docker network. Here, we've created a network named `mynetwork`. Let's run `docker network ls` to verify that the network is created successfully.

```
$ docker network ls
NETWORK ID     NAME        DRIVER    SCOPE
b995772ac197   bridge      bridge    local
7e25f334b07f   host        host      local
```

```
97a158252c99   mynetwork   bridge    local
475e50be0fe0   none        null      local
```

Now we have a new custom network named `mynetwork`, and its type is bridge.

Disconnecting a Container from the Network

This command disconnects a Docker container from the custom `mynetwork`:

```
docker network disconnect mynetwork 0f8d7a833f42
```

Inspecting the Network

The `docker network inspect` command displays detailed information on one or more networks.

```
$ docker network inspect mynetwork
[
    {
        "Name": "mynetwork",
        "Id": "97a158252c995d3632560852c62bd140984769c8714b1f990c8133a5c8ae65d3",
        "Created": "2021-03-02T17:36:30.090173896Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.20.0.0/16",
                    "Gateway": "172.20.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "0f8d7a833f4283202e905e621e6fd5b29a8e3d4eccc6be6ea0f209f5cb3ca81c": {
                "Name": "wizardly_greider",
                "EndpointID": "db21f395ca781523c115706b11063ebe879cf5ef246c24fd128fe621a582cade",
                "MacAddress": "02:42:ac:14:00:02",
                "IPv4Address": "172.20.0.2/16",
                "IPv6Address": ""
            }
```

```
        },
        "Options": {},
        "Labels": {}
    }
]
```

List Available Networks

Docker installation comes with three networks: none, bridge, and host. You can verify this by running the command `docker network ls`:

```
docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
b995772ac197   bridge    bridge    local
7e25f334b07f   host      host      local
475e50be0fe0   none      null      local
```

Removing a Network

The following are the Docker commands to remove a specific or all available networks:

```
$ docker network rm mynetwork
mynetwork
$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
b995772ac197   bridge    bridge    local
7e25f334b07f   host      host      local
475e50be0fe0   none      null      local
```

```
$ docker network prune
WARNING! This will remove all custom networks not used by at least one container.
Are you sure you want to continue? [y/N]
```

Public Networking

Let's talk about how to publish a container port and IP addresses to the outside world. When you start a container using the `docker run` command, none of its ports are exposed. Your Docker container can connect to the outside world, but the outside world cannot connect to the container. To make the ports accessible for external use or with other containers not on the same network, you will have to use the `-P` (publish all available ports) or `-p` (publish specific ports) flag.

For example, here we've mapped the TCP port 80 of the container to port 8080 on the Docker host:

```
docker run -it --rm nginx -p 8080:80
```

Here, we've mapped container TCP port 80 to port 8080 on the Docker host for connections to host IP 192.168.1.100:

```
docker run -p 192.168.1.100:8085:80 nginx
```

You can verify this by running the following curl command:

```
$ curl 192.168.1.100:8085
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Let me briefly mention DNS configuration for containers. Docker provides your containers with the ability to make basic name resolutions:

```
$ docker exec busybox2 ping www.google.com
PING www.google.com (216.58.216.196): 56 data bytes
64 bytes from 216.58.216.196: seq=0 ttl=37 time=9.672 ms
64 bytes from 216.58.216.196: seq=1 ttl=37 time=6.110 ms
$ ping www.google.com
PING www.google.com (216.58.216.196): 56 data bytes
64 bytes from 216.58.216.196: icmp_seq=0 ttl=118 time=4.722 ms
```

Docker containers inherit DNS settings from the host when using a bridge network, so the container will resolve DNS names just like the host by default. To add custom host records to your container, you'll need to use the relevant `--dns*` flags outlined here.

Docker Compose Networking

Docker Compose is a tool for running multi-container applications on Docker, which are defined using the compose YAML file. You can start your applications with a single command: `docker-compose up`.

By default, Docker Compose creates a single network for each container defined in the compose file. All the containers defined in the compose file connect and communicate through the default network.

If you're not sure about the commands supported with Docker Compose, you can run the following:

```
$ docker compose help
Docker Compose

Usage:
  docker compose [command]

Available Commands:
  build       Build or rebuild services
  convert     Converts the compose file to a cloud format (default: cloudformation)
  down        Stop and remove containers, networks
  logs        View output from containers
  ls          List running compose projects
  ps          List containers
  pull        Pull service images
  push        Push service images
  run         Run a one-off command on a service.
  up          Create and start containers

Flags:
  -h, --help   help for compose

Global Flags:
      --config DIRECTORY   Location of the client config files DIRECTORY (default "/Users/ashish/
  -c, --context string     context
  -D, --debug              Enable debug output in the logs
  -H, --host string        Daemon socket(s) to connect to

Use "docker compose [command] --help" for more information about a command.
```

Let's understand this with an example. In the following `docker-compose.yaml` file, we have a WordPress and a MySQL image.

When deploying this setup, `docker-compose` maps the WordPress container port 80 to port 80 of the host as specified in the compose file. We haven't defined any custom network, so it should create one for you. Run `docker-compose up -d` to bring up the services defined in the YAML file:

```
version: '3.7'
services:
  db:
    image: mysql:8.0.19
    command: '--default-authentication-plugin=mysql_native_password'
    restart: always
```

```
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      - MYSQL_ROOT_PASSWORD=somewordpress
      - MYSQL_DATABASE=wordpress
      - MYSQL_USER=wordpress
      - MYSQL_PASSWORD=wordpress
  wordpress:
    image: wordpress:latest
    ports:
      - 80:80
    restart: always
    environment:
      - WORDPRESS_DB_HOST=db
      - WORDPRESS_DB_USER=wordpress
      - WORDPRESS_DB_PASSWORD=wordpress
      - WORDPRESS_DB_NAME=wordpress
volumes:
  db_data:
```

As you can see in the following output, a network named `downloads_default` is created for you:

```
$ docker-compose up -d
Creating network "downloads_default" with the default driver
Creating volume "downloads_db_data" with default driver
Pulling db (mysql:8.0.19)...
```

Verify that we have two containers up and running with the `docker ps` command:

```
$ docker ps
CONTAINER ID   IMAGE              COMMAND                  CREATED         STATUS         PORTS
f68265cd6219   wordpress:latest   "docker-entrypoint.s..."   6 minutes ago   Up 6 minutes   0.0.0
2838f5586c73   mysql:8.0.19       "docker-entrypoint.s..."   6 minutes ago   Up 6 minutes   3306/
```

Navigate to `http://localhost:80` in your web browser to access WordPress.

Now let's inspect this network with the `docker network inspect` command. The following is the output:

```
$ docker network inspect downloads_default
[
    {
        "Name": "downloads_default",
        "Id": "717ea814aae357ceca3972342a64335a0c910455abf160ed820018b8d6690383",
        "Created": "2021-03-05T03:43:42.541707419Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
```

```
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": true,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "2838f5586c735894051498c8ed0e5e103209cd22ee5718cbb29e8c6d169c9bf8": {
                "Name": "downloads_db_1",
                "EndpointID": "10033e064387892253d69ac5813be6bc820a95df1a3e19f84eff99a8d55af1bd",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            },
            "f68265cd6219fb60491c7ebbdae2d7f4c5ea4a74aec9987f5a5082c13b1ec025": {
                "Name": "downloads_wordpress_1",
                "EndpointID": "a4a673479ab3d812713955d461cc4d7032aba3806b49f50dd783a8083588aec5",
                "MacAddress": "02:42:ac:12:00:03",
                "IPv4Address": "172.18.0.3/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {
            "com.docker.compose.network": "default",
            "com.docker.compose.project": "downloads",
            "com.docker.compose.version": "1.27.4"
        }
    }
]
```


In the container sections, you can see that two containers (`downloads_db_1` and `downloads_wordpress_1`) are attached to the default `downloads_default` network driver, which is the bridge type. Run the following commands to clean up everything:

```
$ docker-compose down
Stopping downloads_wordpress_1 ... done
```

```
Stopping downloads_db_1        ... done
Removing downloads_wordpress_1 ... done
Removing downloads_db_1        ... done
Removing network downloads_default
```

You can observe that the network created by Compose is deleted, too:

```
$ docker-compose down -v
Removing network downloads_default
WARNING: Network downloads_default not found.
Removing volume downloads_db_data
```

The volume created earlier is deleted, and since the network is already deleted after running the previous command, it shows a warning that the default network is not found. That's fine.

The example we've looked at so far covers the default network created by Compose, but what if we want to create our custom network and connect services to it? You will define the user-defined networks using the Compose file. The following is the `docker-compose` YAML file:

```
version: '3.7'
services:
  db:
    image: mysql:8.0.19
    command: '--default-authentication-plugin=mysql_native_password'
    restart: always
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    networks:
      - mynetwork
    environment:
      - MYSQL_ROOT_PASSWORD=somewordpress
      - MYSQL_DATABASE=wordpress
      - MYSQL_USER=wordpress
      - MYSQL_PASSWORD=wordpress
  wordpress:
    image: wordpress:latest
    ports:
      - 80:80
    networks:
      - mynetwork
    restart: always
    environment:
      - WORDPRESS_DB_HOST=db
      - WORDPRESS_DB_USER=wordpress
      - WORDPRESS_DB_PASSWORD=wordpress
      - WORDPRESS_DB_NAME=wordpress
volumes:
  db_data:
```

```
networks:
  mynetwork:
```

I've defined a user-defined network under the top-level networks section at the end of the file and called the network `mynetwork`. It's a bridge type, meaning it's a network on the host machine separated from the rest of the host network stack. Following each service, I added the network key to specify that these services should connect to `mynetwork`.

Let's bring up the services again after the changing the Docker Compose YAML file:

```
$ docker-compose up -d
Creating network "downloads_mynetwork" with the default driver
Creating volume "downloads_db_data" with default driver
Creating downloads_wordpress_1 ... done
Creating downloads_db_1        ... done
```

As you can see, Docker Compose has created the new custom `mynetwork`, started the containers, and connected them to the custom network. You can inspect it by using the Docker inspect command:

```
$ docker network inspect downloads_mynetwork
[
    {
        "Name": "downloads_mynetwork",
        "Id": "cb24ed3832dfdd34ca6fdcfcf0065c8e0df6b9b72f7b29a2aaada74970835dd1",
        "Created": "2021-03-05T04:23:14.354570267Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.19.0.0/16",
                    "Gateway": "172.19.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": true,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "334bd5bf1689b067fa24705af0b6ab444976b288d25846349ce5b8f6914d8c19": {
                "Name": "downloads_wordpress_1",
```

```
                "EndpointID": "10674d2ddd9feb67c98e3c08fcf451f32bda58e96c9c256aee70a0487f6a4496",
                "MacAddress": "02:42:ac:13:00:03",
                "IPv4Address": "172.19.0.3/16",
                "IPv6Address": ""
            },
            "9ffc94adab6896620648a1d08d215ba3d9423fee934b905b7bc2a44dd30bd74c": {
                "Name": "downloads_db_1",
                "EndpointID": "927943a0202bfb69692bc172a5c26e05676df6961236596eba3c3464cacbd1ab",
                "MacAddress": "02:42:ac:13:00:02",
                "IPv4Address": "172.19.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {
            "com.docker.compose.network": "mynetwork",
            "com.docker.compose.project": "downloads",
            "com.docker.compose.version": "1.27.4"
        }
    }
]
```


Inspecting the network, you can see there are now two containers connected to the custom network.

Conclusion

In this article, we've covered the what and how of Docker networking in detail, starting with Docker's network drivers available out-of-the-box and then some advanced concepts such as overlay and macvlan. We ran through some examples of the most common Docker network commands, and then discussed some common use cases and general pitfalls of the available network drivers. We also covered port publishing, which allows the outside world to connect with containers, and how Docker resolves DNS names. Finally, we explored Docker Compose networking with some examples.

That should provide you with a decent overview of how Docker networking provides different modes of network drivers so that your containers can communicate on a single or multi-host setup. With this knowledge, you can pick and choose a network driver that fits your use case.

The Complete Guide to Docker Secrets

Even if you've used Docker for your smaller or locally developed software, you might find that it can be daunting for more complex tasks. This can especially be true for secrets management and sharing—areas often overlooked when working with containerized applications.

There isn't a standardized approach for accessing and managing secrets in containers, which has resulted in homegrown or inadequate solutions better geared for more static environments. Fortunately, the Docker ecosystem offers a great option with Docker secrets.

This article will explain the benefits of using Docker secrets. You will learn how to set up Docker Swarm and leverage Docker secrets as part of your development flow.

What Is Docker Secrets?

Docker secrets is Docker's secrets management service, offered as part of its container orchestration stack. In Docker, a secret is any piece of data, like passwords, SSH private credentials, certificates, or API keys, that shouldn't be stored unencrypted in plain text files. Docker secrets automates the process of keeping this data secure.

What Is Docker Swarm?

Docker Swarm is a container orchestration tool that allows the management of containers across multiple host machines. It works by clustering a group of machines together; once they are in the group, you can run Docker commands as you normally would.

To use secrets on your Docker container and through Docker Compose, you will need to make sure that you are running your Docker Engine in Swarm mode.

Why You Need Secrets Management

Secrets management is an important aspect of container security and for any application handling configuration variables, SSH keys, passwords, API tokens, private certificates, or other data that shouldn't be accessible to anyone outside of your organization.

Secrets can be used to prove a user's identity and to authenticate and authorize the user to access applications and services. Once you start running multiple instances of your containerized applications, you need to keep, synchronize, and rotate all secrets.

A common use case is persisting and prepopulating sensitive data in our containers (for example, database credentials) that might change between environments; another common use case in a microservice architecture is sharing a known secure key or token to authenticate communication between services.

There are several commonly used options for managing secrets.

Stored in Docker Compose and Stack Files

Often, `docker-compose.yml` files look something like this:

```
version: '3'

services:

  my_database:
    container_name: my_database
    hostname: my_database
    image: postgres
    volumes:
      - ./volume:/var/lib/postgresql
    environment:
      - POSTGRES_DB=mydb, mydb_dev
      - POSTGRES_USER=notsecure
      - POSTGRES_PASSWORD=aStrongPassword
    ports:
      - 54321:5432
    restart: unless-stopped
```

Unfortunately, this is a common mistake, as this will allow the sensitive information needed by the container to be committed to version control, making it easily accessible by anyone with access to the repository or the file.

Embedded Into Docker Images

Container images should be both reusable and secure. Creating images with embedded configuration or secrets breaks these principles and leads to a multitude of potential problems:

- If you copy files with sensitive information into the image, they're accessible through a previous layer even if a file is later removed.
- If the image is reliant on external files for configuration, this causes the image to couple with the configuration and breaks the reusability principle.

Stored in Environment Variables

Storing configuration in environment variables is common practice and recommended in some situations, following the Twelve-Factor App methodology. Unfortunately, it is also common practice to use `.env` variables for storing sensitive information. There are a few downsides to this:

- Secrets stored in an environment variable are more vulnerable to accidental exposure.

- These variables are available to all processes, and it can be difficult to track access.
- Applications might accidentally print the entire collection of `.env` variables during debugging.
- Secrets can be shared with subprocesses with little oversight.

Docker Secrets to the Rescue

As you can see, the above options can potentially compromise your security. Docker secrets offers a solution. It provides the following advantages:

- Secrets are always encrypted, both in transit and at rest.
- Secrets are hard to unintentionally leak by the consuming service.
- Secrets access follows the principle of least privilege.

You're going to set up Docker Swarm with Docker secrets. For this tutorial, make sure of the following:

- Docker client and daemon versions are at least 1.25. Run `docker version` to confirm:
- Docker is running on Swarm mode. Run `docker info` to confirm:

Enabling Swarm Mode

Swarm mode is not enabled by default, so you will need to initialize your machine by running the following command:

```
docker swarm init
```

Running this command turns your local machine into a Swarm manager.

There are a few concepts you'll need to know when working with Docker Swarm:

- **Node:** An instance of the Docker engine connected to Swarm. Nodes are either managers or workers. Managers schedule which containers run while workers execute tasks, and by default, managers are also workers.
- **Services:** A collection of tasks to be executed by workers.

Creating Your First Secret

Now that you're in Swarm mode, create a sample secret:

```
openssl rand -base64 128 | docker secret create secure-key -
```

Pass your secret to a new service:

```
docker service create --secret="secure-key" redis:alpine
```

To make use of the secret, your application should read the contents from the `in-memory`, the temporary filesystem created under `/run/secrets/secure-key`:

```
> cat /run/secrets/secure-key
Wsjmn/7cqixYLH8hABc8fTuv5/oeki2+5Hn4NzVUdNEQquSUfaDJT/80vh0MA1hl
uTCL504xjCEqogq5xFfLNPupKz9isUAESMCkc0nhGb39UZbt3Rk+Qk+J6M3xBSEe
VzgvNfjLkvk4nJqGfyYIx0mxj7zgLmL2NzQzzLEGhPg=
```

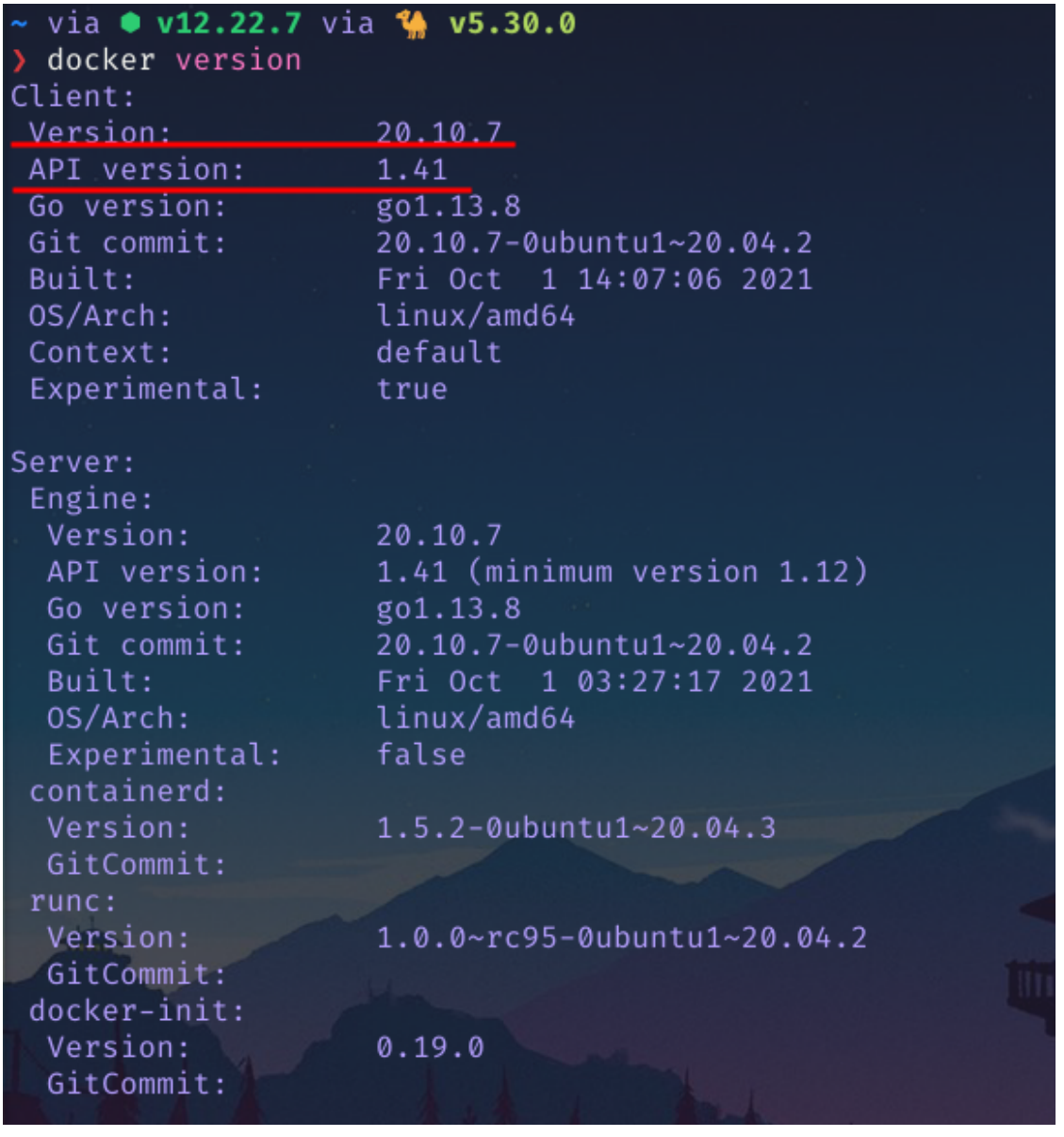


Figure 6: Docker version

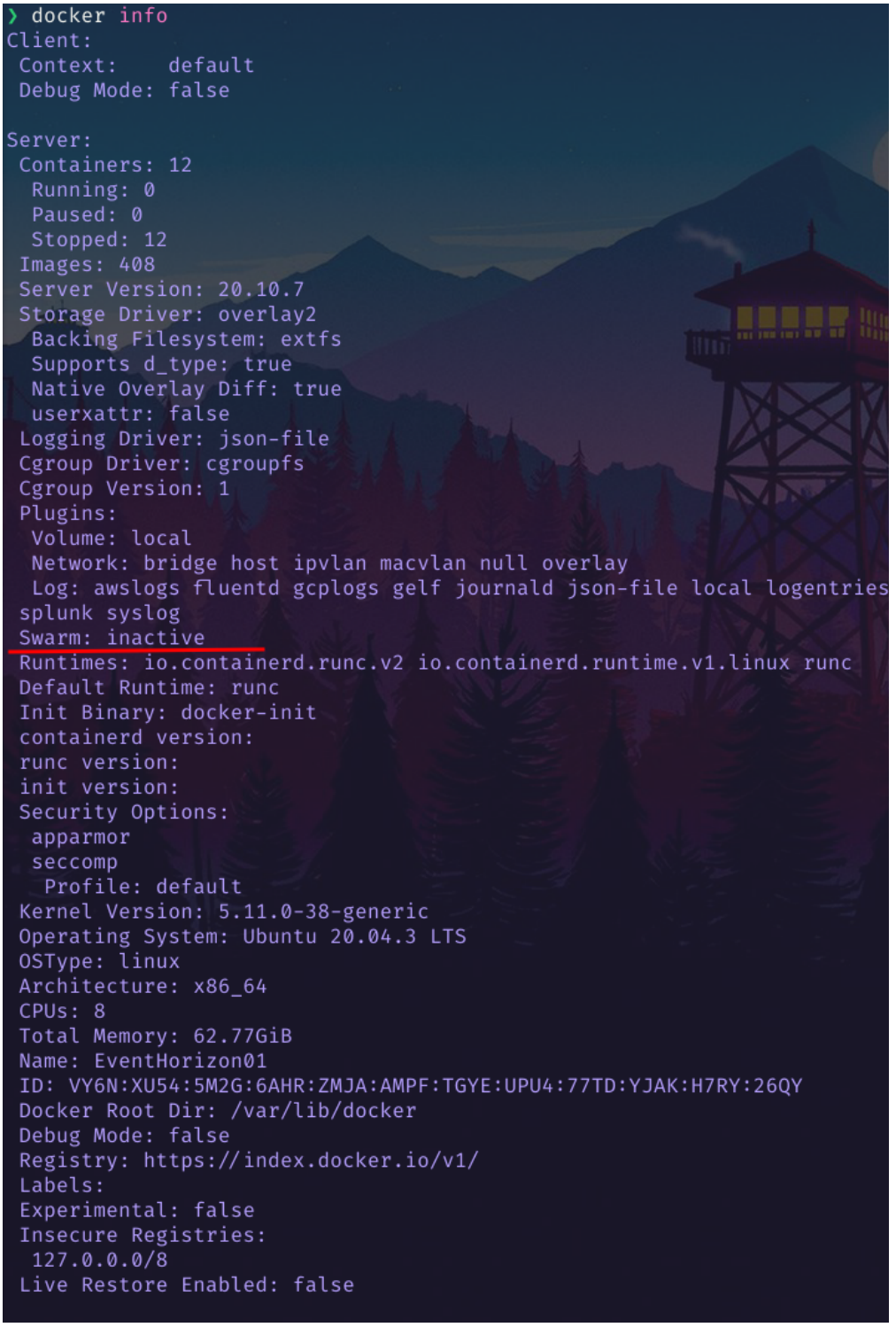


Figure 7: Docker Swarm

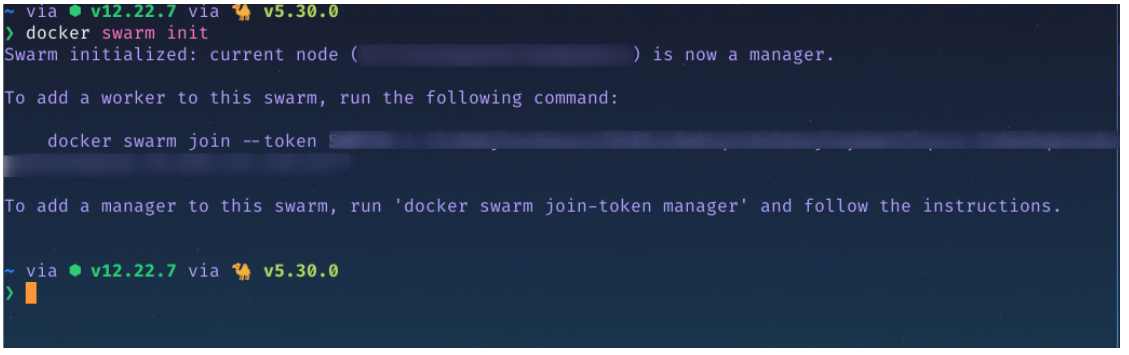


Figure 8: Docker Swarm init

All containers in that service can freely access the secret, making it automatically available on the appropriate hosts. Remember, services might be associated with one or more containerized applications; this makes Docker secrets ideal for shared secrets and API credentials.

About Swarm and Secrets

There are a few more things to note about using Docker Swarm for secrets, according to the documentation:

- A service's access to secrets can be allowed or revoked at any time.
- A newly created or running service can be granted access to a secret; then the decrypted secret is mounted into the container in an in-memory filesystem.
- A node only has access to secrets if the node is a swarm manager or running service tasks that have been granted access to the secret.
- A container task can stop running; then the decrypted secrets shared to it are unmounted from that container's filesystem and deleted from the node's memory.
- A secret that's being used by a running service can't be removed.

Using Secrets With Compose

For a more practical method, take a look at the official `docker-compose.yml` example, which leverages the use of Docker secrets in a WordPress site:

```
version: "3.9"

services:
   db:
     image: mysql:latest
     volumes:
       - db_data:/var/lib/mysql
     environment:
       MYSQL_ROOT_PASSWORD_FILE: /run/secrets/db_root_password
       MYSQL_DATABASE: wordpress
```

```
      MYSQL_USER: wordpress
      MYSQL_PASSWORD_FILE: /run/secrets/db_password
    secrets:
      - db_root_password
      - db_password

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    ports:
      - "8000:80"
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD_FILE: /run/secrets/db_password
    secrets:
      - db_password


secrets:
  db_password:
    file: db_password.txt
  db_root_password:
    file: db_root_password.txt

volumes:
    db_data:
```

Let's break down the above file. Here is what's happening:

- The `secrets` line under each service defines the Docker secrets you want to inject into the specific container.
- The main `secrets` segment defines the variables `db_password` and `db_root_password` and a file that should be used to populate their values.
- The deployment of each container means Docker creates a temporary filesystem mount under `/run/secrets/<secret_name>` with their specific values.

Unlike the other methods, this guarantees that secrets are only available to the services that have been explicitly granted access, and secrets only exist in memory while that service is running.

Conclusion

You should now be familiar with some of the most common mistakes developers make when creating containerized applications with secret or sensitive information. Recognizing and avoiding these mistakes will help you keep your applications secure.

Another way to avoid these mistakes is to use secrets management with Docker. Sensitive data is always immutable, never written to disk, and never sent in clear-text format over the network. If

you plan to leverage Docker Swarm in production, you should also use Docker secrets for local development.

Understanding Docker Logging and Log Files

Docker logging and its management are an important part of the containerization of your application. Once you've deployed your application, logging is one of the best tools to help reveal errors, aid in debugging, and optimize your application's performance.

With that in mind, let's dive into Docker logging and its log files.

Introduction To Docker Logging

Developers, DevOps professionals, and product stakeholders all apply knowledge gained from logging to improve a system's performance and reliability. Much of what an application, server, or OS does should be recorded in the logs and aggregated in an easily accessible location. A log analysis then uses all the log events and audit trails to help chalk out a clear picture of events that happen across your application.

In Docker, containers are isolated and bundled with software, libraries, and configuration files. In a traditional single-server setup log analysis is centralized on a single node, but with a stateless containerized setup logging becomes more complex. Why? Two reasons:

1. Containers are transient. When a Docker container broadcasts logs, it sends them to the application's `stdout` and `stderr` output streams. The underlying container logging driver can start accessing these streams, and the logs are stored on the Docker host in JSON files (`json-file` is the default logging driver used by Docker). It writes JSON-formatted logs to a file on the host where the container is running. Any logs stored in the container will be deleted when it is terminated or shut down.

2. Containers are multi-leveled. There are two levels of aggregation in Docker logging. One refers to the logs from inside the container in your Dockerized application, and the second refers to the logs from the host servers (that is system logs or Docker daemon logs), which are generally located in `/var/log`. A log aggregator that has access to the host pulls application log files and accesses the file system inside the container to collect the logs. Later, you'd need to correlate these log events for analysis.

In this article, you'll learn about different logging strategies you can use in a Dockerized application—how you can access logs and understand Docker logging commands, drivers, configuration, and management to build a highly performant and reliable infrastructure.

Docker Logging Commands

`docker logs` is a command that shows all the information logged by a running container. The `docker service logs` command shows information logged by all the containers participating in a service. By default, the output of these commands, as it would appear if you run the command in a terminal, opens up three I/O streams: stdin, stdout, and stderr. And the default is set to show only stdout and stdout.

- `stdin` is the command's input stream, which may include input from the keyboard or input from another command.
- `stdout` is usually a command's normal output.
- `stderr` is typically used to output error messages.

The `docker logs` command may not be useful in cases when a logging driver is configured to send logs to a file, database, or an external host/backend, or if the image is configured to send logs to a file instead of stdout and stderr. With `docker logs <CONTAINER_ID>`, you can see all the logs broadcast by a specific container identified by a unique ID.

```
$ docker run --name test -d busybox sh -c "while true; do $(echo date); sleep 1; done"
$ date
Tue 06 Feb 2020 00:00:00 UTC
$ docker logs -f --until=2s test
Tue 06 Feb 2020 00:00:00 UTC
Tue 06 Feb 2020 00:00:01 UTC
Tue 06 Feb 2020 00:00:02 UTC
```

`docker logs` works a bit differently in community and enterprise versions. In Docker Enterprise, `docker logs` read logs created by any logging driver, but in Docker CE, it can only read logs created by the json-file, local, and journald drivers.

For example, here's the JSON log created by the hello-world Docker image using the json-file driver:

```
{"log":"Hello from Docker!\n","stream":"stdout","time":"2021-02-10T00:00:00.000000000Z"}
```

As you can see, the log follows a pattern of printing:

- Log's origin
- Either `stdout` or `stderr`
- A timestamp

You can find this log in your Docker host at:

`/var/lib/docker/containers/<container id>/<container id>-json.log`

These Docker logs are stored in a host container and will build up over time. To address that, you can implement log rotation, which will remove a chunk of logs at specified intervals, and a log aggregator, which can be used to push them into a centralized location for a permanent log repository. You can use this repository for further analysis and improvements in the system down the road.

To find `container_id`, run the `docker ps` command. It'll list all the running containers.

```
$ docker ps
CONTAINER ID   IMAGE       COMMAND             CREATED             STATUS              PORTS
cg95e1yqk810   bar_image  "node index.js"     Y min ago           Up Y min            80/tcp
```

Then, the `docker logs container_id` lists the logs for a particular container.

```
docker logs <container_id>
```

If you want to follow the Docker container logs:

```
docker logs <container_id> -f
```

If you want to see the last N log lines:

```
docker logs <container_id> --tail N
```

If you only want to see any specific logs, use `grep`:

```
docker logs <container_id> | grep node
```

If you want to check errors:

```
docker logs <container_id> | grep -i error
```

Logging Drivers Supported by Docker

Logging drivers, or log-drivers, are mechanisms for getting information from running containers and services, and Docker has lots of them available for you. Every Docker daemon has a default logging driver, which each container uses. Docker uses the json-file logging driver as its default driver.

Currently, Docker supports the following logging drivers:

Driver	Description
none	No logs are available for the container and Docker logs do not return any output.
local	Logs are stored in a custom format designed for minimal overhead.
json-file	Logs are formatted as JSON. The default logging driver for Docker.
syslog	Writes logging messages to the Syslog facility. The Syslog daemon must be running on the host machine.
journald	Writes log messages to journald. The journald daemon must be running on the host machine.
gcplogs	Writes log messages to Google Cloud Platform (GCP) logging.
awslogs	Writes log messages to Amazon CloudWatch logs.
splunk	Writes log messages to Splunk using the HTTP Event Collector.

Driver	Description
gelf	Writes log messages to a Graylog Extended Log Format (GELF) endpoint, such as Graylog or Logstash.
fluentd	Writes log messages to fluentd (forward input). The fluentd daemon must be running on the host machine.
etwlogs	Writes log messages as Event Tracing for Windows (ETW) events. Only available on Windows platforms.
logentries	Writes log messages to Rapid7 Logentries.

Configuring the Logging Driver

To configure the Docker daemon to a logging driver, you need to set the value of `log-driver` to the name of the logging driver in the daemon.json configuration file. Then you need to restart Docker for the changes to take effect for all the newly created containers. All the existing containers will remain as they are.

For example, let's set up a default logging driver with some additional options.

```
{
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "25m",
    "max-file": "10",
    "labels": "production_bind",
    "env": "os,customer"
  }
}
```

To find the current logging driver for the Docker daemon:

```
  {% raw %}
$ docker info --format '{{.LoggingDriver}}'

json-file
{% endraw %}
```

You can override the default driver by adding the `--log-driver` option to the docker run command that creates a container.

The following command will start using the Splunk driver:

```
docker run --log-driver splunk httpd
```

Modify the `daemon.json` file, to pass the address of the Splunk host to the driver:

```
{
    "log-driver": "splunk",
    "log-opts": {
        "splunk-url": "172.1.1.1:11111"
```

```
    }
}
```

```
{% raw %}
$ docker inspect -f '{{.HostConfig.LogConfig.Type}}' <CONTAINER>

splunk
{% endraw %}
```

Choosing the Delivery Mode of Log Messages From Container to Log Driver

Docker provides two modes for delivering log messages:

- **Blocking (default):** The container interrupts the application every time it needs to deliver a message to the driver. This will add latency in the performance of your application (for some drivers), but it'll guarantee that all the messages will be sent to the driver. Case in point, the json-file driver writes logs very quickly, therefore it's unlikely to cause latency. But drivers like `gcplogs` and `awslogs` open a connection to a remote server and so are more likely to block and cause latency.
- **Non-Blocking:** The container writes and stores its logs to an in-memory ring buffer until the logging driver is available to process them. This ensures that a high rate of logging will not affect application performance. It does not guarantee that all the messages will be sent to the driver. In cases where log broadcasts are faster than the driver processor, the ring buffer will soon run out of space. As a result, buffered logs are deleted to make space for the next set of incoming logs. The default value for `max-buffer-size` is 1 MB.

To change the mode:

```
{
    "log-driver": "splunk",
    "log-opts": {
        "splunk-url": "172.1.1.1:11111",
        "mode": "non-blocking"
    }
}
```

Understanding Logging Strategies

Docker logging effectively means logging the events of an application, host OS, and the respective Docker service. There are a few defined logging strategies you should keep in mind when working with containerized apps.

1. Application Logging

In this logging strategy, the application running in the containers will have its own logging framework. For example, a Node.js app could use a `winston` library to format and send the logs. With this approach, you have complete control over the logging event.

If you have multiple containers, you need to add an identifier at each container level, so you can uniquely determine the container and its respective logs. But as the size of the logs increases, this will start creating a load on the application process. Due to the transient nature of containers, the logs will be wiped out when a container is terminated or shut down.

To address this, you have two options:

- Configure steady storage to hold these logs, for example, disks/data volumes.
- Forward these logs to a log management solution.

2. Data Volumes

In this logging strategy, your container's directory gets links to one of the host machine directories that will hold all the data for you. Containers can now be terminated or shut down, and access logs from multiple containers.

A regular backup is a good idea in order to prevent data corruption or loss in case of a failure. But if you want to shift the containers on different hosts without loss of data, then the data volumes strategy will make that difficult.

3. Docker Logging Driver

Docker has a bunch of logging drivers that can be used in your logging strategy. The configured driver reads the data broadcast by the container's `stdout` or `stderr` streams and writes it to a file on the host machine. By default, the host machine holds the log files, but you can forward these events using the available drivers, like Fluentd, Splunk, and awslogs.

Note that the `docker log` command will not work if you use anything other than the `json-file` driver. And if log forwarding over a TCP connection fails or becomes unreachable, the containers will shut down.

4. Dedicated Logging Container

Here, you need to set up a dedicated container whose only job is to collect and manage logs within a Docker ecosystem. It'll aggregate, monitor, and analyze logs from containers and forward them to a central repository. The log dependency on the host machine is no longer an issue, and it's best suited for microservices architecture.

This strategy gives you the freedom to:

- Move containers between the hosts.
- Scale up by just adding a new container.
- Tie up all the various streams of log events.

5. Sidecar Logging Container

This logging strategy is similar to dedicated logging in that you have a container to hold logs, but here, each application has its own dedicated logging container (note that the application and its log container can be considered as a single unit). This opens up the flexibility of app-level logging customization. By adding custom identifiers, you can identify the specific container that broadcasts the log.

This is best suited for a complex system where each entity customizes the logs, making it popular in microservices deployment. Maintaining a container per application level will require additional resources for management and setup, so it's complex to implement. Also, you may end up losing data when a container from the unit becomes unserviceable.

6. Third-Party Logging Services

There are a lot of third-party logging services you can use according to your infrastructure and application needs, enabling you to aggregate, manage, and analyze logs and take proactive preventive actions as a result.

- **Splunk:** Splunk is built to scale from a single server to multiple data centers. It has built-in alerting and reporting, real-time search, analysis, and visualization, which helps you take action against malfunctions faster. It comes with a Splunk Enterprise on-premise deployment and Splunk Cloud. The free plan comes with 500 MB data per day, and the paid plan starts from $150 a month for a GB.
- **Logstash:** Logstash is part of the Elastic Stack along with Beats, Elasticsearch, and Kibana. It's a server-side data processing pipeline that ingests data from a multitude of sources simultaneously, transforms it, and then sends it to your favourite "stash." Logstash has over 200 plugins for input, filter/transform, and output. It's free and open source.
- **Fluentd:** Fluentd is an open-source data collector. Its performance has been well proven, handling 5 TB of daily data, 50,000 messages/sec at peak time. The main reason to use it would be performance. It's free and open source.
- **PaperTrail:** PaperTrail is a simple and user-friendly service that provides a terminal-like experience. Once the data is sent over the syslog, you can perform tail and search operations with the PaperTrail UI. It's affordable for low volumes but becomes expensive for higher volume. It comes with a free 50 MB/month plan, and paid plans start at $7/month for 1 GB per month.
- **Loggly:** Loggly focuses on simplicity and ease of use for a DevOps audience. It's one of the more robust log analyzers. Primary use cases are for troubleshooting and customer support scenarios, and it provides richer visualizations and more parsing functionality than PaperTrail. It comes with a free 200 MB/day, and paid plans start at $79 a month for 1 GB per day ingestion with a 2 weeks retention.

Limitations and Challenges of Docker Logging

Docker makes containerized application deployment easier, faster, and streamlines it with limited resources. Log management and analysis are handled differently with Docker as compared to a traditional system, which introduces a new set of challenges and limitations. From storing logs in a file system to forwarding them to a central repository, Docker logging has some pain points you'll need to overcome with a deeper understanding of Docker capability.

The Only Compatible Driver Is json-file

The json-file driver is the only one that works with the `docker logs` command, a limitation of Docker logs. When you start using any other logging drivers, such as Fluentd or Splunk, the `docker logs` command shows an error and the Docker logs API calls will fail. Also, you won't be able to check the container logs in this situation.

Docker Syslog Impacts Container Deployment

The reliable way to deliver logs is via Docker Syslog with TCP or TLS. But this driver needs an established TCP connection to the Syslog server whenever a container starts up. And the container will fail if a connection is not made.

```
docker: Error response from daemon: Failed to initialize logging driver: dial tcp
```

Your container deployment will be affected if you face network problems or latency issues. And it's not recommended that you restart the Syslog server because this will drop all the connections from the containers to a central Syslog server.

Potential Loss of Logs with Docker Syslog Driver

The Docker syslog driver needs an established TCP or TLS connection to deliver logs. Note that when the connection is down or not reachable, you'll start losing the logs until the connection reestablished.

Multi-Line Logs Not Supported

Generally, either of two patterns is followed for logging: single-line per log or multiple lines with extended information per log, like stack traces or exceptions. But with Docker logging, this is a moot point, because containers always broadcast logs to the same output: `stdout`.

Multiple Logging Drivers Not Supported

It's mandatory that you use only a single driver for all of your logging needs. Scenarios where you can store logs locally and push it to remote servers are not supported.

Logs Can Miss or Skip

There's a rate limitation setting at the Docker end for journald drivers that takes care of the rate that logs get pushed. If it exceeds, then the driver might skip some logs. To prevent such issues, increase the rate limitation settings according to your logging needs.

Conclusion

While Docker containerization allows developers to encapsulate a program and its file system into a single portable package, that certainly doesn't mean containerization is free of maintenance. Docker Logging is a bit more complex than traditional methods, so teams using Docker must familiarize themselves with the Docker logging to support full-stack visibility, troubleshooting, performance improvements, root cause analysis, etc.

As we have seen in this post, to facilitate logging, Docker offers logging drivers and commands in the platform which gives you the mechanisms for accessing the performance data and also provides plugins to integrate with third-party logging tools as well. To maximize the logging capabilities there are several methods and strategies which help in designing your logging infrastructure, but each comes with its advantages and disadvantages.

Understanding Docker Multistage Builds

At first glance, writing Dockerfiles appears to be a straightforward process. After all, most basic examples reflect the same set of steps. However, not all Dockerfiles are created equal. There is an optimal way of writing these files to produce the kind of Docker images you want for your final product. If you were to pop the hood, you'd see that Docker images actually consist of file system layers that correlate to the individual build steps involved in the creation of the image.

To build an efficient Docker image, you want to eliminate some of these layers from the final output. Optimizing an image can take a lot of effort as you attempt to filter out what you don't need. Building these kinds of images has a lot to do with keeping them as small as possible, because large images lead to a host of other issues.

One approach to keeping Docker images small is using multistage builds. A multistage build allows you to use multiple images to build a final product. In a multistage build, you have a single Dockerfile, but can define multiple images inside it to help build the final image.

In this post, you'll learn about the core concepts of multistage builds in Docker and how they help to create production-grade images. In addition, I will detail how you can create these types of files, as well as highlight some challenges that they present. I'll end with a better way to do multi-stage builds using Earthly.

(For The Impatient: Skip to 'Better Multi-Stage Builds')

Core Concepts of Docker Multistage Builds

Before getting to the details of multistage builds, it's good to have an understanding of the main idea behind them. You're already familiar with the starting point of container images, the Dockerfile. Dockerfiles are text files that make it easy to assemble all the relevant commands required to create your container image. These commands in the Dockerfile should be combined whenever possible for the sake of optimization.

When I first came across this idea of building the right kind of Docker image, I asked myself a few questions that may be crossing your mind as you read this. What are the implications of *not* optimizing an image? Can it really have that much of a negative impact on your application? The answer to the latter question is yes.

As for the implications, you typically end up with bigger images. The reason big images should be avoided is because they increase both potential security vulnerabilities and the surface area for

attack. You definitely want to keep things lean by ensuring you only have what your application needs to run successfully in a production environment.

The Old Way: Builder Pattern

One way of reducing the size of your Docker images is through the use of what is informally known as the builder pattern. The builder pattern uses two Docker images to create a base image for building assets and the second to run it. This pattern was previously implemented through the use of multiple Dockerfiles. It has become an uncommon practice since the introduction and support of multistage builds.

For context, it's good to understand how this was typically done before multistage builds. The following example makes use of a basic React application that is first built and then has its static content served by an Nginx virtual server. Following are the two Dockerfiles used to create the optimized image. In addition, you'll see a shell script that demonstrates the Docker CLI commands that have to be run in order to achieve this outcome. You can find the source code for this example in this repository.

```
FROM node:12.13.0-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
```

```
FROM nginx
EXPOSE 3000
COPY ./nginx/default.conf /etc/nginx/conf.d/default.conf
COPY /app/build /usr/share/nginx/html
```

```
#!/bin/sh
echo Building lukondefmwila/react:build
docker build -t lukondefmwila:build . -f Dockerfile.build
docker create --name extract lukondefmwila:build
docker cp extract:/app/build ./app
docker rm -f extract

echo Building lukondefmwila/react:latest
docker build --no-cache -t lukondefmwila/react:latest . -f Dockerfile.main
```

While using the builder pattern does give you the desired outcome, it presents additional challenges. This process introduces the management overhead that comes with maintaining multiple Dockerfiles—not to mention the cumbersome procedure of running through several Docker CLI commands, even if this can be streamlined by a shell script.

The Next Way: Docker Multistage Builds

Now that you get the underlying concept, turn your attention to how this translates to the modern implementation of the builder pattern. What the former approach accomplishes with multiple

Dockerfiles, the multistage feature does in one. You can get the same results with your builds without the added complexity.

Multistage builds make use of one Dockerfile with multiple FROM instructions. Each of these FROM instructions is a new build stage that can COPY artifacts from the previous stages. By going and copying the build artifact from the build stage, you eliminate all the intermediate steps such as downloading of code, installing dependencies, and testing. All these steps create additional layers, and you want to eliminate them from the final image.

The build stage is named by appending `AS *name-of-build*` to the FROM instruction. The name of the build stage can be used in a subsequent FROM and COPY command by providing a convenient way to identify the source layer for files brought into the image build. The final image is produced from the last stage executed in the Dockerfile.

Try taking the example from the previous section that used more than one Dockerfile for the React application and replacing the solution with one file that uses a multistage build.

```
FROM node:12.13.0-alpine as build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

FROM nginx
EXPOSE 3000
COPY ./nginx/default.conf /etc/nginx/conf.d/default.conf
COPY --from=build /app/build /usr/share/nginx/html
```

This Dockerfile has two `FROM` commands, with each one constituting a distinct build stage. These distinct commands are numbered internally, stage 0 and stage 1 respectively. However, stage 0 is given a friendly alias of `build`. This stage builds the application and stores it in the directory specified by the WORKDIR command. The resultant image is over 420 MB in size.

The second stage starts by pulling the official Nginx image from Docker Hub. It then copies the updated virtual server configuration to replace the default Nginx configuration. Then the `COPY --from` command is used to copy only the production-related application code from the image built by the previous stage. The final image is approximately 127 MB.

Problems That Docker Multistage Builds Might Encounter

Depending on how they are designed, multistage builds can introduce some serious issues around the speed of the build process. Since these Dockerfiles have multiple stages to produce the production-grade image, they can take a while to build. If you are using the latest docker version and have enabled BuildKit then caching will likely help speed things up. But there is another problem, Readability.

Note: You can enable BuildKit, and get faster multi-stage builds by setting the DOCKER_BUILDKIT environment variable to 1. In newer versions of Docker this is enabled by default.

A Better Way: Earthly

As your multi-stage build grows in complexity, comprehending how each step follows from the next can become a challenge. If the number of stages extends beyond two or if caching is becoming a challenge even with Buildkit enabled, you may want to consider using Earthly to produce your docker images.

Earthly mirrors the dockerfile syntax but allows for naming the stages and for more fine-grained caching.

Here is our previous solution in Earthly:

```
FROM node:12.13.0-alpine
WORKDIR /app

build:
  COPY package*.json ./
  RUN npm install
  COPY . .
  RUN npm run build

final:
  FROM nginx
  EXPOSE 3000
  COPY ./nginx/default.conf /etc/nginx/conf.d/default.conf
  COPY  +build/app/build /usr/share/nginx/html
```

The stages are now named `build` and `final` and the copy syntax has changed slightly from:

```
COPY --from=build /app/build /usr/share/nginx/html
```

To:

```
COPY  +build/app/build /usr/share/nginx/html
```

This is only scratching the surface of what the open source Earthly project can do. But once you have multiple stages in play, I'd recommend converting your Dockerfiles to Earthfiles.

Using Earthly

- Mac users: `brew install earthly`. (Other platforms)
- Rename `Dockerfile` to `Earthfile`.
- Build image (`earthly +final` for above example).
- Read more on the website.

Conclusion

While creating Docker images the right way is not a small task, the final outcome does a great deal of good for the speed and security of your application delivery. Larger images have a high number of security vulnerabilities that shouldn't be overlooked for the sake of speed. The reality is that quality images take time and care.

The builder pattern has evolved over time in its implementation, with multistage builds coming to the rescue from the tedious steps that previously had to be followed. Tools like BuildKit and Earthly further improve on this process.

Though not foolproof, multistage builds have made it much easier to create optimized images that you can be more pleased and confident to have running in your production environment.

A Beginner's Guide to Debugging Docker Containers

Debugging Docker containers can be frustrating. Luckily, Docker provides several commands that make managing and troubleshooting containers easy. In this article, we'll walk you through several real world scenarios and show you some tips and techniques for debugging Docker containers.

Prerequisites

Although this guide will take a beginner-friendly approach, you must have Docker installed and configured correctly on your operating system. Additionally, you would need the following:

- Basic knowledge of Docker, such as pulling and pushing Docker images, starting and stopping containers, etc.
- Basic understanding of the command line, such as executing commands and primary navigation.

With all these in check, you're sure to make the most out of this article by following the examples, which will give you a good grip on these debugging techniques.

Docker Debugging Landscape

Before moving into the various debugging techniques and their respective Docker commands, it is essential to understand some unique challenges that arise from debugging containers. These challenges include Docker containers' short-lived (ephemeral) nature, isolated environments, and limited visibility. Understanding the debugging landscape will help you effectively troubleshoot containerized applications.

Ephemeral Nature of Containers

Docker containers have numerous features that make them highly popular. Some of these features include being lightweight, portable, and disposable. That means you can start, stop and destroy Docker containers with ease.

Although advantageous, these features mean containers would be short-lived, making it challenging to capture and analyze the state of a container at any point in time.

Isolated Environments

Part of the design of Docker containers is their isolation from the underlying host system and other containers. This feature makes it more secure and improves encapsulation; however, debugging can be complex with this design.

Limited Visibility

Imagine a scenario where your application comprises multiple interconnected containers. If one container is not working correctly, it may impact the entire application's functionality. However, pinpointing the exact cause of the issue becomes a puzzle due to limited visibility from the host system into individual containers making it challenging to observe internal states, logs, and network interactions.

Exploring Essential Docker Debugging Commands

The following sections contain various scenarios where debugging and troubleshooting commands are helpful.

Inspecting Container Status and Logs

One of the essential tasks in debugging Docker containers is gaining insight into their status and examining the logs they generate. This information can help you understand how containers are running, detect any errors or issues, and pinpoint potential areas of concern.

To inspect the status of running containers, we will use the following Docker commands:

- **Docker ps**: This command lists all the containers running on your system. They also provide certain information about these containers, which include the container ID, images name, and status. With this command, you can quickly view any container you deploy.

  By simply running `docker ps`, you can easily see the details of all containers. You can also add an option `-a` or `--all` to see all containers (running and stopped containers).

  ```
  docker ps # Only for running containers
  docker ps -a # For all running or stopped containers
  ```

- **Docker logs**: Using this command, you can retrieve important information such as application output, error messages, and other relevant details from a Docker container. During troubleshooting issues, it is crucial to examine these logs to identify errors and understand the application's behavior.

  You can use this command by attaching the container ID to it:

  ```
  docker logs [container]
  ```

- **Docker inspect**: To obtain detailed information about a container, utilize the Docker inspect command. The command generates an output in an object format that includes all relevant details about the container, including its creation date and shell.

  The syntax to use this command is as follows:

  ```
  docker docker_object inspect [OPTIONS] NAME|ID [NAME|ID...]
  ```

Note: `docker_object` refers to the specific Docker object you want to inspect, such as a container, image, network, or volume. You replace `docker_object` with the specific object you want to inspect, for example, container, image, network, or volume.

Since the focus is on debugging Docker containers, the syntax would be as follows:

```
docker container inspect [container]
```

You should replace the `container` placeholder with either the name or the ID of the specific Docker container you want to inspect.

Scenario: Suppose you have deployed multiple containers for your web application, including a web server and a database container. However, you notice that your application throws an error saying **no database connection was found**. Using the following commands above, this is how you would troubleshoot the issue.

1. Using `docker ps`, check whether the database container runs successfully. If not, rerun the docker image to restart the container. If it is, copy its container ID and proceed with further debugging.

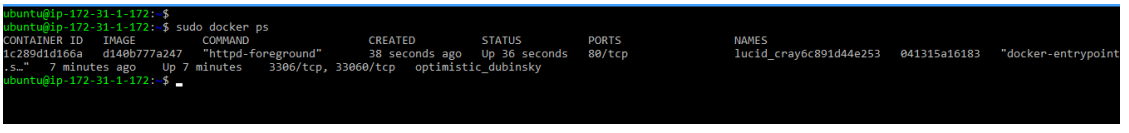


Figure 9: **Figure 1**. Docker ps output

2. Use the `docker logs` command with the container ID to view the logs of the database container. Make sure to read slowly on each line for error messages from the database connection.

   ```
   docker logs [container]
   ```

3. After reading the error messages, you would identify some issues such as incorrect database credentials or network configuration problems. Apply the necessary changes, and restart your container.

Although the commands mentioned may be helpful in solving some issues, it's important to note that more complex problems may require additional tools. You must gain knowledge and proficiency in using these tools to handle such situations effectively.

Accessing Container Shell

Most times, while debugging, there is just so little you can do outside a running container. However, Docker provides commands allowing you to access any container's shell or command line. This ability can be very efficient during troubleshooting, allowing you to interact with the container's environment, execute commands, and investigate issues directly.

You can use the following commands to access the shell of any container:

- **Docker exec**: There are two ways to access the command line of a container using this command. They are as follows:

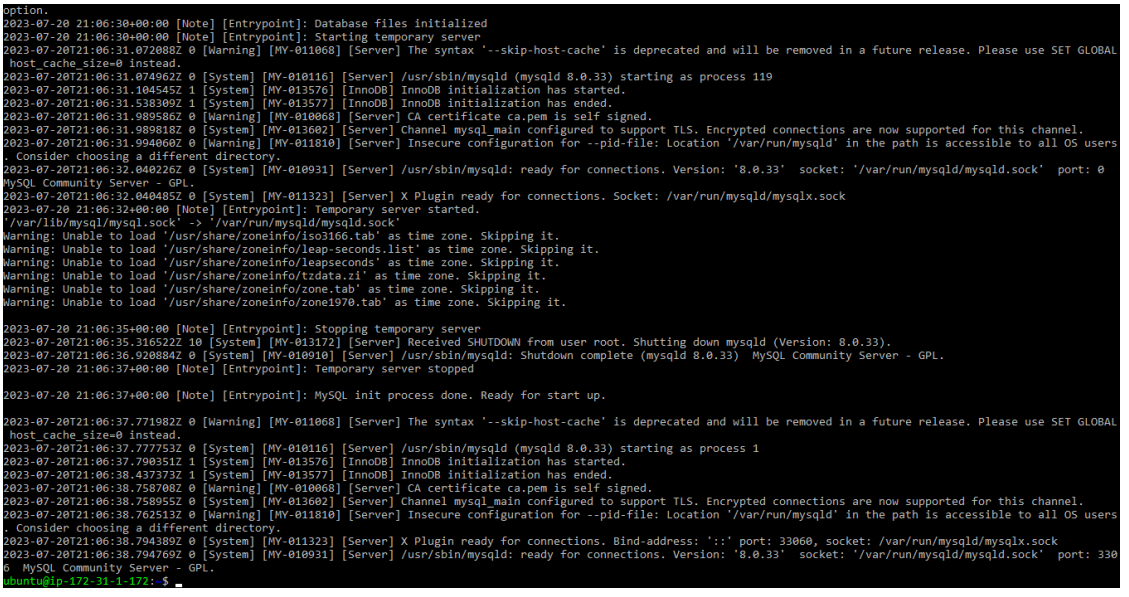


Figure 10: **Figure 2**. Docker logs output

1. **Executing Commands Directly:** This method of using `docker exec` involves executing commands directly into a running container without the `-it` flags. The `it` flag means the following:
   - `i` (interactive): It enables you to interact with the container, for example, by providing input to the command running inside the container.
   - `t` (terminal): This command allocates a terminal for the container.

   Note: By using the `-it` flag, you can run interactive commands inside a Docker container, enabling you to interact with the running process and see its output as if you were working directly in a terminal.

   This method allows you to run a specific command within the container's environment and receive the output directly in your terminal. It is helpful for one-off commands or situations where you don't require continuous interaction with the container.

   To execute a command directly within a container, use the following syntax:

   ```
   docker exec [container] [command]
   ```

   Replace `container` with the container's ID or name and `command` with the command you want to execute.

   For example, you might use the following command to list the files in a container:

   ```
   docker exec [container] ls
   ```

2. **Starting an Interactive Shell:** The second method involves starting an interactive

shell within a running container using the `docker exec` command with the `-it` flags. This approach allows you to establish an interactive session, like opening a shell directly within the container. It provides a more immersive experience, allowing you to execute multiple commands, navigate the file system, and perform complex troubleshooting tasks.

To begin an interactive shell in a container, use this syntax:

```
docker exec -it [container] [shell]
```

For example, you might use the following command to start an interactive shell session in a container:

```
docker exec -it [container] sh
```

This command starts an interactive shell using the sh command within the container.

Both methods have advantages and can be utilized based on your specific debugging needs. Using the `docker exec` command directly might be sufficient if you only need to execute a single command. However, if you need continuous interaction and exploration within the container, starting an interactive shell using `docker exec -it` is the way to go.

- **Docker attach**: In addition to the `docker exec` command discussed earlier, there's another useful command called `docker attach` that allows you to attach your terminal to a running container's shell. This command is particularly helpful when interacting with a container already running and viewing its output in real time.

  The `docker attach` command lets you connect your terminal directly to a running container and interact with its standard input (`stdin`), output (`stdout`), and error (`stderr`). It's like **attaching** your terminal to the container, allowing you to see its ongoing output and interact with the running processes.

  To use `docker attach`, you need to know the ID or name of the container you want to connect to. Once you have that information, you can execute the following command:

  ```
  docker attach [container]
  ```

Scenario: Suppose you are running a Docker container hosting a web application but encounter issues due to misconfigured application settings or script errors. To troubleshoot and resolve this problem, you can utilize the `docker exec`, and `docker attach` commands. Follow the steps below:

1. Using `docker ps`, identify the container running the web application. Make sure to get the container ID or name.
2. Observe the application output. To observe the application's output in real-time, you can attach it to the container using the `docker attach` command:

   ```
   docker attach [container]
   ```

   This connects your terminal to the container's output stream, allowing you to see any print statements or error messages generated by the application. Take note of any relevant error messages or unexpected behavior displayed in the terminal.

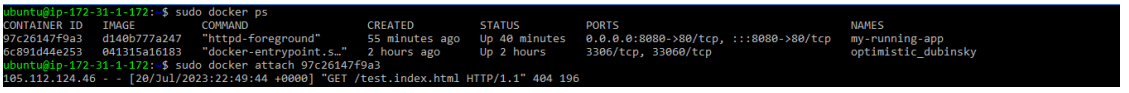


Figure 11: **Figure 3** docker ps and docker attach command

3. Debug the application. You can now debug the application with the container's shell attached. Examine the error messages, log files, or any other available information to identify the root cause of the issue.

   You can execute commands within the attached shell to investigate further:

   ```
   docker exec -it [container] /bin/sh
   ```

   ```
   # ls
   bin build cgi-bin conf error htdocs icons include logs modules
   ```

   This command gives you access to the shell of the container, where you can debug errors according to the logs thrown by the application.

4. Make necessary corrections. Make the necessary corrections within the container based on the errors or misconfigurations you identified. Edit configuration files, update scripts, or modify environment variables as required to rectify the issue.

Using the `docker exec` and `docker attach` commands, you can troubleshoot and resolve issues related to misconfigured application settings or script errors within a Docker container. These commands enable you to interact with the container's shell, execute commands, and observe real-time output, facilitating effective debugging and resolution of such issues.

Examining Resource Usage and Performance

When working with Docker containers, monitoring their resource usage and performance is essential to ensure optimal operation. This section will teach you various Docker commands to examine resource usage and identify performance bottlenecks.

The following commands are essential in monitoring resource usage:

- **Docker stats**: This command provides a continuously updating table showing the resource consumption of each container. It displays CPU and memory utilization, network I/O, and container uptime. You can quickly identify the problematic container and investigate further by running docker stats.

  By running `docker stats`, you can easily get this table. The syntax for this command is as follows:

  ```
  docker stats
  ```

- **Docker top**: The `docker top` command lets you view all the processes inside a container. It displays the running processes' details, such as the process ID (PID), CPU and memory usage, and the associated command.

  To utilize this command, run:

```
docker top [container]
```

Replace `container` with the container's name or ID.

Scenario: Imagine you have a container running a database server and notice a significant increase in CPU usage. By executing `docker stats` and `docker top` command together, you can monitor the container's overall resource usage while identifying the specific processes within the container that is responsible for the high CPU consumption.

Below are the steps you would take to troubleshoot this issue:

1. To keep track of the resource usage of your running containers, begin by entering the command `docker stats` into your terminal. This will display a continuously updating table showing the resource consumption of each container:

```
CONTAINER ID   NAME                   CPU %     MEM USAGE / LIMIT    MEM %   NET I/O
97c26147f9a3   my-running-app         0.00%     8.742MiB / 965.7MiB  0.91%   3.28kB / 3.68kB
6891d44e253    optimistic_dubinsky    0.41%     352.1MiB / 965.7MiB  36.46%  2.04kB / 0B
```

2. Identify the container exhibiting abnormal CPU or memory usage based on the information provided by `docker stats` command. Take note of the container's name or ID.

   With the container name or ID in hand, execute the following command to inspect the processes inside the container.

   ```
   docker top [container]
   ```

   Replace `container` with the name or ID of the problematic container.

   ```
   UID      PID   PPID  STIME  TTY  TIME      CMD
   lxd      2523  2502  21:06  -    00:00:38  mysqld
   ```

3. Review the output of docker top to identify any processes within the container that consume excessive resources. Look for processes with high CPU or memory usage, as well as the associated command that is being executed.

4. Analyze the resource-intensive processes and investigate the root cause of their resource consumption. This may involve examining application code, configuration files, or debugging scripts to identify misconfigurations or inefficiencies.

5. Once you have identified the underlying issue, take appropriate steps to address it. This may involve optimizing code, adjusting application settings, or resolving script errors.

6. Monitor the container's resource usage using docker stats again to verify that the problem has been resolved. Ensure that the CPU and memory utilization return to expected levels.

By following these steps, you can effectively use `docker stats` and `docker top` together to monitor resource usage and identify resource-intensive processes within containers.

Debugging Network Connectivity

Learning Docker networking can be tricky. Even with a solid understanding of the fundamentals, debugging Docker network connectivity errors can be frustrating. In this section, we'll take a look

at some strategies you can use next time you encounter a network error.

You can use the following commands to resolve network difficulties:

- **Docker network inspect**: The `docker network inspect` command allows you to analyze Docker networks and their configuration. It provides detailed information about network settings, IP addresses, and connected containers. To utilize this command, run:

  ```
  docker network inspect [network]
  ```

  To inspect a specific network, replace `network` with its name or ID.

- **Docker exec**: The `docker exec` command, as you learned earlier, allows you to execute commands within a running container. It can also be utilized in two ways to debug network connectivity issues:

  1. `ping`: The `docker exec` command, combined with the `ping` utility, allows you to verify network connectivity between containers and external hosts:

     ```
     docker exec -it [container] ping [host]
     ```

     To test a specific container, replace `container` with its name or ID. Similarly, replace `host` with the hostname or IP address of the target host.

  2. `nc`: The `docker exec` command, combined with the nc utility, allows you to check connectivity to a specific port on a host. To perform a check on the connection between a container and a specific host and port, use the command below:

     ```
     docker exec -it [container] nc -zv [host] [port]`
     ```

     To test a specific container, replace `container` with its name or ID. For the target host, use the hostname or IP address and replace `host`. Lastly, specify the port number by replacing `port`.

     By using the `docker exec` command in different ways, you can effectively debug network connectivity problems in your Docker environment. Whether you need to check connectivity to a specific host or verify port access, these commands provide valuable insights into your containerized applications' network behavior.

Scenario: Let's say you are working on a microservices-based application that consists of multiple Docker containers. One of the containers is a frontend web server that needs to communicate with a backend database container. However, you encounter network connectivity issues, and the web server cannot establish a connection with the database.

To troubleshoot and resolve the network problem, you can follow these steps:

1. Verify Docker network configuration: Start by inspecting the Docker network associated with the containers using the following command:

   ```
   docker network inspect [network]
   ```

   Replace `network` with the name of the network your containers are connected to.

   This command will provide detailed information about the network, including the container configurations, IP addresses, and more. Ensure that the front and backend containers are

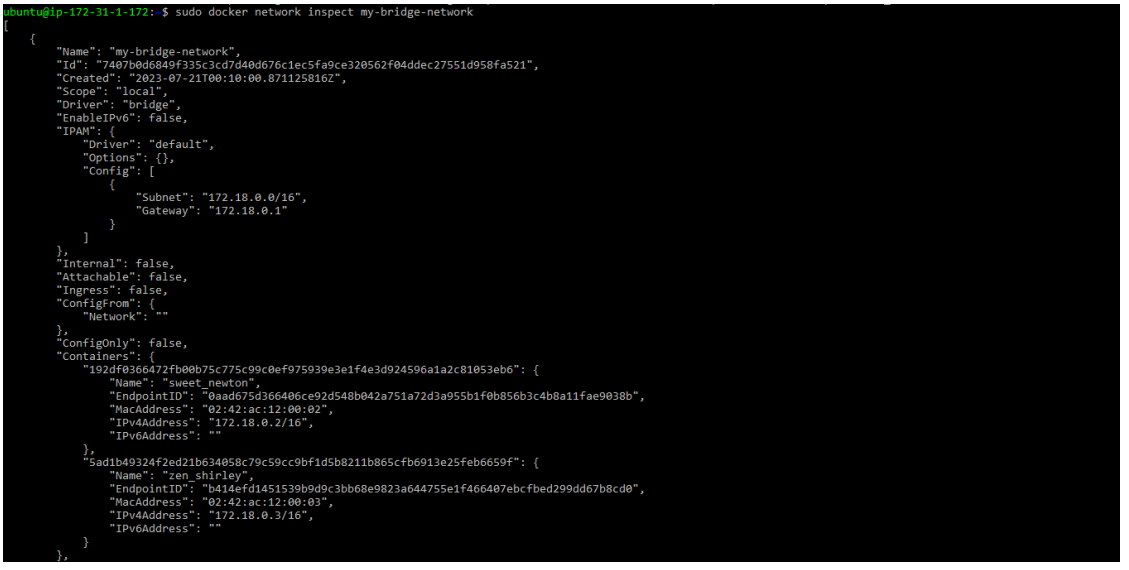


Figure 12: **Figure 4**. Docker network inspect command

connected to the same network.

2. Check host and port availability

   ```
   docker exec -it 192df0366472 nc 54.91.248.13 8080
   ```

   ```
   Connection to 54.91.248.13 8080 port [tcp/*] succeeded!
   ```

   Replace `container` with the name or ID of the frontend container, `host` with the IP address or hostname of the backend database container, and `port` with the port number the database is running on.

   This command will check if the frontend container can connect to the backend container on the specified host and port. It will provide information on whether the port is open and accessible.

   Ensure that the host and port are correctly specified and that no firewall or network configuration issues block the connection.

3. Verify Network Connectivity: Next, check the network connectivity between the containers by running the command:

   ```
   docker exec -it [container] ping [host]
   ```

   Replace `container` with the name or ID of the frontend container and `host` with the IP address or hostname of the backend database container.

   This command will send ICMP echo also known as ping requests, from the frontend container to the backend container. It helps determine if there are any network connectivity issues, such as packet loss or unreachable hosts.

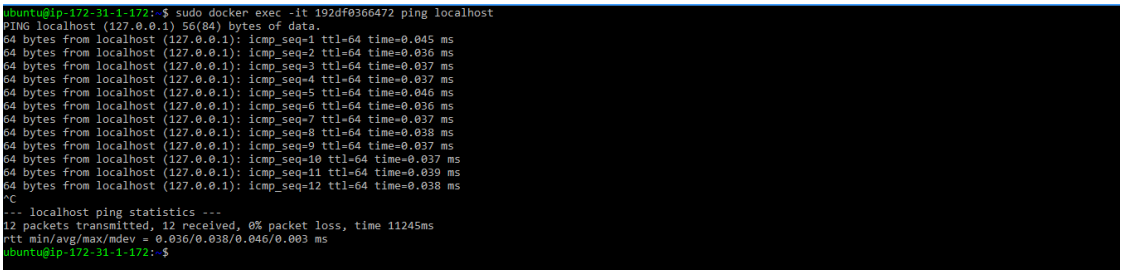


Figure 13: **Figure 5**. ping command

Note: Internet Control Message Protocol (ICMP) is a network protocol used for troubleshooting purposes. When a device or computer sends an ICMP echo request (ping) to another device, it is essentially asking for a response to confirm the target device's availability and round-trip time.

4. Analyze the ping results to identify connectivity problems like high latency or timeouts. Ensure that the frontend container can reach the backend container through the network.

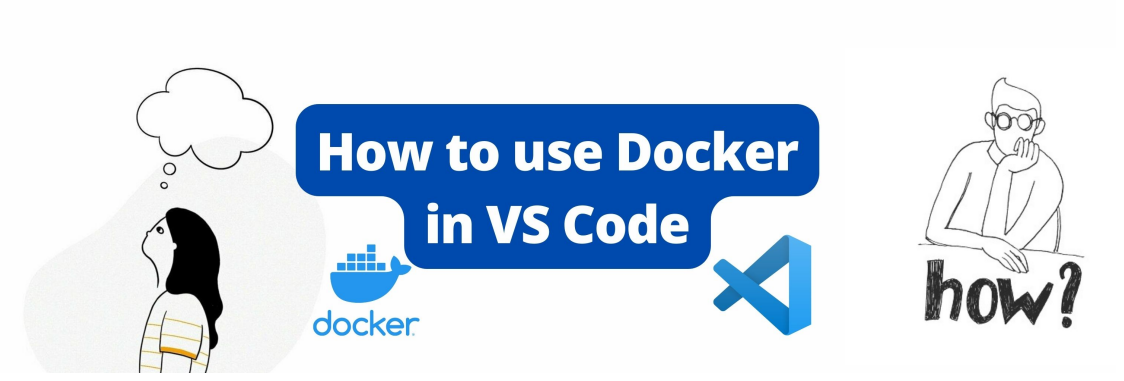
By combining the `docker exec -it nc` command to check host and port availability, the `docker exec -it ping` command to verify network connectivity, and the `docker network inspect` command to examine the network configuration, you can diagnose and troubleshoot network-related issues between Docker containers.

Conclusion

This comprehensive guide has explored essential Docker commands for debugging containers and provided practical scenarios to demonstrate their usage. By adopting a systematic approach to debugging, developers can effectively identify and resolve issues in their Dockerized environments.

We began by understanding the challenges of debugging Docker containers, including their ephemeral nature and limited visibility. Recognizing the need for a systematic approach, we delved into essential Docker debugging commands. We learned how to inspect container status and logs, access container shells, examine resource usage and performance, and debug network connectivity. Through clear explanations and real-world scenarios, we demonstrated the practical application of these commands in troubleshooting common Docker-related problems.

By following the techniques and commands outlined in this guide, developers can gain a comprehensive understanding of Docker debugging and enhance their troubleshooting skills. Remember to approach debugging systematically, using commands such as `docker ps`, `docker logs`, `docker exec`, `docker attach`, `docker stats`, `docker top`, `docker network inspect`, `docker exec -it ping`, and `docker exec -it nc`. With this knowledge, developers can confidently debug Docker containers, resolve issues efficiently, and ensure the smooth operation of their containerized applications.



How to use Docker in VS Code

Created by Microsoft, the Docker extension makes it easy to build, manage, and deploy containerized applications without leaving your code editor. Simply put, it helps you manage Docker better.

In this article, I'll walk you through how to use Docker in VS Code using the Docker extension. This extension has some exciting features that can make working with Docker easier.

I'll be using this extension to work with Docker by adding a `dockerfile`, building an image, and also running it. I'll do all this without having to use the terminal at all. Let's get started!

Prerequisites

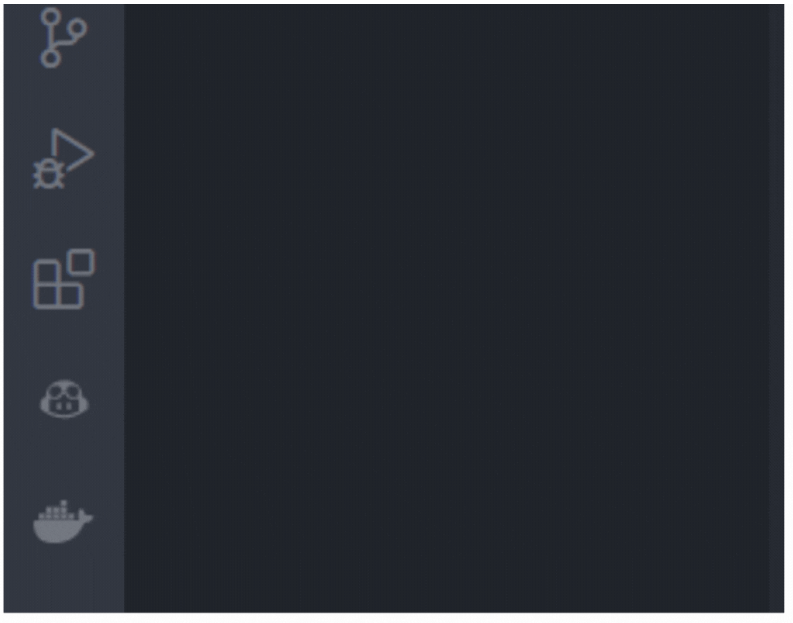
To follow along with this article, you will need to have Docker installed on your workstation. Instructions on installing and running Docker are available, and they should be specific to the operating system you are running. You also need to have VS Code installed.

Installing the Docker Extension

Head to the extension section in VS Code and type `docker` in the search box. You should see something like this:

The first selection in this image is what you want to look for to install. Click on it to install. Once it's done downloading, you will notice a Docker icon or logo at the bottom left corner of your window. This is the Docker explorer.

Figure 14: Searching for docker extension



Click on it. All our running and stopped containers are highlighted inside the Docker explorer. Here, you can also see your images, registries, volumes, networks, e.t.c. :

Building Our Project

Let's create a basic node express.js app to show how you can use the extension.

First, create a folder where you want your project to be stored and open it up in VS Code. Next, create a file called `index.js` and paste the following code:

```
const express = require("express");
const app = express();
const port = 3000;

app.get("/", (req, res) => {
  res.send("Hello World!");
});

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`);
});
```

Basically, the code above is just to create a hello world application. All the code does is make the `app` start a server and listen on port 3000 for connections. Then, the app responds with "Hello World!" for requests to the root URL (/).

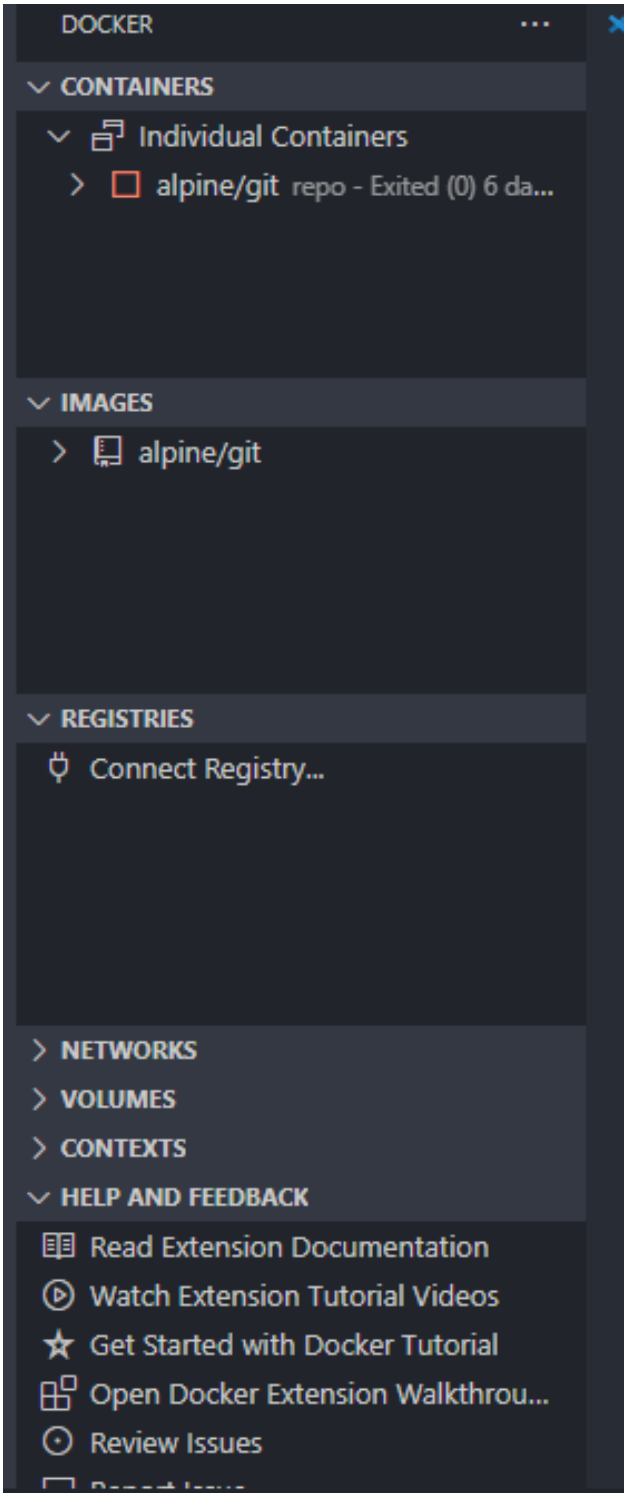


Figure 15: Containers ,volumes, images, and more

You won't do anything else with this app as it's just being used to play around in our Dockerfile

Adding a Docker File

Traditionally, to add Docker, you would need to: Create a `dockerfile`. Add Docker instructions to it. Build the Docker image from the terminal. Run the Docker image, also from the terminal.

But with the Docker extension, you can have VS Code do most of the heavy lifting for you.

If you have existing projects with Docker or docker-compose files or you simply prefer to write your own, don't worry, the extension still has a lot to offer you. But if you do want to speed up creating these files, you have the option of letting the extension do some of the heavy-lifting for you.

To generate the Docker files automatically, open the Command Palette by pressing `Command+Shift+P` on a Mac or `Control+Shift+P` on a Windows PC. Then type:

`Docker: Add Docker files to Workspace command`:

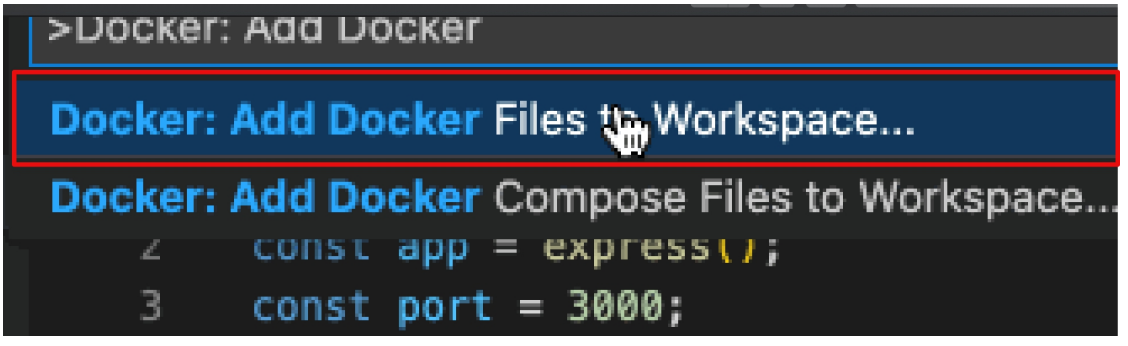


Figure 16: Adding docker files

You'll be asked to choose the application platform you're working with. Just proceed to choose `node`, and you'll also be asked whether to include Docker compose or not. A compose file is typically used when you want to start up multiple containers, say if you also wanted a database, or if you were trying to run a front-end and a back-end together. Since that's not the case with our project you can choose `No`.

You will then be prompted to select a port. Select `3000` because `3000` is the port on which our app will listen. Now, the following files are added to your workspace: `.dockerignore` and `Dockerfile`.

Inside the `Dockerfile` should have something like this:

```
FROM node:lts-alpine
ENV NODE_ENV=production
WORKDIR /usr/src/app
COPY ["package.json", "package-lock.json*", "npm-shrinkwrap.json*", "./"]
RUN npm install --production --silent && mv node_modules ../
COPY . .
```

```
EXPOSE 3000
RUN chown -R node /usr/src/app
USER node
CMD ["node", "index.js"]
```

You're probably already familiar with Docker, but just to highlight all the things the extension gives you, here's a brief explanation of what was generated: **FROM:**Sets the base image to use for subsequent instructions. **ENV:** Sets the environment variable key to the value. **WORKDIR:**Sets the working directory to /usr/src/app. **COPY:** Copies files or folders from the source to the destination path in the image's filesystem. **RUN:** Executes any commands on top of the current image as a new layer and commits the results. **EXPOSE:** Defines the network ports on which this container will listen at runtime. **USER:**Sets the user name or UID to use when running the image in addition to any subsequent CMD, ENTRYPOINT, or RUN instructions that follow it in the Dockerfile. **CMD:** Provides defaults for an executing container.

Similar to the `.gitignore`, the `.dockerignore` instructs Docker to hold files and folders that should not be replicated when creating the image.

Building and Running Your Docker File

To build the Docker image, open the `Command Palette` and execute `Docker Images: Build Image`. You can also right-click the `Dockerfile` in the navigation panel and select `Build image:`

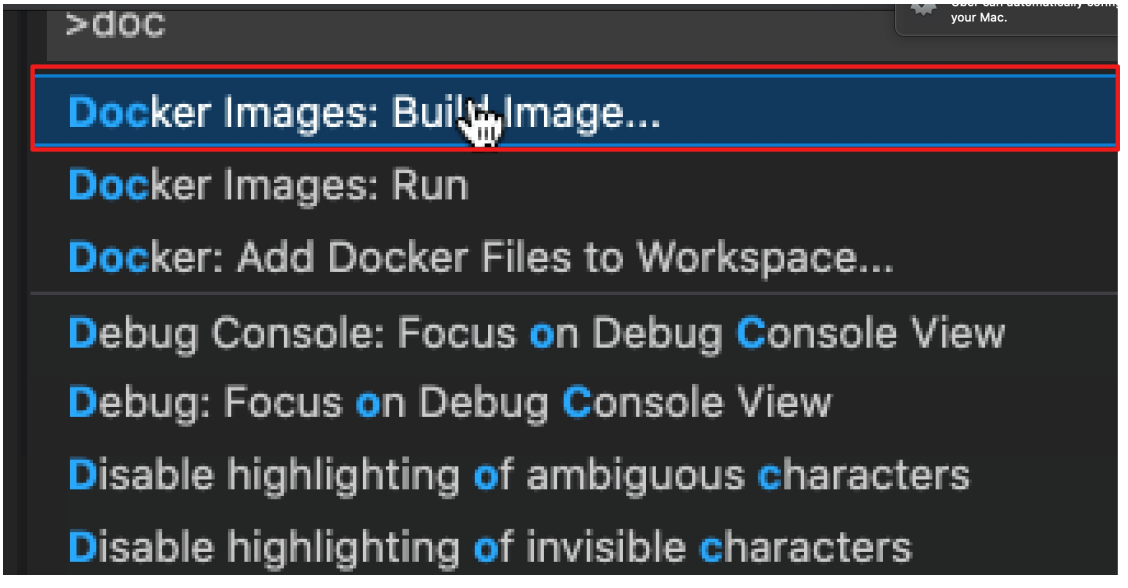


Figure 17: Building an image

If you check the extension pane and look at the `images` section inside the Docker explorer, you should see the latest project has been added to the docker explorer.

The following step is to run our `image`. Open the command palette once more, type `docker run`,

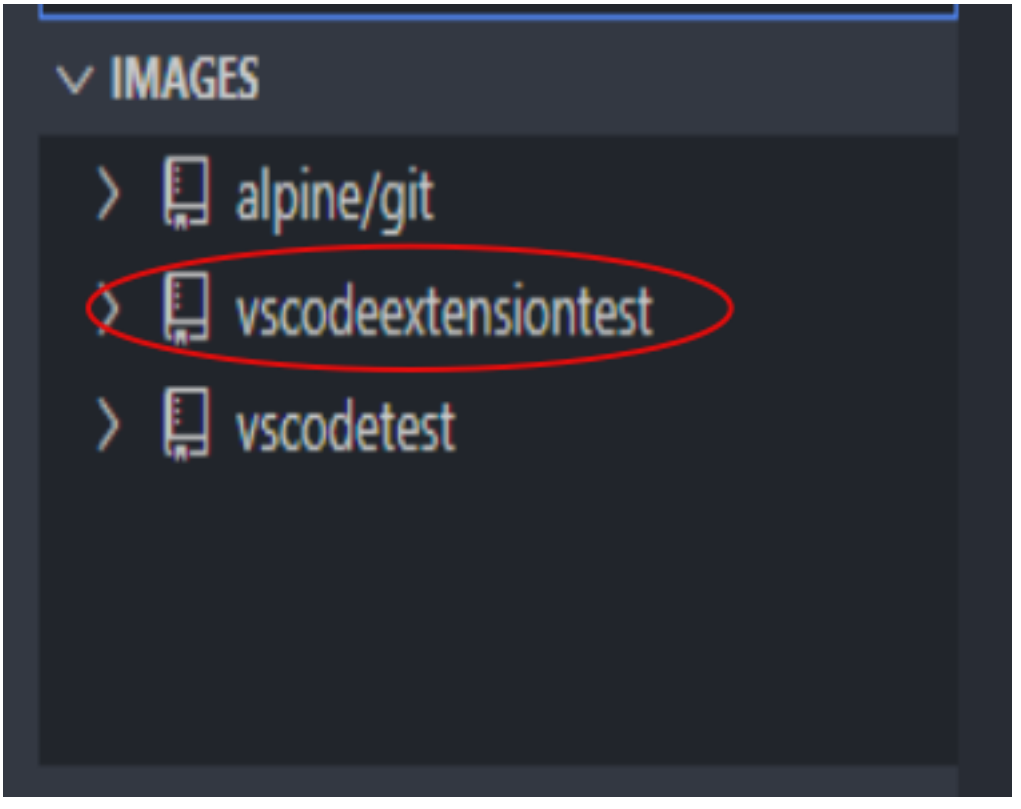


Figure 18: Our latest project

and then pick `Docker: Run`. It will display a list of all the containers on your system. Select the `docker-node:latest` tag and click `Enter`.

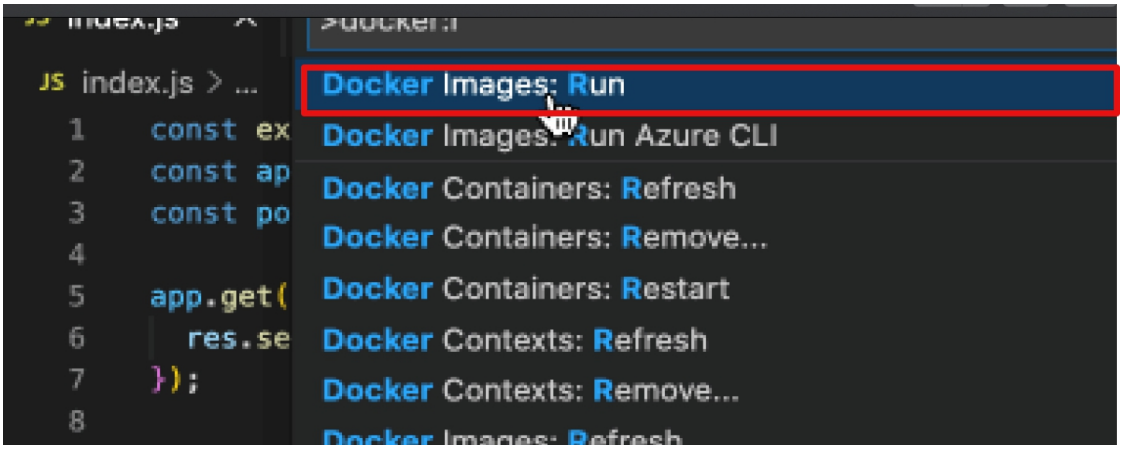


Figure 19: Running a container

You can also run the container by going to the left pane, selecting the Docker explorer, then under `IMAGES`, choose the image you want. Right-click on `latest`. and click run. You will get the same logs running on the terminal.

Once the `docker-node` container runs, You can check the running containers in the same section in our Docker explorer. You can also stop them from here.

You have successfully built an image and ran your image all from VS Code without having to open the command terminal.

You can view the app running in the container in the browser. To do this, right-click on the running container in the docker explorer and click on “open in browser”:

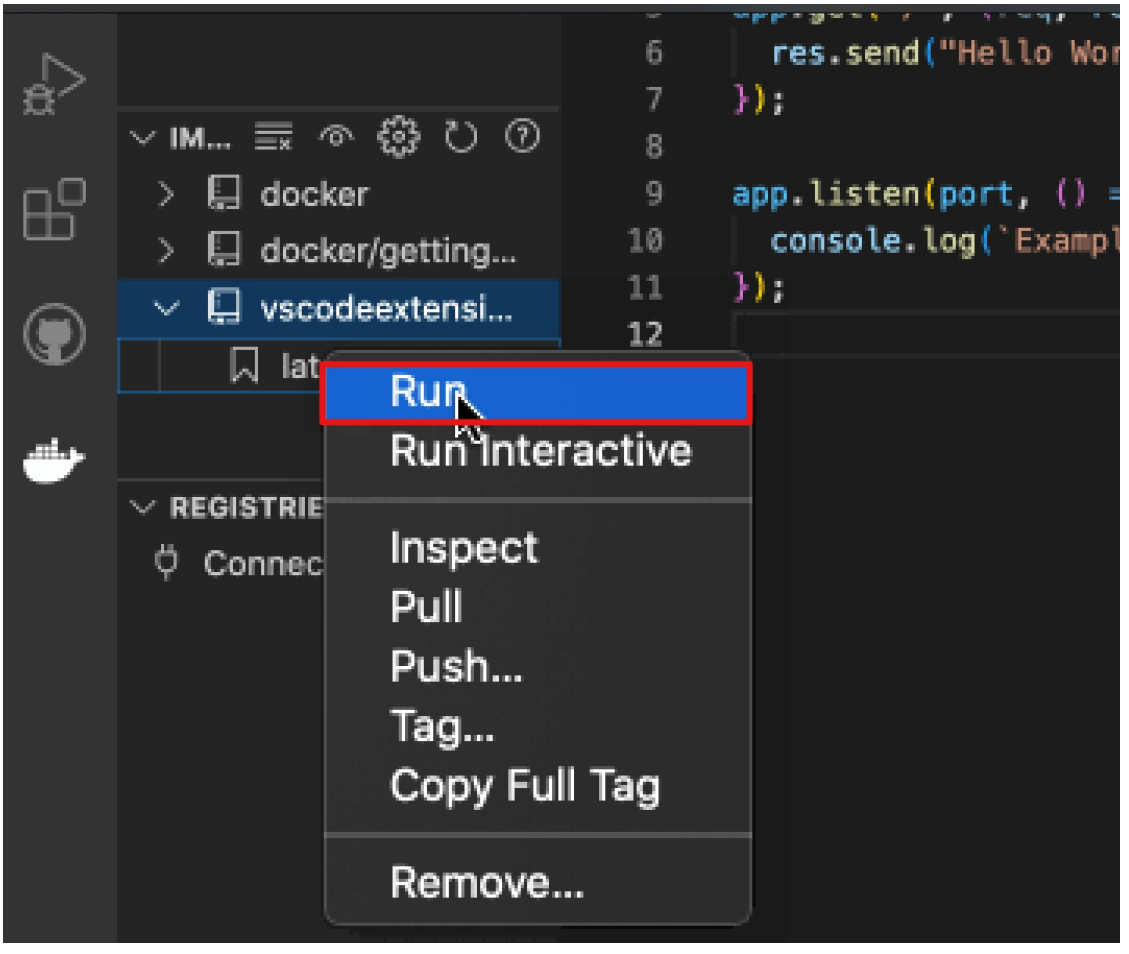
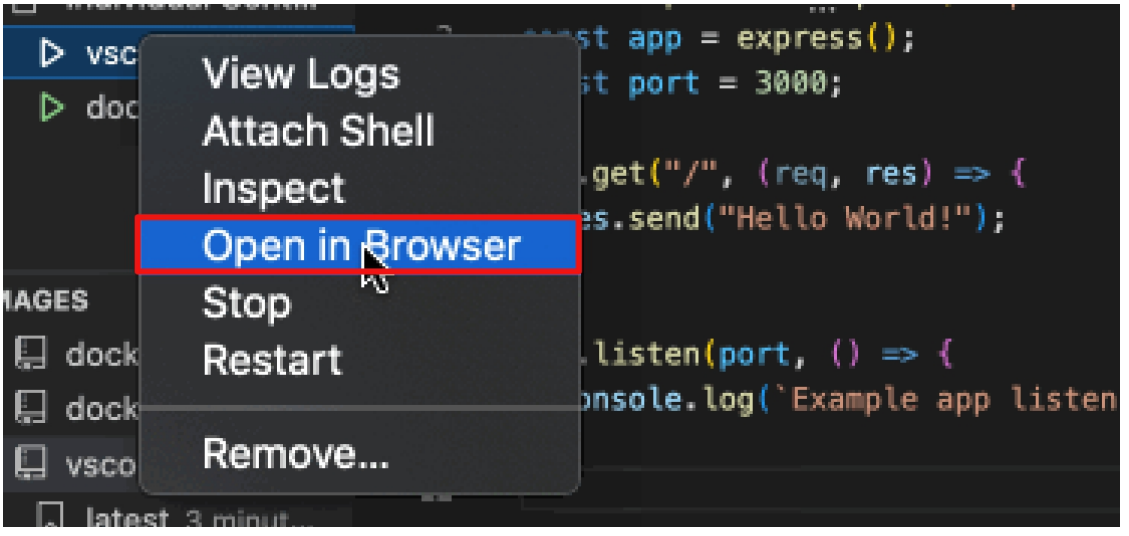


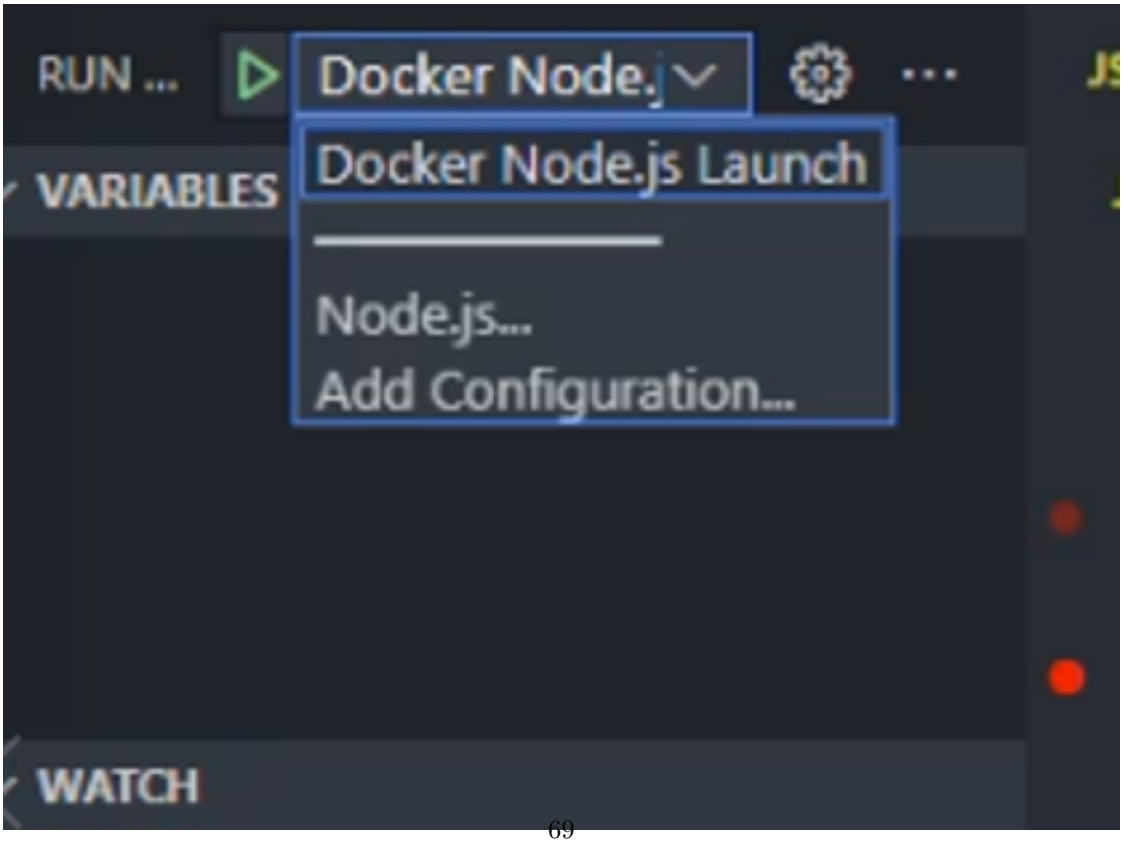
Figure 20: Running docker inside the extension



Debugging Our Container

The docker extension includes a VS Code debugger configuration inside `.vscode/launch.json` for debugging when running inside a container. To do this, set a breakpoint in the `get()` handler for the '/' in `index.js` by pressing the f9 key. Then go to the `run and debug section` in VS Code and select `Docker Node.js launch` debugger and start debugging by pressing `f5` :

Figure 21: The run and debug section



What you'll notice is that the debugger comes to a halt in `index.js` at the breakpoint:

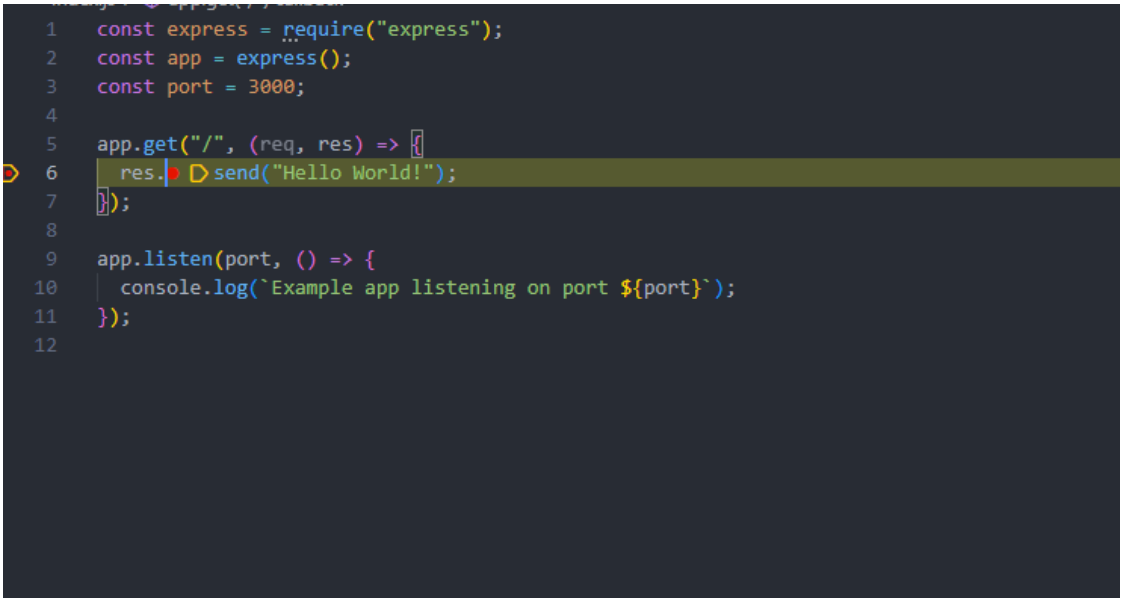


Figure 22: Using breakpoints

You can then continue by pressing the play button at the top to continue running.

Viewing Container Logs

This option is available in the context menu for running containers. The integrated terminal will display the logs.

Using the running Node.js container as our example, all you have to do is navigate to Docker Explorer. In the Containers tab, right-click on your container and choose View Logs. You should see it being displayed in the terminal

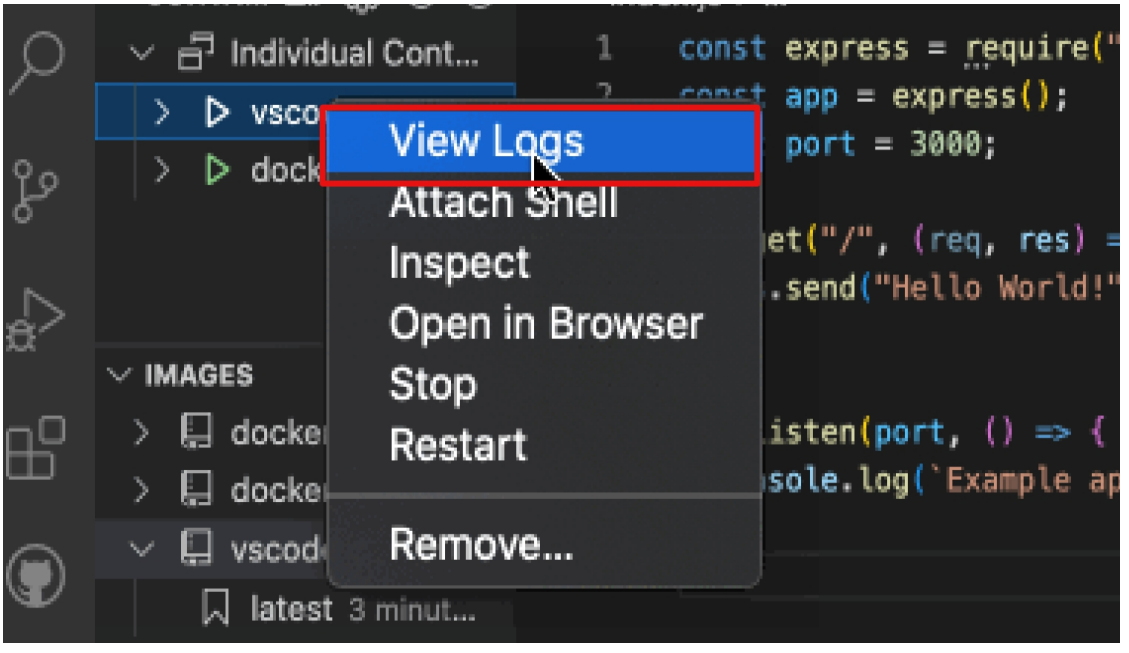


Figure 23: Viewing container logs

```
  express:application set "jsonp callback name" to 'callback' +0ms
  express:router use '/' query +36s
  express:router:layer new '/' +1ms
  express:router use '/' expressInit +0ms
  express:router:layer new '/' +0ms
  express:router:route new '/' +1ms
  express:router:layer new '/' +0ms
  express:router:route get '/' +0ms
  express:router:layer new '/' +0ms
Example app listening on port 3000
  express:router dispatching GET / +123ms
  express:router query  : / +1ms
  express:router expressInit  : / +1ms
  express:router dispatching GET /favicon.ico +168ms
  express:router query  : /favicon.ico +1ms
```

Docker Inspect Images

The Docker inspect images is a feature that allows you to inspect the images built and see the details in a JSON file. This allows you to see important information about our image i.e the

image ID, when it was created, volumes, and many more. Inside our docker explorer, navigate to `IMAGES` and locate the `project folder/latest` right click and click on inspect:

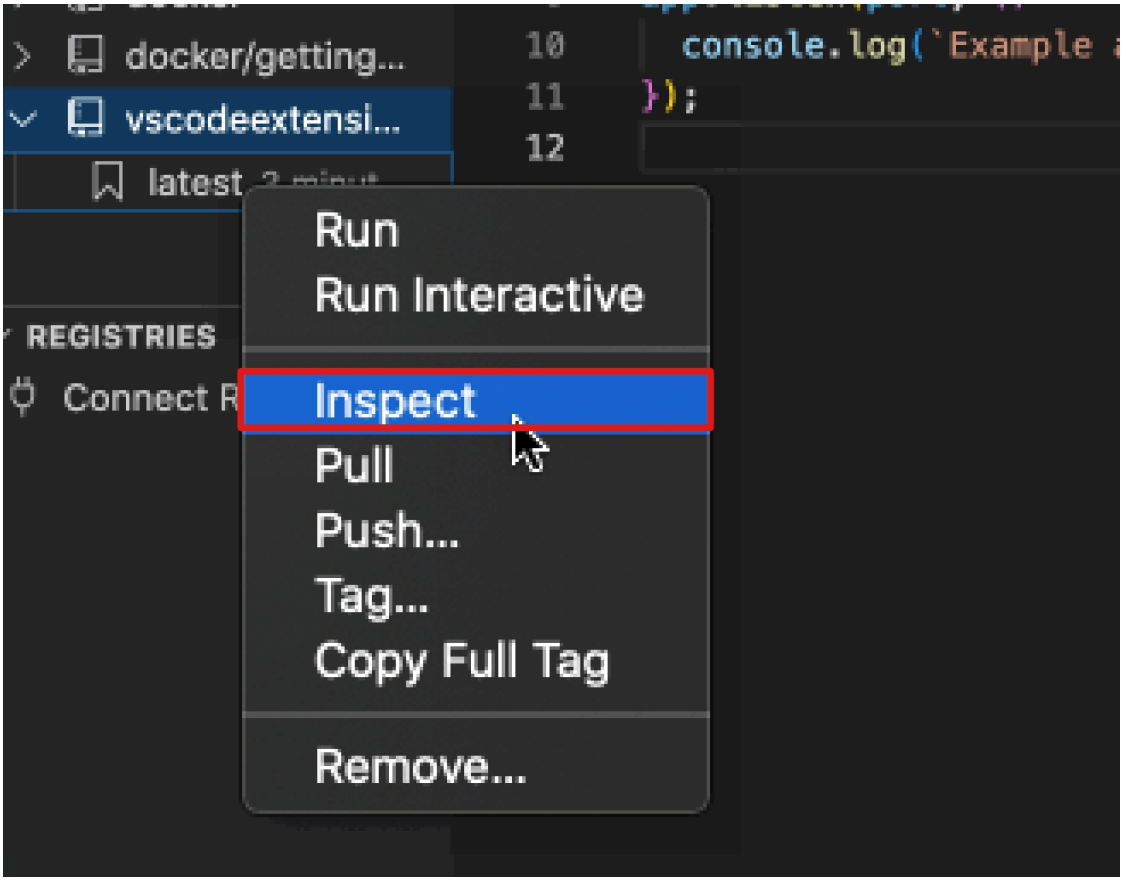


Figure 24: Docker inspect

Other Features

I've barely scratched the surface of what the Docker Extension can do. Some other features you may want to look into further include:

- **Docker commands:** The Command Palette contains most of the commands for Docker images and containers. Typing `Docker:` will find Docker commands.
- **Azure CLI integration:** The Docker extension for VS Code comes with azure CLI integration, which means the Azure CLI can be executed in a separate, Linux-based container by simply running `Docker Images: Run Azure CLI` command. See Get started with Azure CLI.
- **system Prune:** You can use system prune to run Docker system prune, which clears unused images from your system. To do this, run `Docker:prune system` command

- **IntelliSense:** You can use the IntelliSense feature provided by VS Code when creating and editing Docker files by hand. To use the IntelliSense feature, press `ctrl` + space.

Conclusion

The VS Code docker extension enhances your productivity by streamlining terminal operations and offering features for faster, error-free Dockerfile creation. The added bonus is its ability to provide vital insights into your Docker environments, including debugging tools, logs, and visual mapping of running images and networks.

Using GitHub Actions to Run, Test, Build, and Deploy Docker Containers

GitHub Actions is a flexible tool that enables developers to automate a variety of processes, including developing, testing, and deploying, right from their GitHub repositories. The automation of Docker containers is no exception since GitHub Actions also enables developers to automate the process of developing containerized applications. As a result, developers can save time and focus on improving the overall quality of their software.

In this article, you'll learn how to use Github Actions to run, test, build, and deploy Docker containers using GitHub Actions.

How to Run, Test, Build, and Deploy Docker Containers Using GitHub Actions

Before you begin this tutorial, you'll need the following:

- GitHub account
- Basic knowledge of Docker and Docker Compose
- Basic knowledge of YAML files

Once these prerequisites are met, it's time to begin. You'll start by creating a sample workflow, and then set a runner for the workflow. After that, you'll learn how to set up GitHub Actions locally and then how to set up the build and test stage. Finally, you'll execute the workflow by running the action.

Create a Workflow

The first thing you need to do is create a workflow using GitHub Actions. This workflow defines the steps that GitHub Actions should take when triggered, including checking out the code, building a Docker container, running tests, and deploying the application.

To create a workflow, log into your GitHub account and navigate to the repo you want to automate. Select **Actions** from the navigation bar, which will take you to the **Actions** page:

At this point, you have two options: you can either select any of the workflow examples/templates or create a new one from scratch. Here, the **Deploy to Amazon ECS** example was chosen

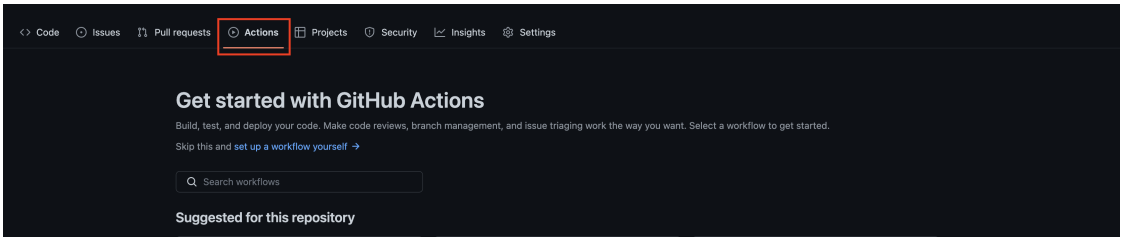


Figure 25: GitHub Actions

under the **Deployment** template. This workflow example essentially deploys a container to an Amazon Elastic Container Service (Amazon ECS):

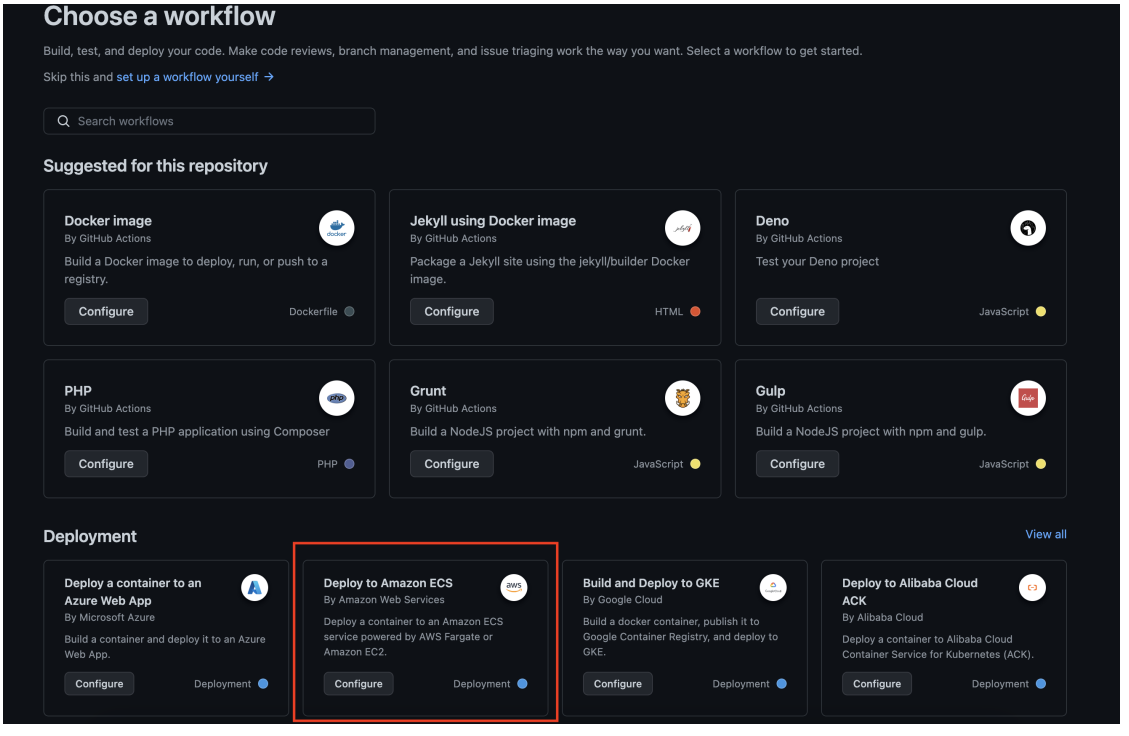


Figure 26: **Deploy to Amazon ECS** example template

Note: The GitHub workflow configuration is always in YAML format, and you'll see many of the following popular parent key-value pairs in your workflows:

- `name` defines a unique name for the workflow.
- `on` specifies the trigger for the workflow, such as when a push is made to the repository or a pull request is opened.
- `jobs` contains one or more jobs that make up the workflow.
- `steps` specifies a list of steps to run in a job.

- `env` defines environment variables that will be used in the workflow.
- `runs-on` specifies the type of runner to use for a job.

Set Up a Runner

Once you've created your workflow, you need to set up a runner. A runner, in this context, is the environment or operating system that processes the actions when the workflow is executed. There are two types of runners in GitHub Actions: self-hosted runners and GitHub-hosted runners.

To set up a runner, open your workflow YAML file. Following the **Deploy to Amazon ECS** workflow template, you should see the `jobs` key with a `deploy` child key, followed by the `runs-on` key. The `runs-on` key is what defines the runner to be used for executing the job. See the following code snippet as an example:

```
jobs:
  deploy:
    name: Deploy
    runs-on: ubuntu-latest
    environment: production

    steps:
    - name: Checkout
      uses: actions/checkout@v3
...
```

In this code snippet, the runner is the `ubuntu-latest` runner, which is a GitHub-hosted runner provided by GitHub Actions. Aside from `ubuntu-latest`, there are several other GitHub-hosted runners, such as `windows-latest`, `macos-latest`, and `centos-8`, that you can use in GitHub Actions.

When the workflow is triggered, GitHub Actions will allocate an available runner of the specified type `ubuntu-latest`. The runner will then execute each step defined in the `steps` key, starting with the `Checkout` step. You can review the `steps` section in the **Deploy to Amazon ECS** as an example:

Set Up GitHub Actions Locally

Considering the limited build minutes that GitHub Actions provides, it's recommended that you conduct a test run of the workflow locally as the workflow will fail to complete if the allotted build time is exceeded. Additionally, setting up GitHub Actions locally allows you to test your workflow locally before you push it to your GitHub repository. This helps ensure that your workflow is working as expected and will run successfully on a GitHub Actions runner.

To set up GitHub Actions locally, you need to first clone or pull the latest changes from your GitHub repository that contains the workflow you want to run locally. Then change the directory to the root directory of the code.

Then install Act on your local machine. Act is a tool that will enable you to run your GitHub Actions locally. After the installation, connect your GitHub token to Act by logging into your GitHub account and navigating to **Settings > Developer settings > Personal access tokens**

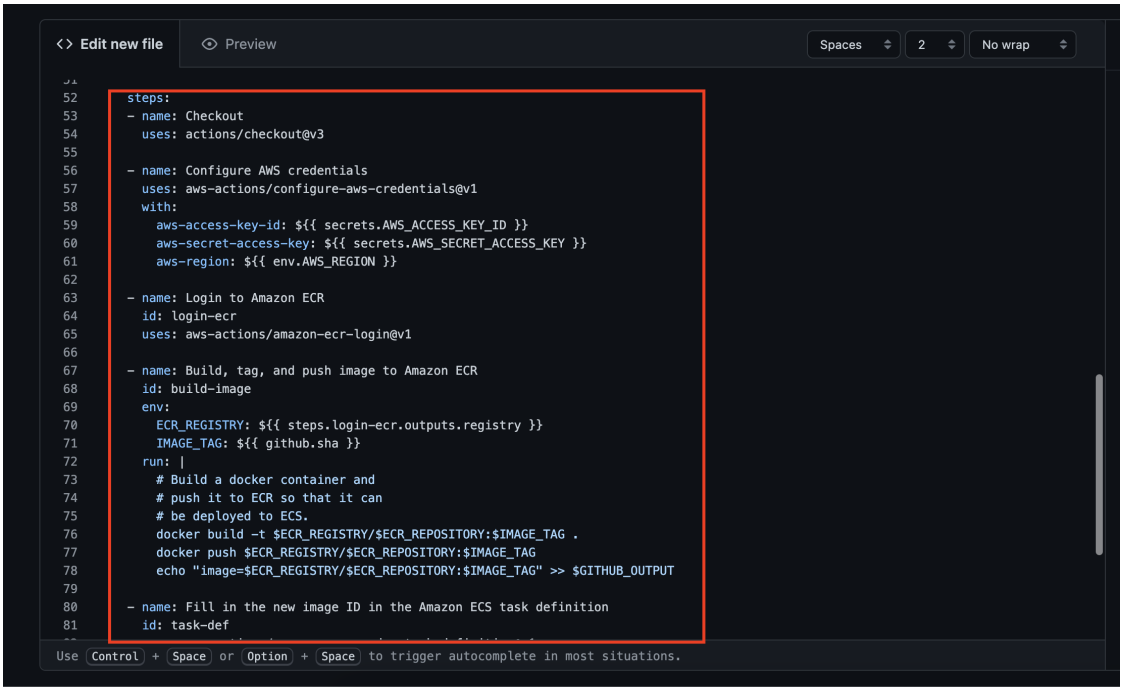


Figure 27: *Deploy to Amazon ECS* steps section

(classic). You can either use an existing token if you still have access to it or generate a new token:

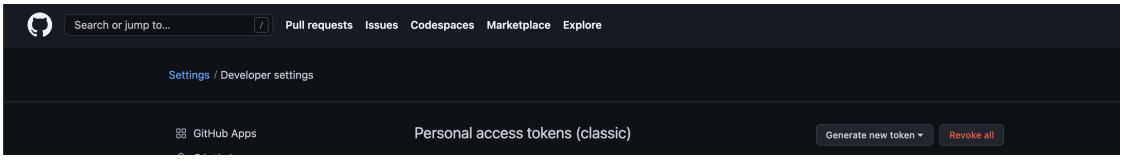


Figure 28: **Personal access tokens** page

Copy the generated token and run the following command:

```
act -s GITHUB_TOKEN={{YOUR_GITHUB_TOKEN}}
```

Make sure to replace `{{YOUR_GITHUB_TOKEN}}` with your generated token. If it is your first time running the command, you will be asked to select the default image you want to use with `act`. You can select the "medium" image.

When finished, clone your GitHub repo with the workflow file if you haven't, and proceed to use Act to run your GitHub Actions locally by running the `act -n` command to dry run the workflow. You should see the run log:

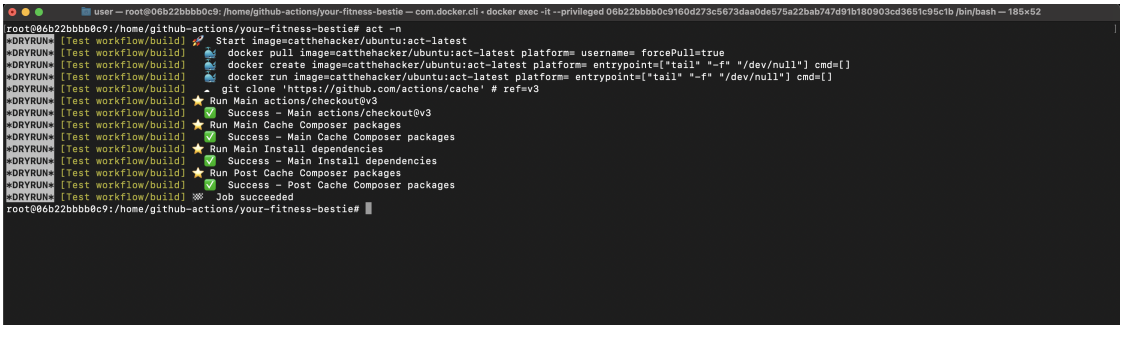


Figure 29: Test workflow run log

You can also run other commands, including the command that runs a specific job, lists all actions for all events, or runs a specific event. You can check out the documentation to see a list of all the available commands.

Note: `Act` can run simple GitHub Action workflow locally using a fairly large docker container. But not all features of GitHub Actions work well with `Act`.

Set Up the Build Stage

Now that you've gotten your GitHub Actions to work locally, you can use it to debug and test your workflow locally, make any necessary changes to the workflow, and rerun it until you're satisfied with the results before pushing your code to GitHub.

Once each step defined in the `steps` key of your workflow file is tested and working as expected, you need to run your `docker-compose` file or build your Docker images. To do that, add a new value to the `steps` key with a `name`, `id`, and `run` key. It may also have an `env` key if the Docker image requires a passkey.

The `name` key can be `Build docker images`, the `id` key will have any unique string, and the `run` key will contain the build command as the value:

```
{% raw %}
- name: Build docker images
  Id: build-image
  run: |
    echo ---Building images and starting up docker---
    {{docker build [image-url] or docker-compose -f \
    [docker-compose file] up -d }}
    echo ---Containers up-
{% endraw %}
```

Following the **Deploy to Amazon ECS** workflow template that's been used here, the build step can be found on line 67:

```
{% raw %}
- name: Build, tag, and push image to Amazon ECR
      id: build-image
      env:
        ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
        IMAGE_TAG: ${{ github.sha }}
      run: |
        # Build a docker container and
        # push it to ECR so that it can
        # be deployed to ECS.
        docker build -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
        docker push $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG
        echo "image=$ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG" >> \
        $GITHUB_OUTPUT
{% endraw %}
```

This template's build step uses the `env` key since `ECR_REGISTRY` requires a login and SHA key.

Set Up the Test Stage

Once you're done setting up the build stage, the next thing you need to do is set up the test stage. Because the **Deploy to Amazon ECS** workflow template is primarily a deployment template, the test stage isn't factored into it. However, the test stage is the recommended phase since it helps you verify the functionality and connections of your applications (including the workflow file) before pushing to GitHub to run the workflow actions.

To set up the test stage, you can either create a test folder in your application or in a different folder outside your application, then develop all your test cases. Afterward, ensure your test folder is containerized in the same network as your application. If you're using Docker Compose, GitHub

Actions will run the workflow in a user-defined network. And for communication to happen, your application and test folder must be in the same network. The network in this context is the Docker Compose network.

The purpose of this network is to provide a Docker network for the containers defined in the `docker-compose` file, allowing them to communicate with each other and with the host system. Containers in separate networks cannot communicate, which means that API test cases won't be able to get a response from the services in other networks. You can learn more about creating a `docker-compose` file with a network in this article.

When you're done adding the test cases folder to the same container network, you need to add a new value to the `steps` key of your workflow file. The new value will contain a `name`, `id`, and `run` key:

```
- name: Run test cases
   id: run-test-cases
  run: |
    echo --- Running test cases ---
    docker-compose -f {{docker-compose-file}} -p {{project-name}} \
    up --build --exit-code-from {{container-name}}
    echo --- Completed test cases ---
```

In this code, `{{docker-compose-file}}` is the name of your `docker-compose` file, `{{project-name}}` is the project name, and `[{{container-name}}` is the container name.

As an alternative to using a `docker-compose` file, you can leverage Earthly, which lets you define your containers and their dependencies, as well as specify your entire build process, including testing and deployment, in one **Earthfile**.

In addition, Earthly allows you to define reusable builds that you can use across different machines, making it easier to collaborate and enabling you to run your builds anywhere.

Moreover, Earthly caches your build components, making the process efficient and fast. Check out the official documentation to learn more about getting started with Earthly.

Run the Action

After you've set up the test stage, the final step is to run the action. To do that, you need to push the changes made to your workflow file to your GitHub repository. The workflow will be triggered based on the branches you set it to monitor on the `on` key of your workflow file.

You can also monitor the progress of the workflow by visiting the **Actions** tab in your repository on GitHub and selecting a workflow that has previously been run or the currently running workflow:

Conclusion

In this tutorial, you learned how to create a new workflow, edit an existing workflow, set up the runner for your workflow, and locally set up and work with GitHub Actions. At this point, you should be confident that you can build GitHub Actions for your projects and speed up the development processes.

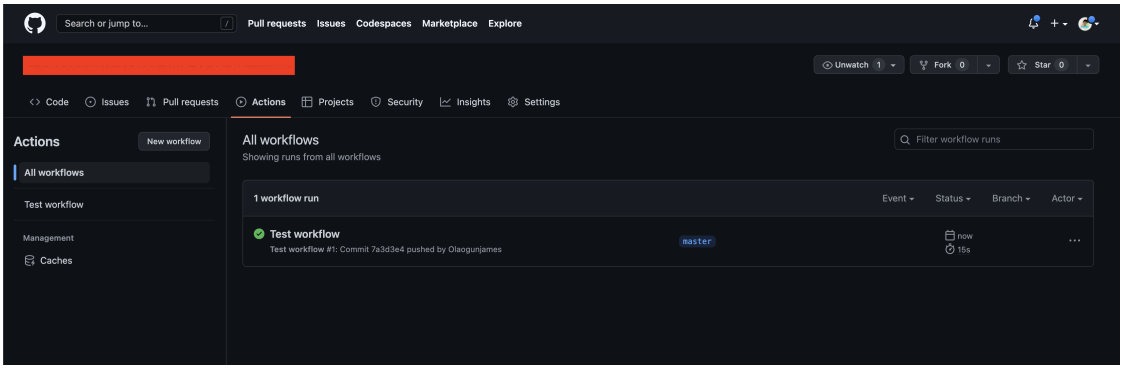


Figure 30: Workflow history