Docker Layers Explained

by Gunter Rotsaert MVB · Mar. 18, 19 · Cloud Zone · Tutorial

When you pull a Docker image, you will notice that it is pulled as different layers. Also, when you create your own Docker image, several layers are created. In this post we will try to get a better understanding of Docker layers.

1. What Is a Docker Layer?

A Docker image consists of several layers. Each layer corresponds to certain instructions in your `Dockerfile`. The following instructions create a layer: `RUN`, `COPY`, `ADD`. The other instructions will create intermediate layers and do not influence the size of your image. Let's take a look at an example. We will use a Spring Boot MVC application which we have created beforehand and where the Maven build creates our Docker image. The sources are available at GitHub.

We use the `feature/dockerbenchsecurity` branch, which is a more secure version of the `master` branch. Here is the Dockerfile:

```
FROM openjdk:10-jdk
VOLUME /tmp

RUN useradd -d /home/appuser -m -s /bin/bash appuser
USER appuser

HEALTHCHECK --interval=5m --timeout=3s CMD curl -f http://localhost:8080/actuator/health/ ||

ARG JAR_FILE
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

We build the application with `mvn clean install` which will also create the Docker image. We don't list the pulling of all layers of the `openjdk:10-jdk` image for brevity.

```
Image will be built as mydeveloperplanet/mykubernetesplanet:0.0.3-SNAPSHOT

Step 1/8 : FROM openjdk:10-jdk

Pulling from library/openjdk
Image 16e82e17faef: Pulling fs layer
...
Image a9448aba0bc3: Pull complete
Digest: sha256:9f17c917630d5e95667840029487b6561b752f1be6a3c4a90c4716907c1aad65
Status: Downloaded newer image for openjdk:10-jdk
 ---> b11e88dd885d
Step 2/8 : VOLUME /tmp

 ---> Running in 21329898c3a6
Removing intermediate container 21329898c3a6
 ---> b6f9ca000de6
Step 3/8 : RUN useradd -d /home/appuser -m -s /bin/bash appuser

 ---> Running in 82645047e6e7
Removing intermediate container 82645047e6e7
 ---> 04f6b2716819
Step 4/8 : USER appuser

 ---> Running in 697b663dadbb
Removing intermediate container 697b663dadbb
 ---> eaf6b8af5709
Step 5/8 : HEALTHCHECK --interval=5m --timeout=3s CMD curl -f http://localhost:8080/actuator/

 ---> Running in f420b9d060c5
Removing intermediate container f420b9d060c5
 ---> 77f95436a3ff
Step 6/8 : ARG JAR_FILE

 ---> Running in 60b9d25ad2ac
Removing intermediate container 60b9d25ad2ac
 ---> 135fa7df95ac
Step 7/8 : COPY ${JAR_FILE} app.jar

 ---> 63c18567012b
Step 8/8 : ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

```
 ---> Running in 79203446934a
Removing intermediate container 79203446934a
 ---> 8e2b049f9783
Successfully built 8e2b049f9783
Successfully tagged mydeveloperplanet/mykubernetesplanet:0.0.3-SNAPSHOT
```

What happens here? We notice that layers are created and most of them are removed (*removing intermediate container*). So why does it say *removing intermediate container* and not *removing intermediate layer*? That's because a build step is executed in an intermediate container. When the build step is finished executing, the intermediate container can be removed. Besides that, a layer is read-only. A layer contains the differences between the preceding layer and the current layer. On top of the layers, there is a writable layer (the current one) which is called the *container layer*. As mentioned before, only specific instructions will create a new layer. Let's take a look at our Docker images:

```
$ docker image ls
REPOSITORY                              TAG               IMAGE ID        CREATED

mydeveloperplanet/mykubernetesplanet    0.0.3-SNAPSHOT    8e2b049f9783    About a minute ago

openjdk                                 10-jdk            b11e88dd885d    2 months ago

```

And take a look at the history of our *mykubernetesplanet* Docker image:

```
$ docker history 8e2b049f9783
IMAGE           CREATED               CREATED BY                                      SIZE

8e2b049f9783    About a minute ago    /bin/sh -c #(nop) ENTRYPOINT ["java" "-Djav…    0B
63c18567012b    About a minute ago    /bin/sh -c #(nop) COPY file:2a5b71774c60e0f6…   17.4MB

135fa7df95ac    About a minute ago    /bin/sh -c #(nop) ARG JAR_FILE                  0B
77f95436a3ff    2 minutes ago         /bin/sh -c #(nop) HEALTHCHECK &{["CMD-SHELL…    0B
eaf6b8af5709    2 minutes ago         /bin/sh -c #(nop) USER appuser                  0B
04f6b2716819    2 minutes ago         /bin/sh -c useradd -d /home/appuser -m -s /b…   399kB

b6f9ca000de6    2 minutes ago         /bin/sh -c #(nop) VOLUME [/tmp]                 0B
b11e88dd885d    2 months ago          /bin/sh -c #(nop) CMD ["jshell"]                0B
<missing>       2 months ago          /bin/sh -c set -ex; if [ ! -d /usr/share/m…     697MB

<missing>       2 months ago          /bin/sh -c #(nop) ENV JAVA_DEBIAN_VERSION=1…    0B
```

```
...
```

We notice here, that the intermediate containers do have a size of 0B, just what was expected. Only the `RUN` and `COPY` command from the `Dockerfile` contribute to the size of the Docker image. The layers of the `openjdk:10-jdk` image are also listed and are recognized by the *missing* keyword. This only means that those layers are built on a different system and are not available locally.

2. Recreate the Docker Image

What happens if we run our Maven build again without making any changes to the sources?

```
Image will be built as mydeveloperplanet/mykubernetesplanet:0.0.3-SNAPSHOT

Step 1/8 : FROM openjdk:10-jdk

Pulling from library/openjdk
Digest: sha256:9f17c917630d5e95667840029487b6561b752f1be6a3c4a90c4716907c1aad65
Status: Image is up to date for openjdk:10-jdk
---> b11e88dd885d
Step 2/8 : VOLUME /tmp

---> Using cache
---> b6f9ca000de6
...
Step 6/8 : ARG JAR_FILE

---> Using cache
---> 135fa7df95ac
Step 7/8 : COPY ${JAR_FILE} app.jar

---> 409f2fee0cde
Step 8/8 : ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]


---> Running in 75f07955bbc8
Removing intermediate container 75f07955bbc8
---> e5d7b72aad05
Successfully built e5d7b72aad05
Successfully tagged mydeveloperplanet/mykubernetesplanet:0.0.3-SNAPSHOT
```

We notice that the first layers are identical to our previous build. The layer ID’s are the same. In the log, we notice that the layers are taken from the cache. In step 7 a new layer is created with a new ID. We did create a new JAR file, Docker interprets this as a new file and therefore a new layer is created. In step 8 also a new layer is created, because it is build on top of the new layer.

Let's list the Docker images again:

```
$ docker image ls
REPOSITORY                              TAG               IMAGE ID        CREATED          S
mydeveloperplanet/mykubernetesplanet    0.0.3-SNAPSHOT    e5d7b72aad05    13 seconds ago   1
<none>                                  <none>            8e2b049f9783    5 minutes ago    1
openjdk                                 10-jdk            b11e88dd885d    2 months ago     9
```

Our tag `0.0.3-SNAPSHOT` has received the *image ID* of our last build. The repository and tag of our old image ID are removed, which is indicated with the *none* keyword. This is called a *dangling* image. We will explain this in more detail at the end of this post.

When we take a look at the history of the newly-created image, we notice that the two top layers are new, just as in the build log:

```
$ docker history e5d7b72aad05
IMAGE           CREATED           CREATED BY                                      SIZE     CC
e5d7b72aad05    38 seconds ago    /bin/sh -c #(nop) ENTRYPOINT ["java" "-Djav…    0B
409f2fee0cde    42 seconds ago    /bin/sh -c #(nop) COPY file:4b04c6500d340c9e…   17.4MB
135fa7df95ac    6 minutes ago     /bin/sh -c #(nop) ARG JAR_FILE                  0B
...
```

When we make a source code change, the result is identical because also, in this case, a new JAR file is generated.

```
$ docker image ls
REPOSITORY                              TAG               IMAGE ID        CREATED          S
mydeveloperplanet/mykubernetesplanet    0.0.3-SNAPSHOT    eced642d4f5c    30 seconds ago   1
<none>                                  <none>            e5d7b72aad05    3 minutes ago    1
<none>                                  <none>            8e2b049f9783    8 minutes ago    1
openjdk                                 10-jdk            b11e88dd885d    2 months ago     9
```

```
$ docker history eced642d4f5c
IMAGE           CREATED              CREATED BY                                      SIZE
eced642d4f5c    About a minute ago   /bin/sh -c #(nop) ENTRYPOINT ["java" "-Djav…    0B
44a9097b8bad    About a minute ago   /bin/sh -c #(nop) COPY file:1d5276778b53310e…   17.4MB
135fa7df95ac    9 minutes ago        /bin/sh -c #(nop) ARG JAR_FILE                  0B
...
```

3. What About Size?

Let’s take a closer look at the latest output of the `docker image ls` command. We notice two dangling images, which are 1 GB in size. But what does this really mean for storage? First, we need to know where the data for our images are stored. Use the following command in order to retrieve the storage location:

```
$ docker image inspect eced642d4f5c
  ...
  "GraphDriver": {
    "Data": {
      "LowerDir": "/var/lib/docker/overlay2/655be8bea8e54c31ebb7e3adf05db227d194a49c1e2f95552
      "MergedDir": "/var/lib/docker/overlay2/205b55ee2f0e06394b6d17067338845410609887ccd18f53
      "UpperDir": "/var/lib/docker/overlay2/205b55ee2f0e06394b6d17067338845410609887ccd18f53b
      "WorkDir": "/var/lib/docker/overlay2/205b55ee2f0e06394b6d17067338845410609887ccd18f53bf
    },
    "Name": "overlay2"
  },
  ...
```

Our Docker images are stored at `/var/lib/docker/overlay2`. We can simply retrieve the size of the `overlay2` directory in order to have an idea of the allocated storage:

```
$ du -sh -m overlay2
1059 overlay2
```

The `openjdk:10-jdk` image is 987 MB. The JAR file in our image is 17.4 MB. The total size should be somewhere around 987 MB + 3 * 17.4 MB (two dangling images and one real image). This is about 1,040 MB. We can conclude that there is some sort of smart storage for the Docker images and that we cannot simply add all the sizes of the Docker images in order to retrieve the real storage size. The difference is due to the existence of intermediate images. These can be revealed as follows:

```
$ docker images -a
REPOSITORY                             TAG              IMAGE ID       CREATED        SIZE
mydeveloperplanet/mykubernetesplanet   0.0.3-SNAPSHOT   eced642d4f5c   7 days ago     1GB
<none>                                 <none>           44a9097b8bad   7 days ago     1GB
<none>                                 <none>           e5d7b72aad05   7 days ago     1GB
<none>                                 <none>           409f2fee0cde   7 days ago     1GB
<none>                                 <none>           8e2b049f9783   7 days ago     1GB
<none>                                 <none>           63c18567012b   7 days ago     1GB
<none>                                 <none>           135fa7df95ac   7 days ago     987MB
<none>                                 <none>           77f95436a3ff   7 days ago     987MB
<none>                                 <none>           eaf6b8af5709   7 days ago     987MB
<none>                                 <none>           04f6b2716819   7 days ago     987MB
<none>                                 <none>           b6f9ca000de6   7 days ago     987MB
openjdk                                10-jdk           b11e88dd885d   2 months ago   987MB
```

4. Get Rid of Dangling Images

How do we get rid of these dangling images? We don't need them anymore and they only allocate storage. First, list the dangling images:

```
$ docker images -f dangling=true
REPOSITORY    TAG     IMAGE ID       CREATED       SIZE
<none>        <none>  e5d7b72aad05   7 days ago    1GB
<none>        <none>  8e2b049f9783   7 days ago    1GB
```

We can use the `docker rmi` command to remove the images:

```
$ docker rmi e5d7b72aad05
Deleted: sha256:e5d7b72aad054100d142d99467c218062a2ef3bc2a0994fb589f9fc7ff004afe
```

```
Deleted: sha256:409f2fee0cde9b5144f8e92887b61e49f3ccbd2b0e601f536941d3b9be32ff47
Deleted: sha256:2162a2af22ee26f7ac9bd95c39818312dc9714b8fbfbeb892ff827be15c7795b
```

Or you can use the `docker image prune` command to do so.

Now that we have removed the dangling images, let's take a look at the size of the `overlay2` directory:

```
$ du -sh -m overlay2
1026 overlay2
```

We got rid of 33 MB. This seems like it's not so much, but when you often build Docker images, this can grow significantly over time.

5. Conclusion

In this post we tried to get a better understanding of Docker layers. We noticed that intermediate layers are created and that dangling images remain on our system if we don't clean them regularly. We also inspected the size of the Docker images on our system.

Topics: DOCKER BUILD , DOCKER , LAYERS , DANGLING IMAGES , INTERMEDIATE LAYERS , CLOUD

Deployment Manifests That Work in

Development and Production

by Gabriel Garrido · Aug 19, 19 · Cloud Zone · Analysis

Kubernetes has swiftly become the de facto standard platform for running containerized workloads. This is because Kubernetes gets a lot of things right straight out of the box, and deployment manifests for application releases can be one of them.

There are two components to typical containerized apps on Kubernetes, if you're following current recommended practices:

1. The application side—known as the d*eployment*: Users outline a *desired state* through a deployment object and the deployment controller adjusts the actual state to meet the new configuration.
2. The access endpoint definition side—known as a *service*: As pods (and their assigned IP addresses) are ephemeral by nature, thanks to their creation and destruction by replicasets, they require the abstraction of a service to define a logical set of pods as well as an access policy to communicate with them.

Put simply, deployments define which image you want to use, how many containers you want to spin up and run as well as what info is passed into them when they start. Deployments can be created using the kubectl run, kubectl apply, or kubectl create commands. In contrast, services outline the load balancer you want sitting in front of those containers and which containers will receive traffic accordingly. Services are defined by being POSTed to the apiserver to create a new instance.

More often than not, devs use plain Kubernetes manifests to roll out an application and its resources. Deployment manifests are singular to Kubernetes and refer to the file which holds a definition of a deployment object. Here's an example of a YAML Deployment manifest file:

```

apiVersion: app/v1
kind: deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

The Results:

- We create an `nginx` container deployment, as shown by the `metadata: name field`.
- The deployment launches three replicated pods, as shown by the `replicas` field.
- Each pod will be labelled `app: nginx`.
- Run version `1.7.9` the nginx image.
- Send and receive traffic through port `80`, as shown by the `containerPort` field.

Deployment manifests can be used in such a way that makes regular deployments repeatable over and over again. Using certain technologies and tools for process iteration allows users to deploy containers at scale through a consistent method without starting from scratch every time.

Manifests also become highly useful for use cases in which a developer working from a laptop needs to refactor the application release heavily when it moves from the development environment on the laptop to in production in the cloud. Here though, it's possible to make some different choices which make these environments more similar to mitigate the level of refactoring that the manifest will require. But the method requires some planning in order to do so.

To use deployment manifests that work in development and production, rather than using hard-coded host-path volumes in their dev environment manifests, it's possible to use a better pattern through persistent volume claims. Then you can exchange the storage class for the persistent volume claim based upon whether it was a dev environment or a real cloud one; so the deployment manifest will work in either place. It's just a case of provisioning a different storage class manifest for dev laptops rather than for the cloud. This method also applies in other areas where this same kind of choice is made. Such as around secrets and the environment variables passed to containers, etc.

Consider an example of what we mean as a manifest example for such purposes:

Local PV:

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv-1
  labels:
    type: local
spec:
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /tmp/data/pv-1
```

GCE PV:

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: gce-pv-1
spec:
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteOnce
  gcePersistentDisk:
    pdName: gce-1
    fsType: ext4
```

Do the two look familiar? Well, they should; each outline is similar by design. The two manifests are essentially a kind of matching template (such as can be leveraged through Helm Charts) with definitions of objects to be replicated in different environments. As you can see, here we're specifying the apiVersion as v1 and specifying the same Persistent Volume (PV) but optimizing different providers for test purposes. The manifest will work both locally on your dev machine and in Staging/Production through a GCE Provider in this instance to scale as necessary.

The next step would be to bind those Persistent Volumes with a PVC:

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
  labels:
    app: wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
```

This PVC will attach the PV that we defined previously (as the size and access modes match). If we are working locally we would need to use the PV according to the local type. On GCE, we use gcePersistentDisk as marked by `name: mysql-pv-claim`. This is the part that matters here, as we need to use it later in the pod spec.

The last step now would be to mount it:

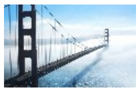
```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: wordpress-mysql
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: mysql
    spec:
      containers:
      - image: mysql:5.6
        name: mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-pass
              key: password
        livenessProbe:
          tcpSocket:
            port: 3306
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-pv-claim
```

The last section defines the volume here in the deployment manifest as `mysql-persistent-storage`, and in the spec of the pod it gets mounted using `volumeMounts`.

By writing out deployment manifest files in this manner, you can both version control and use them in a repeatable and scalable way in both development and production.

This post was originally published here.

Like This Article? Read More From DZone


Overcoming the Gap Between Kubernetes in Production and Kubernetes in Development

Configuring Your Stack on Cloud 66 With a Manifest File

CloudNativeCon + KubeCon Europe 2017 Impressions [Videos]

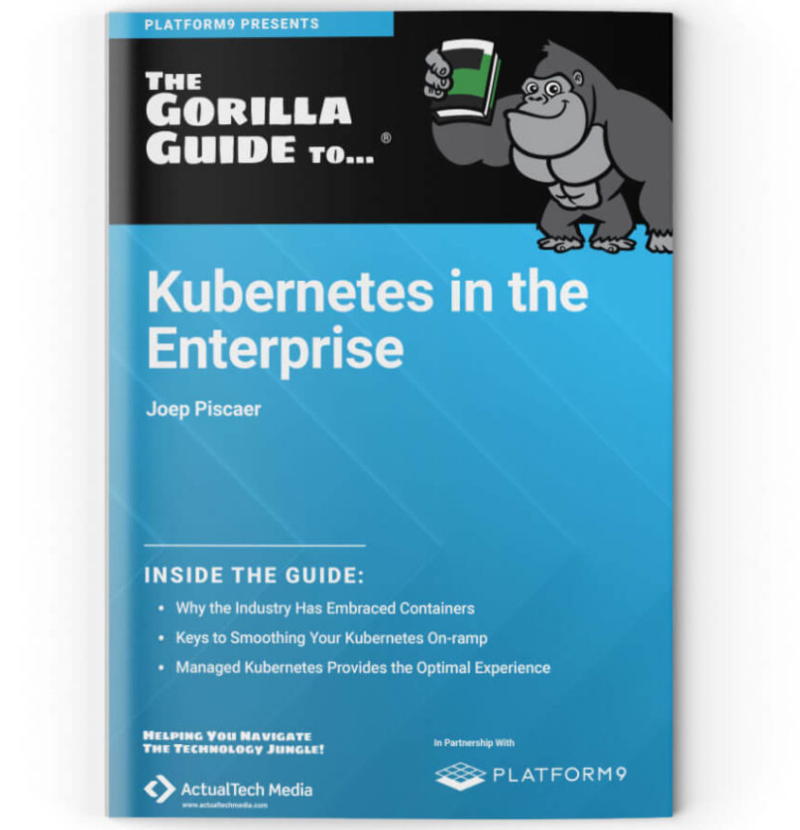
Free DZone Refcard
Cloud Native Data Grids: Hazelcast IMDG With Kubernetes

Topics: DEVELOPMENT, MANIFEST, KUBERNETES, PRODUCTION, CLOUD

Kubernetes RBAC, Monitoring

Kubernetes RBAC, Monitoring, Logging, Storage: What You Need to Know for Enterprise Use

by Vamsi Chemitiganti MVB · Aug 19, 19 · Cloud Zone · Analysis


PLATFORM9 PRESENTS

THE GORILLA GUIDE TO...®

Kubernetes in the Enterprise

Joep Piscaer

INSIDE THE GUIDE:

- Why the Industry Has Embraced Containers
- Keys to Smoothing Your Kubernetes On-ramp
- Managed Kubernetes Provides the Optimal Experience

HELPING YOU NAVIGATE THE TECHNOLOGY JUNGLE!

ActualTech Media

In Partnership With

PLATFORM9

This is an excerpt from The Gorilla Guide to Kubernetes in the Enterprise, written by Joep Piscaer. Previous Chapters: You can download the full guide here.

Now that you've deployed a basic Kubernetes cluster, let's review what it takes to put it to work for mission-critical applications in the enterprise.

Kubernetes RBAC: Giving Users Access

Kubernetes RBAC: Giving Users Access

Kubernetes uses Role-based Access Control (RBAC) to regulate user access to its resources by assigning roles to users (see illustration below). While it's possible to let all users log in using full administrator credentials, most organizations will want to limit who has full access for security, compliance and risk management reasons.

Kubernetes' approach allows administrators to limit the number of operations a user is allowed, as well as limit the scope of said operations. In practical terms, this means users can be allowed or disallowed access to resources in a namespace, as well as granular control over who can change, delete or create resources.

RBAC in Kubernetes is based on three key concepts:

1. **Verbs:** This is a set of **operations** that can be executed on resources. There are many verbs, but they're all Create, Read, Update, or Delete (also known as CRUD).
2. **API Resources:** These are the objects available on the clusters. They are the pods, services, nodes, PersistentVolumes and other things that make up Kubernetes.
3. **Subjects:** These are the objects (users, groups, processes) allowed access to the API, based on Verbs and Resources.

These three concepts combine into giving a user permission to execute certain operations on a set of resources by using **Roles** (which connects API Resources and Verbs) and **RoleBindings** (connecting subjects like users, groups and service accounts to Roles).

Users are **authenticated** using one or more authentication modes. These include client certificates, passwords, and various tokens. After this, each user action or request on the cluster is authorized against the rules assigned to a user through roles.

There are two kinds of users: service accounts managed by Kubernetes, and normal users. These normal users come from an identity store outside Kubernetes. This means that accessing Kubernetes with multiple users, or even multiple roles, is something that needs to be carefully thought out. Which **identity source** will you use? Which access control mode most suits you? Which attributes or roles should you define? For larger deployments, it's become standard to give each app a **dedicated service account** and launch the app with it. Ideally, each app would run in a **dedicated namespace**, as it's fairly easy to assign roles to namespaces.

Kubernetes does lend itself to securing namespaces, granting only permissions where needed so users don't see resources in their authorized namespace for isolation. It also limits resource creation to specific namespaces, and applies quotas.

Many organizations take this one step further and lock down access even more, so only tooling in their CI/CD pipeline can access Kubernetes, via service accounts. This locks out real, actual humans, as they're expected to interact with Kubernetes clusters only indirectly.

Monitoring and Ensuring Cluster Health

The easiest way of manually checking your cluster after deployment is via the Kubernetes Dashboard. This

The easiest way of manually checking your cluster after deployment is via the Kubernetes Dashboard. This is the default dashboard and is usually included in new clusters. The dashboard gives a graphical overview of resource usage, namespaces, nodes, volumes, and pods. The dashboard provides a quick and easy way to display information about the cluster. Because of its ease of use, it's usually the first step in a health check.

You can also use the dashboard to deploy applications, troubleshoot deployments and manage cluster resources. It can fetch an overview of applications running on your cluster, as well as create or modify individual Kubernetes resources. For example, you can scale a deployment, initiate a rolling update, restart a pod, or deploy new applications using a wizard.

The dashboard also provides information on the state of Kubernetes resources in your cluster, and any errors that may have occurred.

After deployment, it's wise to run standard conformance tests to make sure your cluster has been deployed and configured properly. The standard tool for these tests is Sonobuoy.

Clusters running as part of a service in the public cloud, like Amazon EKS, Google GKE or Azure AKS, will benefit from the managed service aspect: the cloud provider takes care of the monitoring and issue mitigation within the Kubernetes cluster. An example is Google's Cloud Monitoring service.

Monitoring and Ensuring Application Health

Most real-world Kubernetes deployments feature native, full metrics solutions. Generally speaking, there are two main categories to monitor: the cluster itself, and the pods running on top.

Kubernetes Cluster Monitoring

For cluster monitoring, the goal is to monitor the health of the Kubernetes cluster, nodes, and resource utilization across the cluster. Because the performance of this infrastructure dictates your application performance, it's a critical area.

Monitoring tools look at infrastructure telemetry: compute, storage, and network. They look at (potential) bottlenecks in the infrastructure, such as processor and memory usage, or disk and network I/O. These resources are an important part of your monitoring strategy, as they're limited to the capacity procured, and costly to expand.

There's another important reason to study these metrics: they define the behavior of the infrastructure on which the applications run, and they can serve as an early warning sign of potential issues. Should issues be identified, you can mitigate the issue before applications dependent on that infrastructure are impacted.

Kubernetes Pod Monitoring

Pod monitoring is slightly more complex. Not only do you want to correlate metrics from Kubernetes with container metrics, you also want application metrics. This requires a metrics and monitoring solution that hooks into all layers, and possibly into layers outside of the Kubernetes cluster.

As applications become more complex and distributed across multiple services, pods and containers, monitoring tools need to be aware of the taxonomy of applications, and understand dependencies between services and the business context in which they operate.

This is where solutions like DataDog, NewRelic, and AppD come in. While these are proprietary solutions, they cover the whole stack: from infrastructure, Kubernetes and containers, to application tracing. This provides a complete picture of an application, as well as transactional traces across the entire system for monitoring of the end-user experience. These solutions offer a unified metrics and monitoring experience and include rich dashboarding and alerting feature sets. Often, these products include default dashboard visualizations for monitoring Kubernetes, encompassing many standard integrations with components in the application stack to monitor up and down.

Kubernetes Monitoring with Prometheus

The most popular Kubernetes monitoring solution is the open-source Prometheus, which can natively monitor the clusters, nodes, pods, and other Kubernetes objects.

It's easily deployed via kube-prometheus, which includes **AlertManager** for alerting, **Grafana** for dashboards, and Prometheus rules combined with documentation and scripts. It provides an easy to operate, end-to-end Kubernetes cluster monitoring solution.

This stack initially monitors the Kubernetes cluster, so it's pre-configured to collect metrics from all Kubernetes components. It also delivers a default set of dashboards and alerting rules. But it's easily extended to target multiple other metric APIs to monitor end-to-end application chains.

Prometheus can monitor custom application code and has integrations with many database systems, messaging systems, storage systems, public cloud services, and logging systems. It automatically discovers new services running on Kubernetes.

** BTW - If you don't want to go through the trouble setting up and managing Prometheus on your own, check out our Managed Prometheus solution for multi-tenant, out-of-the-box Prometheus monitoring with 99.9% SLA on any environment.*

Kubernetes Logging and Tracing

Collecting metrics is just part of the puzzle. In a microservices landscape, we need to observe behavior across the multitude of microservices to get a better understanding of the application's performance. For this reason, we need both tracing and logging.

For centralized log aggregation, there are numerous options. The default option is Fluentd, a sister project of Kubernetes. On top of that, transactional tracing systems like Jaeger give insights into the user experience as they traverse the microservice landscape.

To dive deeper into Kubernetes Logging and monitoring, see our blog series on the EFK Stack.

Persistent Storage

Managing storage in production is traditionally one of the most complex and time-consuming administrative tasks. Kubernetes simplifies this by separating supply and demand.

Admins make existing, physical storage and cloud storage environments alike available using **PersistentVolumes.** Developers can consume these resources using **Claims**, without any intervention of the admins at development or deploy time. This makes the developer experience much smoother and less dependent on the admin, who in turn is freed up from responding ad-hoc to developer requests.

To learn more about Dynamic Volumes and the Container Storage Interface (CSI) see our Kubernetes Storage blog, which also includes a Minikube tutorial so you can hack on the inner workings of storage in Kubernetes!

There's More:

PLATFORM9 PRESENTS

THE GORILLA GUIDE TO...

Kubernetes in the Enterprise

Joep Piscaer

INSIDE THE GUIDE:

- Why the Industry Has Embraced Containers
- Keys to Smoothing Your Kubernetes On-ramp
- Managed Kubernetes Provides the Optimal Experience

HELPING YOU NAVIGATE THE TECHNOLOGY JUNGLE!

ActualTech Media

In Partnership with PLATFORM9

ENTERING THE JUNGLE

On the next posts we'll dive deeper into enterprise Kubernetes solutions, key use cases, and best practices for operating Kubernetes in Production, at scale.

Can't wait?

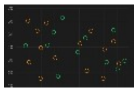
To learn more about Kubernetes in the Enterprise, download the complete guide now.

Learn About Kubernetes Security Best Practices With Our FREE Guide: Build, run-time, vul management, scanning, compliance, forensics and more.

Presented by Sysdig

Like This Article? Read More From DZone

The Gorilla Guide to Serverless on Kubernetes, Chapter 6: Simplify



Logging and Monitoring Your Kubernetes Containers

Topics: KUBERNETES, SERVERLESS, MONITORING, LOGGING, RBAC, STORAGE

Docker: Ready! Steady! Deploy!

by Dmitry Kotlyarenko · Aug 16, 19 · Cloud Zone · Tutorial

How often have you had to set up a server environment to deploy your application? Probably more often than you would like.

At best, you had a script performing all the operations automatically. At worst, you had to:

- Install the D database of the x.x.x version;
- Install the N web server of the x.x version;
- and so on...

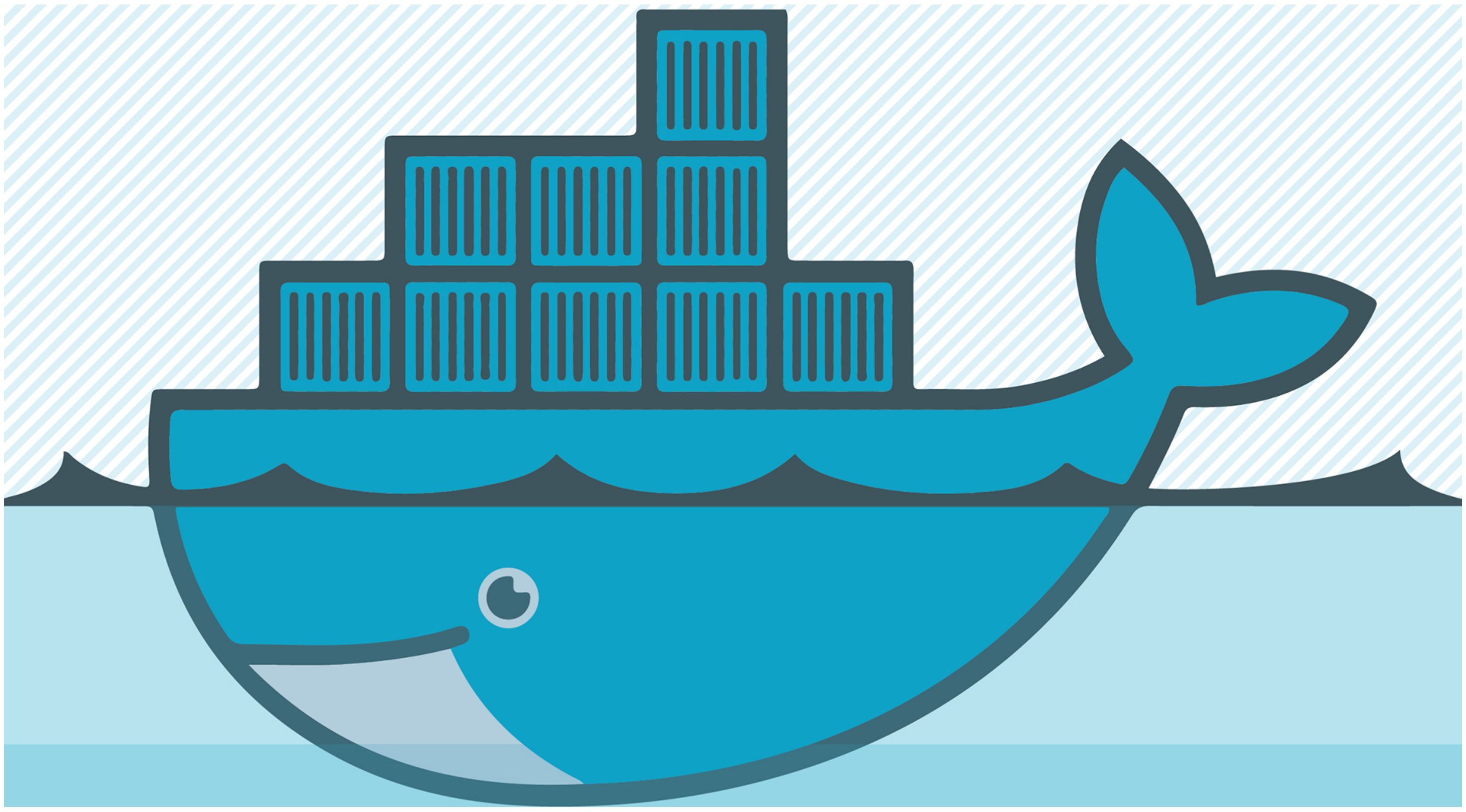
Over time, environmental management configured in this way becomes very resource-intensive. Any minor change in configuration means at least that:

- every member of a development team should be aware of this change;
- this change should be safely added to the production environment.

It is challenging to track changes and manage them without special tools. One way or another, problems with the configuration of environment dependencies arise. As development continues, it becomes more and more challenging to find and fix bottlenecks.

I have described what is called vendor lock-in. This phenomenon becomes quite a challenge in the development of applications, in particular, server-type ones.

In this article, we will consider one of the possible solutions — Docker. You will learn how to create, deploy, and run an application based on this tool.



Disclaimer: This is not a review of Docker. This is the first entry point for developers who plan to deploy Node.JS applications using Docker containers.

While developing one of our projects, my team faced a lack of detailed articles on the topic, and this greatly complicated our work. This post was written to simplify the pathways of colleague developers who will follow our footsteps.

What is Docker?

Simply put, Docker is an abstraction over LXC containers. This means that processes launched using Docker will see only themselves and their descendants. Such processes are called Docker containers.

Images (`docker image`) are used for creating abstractions based on such containers. It is also possible to configure and create containers based on Docker images.

There are thousands of ready-made Docker images with pre-installed databases, web servers, and other important elements. Another advantage of Docker is that it is a very economical (in terms of memory consumption) tool since it uses only the resources it needs.

Let’s Get Closer

We will not dwell on the installation. Over the past few releases, the process has been simplified to a few clicks/commands.

In this article, we will analyze the deployment of a Docker application using the example of a server-side Node.js app. Here is its primitive, source code:

```
// index

const http = require('http');

const server = http.createServer(function(req, res) {

res.write('hello world from Docker');

res.end();

});

server.listen(3000, function() {

console.log('server in docker container is started on port : 3000');

});
```

We have at least two ways to package the application in a Docker container:

- create and run a container using an existing image and the command-line-interface tool;
- create your own image based on a ready sample.

The second method is used more often.

To get started, download the official node.js image:

docker pull node

The "docker pull" command downloads a Docker image. After that, you can use the "docker run" command. This way you will create and run the container based on the downloaded image.

docker run -it -d --rm -v "$ PWD": / app -w = / app -p 80: 3000 node node index.js

This command will launch the index.js file, map a 3000 port to an 80 port, and display the ID of the created container. Much better! But you will not go far with CLI only. Let's create a Dockerfile for a server.

```
FROM node

WORKDIR /app

```

```
RUN cp . /app

CMD ["node", "index.js"]
```

This Dockerfile describes the parent image of the current version, as well as the directory for starting container commands and the command for copying files from the directory where the image assembly is being launched. The last line indicates which command will run in the created container.

Next, we need to build an image that we will deploy based on this Dockerfile: `docker build -t username / helloworld-with-docker: 0.1.0.` This command creates a new image, marks it with username `helloworld-with-docker` and creates a 0.1.0 tag.

Our container is ready. We can run it with the `docker run` command. The vendor lock-in problem is solved. The launch of the application is no longer dependent on the environment. The code is delivered along with the Docker image. These two criteria allow us to deploy the application to any place where we can run Docker.

Deploy

The first part is not as tricky as the remaining steps.

After we have completed all the instructions above, the deployment process itself becomes a matter of technics and your development environment. We will consider two options for deploying Docker:

- Manual deployment of Docker image;
- Deployment using Travis-CI.

In each case, we will consider delivering the image to an independent environment, for example, a staging server.

Manual Deployment

This option should be chosen if you do not have a continuous integration environment. First, you need to upload the Docker image to a place accessible to the staging server. In our case, it is a DockerHub. It provides each user with a free private image repository and an unlimited number of public repositories.

Log in to access the DockerHub:

```
docker login -e username@gmail.com -u username -p userpass
```

Upload the image:

```
docker push username/helloworld-with-docker:0.1.0
```

Next, go to the staging server (Docker must be already preinstalled on it). To deploy the application on the server, we need to execute only one command: `docker run -d --rm -p 80: 3000 username / helloworld-with-docker: 0.1.0` . And that's all!

Check the local register of images. If you don't find your image, enter username `helloworld-with-docker` to check the DockerHub registry. An image with the name you indicate must be in the register since we have already uploaded it there. Docker will download it, create a container on its basis, and launch your application in it.

From this moment, every time you need to update the version of your application, you can make a push with a new tag and just restart the container on the server.

P.S. This method is not recommended if Travis-CI is available.

Deployment Using Travis-CI

First, we should add DockerHub data to Travis-CI. They will be stored in environment variables.

```
travis encrypt DOCKER_EMAIL=email@gmail.com

travis encrypt DOCKER_USER=username

travis encrypt DOCKER_PASS=password
```

Then we should add the received keys to the .travis.yml file. We should also add a comment to each key in order to distinguish between them in the future.

```
env:

  global:

    - secure: "UkF2CHX0lUZ...VI/LE=" # DOCKER_EMAIL

    - secure: "Z3fdBNPt5hR...VI/LE=" # DOCKER_USER

    - secure: "F4XbD6WybHC...VI/LE=" # DOCKER_PASS
```

Next, we need to login and upload the image:

```
after_success:

  - docker login -e $DOCKER_EMAIL -u $DOCKER_USER -p $DOCKER_PASS

  - docker build -f Dockerfile -t username/hello-world-with-travis.

  - docker tag username/hello-world-with-travis 0.1.0

  - docker push username/hello-world-with-travis
```

Also, image delivery can be launched from Travis-CI in various ways:

- manually;
- via SSH connection;
- online deploy services (Deploy Bot, deployhq);
- AWS CLI;
- Kubernetes;
- Docker deployment tools.

Conclusion

We have considered two ways (automatic and via Travis-CI) of preparation and deployment of Docker using an example of a simple node.js server. The knowledge gained should greatly simplify your work. Thanks for your time!

Topics: DOCKER APPLICATION, DEPLOY, APP DEV, NODE, TRAVIS-CI, CLOUD, DEPLOYMENT, AUTOMATION