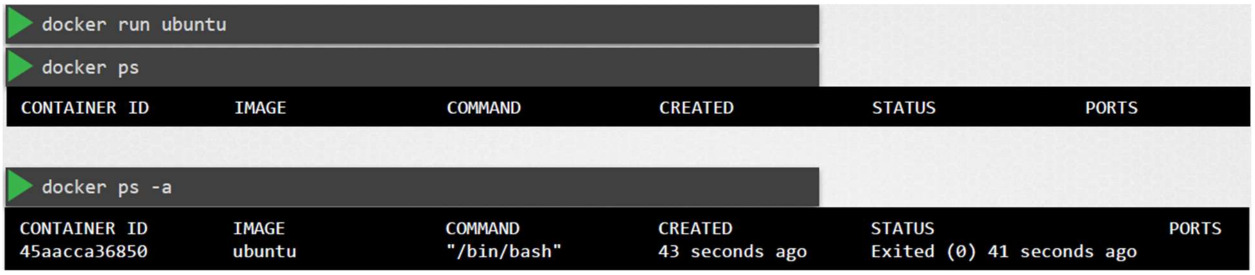
Commands

In this section, we will talk about commands and arguments in a pod definition file. Let's first refresh our memory on commands and containers and Docker. We will then translate this into pods in the next lecture. In this lecture, we will look at commands, arguments, and entry points in Docker.

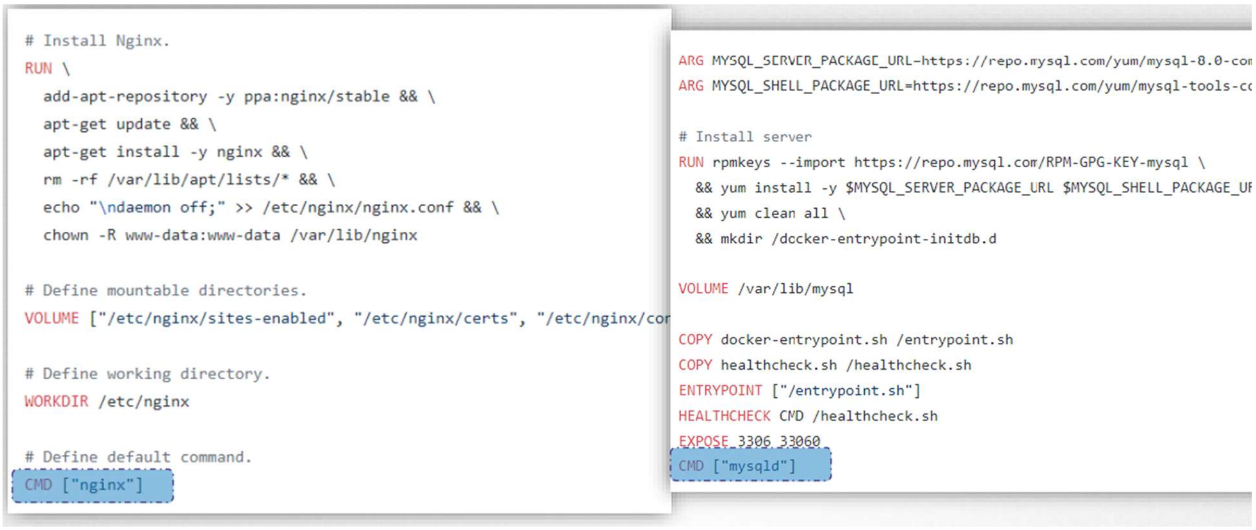
Let's start with a simple scenario. Say you were to run a Docker container from an Ubuntu image. When you run the Docker run Ubuntu command, it runs an instance of the Ubuntu image and exits immediately. If you were to list the running containers, you wouldn't see the container running. If you list all containers, including those that are stopped, you will see that the new container you ran is in an exited state.



```
docker run ubuntu

docker ps
CONTAINER ID     IMAGE          COMMAND        CREATED          STATUS          PORTS

docker ps -a
CONTAINER ID     IMAGE          COMMAND        CREATED          STATUS                        PORTS
45aacca36850     ubuntu         "/bin/bash"    43 seconds ago   Exited (0) 41 seconds ago
```

Now, why is that? Unlike virtual machines, containers are not meant to host an operating system. Containers are meant to run a specific task or process, such as to host an instance of a web server, application server, or a database, or simply to carry out some computation or analysis. Once the task is complete, the container exits. A container only lives as long as the process inside it is alive. If the web service inside the container is stopped or crashes, the container exits.

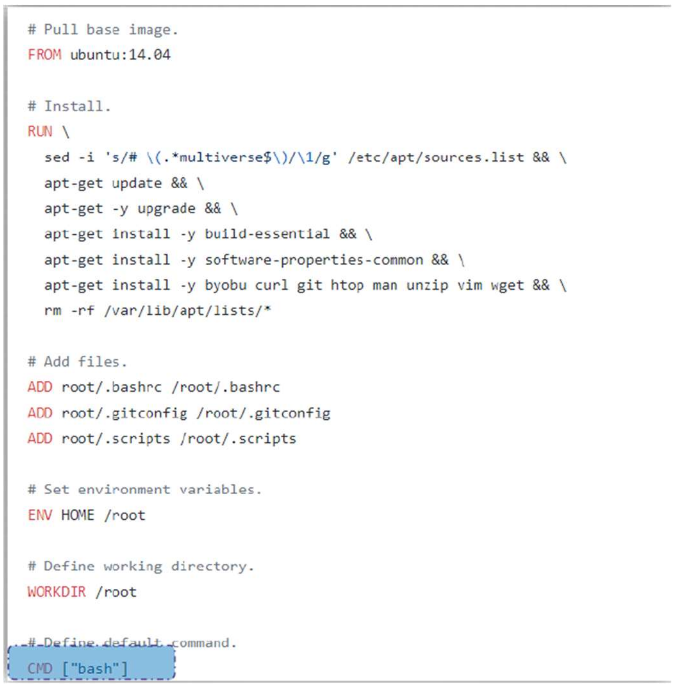
So who defines what process is run within the container? If you look at the Docker file for popular Docker images, like NGINX, you will see an instruction called CMD, which stands for command, that defines the program that will be run within the container when it starts. For the NGINX image, it is the NGINX command. For the MySQL image, it is the MySQL D command.



```
# Install Nginx.
RUN \
  add-apt-repository -y ppa:nginx/stable && \
  apt-get update && \
  apt-get install -y nginx && \
  rm -rf /var/lib/apt/lists/* && \
  echo "\ndaemon off;" >> /etc/nginx/nginx.conf && \
  chown -R www-data:www-data /var/lib/nginx

# Define mountable directories.
VOLUME ["/etc/nginx/sites-enabled", "/etc/nginx/certs", "/etc/nginx/cor

# Define working directory.
WORKDIR /etc/nginx

# Define default command.
CMD ["nginx"]
```

```
ARG MYSQL_SERVER_PACKAGE_URL-https://repo.mysql.com/yum/mysql-8.0-com
ARG MYSQL_SHELL_PACKAGE_URL=https://repo.mysql.com/yum/mysql-tools-cc

# Install server
RUN rpmkeys --import https://repo.mysql.com/RPM-GPG-KEY-mysql \
  && yum install -y $MYSQL_SERVER_PACKAGE_URL $MYSQL_SHELL_PACKAGE_UF
  && yum clean all \
  && mkdir /docker-entrypoint-initdb.d

VOLUME /var/lib/mysql

COPY docker-entrypoint.sh /entrypoint.sh
COPY healthcheck.sh /healthcheck.sh
ENTRYPOINT ["/entrypoint.sh"]
HEALTHCHECK CMD /healthcheck.sh
EXPOSE 3306 33060
CMD ["mysqld"]
```

What we tried to do earlier was to run a container with a plain Ubuntu operating system. Let us look at the Docker file for this image. You will see that it uses bash as the default command.

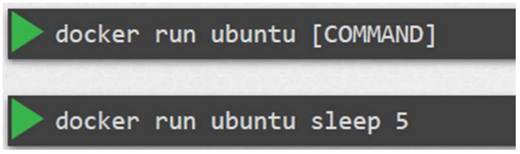


```
# Pull base image.
FROM ubuntu:14.04

# Install.
RUN \
  sed -i 's/# \(.*multiverse$\)/\1/g' /etc/apt/sources.list && \
  apt-get update && \
  apt-get -y upgrade && \
  apt-get install -y build-essential && \
  apt-get install -y software-properties-common && \
  apt-get install -y byobu curl git htop man unzip vim wget && \
  rm -rf /var/lib/apt/lists/*

# Add files.
ADD root/.bashrc /root/.bashrc
ADD root/.gitconfig /root/.gitconfig
ADD root/.scripts /root/.scripts

# Set environment variables.
ENV HOME /root

# Define working directory.
WORKDIR /root

# Define default command.
CMD ["bash"]
```

Now, bash is not really a process like a web server or database server. It is a shell that listens for inputs from a terminal. If it cannot find a terminal, it exits.

When we ran the Ubuntu container earlier, Docker created a container from the Ubuntu image and launched the bash program. By default, Docker does not attach a terminal to a container when it is run. And so the bash program does not find the terminal and so it exits. Since the process that was started when the container was created finished, the container exits as well.

So how do you specify a different command to start the container? One option is to append a command to the Docker run command, and that way, it overrides the default command specified within the image. In this case, I run the Docker run Ubuntu command with the sleep five command as the added option. This way, when the container starts, it runs the sleep program, waits for five seconds, and then exits.



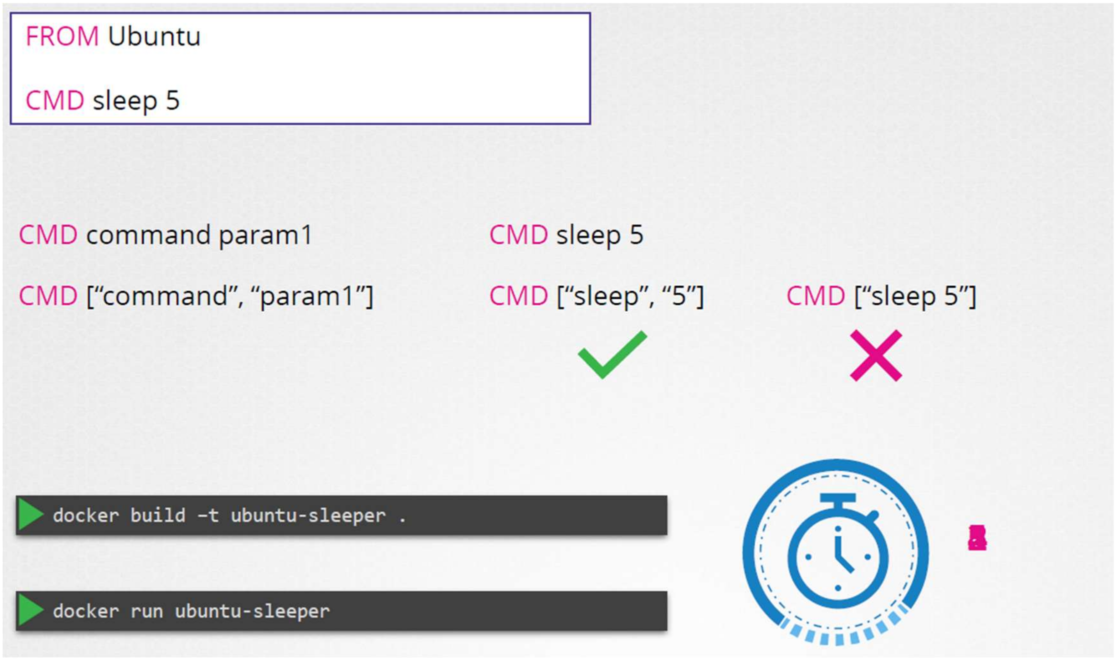
```
docker run ubuntu [COMMAND]
```

```
docker run ubuntu sleep 5
```

But how do you make that change permanent? Say you want the image to always run the sleep command when it starts. You would then create your own image from the base Ubuntu image and specify a new command. There are different ways of specifying the command. Enter the command simply as is in a shell form or in a JSON array format, like this.

But remember, when you specify in a JSON array format, the first element in the array should be the executable, in this case, the sleep program. Do not specify the command and parameters together, like this. The command and its parameters should be separate elements in the list.

So I now build my new image using the Docker build command and name it as Ubuntu sleeper. I could now simply run the Docker Ubuntu sleeper command and get the same results. It always sleeps for five seconds and exits.



```
FROM Ubuntu

CMD sleep 5
```

```
CMD command param1
CMD ["command", "param1"]
```

```
CMD sleep 5
CMD ["sleep", "5"]
```

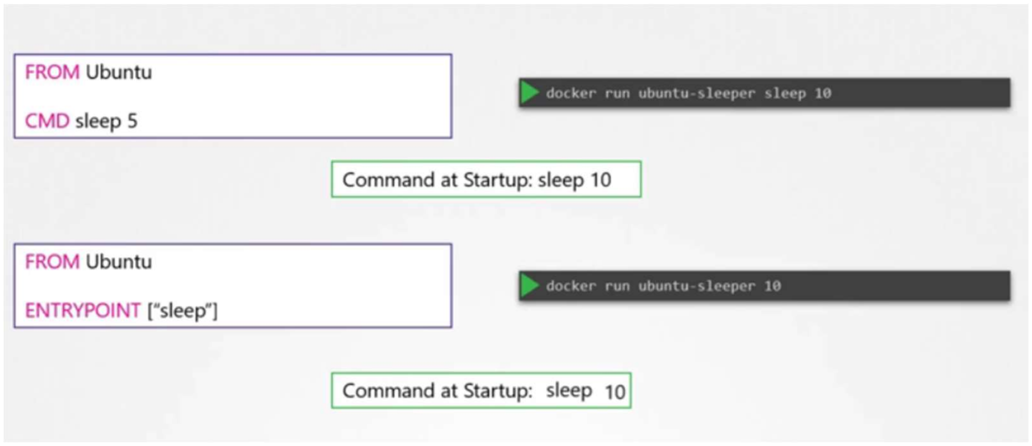
```
CMD ["sleep 5"]
```

```
docker build -t ubuntu-sleeper .
```

```
docker run ubuntu-sleeper
```

But what if I wish to change the number of seconds it sleeps? Currently, it is hard-coded to five seconds. As we learned before, one option is to run the Docker run command with the new command appended to it. In this case, sleep 10. And so the command that will be run at start-up will be sleep 10. But it doesn't look very good. The name of the image, Ubuntu sleeper, in itself implies that the container will sleep. So we shouldn't have to specify the sleep command again. Instead, we would like it to be something like this: **Docker run Ubuntu-sleeper 10**. We only want to pass in the number of seconds the container should sleep, and the sleep command should be invoked automatically. And that is where the entry point instruction comes into play.

The entry point instruction is like the command instruction, as in, you can specify the program that will be run when the container starts. And whatever you specify on the command line, in this case, 10, will get appended to the entry point. So the command that will be run when the container starts is sleep 10. So that's the difference between the two. In the case of the CMD instruction, the command line parameters passed will get replaced entirely. Whereas in the case of entry point, the command line parameters will get appended.



```
FROM Ubuntu

CMD sleep 5
```

```
docker run ubuntu-sleeper sleep 10
```

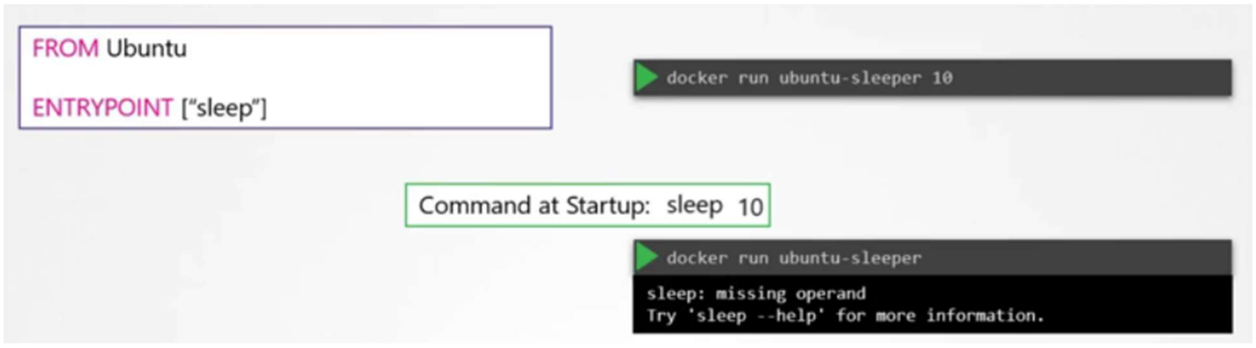
Command at Startup: sleep 10

```
FROM Ubuntu

ENTRYPOINT ["sleep"]
```

```
docker run ubuntu-sleeper 10
```

Command at Startup: sleep 10

Now, in the second case, what if I run the Ubuntu sleeper image command without appending the number of seconds? Then, the command at startup will be just sleep, and you get the error that the operand is missing.



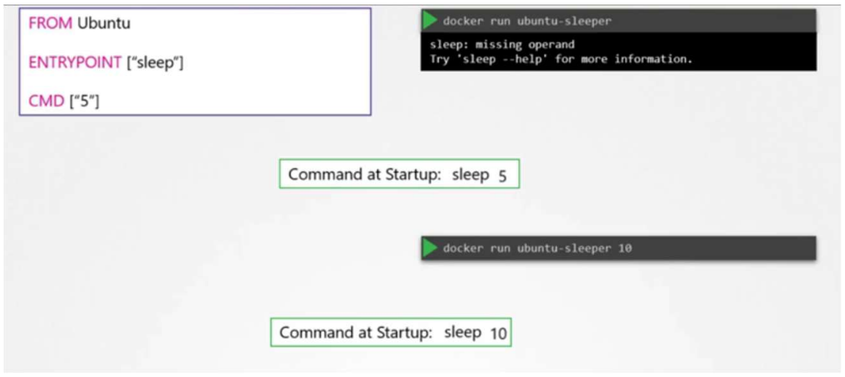
```
FROM Ubuntu

ENTRYPOINT ["sleep"]
```

```
docker run ubuntu-sleeper 10
```

Command at Startup: sleep 10

```
docker run ubuntu-sleeper
sleep: missing operand
Try 'sleep --help' for more information.
```

So how do you configure a default value for the command if one was not specified in the command line? That's where you would use both entry point, as well as the command instruction. In this case, the command instruction will be appended to the entry point instruction. So at startup, the command would be sleep five if you didn't specify any parameters in the command line. If you did, then that will override the command instruction. And remember, for this to happen, you should always specify the entry point and command instructions in a JSON format.



```
FROM Ubuntu

ENTRYPOINT ["sleep"]

CMD ["5"]
```

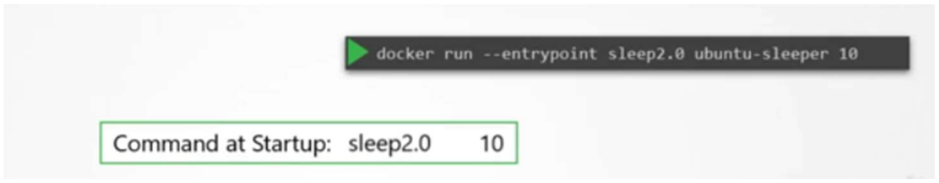
```
docker run ubuntu-sleeper
sleep: missing operand
Try 'sleep --help' for more information.
```

Command at Startup: sleep 5

```
docker run ubuntu-sleeper 10
```

Command at Startup: sleep 10

Finally, what if you really, really want to modify the entry point during runtime? Say, from sleep to an imaginary sleep 2.0 command. Well, in that case, you can override it by using the entry point option in the Docker run command. The final command at startup would then be sleep 2.0 10



```
docker run --entrypoint sleep2.0 ubuntu-sleeper 10
```

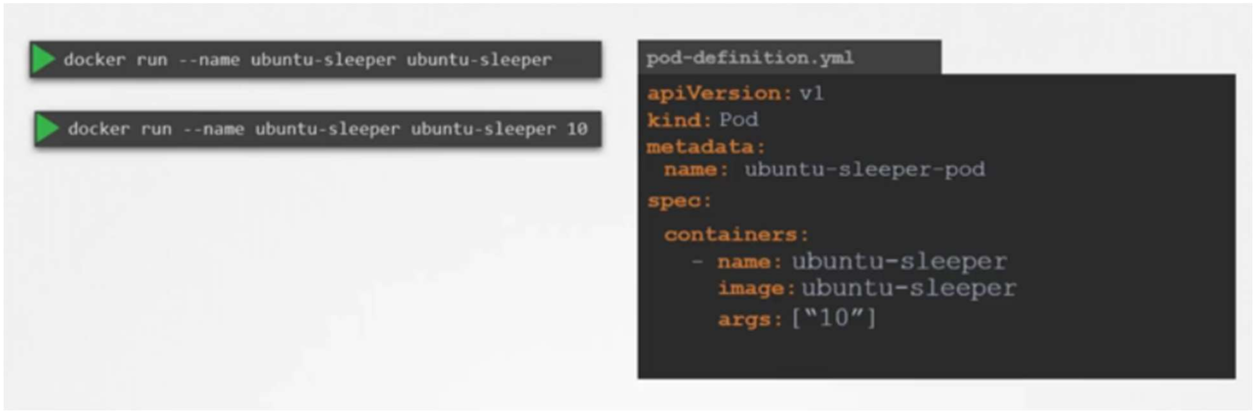
```
Command at Startup:  sleep2.0     10
```

Commands and Arguments

In this lecture, we will look at commands and arguments in a Kubernetes pod. In the previous lecture, we created a simple Docker image that sleeps for a given number of seconds. We named it Ubuntu sleeper and ran it using the Docker command: docker run ubuntu-sleeper. By default, it sleeps for five seconds, but you can override it by passing a command line argument.

We will now create a pod using this image. We start with a blank pod definition template, input the name of the pod, and specify the image name. When this pod is created, it creates a container from the specified image, and the container sleeps for five seconds before exiting.

Now, if you need the container to sleep for 10 seconds, as in the second command, how do you specify the additional argument in the pod definition file? Anything that is appended to the docker run command will go into the args property of the pod definition file, in the form of an array like this.



```
docker run --name ubuntu-sleeper ubuntu-sleeper
```

```
docker run --name ubuntu-sleeper ubuntu-sleeper 10
```

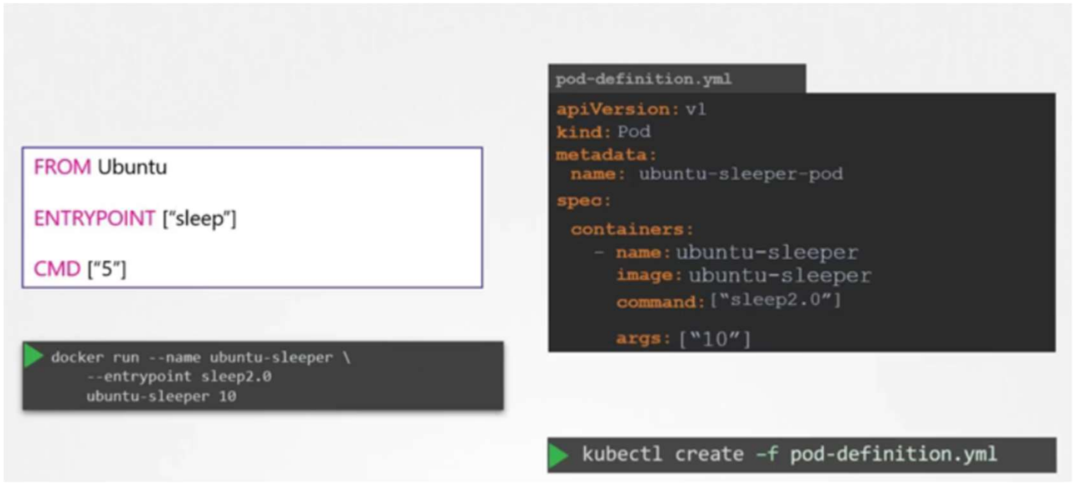
pod-definition.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper-pod
spec:
  containers:
    - name: ubuntu-sleeper
      image: ubuntu-sleeper
      args: ["10"]
```

Let us try to relate that to the Docker file we created earlier. The Docker file has an entry point, as well as a CMD instruction specified. The entry point is the command that is run at startup, and the CMD is the default parameter passed to the command. With the args option in the pod definition file, we override the CMD instruction in the Docker file.

But what if you need to override the entry point, say from sleep to an imaginary sleep 2.0 command? In the Docker world, we would run the docker run command with the entry point option set to the new command. The corresponding entry in the pod definition file would be using a command field. The command field corresponds to the entry point instruction in the Docker file.

So to summarize, there are two fields that correspond to two instructions in the Docker file. The command field overrides the entry point instruction, and the args field overrides the CMD instruction in the Docker file. Remember, it is not the command field that overrides the CMD instruction in the Docker file.



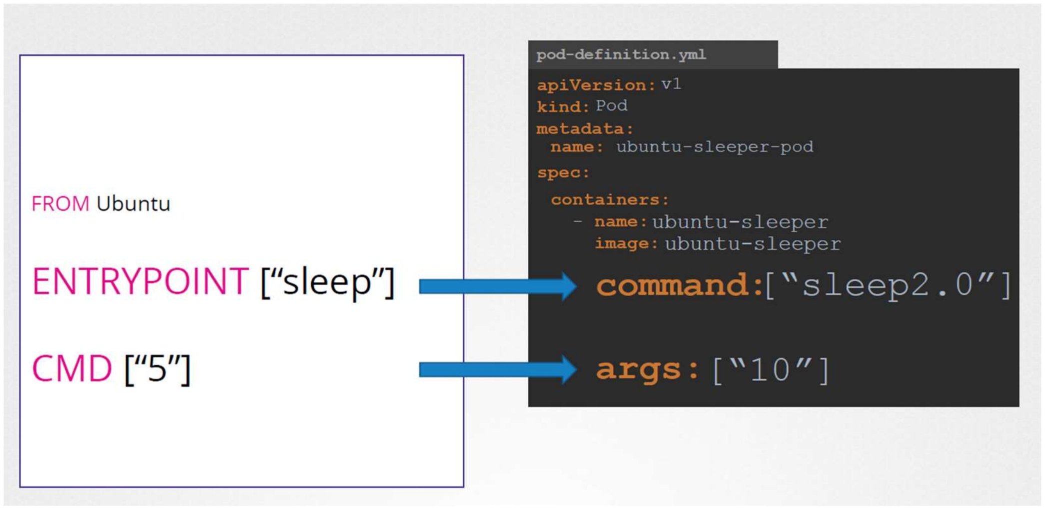
```
FROM Ubuntu

ENTRYPOINT ["sleep"]

CMD ["5"]
```

```
docker run --name ubuntu-sleeper \
    --entrypoint sleep2.0
    ubuntu-sleeper 10
```

pod-definition.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper-pod
spec:
  containers:
    - name: ubuntu-sleeper
      image: ubuntu-sleeper
      command: ["sleep2.0"]

      args: ["10"]
```

```
kubectl create -f pod-definition.yml
```



```
FROM Ubuntu

ENTRYPOINT ["sleep"]

CMD ["5"]
```

pod-definition.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper-pod
spec:
  containers:
    - name: ubuntu-sleeper
      image: ubuntu-sleeper
      command:["sleep2.0"]

      args: ["10"]
```