Part 44
-
Docker

Version: 2020-02-20

Introduction

Don't know what Docker is yet or what you could use it for? Check out 'Understanding Docker' from the offical docs. https://docs.docker.com/engine/docker-overview/

Docker does run on Raspberry Pi 2, 3 and 4, and you don't need any other OS beside Raspbian.

Docker, Docker CE, Docker EE?
Docker is the general name of the software. Docker CE stands for 'Community Edition' as there is also a Docker EE which stands for entreprise Edition. We will use only Docker CE.

Preparing the SD Card

Start with a fresh image on your Pi. So download the latest Raspbian lite image and put this on an micro SD Card. Now enable SSH on the SD Card by just creating a text file with the simple name '`ssh`' in `/boot`. It can be blank or you can type anything you want inside it. Make sure it has no extension such as `ssh.txt`. It must just be `ssh`. Now insert the SD card, networking and power etc. At this point you may want to do your own standard setting here (WiFi, logging to RAM, passwords, hostname, …).
If you are using the Pi for a headless application then you can reduce the memory split between the GPU and the rest of the system down to 16mb. Edit /boot/config.txt and add this line:

```
gpu_mem=16
```

Note: Once you boot up the Raspberry Pi you will be able to locate it on your network through the bonjour/avahi service. Connect with SSH using the following hostname `pi@raspberrypi.local`

Installing Docker CE on Raspbian (Stretch or Buster only!) for Raspberry Pi is straightforward, and it's fully supported by Docker. Docker CE is not supported on Raspbian Jessie anymore, so I'd recommend upgrading to a more recent release.
We're going to install Docker from the official Docker repositories. While there are Docker packages on the Raspbian repos too, those are not kept up to date, which is an issue with a fast-evolving software like Docker.
To install Docker CE on Raspbian Stretch and Buster, first install or update required packages

```
sudo apt update
sudo apt -y install apt-transport-https ca-certificates curl
sudo apt -y install gnupg2 software-properties-common
```

Get the Docker signing key for packages

```
curl -fsSL https://download.docker.com/linux/$(. /etc/os-release; echo
"$ID")/gpg | sudo apt-key add -
```

Add the Docker official repos

```
echo "deb [arch=armhf] https://download.docker.com/linux/$(. /etc/os-release; echo "$ID")
$(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list
```

Install Docker. Note that the aufs package, part of the "recommended" packages, won't install on Buster just yet, because of missing pre-compiled kernel modules. We can work around that issue by using "--no-install-recommends"

```
sudo apt update
sudo apt install -y --no-install-recommends docker-ce cgroupfs-mount
```

That's it! The next step is about starting Docker and enabling it at boot

```
sudo systemctl enable docker
sudo systemctl start docker
```

Enable Docker client
The Docker client can only be used by root or members of the docker group. Add pi (or your equivalent user) to the docker group:

```
sudo usermod -aG docker pi
```

After making this change, log out and reconnect with `ssh`.

Selecting the right image
This should hardly come as a surprise, but there's a caveat with running Docker on a Raspberry Pi. Since those small devices do not run on x86_64, but rather have ARM-based CPUs, you won't be able to use all the packages on the Docker Hub.
So you need to be careful while selecting the base image for your new Docker Image, as most these bases images are created for specific CPU Architectures.

You can use the "*cat /proc/cpuinfo | grep model*" command on your Raspberry Pi to find the CPU Architecture and use the corresponding Docker Hub repository

```
cat /proc/cpuinfo | grep model
model name : ARMv7 Processor rev 4 (v7l)
model name : ARMv7 Processor rev 4 (v7l)
model name : ARMv7 Processor rev 4 (v7l)
model name : ARMv7 Processor rev 4 (v7l)
```

Based on this command output from my Raspberry Pi which shows the CPU architecture as ARMv7, you need to look for images distributed by the **arm32v7** organization (called **armhf** before), or tagged with those labels. Good news is that the arm32v7 organization is officially supported by Docker, so you get high-quality images.
While the CPUs inside Raspberry Pi 3's and 4's are using the ARMv8 (or ARM64) architecture, Raspbian is compiled as a 32-bit OS, so using Raspbian you're not able to run 64-bit applications or containers.
Many common applications are already pre-built for ARM, including a growing number of official images, and you can also find a list of community-contributed arm32v7 images on Docker Hub. (https://hub.docker.com/r/arm32v7).

Testing your Docker installation
Now that we have Docker running, we can test it by running the "hello world" image:

```
sudo docker run --rm arm32v7/hello-world
```

If everything is working fine, the command above will output something similar to:

```
Unable to find image 'arm32v7/hello-world:latest' locally
latest: Pulling from arm32v7/hello-world
4ee5c797bcd7: Pull complete
Digest: sha256:d32a4c07ce3055032a8d2d59f49ca55fafc54a4e840483b590f7565769dc7e00
Status: Downloaded newer image for arm32v7/hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (arm32v7)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Running your first ARM image

Let's try starting Docker's official Alpine Linux image. Alpine Linux is a very compact 1.8MB download.
Check on https://hub.docker.com/r/arm32v7 to find the latest release of Apine Linux for arm32V7. As I write this, it was 3.11.3.

```
docker run -ti arm32v7/alpine:3.11.3 /bin/sh

Unable to find image 'arm32v7/alpine:3.11.3' locally
3.11.3: Pulling from arm32v7/alpine
3a2c5e3c37b2: Pull complete
Digest: sha256:2c26a655f6e38294e859edac46230210bbed3591d6ff57060b8671cda09756d4
Status: Downloaded newer image for arm32v7/alpine:3.11.3

/ # cat /etc/os-release
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.11.3
PRETTY_NAME="Alpine Linux v3.11"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://bugs.alpinelinux.org/"

/ # echo "Hi, this is a tiny Linux distribution!" | base64
SGksIHRoaXMgaXMgYSB0aW55IExpbnV4IGRpc3RyaWJ1dGlvbiEK

/ # echo "SGksIHRoaXMgaXMgYSB0aW55IExpbnV4IGRpc3RyaWJ1dGlvbiEK" | base64 -d
Hi, this is a tiny Linux distribution!

/ # exit
```

Note: If you just want the lastest version to be installed, you can use ‘latest’ instead of the specific version.

We launched sh - a BusyBox shell, but we could have launched any command directly.
Here's another example, justing the latest version of Alpine, without using the shell - we just run the date binary directly:

```
docker run arm32v7/alpine:latest date

Unable to find image 'arm32v7/alpine:latest' locally
latest: Pulling from arm32v7/alpine
Digest: sha256:2c26a655f6e38294e859edac46230210bbed3591d6ff57060b8671cda09756d4
Status: Downloaded newer image for arm32v7/alpine:latest
Fri Feb 21 13:24:34 UTC 2020

docker run arm32v7/alpine:latest date
Fri Feb 21 13:24:54 UTC 2020
```

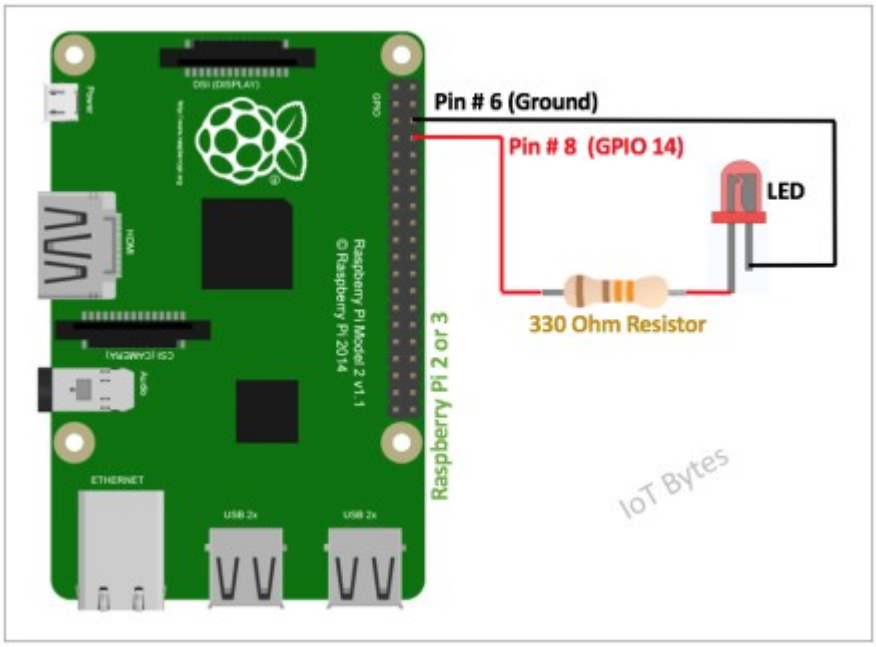
Creating your own Docker image

We are going to create a Docker Container Pi to Blink an LED

Create a New Directory
Create a new directory for all the files that you will use for creating a new Docker Image and get into this new directory. Let's name it "`docker_test`"

```
mkdir docker_test
cd docker_test
```

Prepare the Circuit
Connect an LED via a 330 Ohm register (or any value between 200 – 300 Ohm) to Raspberry Pi GPIO Pins, as shown in the following image



Create a Python Script to Blink an LED
Create a Python Script to blink the LED with the name "`led_blinker.py`", using the following code and save it in the "`docker_test`" directory created in the previous step

```
nano led_blinker.py

import RPi.GPIO as GPIO
import time

# Configure the PIN # 8
GPIO.setwarnings(False)
GPIO.setmode(GPIO.BOARD)
GPIO.setup(8, GPIO.OUT)

# Blink Interval
blink_interval = .5 #Time interval in Seconds

# Blinker Loop
while True:
```

```
        GPIO.output(8, True)
        time.sleep(blink_interval)
        GPIO.output(8, False)
        time.sleep(blink_interval)

    # Release Resources
    GPIO.cleanup()
```

Test your script

```
python3 led_blinker.py
```

Correct as needed. Once it is working, continue with the next step

Create Dockerfile
Create a new file in the "docker_test" directory with the name "Dockerfile", adding the following contents

```
nano Dockerfile

# Python Base Image from https://hub.docker.com/r/arm32v7/python/
FROM arm32v7/python:latest

# Copy the Python Script to blink LED
COPY led_blinker.py ./

# Intall the rpi.gpio python module
RUN pip3 install --no-cache-dir rpi.gpio

# Trigger Python script
CMD ["python", "./led_blinker.py"]
```

What is means is

build a Docker image from the latest Python Docker image
copy led_blinker.py from current directory to current directory on image
install Python module rpi.gpio (as this is not part of the default Python installation) using pip3 and do not use a cache directory
and start the command python with arguments ./led_blinker.py

If you need more information about Dockerfile, you can refer to the Dockerfile reference from the following URL

https://docs.docker.com/engine/reference/builder/

Create Docker Image from Dockerfile
Create Docker Image with the image name as "docker_blinker" and tag as "v1" using the following command

```
docker build -t "docker_blinker:v1"
```

Once the command execution gets completed you should be able to list the image using the following command

```
docker image ls
```

Start the Container to Blink the LED
As we are interacting with the hardware components i.e. the GPIO pins from this container, we need to use "`docker container run`" command with either the "`--privileged`" option or by specifying the Linux GPIO Device ("`/dev/gpiomem`") using the "`-device`" option.
You can use one of the following commands to run the Docker container, which in turn would blink the LED connected to your Raspberry Pi

```
docker container run --device /dev/gpiomem -d docker_blinker:v1
```

or

```
docker container run --privileged -d docker_blinker:v1
```

A slightly other way to do things
An other way to make your image could be to start from a base raspbian image. The Balenalib account in Docker Hub provides such an image (that is regularly updated. When I checked, update was 2 days ago). So your base Docker build file would be

```
nano Dockerfile

FROM balenalib/rpi-raspbian:latest
ENTRYPOINT []

RUN apt -q update && apt -qy install python3 python3-pip python3-dev gcc make

RUN pip3 install --no-cache-dir rpi.gpio
```

What is means is

> build a Docker image from the latest balenalib/rpi-raspbian Docker image
> update Raspbian
> install Python3, Python3 PIP, Python3 Dev package, gcc compiler and make tools
> install rpi.gpio (as this is not part of the default Python installation) using pip3 and do not use a cache directory

Build this image as a basis for adding your GPIO scripts at a later date.

```
docker build -t gpio-base .
```

Now use ADD to transfer the script into a new image depriving from gpio-base

```
rm Dockerfile
nano Dockerfile

FROM gpio-base:latest

ADD ./led_blinker.py ./led_blinker.py

CMD ["python3", "./led_blinker.py"]
```

What is means is

> build a Docker image from the latest gpio-base Docker image
> add led_blinker.py from current directory to current directory on image
> and last start the command python with arguments ./led_blinker.py

You will need to run this container in --privileged mode in order to be able to access the GPIO pins.

```
docker build -t blink .
docker run -ti --privileged blink
```

Optional Step : Use Docker Hub to Share Container Image
You can upload this image on any Container Registry and share it with other devices/users. To keep things simple I will use Docker Hub for this article. If you don't have a Docker Hub ID, you can register on Docker Hub and create one.

1. Login to Docker Hub from Raspberry Pi using the "`docker login`"command. Use your Docker ID and password to at the login prompt.

2. Re-tag your container image to add your Docker Hub ID with it, using the "`docker image tag SOURCE_IMAGE[:TAG] DOCKER_HUB_ID/TARGET_IMAGE[:TAG]`" command. For example, my docker id is "*mydockerid*", so the command that I would use is "`docker image tag docker_blinker:v1 mydockerid/docker_blinker:v1`"

3. Now you can upload the container image to Docker Hub using the "`docker image push DOCKER_HUB_ID/IMAGE_NAME:TAG`". I would use the command "`docker image push mydockerid/docker_blinker:v1`"

4. With this your image is ready to be pulled on any Raspberry Pi. If you have another Raspberry Pi you can pull this image using the "`docker image pull DOCKER_HUB_ID/IMAGE_NAME: TAG`". If you want to pull my image you can use the "`docker image pull mydockerid/docker_blinker:v1`" command.

5. Once you have this image pulled onto a new Raspberry Pi, you just need to connect the LED to the GPIO pins as explained earlier in this article, and start the container to blink the LED. You need not write any code or install any libraries.