Docker Essentials: Creating Custom Image Explained

Working with docker means, using Images of other engineers as base and add our configuration layer. This time we are going to demonstrate baking our own docker image. To create our own image, we have to do the followings,

- Create a `dockerfile`.
- Once a `dockerfile` is created, we will pass it to the `docker client`
- This `docker client` will provide the `image` to the `docker server`
- The `docker-server` will make the `image`. This image can be used by other engineers.

  A `dockerfile` is a plain text file. It contains all the configs along with the startup commands to define how a docker `container` should behave. It determine what programs should be inside the container and how will these program will work around in the `container` star up.

  `Docker server` does the most of heavy lifting while creating a `image`. It takes the docker file, go through the configuration and build the useable `image`

For each `docker file`, there's always should be a pattern,

- A `base image`
- Some config to install additional programs and dependencies that is required to create and execute the container program
- A start up command, that will be executed on the moment a container start

Hands On

Let's create our own docker `image` that will run a `redis-server`. We will do the following procedures

1. Define base `image`
2. Download and install the `dependencies`
3. Instruct the images initial behaviour

So,

- First create a file named `Dockerfile`

```
bash touch Dockerfile
```

- In the `Dockerfile`, add a base image

```
bash FROM alpine
```

- Download the dependency

```
bash RUN apk add --update redis
```

- Define the initial command

```
bash CMD ["redis-server"]
```

Finally our `Dockerfile` should be the following

```
FROM alpine
RUN apk add --update redis
CMD ["redis-server"]
```

Now let's build the container

```
docker build .
```

This will create the image and return the `image_id`.

Now we can run the `container` from the `image_id` using the followings,

```
docker run image_id
```

This will run the redis server.

> `Dockerfile` is a plain file with no extension

Dockerfile Teardown

We just going through the process of creating a docker image. But we don't explain what really happen there. Now lets explain what actually we have done inside the `Dockerfile` configuration.

We ran 3 commands to build the image, and all three has a very similar pattern. Each command start with `FROM`, `RUN` or `CMD`, called docker `instruction` and they took some `arguments`.

> The `FROM` instruction specify a `docker image` we want to use as base. While we are preparing our custom image, the `RUN` execute some commands. The `CMD` specify, what should run on startup when our image will be used to create a new `container`.

Every line of configuration we are going to add inside the `Dockerfile` will always start with a `instruction`

> A base image is an initial set of programs that can be used to to further customize the the image. `Alpine` is a base image that comes with a package manager named `apk` (Apache Package Manager). `apk` can be used to download the `redis-server` and install in the system.

With `FROM alpine`, we are using an initial operating system named `alpine`. This `alpine` operating system has couple of preinstalled program, that are very much useful for what we are trying to accomplish. Since we are here to create `redis server` and we need to install some dependencies, `alpine` has these tools and programs preinstalled like `apk`.

With `RUN apk add --update redis` we download and install the `redis` server. Here `apk` is nothing related to the `docker`. It's a dependency manager preinstalled in the `base image`, download and install the `redis` server.

The `CMD ["redis-server"]` ensure, when we create a container from this docker `image`, the redis server will be started using `redis-server`command.

Image build process

To build a image from the `Dockerfile`, we use

```
docker build .
```

This ship our `Dockerfile` to the `docker client`. `build` is responsible for take the `Dockerfile` and build an `image` out of it.

The `.` is the build context. The build context contains all the set of files and folders that belongs to our project needs to wrap or encapsulate in our docker `container`.

If we notice the logs of the `docker build .`, we can see except the first instruction `FROM alpine`, every other instruction has an `intermediary container` that being removed automatically.

This means, except first step, each step took the container from the previous step and step itself acts as the startup instruction. Then when step is done, it simply pass the updated file system snapshot and remove the temporary container from itself.

For the last step, it took the container from the previous step, and do not run itself as the start-up command. Instead it only set itself as the first instruction and ensure if someone in future create and run the `container` out of the image, it then execute this last step as the `startup instruction`.

> For each step, except first one, we take the `image` from the previous step, create a temporary container, execute instructions, make changes, took the snapshot of the file system and return the file system output as output, so it can be used as the image for the next step. For last step, the final instruction is considered as the `start-up` instruction of the `docker container`.

Rebuild image from cache

Let's update the `Dockerfile` with an additional command

```
RUN apk add --update gcc
```

Now the `Dockerfile` should be like

```
FROM alpine
RUN apk add --update redis
RUN apk add --update gcc
CMD ["redis-server"]
```

Let's build a image out of this `Dockerfile`,

```
docker build .
```

If we observer the `logs` closely, we see, in the second step, it does not create a `intermediary container` from the previous step. Instead it is using the cache. So we do not need to install the `redis-server` multiple times. This gives the docker robustness and faster build performance.

From the instruction `RUN apk add --update gcc` it will create the `intermediary container` and since the file snapshot is being changed, it will do the same from the next steps.

If we build the image from the same `Dockerfile` for the 3rd time, it will take all the changes from the cache instead of the `intermediary container`.

> An `intermediary container` is created from the previous step image and use to run current instruction, make changes, take the snapshot of the new changed file system and got removed. This snapshot is being used for the next step to create another `intermediary container` and goes on.
>
> Altering the instruction sequence will not use the cache.

Tagging a docker image

Till now, we have created `image` from the `Dockerfile` and getting an `image_id`.

When we want a name instead of an `image_id` we can use the `tagging`.

To get a name, after a `image` being created, we can use the following

```
docker build -t user_name/image_name:version_number .
```

This will return

```
Successfully tagged user_name/image_name:version_number
```

> Here the `user_name` is the `username`, that is used to login to the `docker-hub`. `image_name` is our desired image name. The `version_number` is the `image` version number. Instead of `version_number` we can use `latest` keyword to use the latest version number handled by the docker itself.

We can now run the docker with the tag

```
docker run user_name/image_name:version_number
```

Here only the `version_number` is the tag itself. The `user_name` itself is always the `docker id` and `image_name` is the project name or the `repo name`.

> While running our tagged custom `image` we can ignore the `version_number`. It will simply take the latest version.