SoftWare Repository for Container

Best Practices

Issue 01

Date 2022-09-30

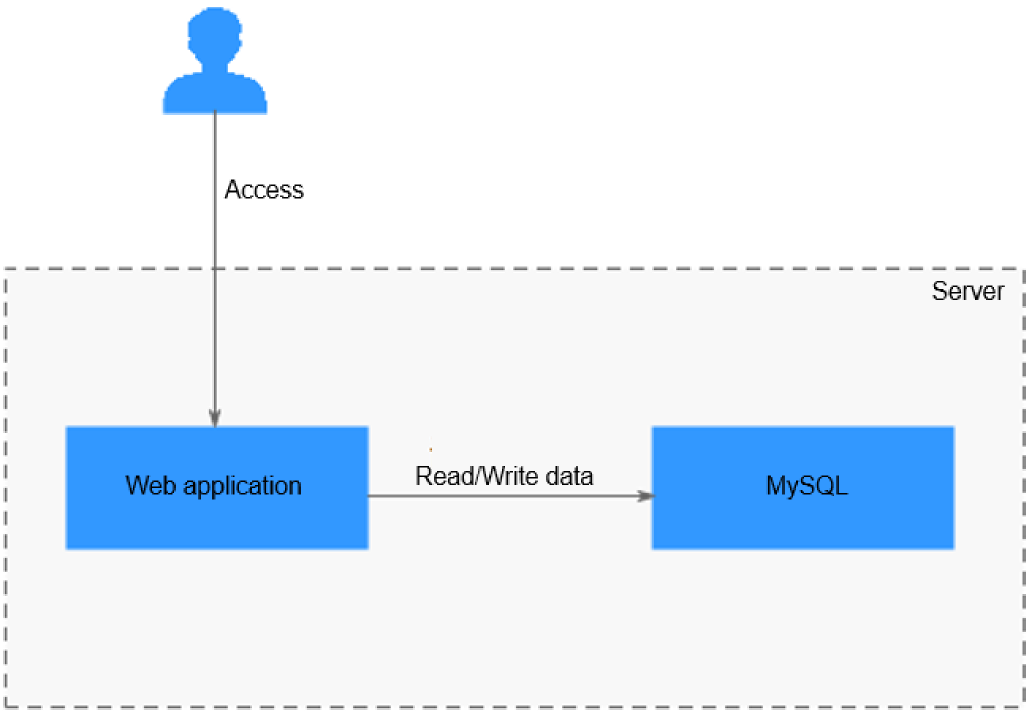
HUAWEI TECHNOLOGIES CO., LTD.

Contents

1 Writing a Quality Dockerfile

This document walks you through how to compile an efficient Dockerfile, using the containerization of an application as an example. Based on the practices of SWR, this file exemplifies how to create images of fewer layers and smaller size to speed up image build process.

The following figure shows a common architecture of an enterprise portal website. This website consists of a web application that provides web services, and a database that saves user data. Normally, the website is deployed on a single server.



To containerize the application, a Dockerfile may be written as follows:

```
FROM ubuntu

ADD . /app

RUN apt-get update
RUN apt-get upgrade -y
RUN apt-get install -y nodejs ssh mysql
RUN cd /app && npm install

# this should start three processes, mysql and ssh
# in the background and node app in foreground
# isn't it beautifully terrible? <3
CMD mysql & sshd & npm start
```

However, the preceding Dockerfile, including the **CMD** command, is problematic.

To rectify and optimize the Dockerfile, here are some tips:

- **Run Only One Process in Each Container**
- **Do Not Upgrade a Version During Image Build**
- **Merge Multiple RUN Commands that Are of Similar Updating Probability**
- **Specify Image Tags**
- **Delete Unnecessary Files**
- **Select a Suitable Base Image**
- **Set WORKDIR and CMD**
- **(Optional) Use ENTRYPOINT**
- **Run the exec Command in ENTRYPOINT**
- **Use the COPY Instruction Preferentially**
- **Adjust the Order of COPY and RUN Commands**
- **Set Default Environment Variables, Mapping Ports, and Data Volumes**
- **Use the EXPOSE Command to Specify Listening Ports**
- **Use the VOLUME Command to Manage Data Volumes**
- **Use Labels to Configure Image Metadata**
- **Add the HEALTHCHECK Instruction**
- **Compile the .dockerignore File**

Run Only One Process in Each Container

Technically, multiple processes, including database, frontend, backend, and SSH, can run on the same Docker container. However, this is not what containers are built for. Stuffing all the processes into one container not only makes the image extremely large in size, but also prolongs the container building time and wastes resources when you perform horizontal scaling. This is because the whole container has to be rebuilt every time you make small adjustments and the number of containers for each application can only be equally added during scaling in.

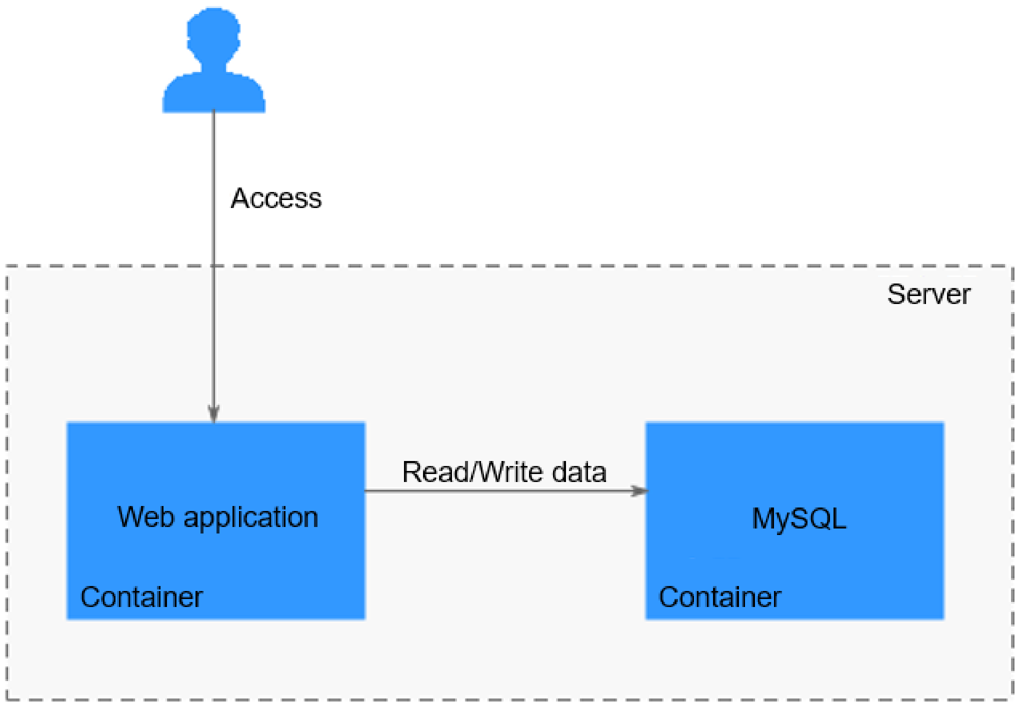
Therefore, usually, an application is split into microservices before containerization. You will benefit a lot from the microservice architecture:

- **Independent scaling**: After an application is split into independent microservices, you can adjust the number of pods for each microservice separately.
- **Faster development**: Since microservices are decoupled, they can be coded independently from each other.
- **Security assurance through isolation**: For an overall application, if a security vulnerability exists, attackers can use this vulnerability to obtain the

permission to all functions of the application. However, in a microservice architecture, if a service is attacked, attackers can only obtain the access permission to this service, but cannot intrude other services.

- **Stabler service**: If one microservice breaks down, other microservices can still run properly.

To optimize the preceding sample Dockerfile, run the web application and MySQL in different containers.



You can modify them separately. As shown in the following example, MySQL is deleted from the sample Dockerfile. Only Node.js is installed.

```
FROM ubuntu

ADD . /app

RUN apt-get update
RUN apt-get upgrade -y

RUN apt-get install -y nodejs
RUN cd /app && npm install

CMD npm start
```

Do Not Upgrade a Version During Image Build

To reduce image complexity, dependency, size, and build time, do not install any unnecessary packages in your images. For example, do not include a text editor in a database image.

Contact the package maintenance personnel if a package in the base image is out of date but you do not know which package it is. To upgrade a specific package automatically, for example, **foo**, run the **apt-get install -y foo** command.

apt-get upgrade brings great uncertainty to image build. Inconsistency between images might occur as you are not sure what packages have been installed by

apt-get upgrade during image build. Therefore, **apt-get upgrade** is usually deleted.

The following is the sample Dockerfile without **apt-get upgrade**:

```
FROM ubuntu

ADD . /app

RUN apt-get update

RUN apt-get install -y nodejs
RUN cd /app && npm install

CMD npm start
```

Merge Multiple RUN Commands that Are of Similar Updating Probability

Like an onion, a Docker image consists of many layers. To modify an inner layer, you need to delete all outer layers. Docker images have the following features:

- Each instruction in a Dockerfile creates an image layer.
- Image layers are cached and reused.
- Cached image layers expire when the files they copy or variables specified in image build change.
- When a cached image layer expires, its subsequent cached image layers expire accordingly.
- Image layers are immutable. If a file is added into a layer and then deleted in the next layer, the file still exists in the image. The file just turns unavailable in the Docker container.

Therefore, merge multiple instructions that are of similar updating probability to avoid unnecessary costs. In the sample Dockerfile, **Node.js** and **npm** are installed together. That means **Node.js** is reinstalled each time the source code is modified, which is time and resource consuming.

```
FROM ubuntu

ADD . /app

RUN apt-get update \
    && apt-get install -y nodejs \
    && cd /app \
    && npm install

CMD npm start
```

It would be better to write the Dockerfile as follows:

```
FROM ubuntu

RUN apt-get update && apt-get install -y nodejs
ADD . /app
RUN cd /app && npm install

CMD npm start
```

Specify Image Tags

When no tag is specified to an image, the tag **latest** is automatically used. Therefore, the **FROM ubuntu** instruction is the same as **FROM ubuntu:latest**.

During an image update, when the **latest** tag points to a new version, image build may fail.

To specify a tag for the **ubuntu** image in the sample Dockerfile, tag the image with **16.04** as follows:

```
FROM ubuntu:16.04

RUN apt-get update && apt-get install -y nodejs
ADD . /app
RUN cd /app && npm install

CMD npm start
```

Delete Unnecessary Files

Assume that you have updated the **apt-get** sources, installed some software packages, and saved them in the **/var/lib/apt/lists/** directory.

However, these files are not required in application running. To make the Docker image more lightweight, it is advised to delete these unnecessary files.

Therefore, in the sample Dockerfile, the files in the **/var/lib/apt/lists/** directory are deleted.

```
FROM ubuntu:16.04

RUN apt-get update \
    && apt-get install -y nodejs \
    && rm -rf /var/lib/apt/lists/*

ADD . /app
RUN cd /app && npm install

CMD npm start
```

Select a Suitable Base Image

In our sample Dockerfile, **ubuntu** is selected as the base image. However, as you only need to run the node program, there is no need to use a general-purpose base image. A node image would be a better choice.

The node image of the Alpine version is recommended. It is a mini-Linux operating system with a size of only 4 MB.

```
FROM node:7-alpine

ADD . /app
RUN cd /app && npm install

CMD npm start
```

Set WORKDIR and CMD

The **WORKDIR** instruction can be used to set a default directory where **RUN**, **CMD**, and **ENTRYPOINT** instructions will be run.

The **CMD** instruction provides default commands for an executing container. Write the instruction in an array with each element of the array being a word of the command.

```
FROM node:7-alpine

```

```
WORKDIR /app
ADD . /app
RUN npm install

CMD ["npm", "start"]
```

(Optional) Use ENTRYPOINT

The **ENTRYPOINT** instruction is optional because it increases complexity. **ENTRYPOINT** is a script that is executed by default. It uses the specified commands as its arguments. It is usually used to create executable container images.

```
FROM node:7-alpine

WORKDIR /app
ADD . /app
RUN npm install

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

Run the exec Command in ENTRYPOINT

In the preceding **ENTRYPOINT** script, the **exec** command is used to run the node application. If the **exec** command is not used, the container cannot be successfully closed since the SIGTERM signal will be swallowed by the bash script process. The process started by running the **exec** command can replace the script process. In this way, all signals work normally.

Use the COPY Instruction Preferentially

The **COPY** instruction is simple. It is only used to copy files to images. Compared with it, the **ADD** instruction is more complex. **ADD** can be used to download remote files and decompress compressed packages.

```
FROM node:7-alpine

WORKDIR /app

COPY . /app
RUN npm install

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

Adjust the Order of COPY and RUN Commands

Place the parts that are not to be changed frequently at the front of your Dockerfile to make the most out of the image cache.

In the example Dockerfile, its source code changes frequently. Every time the image is built, the NPM is reinstalled. For example, copy **package.json** first, then install NPM, at last copy the rest of the source code. In this way, changes of the source code will not result in repetitive installation of NPM.

```
FROM node:7-alpine

WORKDIR /app

COPY package.json /app
```

```
RUN npm install
COPY . /app

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

Set Default Environment Variables, Mapping Ports, and Data Volumes

Environment variables may be required when running a Docker container. Setting default environment variables in Dockerfile is a good choice. In addition, you can set mapping ports and data volumes in the Dockerfile. Example:

```
FROM node:7-alpine

ENV PROJECT_DIR=/app

WORKDIR $PROJECT_DIR

COPY package.json $PROJECT_DIR
RUN npm install
COPY . $PROJECT_DIR

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

The environment variable specified by the **ENV** instruction can be used in a container. If you only need to specify the variables used in image build, you can use the **ARG** instruction.

Use the EXPOSE Command to Specify Listening Ports

The **EXPOSE** instruction is used to specify listening ports to containers. Select common ports for your applications. For example, for the image that provides the Apache web service, use **EXPOSE 80**; while for the image that provides the MongoDB service, use **EXPOSE 27017**.

For external access, use a flag to indicate how to map a specified port to the selected port during the execution of the **docker run** command.

```
FROM node:7-alpine

ENV PROJECT_DIR=/app

WORKDIR $PROJECT_DIR

COPY package.json $PROJECT_DIR
RUN npm install
COPY . $PROJECT_DIR

ENV APP_PORT=3000
EXPOSE $APP_PORT

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

Use the VOLUME Command to Manage Data Volumes

The **VOLUME** instruction is used to access database storage files, configuration files, or files and directories of created containers. It is strongly recommended that the **VOLUME** instruction be used to manage the image modules that can change or the modules modifiable for users.

In the example Dockerfile, a media directory is entered.

```
FROM node:7-alpine

ENV PROJECT_DIR=/app

WORKDIR $PROJECT_DIR

COPY package.json $PROJECT_DIR
RUN npm install
COPY . $PROJECT_DIR

ENV MEDIA_DIR=/media \
    APP_PORT=3000

VOLUME $MEDIA_DIR
EXPOSE $APP_PORT

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

Use Labels to Configure Image Metadata

Add labels to images to help organize images, record permissions, and automatic image build. Starting with **LABEL**, add one or more labels with each label occupying one line.

NOTICE

If your string contains spaces, put the string in quotation marks ("") or convert it into escape sequence. If the string itself contains quotation marks, convert the quotation marks.

```
FROM node:7-alpine
LABEL com.example.version="0.0.1-beta"
```

Add the HEALTHCHECK Instruction

When running a container, you can enable the **--restart always** option. In this case, the Docker daemon restarts the container when the container crashes. This option is useful for containers that need to run for a long time. What if a container is running but unavailable? The **HEALTHCHECK** instruction enables Docker to periodically check the health status of containers. You only need to specify a command. If the containers are normal, **0** is returned. Otherwise, **1** is returned. When the request fails and the **curl --fail** command is run, a non-zero state is returned. Example:

```
FROM node:7-alpine
LABEL com.example.version="0.0.1-beta"

ENV PROJECT_DIR=/app
WORKDIR $PROJECT_DIR

COPY package.json $PROJECT_DIR
RUN npm install
COPY . $PROJECT_DIR

ENV MEDIA_DIR=/media \
    APP_PORT=3000

VOLUME $MEDIA_DIR
EXPOSE $APP_PORT
HEALTHCHECK CMD curl --fail http://localhost:$APP_PORT || exit 1
```

```
ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

Compile the .dockerignore File

The functions and syntax of the **.dockerignore** file are similar to those of the **.gitignore** file. You can ignore unnecessary files to accelerate image creation and reduce the sizes of Docker images.

Before image build, Docker needs to prepare the context by collecting all required files to the process. By default, the context contains all files in the Dockerfile directory. However, files such as the **.git** directory are unnecessary.

Example:

```
.git/
```