Container Image Optimisation and Security Practices

Aarush Gupta
Computer science and Engineering
R.V College of Engineering
Bangalore, India

Prof. Prapulla SB
Computer science and Engineering
R.V College of Engineering
Bangalore, India

***Abstract* – Cloud Computing is one of the major developments that offers a promising future. One of the basic tools which makes the Cloud a reality is virtualization. Hypervisor-based virtualization comes with high performance and overhead performance due to the added level of abstraction. Virtual Machine results in high service downtime whenever the application is updated to the new version. Containers have various advantages over virtual machines due to performance enhancements and reduced start time. Docker is widely used in the business environment and at the personal level container environment. If the Docker image has been stored locally on the server, Docker technology can frequently reduce the launch time from a few minutes to less than 5 seconds. However, these container images are very customisable and are typically created at runtime by executing script instructions from a remote base image (the Docker file). Many input files, including the packages and dependencies, may need to be acquired from the Internet during the execution of the instruction. The process of creating an image can be time-consuming and iterative.**

Keywords – Docker, Docker Images, Base Images, Multi-staging and Multilayering

I. Introduction

The ability to replicate and reproduce scientific results has become a major topic in many academic disciplines. In computer science and, in particular, software and web engineering, the contributions of scientific work in computer science, and particularly in software and web engineering, are based on sophisticated algorithms, tools and prototypes, quantitative tests, and other computer-based analyses[1]. It is challenging to replicate published code and data because it contains several unstated assumptions, dependencies, and configurations that constitute internal knowledge.

A lightweight virtualization technology called containerization makes it possible to install and run distributed applications on platforms including the cloud[2], the edge, and the Internet of Things. There are numerous unresolved scientific difficulties and container technologies are developing at the speed of light. Adoption of container technologies in High Performance Computing, Big Data, and spatially dispersed applications faces obstacles.Those challenges include performance, orchestration and cyber-security.

Because all applications with the exception of those hosted by a particular system look the same, Docker[3] facilitates easier design decisions. It facilitates the creation and sharing of tools between apps. Nothing in our world is without advantages and obstacles, but Docker is extraordinarily tilted in its favour. In this paper, Our main focus would be to discuss the best practices in order to reduce the redundancies in building the docker file and getting an optimised container with minimal size.

II. Related Terminologies and Process flow

Here are the few terms which we should be aware about before moving further:

- Docker client : Docker command which used to control Docker workflow. It is also useful in talking to remote Docker servers.
- Docker server : The Docker command starts the Docker server process, which then creates and launches containers via a client.
- Docker images : It comprises one or more file system layouts and important metadata for each file necessary to operate a containerized application. Multiple hosts can be copied from a single Docker image. Typically, an image has a name and a tag. The tag is typically used to denote a particular image release. From Docker Hub, you may download the base images. [4]
- Docker container : A Linux container that has been created from a Docker image is known as a Docker container. There can only be one instance of a given container, but you may quickly make more containers from the same image.
- Atomic host : It is a small, finely tuned Operating System image similar to Fedora CoreOS , which supports container hosting and atomic OS upgrades.

The Docker Container Lifecycle[5] describes the many stages that a Docker container goes through. A few of the states include:

- **Created:** A newly created but unstarted container
- **Started/Running**: A container which is running with all its processes

docker start 6b785f78b75e

- **Paused:** A container in which processes have been paused. To pause use the SIGSTOP signal, and to un-pause, the SIGCONT signal.

  docker pause 6b785f78b75e

- **Stopped:** The antithesis of Running, a container that houses stopped operations. Sending the SIGTERM signal to the primary container process, or PID 1, which is located in the container namespace, stops a container.

  docker stop 6b785f78b75e

- **Destroyed/Deleted**: A container in a dead state goal is to assess and evaluate numerous dashboard templates in light of the established criteria.

  docker kill 6b785f78b75e

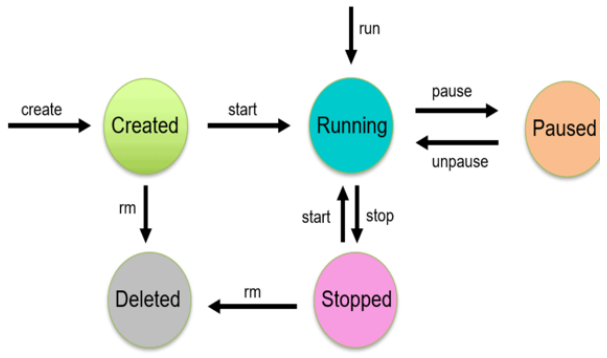

Fig1. Docker Life Cycle

The Docker daemon manages Docker objects like images, containers, networks, and volumes while listening for requests made over the Docker API. To manage Docker services, a daemon can also talk to other daemons.

III. PRACTICES INVOLVED FOR MINIMIZATION OF SIZE OF CONTAINER[6]

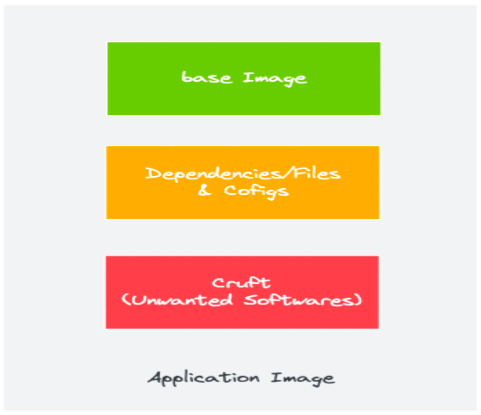

Fig 2. Image Composition

- **Using minimal and distort less base-images**

The selection of an appropriate base image with the smallest possible operating system footprint would be the first priority.

Alpine is the perfect candidate for such purpose. This image is as small as 4-6 Megabytes. It is small as well as secure. The other image which can be used is Nginx alpine which is of 22 Megabytes. This image by default comes with sh shell which helps in debugging the container.

The image size can be again reduced by using distort less images. Distort-less image[7] is a stripped down version of a operating system. These images are available in particular for python ,rust java etc. However, it is not at all recommended to use publicly available base images for enterprise use due to security reason.

Base Image Comparison (in Mega Bytes)

ubuntu	busybox	centos	opensuse	alpine
188 MB	2 MB	172 MB	82 MB	5 MB

Busy box does not comes with package manager therefore not preferred. When changing the base image to alpine, caution should be taken about mapping and equivalent packages from traditional say, Debian based OS which support apt package manager to apk based package manager. The package might differ in name and the version should be matched carefully while migrating from one to another.

Package Installation in Debian (Linux)

apt install [package name]

While in Alpine

apk add [package name]

- **Use Docker Multistage Builds**

The multi-stage build divides the Docker-file into several sections in order to transfer the required artifact from one stage to another and ultimately bring the final artifact to the final stage. This way, our final image will not contain unnecessary content except the required artifact.

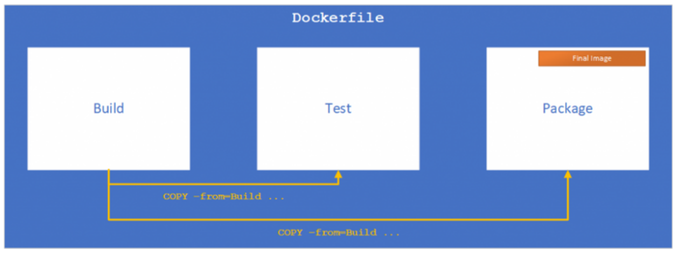

Fig 3. Multi-Staging in Docker

Code Snippet:

```
FROM alpine AS base
RUN apk add -no-cache curl wget
FROM nginx
COPY nginx.conf/nginx/conf.d/default.conf
COPY --from=builder /app/build /usr/nginx/html
FROM base
COPY --from=go-builder /go/main /main
CMD ["/main "]
```

Memory Idle	Normal	Multi Staging
RAM	242.9 MB	6.4 MB
Storage	369 MB	32 MB

This is RAM utilisation and storage comparison while working with Normal docker file and while using multi-staging[8] to build the docker file. Another benefits of using multistaging gains extra minutes and bandwidth to upload. It also helps in removing unnecessary dependencies and reducing the exposition of your image, you drastically reduce the security risks. It minimizes the surface attacks. While boosting up the security of the container. It reduces the cost and makes easy to incorporate multiple dependencies inside the container.

- **Minimizing the Number of Layers**

Each RUN, COPY, FROM Docker-file instruction adds a new layer to a Docker image, which increases the image's storage needs and lengthens the build's execution time. Minimizing the layers means to install all the packages on a single RUN command which reduce the number of steps in the build process and further reduce the size of the image.
However, If a new package is added, the entire needs to be rebuild again. To demonstrate multilayering, For instance

Code Snippet:

```
FROM alpine
RUN apk add <packageA>
RUN apk add <packageB>
```

This can be translated to

```
FROM alpine
RUN apk add <packageA><packageB>
```

Sometimes repositories used by package managers have links between dependencies, of the form “A requires B to work” (strong link), or “A profits from also having B installed” (weak link). Some package managers have an switch to disable the latter link-type (which you need to enable explicitly).In the above command, using --no-install-recommends flag to disable recommended packages also significantly reduces the size of the image. Avoiding packages that are suggested for installation with the necessary packages even when they aren't really dependent is beneficial.

- **Use Caching**

When the docker image needs to be rebuilt repeatedly with minor modifications to the packages and code, the caching concept[9] can be utilized. The caching feature of Docker offers assistance in these situations by preserving the cache level data for each build layer that may be useful for subsequent building.
It is advised to put lines of code for installation and packages before the COPY instructions in the Docker file. The reasoning behind this is that the docker will be able to cache the required dependencies and use this cache when the code is modified. An snippet is given for reference

Code Snippet:

```
FROM alpine:latest
COPY . .
RUN apk add update
RUN apk add upgrade
RUN apk add vim
```

- **Using .dockerignore and keeping the application data elsewhere separately**

In git, .gitignore is used to ignore unnecessary node modules, document and git files. Similar to this, if it is configured in the .dockerignore file, Docker will likewise ignore the files that are present in the working directory. As a general guideline, only files that are required and necessary should be copied over to the docker image.
An example are data files (e.g. big CSV files with raw data required only for automated tests), which someone incorrectly placed in the "src" folder whose entire content is copied into the image via a COPY statement in your Dockerfile.

Storing the application data within the image will redundantly increase the size of the images, therefore volume feature of the container runtimes is used to keep the image separate from the data.

- **ADD rm -rf /lib/var/apt/lists/* TO SAME LAYER AS apt-get installs**

Add rm -rf /var/lib/apt/lists/* at the end of the apt-get -y install command to do clean up after installing packages if some of the packages are intermediate dependencies and are no longer needed. The most useful use case is to bundle them all in a single RUN sentence while installing wget or curl to download a package. When curl or wget are no longer needed, the statement executes apt-get remove at the conclusion of the run.

IV. PRACTICES INVOLVED FOR SECURING[10] THE CONTAINER

- **Least privileged user**

It is advised to set up a specific user or group on the image with the bare minimum of rights in order to run the application[11]. This process is executed by the same user. a Node.js image with a built-in Node generic user, for instance,

Code Snippet:

```
FROM node:12-alpine
USER node
CMD node index.js
```

- **Using signatures and image verification to prevent Man in the middle[12] attacks**

We place great faith in docker images. It is important to make sure that the image we are taking is the one is the original and checked ones and not been tampered. Therefore recommended to verify the trust and authenticity of the image using notary

- **Use fixed tags for immutability**

Owners of container images can ignore risks and push new versions to the same tags, which could lead to inconsistency between the images during builds and make it difficult to determine if a vulnerability has been patched or not. Consequently, choose one of the following:

a. A verbose image tag with which to pin both version and operating system, for example:

FROM node:8-alpine

b. A contact-specific image hash, for example:

FROM node:<hash>

- **Use labels for metadata**

Users can find useful information, such as security information, in labels with image metadata. Therefore, adopt an SECURITY.TXT policy file, refer to the Responsible Security Disclosure Policy, and provide this information to image labels.

V. CONCLUSION

As was mentioned, there are many ways to reduce the size of the container image. Some of them take efficiency a step further and rely on choices that work for every facet of your application. In the end, efficient images are a trade-off between size and what is required to accurately and simply support your application. The best route for you will rely on your architecture and your objectives, but you should also consider how heavy or light you want your images to be in the end because there may be drawbacks in handling the packages. The most widely used container management option at the moment is Docker.
Given this prevalence and the effects it has on the Docker ecosystem, it is crucial to understand optimisation trends and security concerns so that container - based apps can be built in the future.

REFERENCES

[1] N. Kratzke, "A brief history of cloud application architectures," Applied Sciences, vol. 8, no. 8, p. 1368, 2018.
[2] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: a state-of-the-art review," IEEE Transactions on Cloud Computing, 2017.
[3] H. Zhu and I. Bayley, "If docker is the answer, what is the question?" in 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE). IEEE, 2018, pp. 152–163.
[4] "Docker hub quickstart," https://docs.docker.com/docker-hub/, 2020,
Online; accessed April-8-2020].

[5] Senthil Nathan, Rahul Ghosh, Tridib Mukherjee, and Krishnaprasad Narayanan. CoMICon: A Co-Operative Management System for Docker Container Images. In Proceedings of the 5th IEEE International Conference on Cloud Engineering (IC2E), 2017.

[6] "Docker development best practices," https://docs.docker.com/develop/dev-best-practices/, 2020, [Online; accessed May-6-2020].

[7] T. Xu and D. Marinov, "Mining container image repositories for software configuration and beyond," in 2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER). IEEE, 2018, pp. 49–52.

[8] Boettiger, C. (2015). An introduction to Docker for reproducible research. ACM SIGOPS Operating Systems Review, 49(1), 71-79.

[9] Vase, T. (2015). Advantages of Docker.

[10] Bui, T. (2015). Analysis of docker security. arXiv preprint arXiv:1501.02967.

[11] B. Tak, H. Kim, S. Suneja, C. Isci, and P. Kudva, "Security analysis of container images using cloud analytics framework," in International Conference on Web Services. Springer, 2018, pp. 116–133.

[12] McArdle, Gavin & Kitchin, Rob. (2016). THE DUBLIN DASHBOARD: DESIGN AND DEVELOPMENT OF A REAL-TIME ANALYTICAL URBAN DASHBOARD. ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences. IV-4/W1. 19-25. 10.5194/isprs-annals-IV-4-W1-19-2016.