# 16

# WHAT'S NEXT FOR DOCKER

## OVERVIEW

In the final chapter of this book, we will look at ways to help you troubleshoot and report issues when things do go wrong, as well as the ways in which you can try to resolve these issues yourself. You will learn how to compile the Docker application from source code, utilize some exciting new features, and explore the wider Docker community for support and guidance. The final part of this chapter will look at the current state of Docker Enterprise, the future of the company, and where the next big moves will be made when it comes to the usage and development of Docker.

## INTRODUCTION

Development and programming jobs have changed dramatically over the years, from simply writing and deploying code to a mainframe to running code on virtual environments and the cloud. In today's world, developers need to be mindful of agile processes, be more diligent, and need to consider implications for continuous integration and deployment. Whether you're administrating or developing with Docker, there is the potential for issues to arise, especially when you are working with cutting-edge applications or services.

There is enterprise support for Docker, but this is reserved for customers who have paid for Docker Enterprise Edition, Universal Control Plane, and Docker Datacenter, and can be accessed at the following URL:

https://success.docker.com/support.

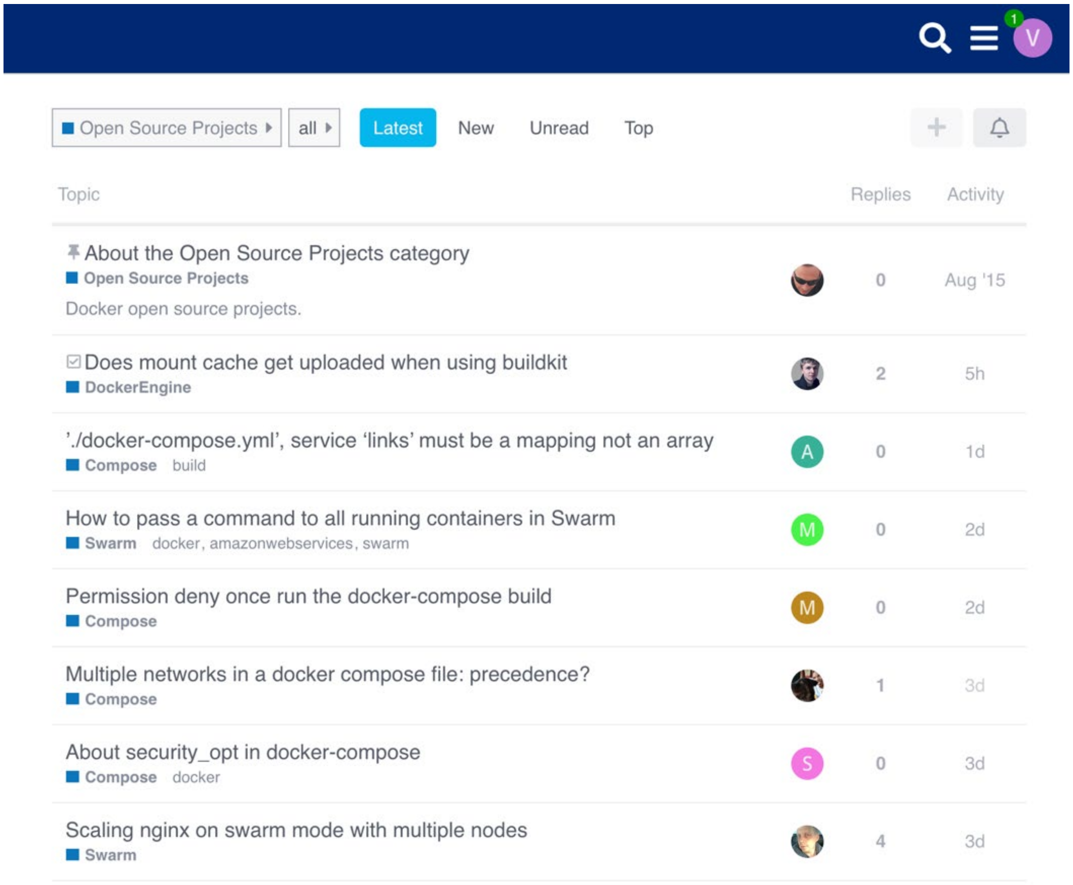
But for the rest of us who are using the open-source or freely available versions of Docker, what can we do when something goes wrong, and we need help? As we've mentioned throughout this book, one of the main benefits of using an application such as Docker is the fact that it is open source and has a large number of users supporting the community. The following topics will introduce some of the ways in which you can utilize this community for support.

## THE DOCKER COMMUNITY FORUM

When you come across an error or issue with your Docker environment, pasting your error into a search engine will most often bring you to the Docker community forum. If you've been working through this book and gotten this far, there is a good chance you have looked for answers there already.

Even though a lot of Docker's sites are linked by your username and the account created on Docker Hub, this is unfortunately not the case for the Docker community forum. You will need to create a new account when you first access the site at https://forums.docker.com.

When you log in, you'll be presented with a screen (similar to the one in the following screenshot) where you can start to search through topics that are relevant to you. You can search via category, application, publication date, and so on:



| Topic | Replies | Activity |
|---|---|---|
| About the Open Source Projects category<br>Open Source Projects<br>Docker open source projects. | 0 | Aug '15 |
| Does mount cache get uploaded when using buildkit<br>DockerEngine | 2 | 5h |
| './docker-compose.yml', service 'links' must be a mapping not an array<br>Compose build | 0 | 1d |
| How to pass a command to all running containers in Swarm<br>Swarm docker, amazonwebservices, swarm | 0 | 2d |
| Permission deny once run the docker-compose build<br>Compose | 0 | 2d |
| Multiple networks in a docker compose file: precedence?<br>Compose | 1 | 3d |
| About security_opt in docker-compose<br>Compose docker | 0 | 3d |
| Scaling nginx on swarm mode with multiple nodes<br>Swarm | 4 | 3d |

**Figure 16.1: The Docker community forum home screen**

If you can't find what you need, you can always create a new topic and see whether someone can help you with any issue you may be experiencing. You can do this by clicking on the **`Create Topic`** button or the plus (**+**) button at the top of the screen. You'll be presented with a screen like the following:

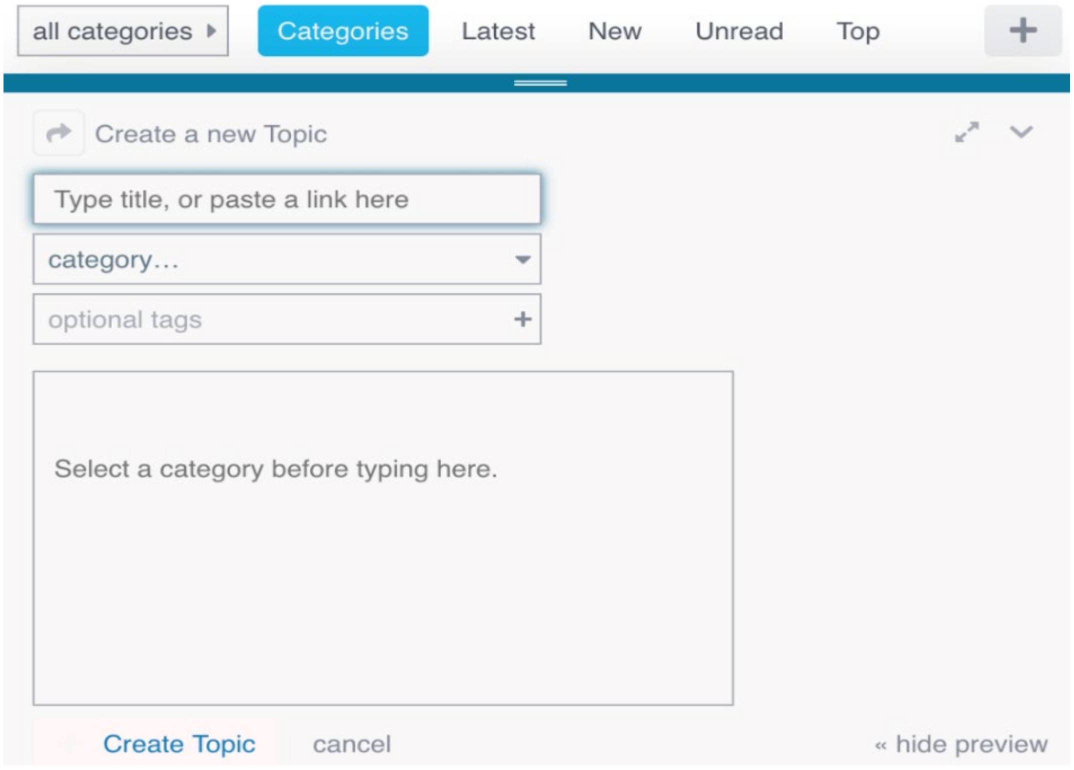


**Figure 16.2: Creating a new topic on the Docker community forum**

The form is self-explanatory, but when creating a new topic for the community forum, add a title and select a category you would like to post the topic to. When entering details of the topic, make sure you include the issue you are experiencing, the operating system and build of your system, the application version, and the steps to reproduce your issue.

Logging your issue with a community forum will be one of the last resorts to resolving your issue. There are a few steps you should take beforehand, including rubber duck debugging, which we will cover later in this chapter. Docker also has its own Slack channel, where a number of users are available to provide solutions to your issues. In the following section, we will understand how the Docker Slack channel helps users to resolve their problems.

## THE DOCKER SLACK CHANNEL

If you haven't used Slack before, it's a collaboration medium similar to a messaging or group chat service that you can subscribe to. Slack helps you organize discussions into different channels and workspaces. A channel can be used by a team to share tools, messages, and files. Slack does offer a paid service for enterprises, but a lot of groups and open source projects are using free access to host discussions as it still provides a lot of functionality without the cost.

Using the Docker Slack channel is a great option if you are looking for quick feedback on an issue that you may be experiencing. Due to a large number of users on Slack at any one time, there is usually someone from the community available to help you when you have an issue.

There is a desktop and a smartphone app available to access Slack, but you can access Slack via a web browser too. To access the Docker Slack channel, simply go to https://dockercommunity.slack.com/.

After accessing the preceding URL, you should be presented with a prompt as in the following screenshot. You'll be asked for your email address. If you already have an account with Slack, provide the email address you have attached to your Slack account:

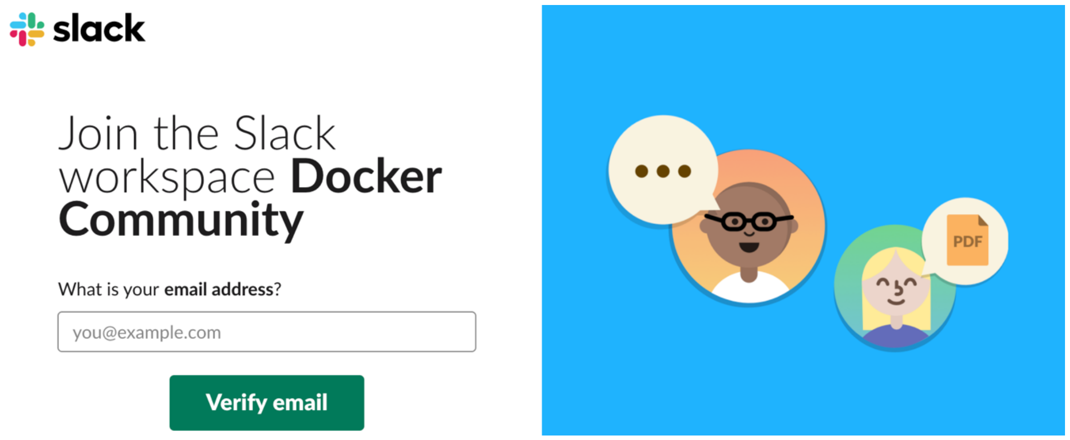
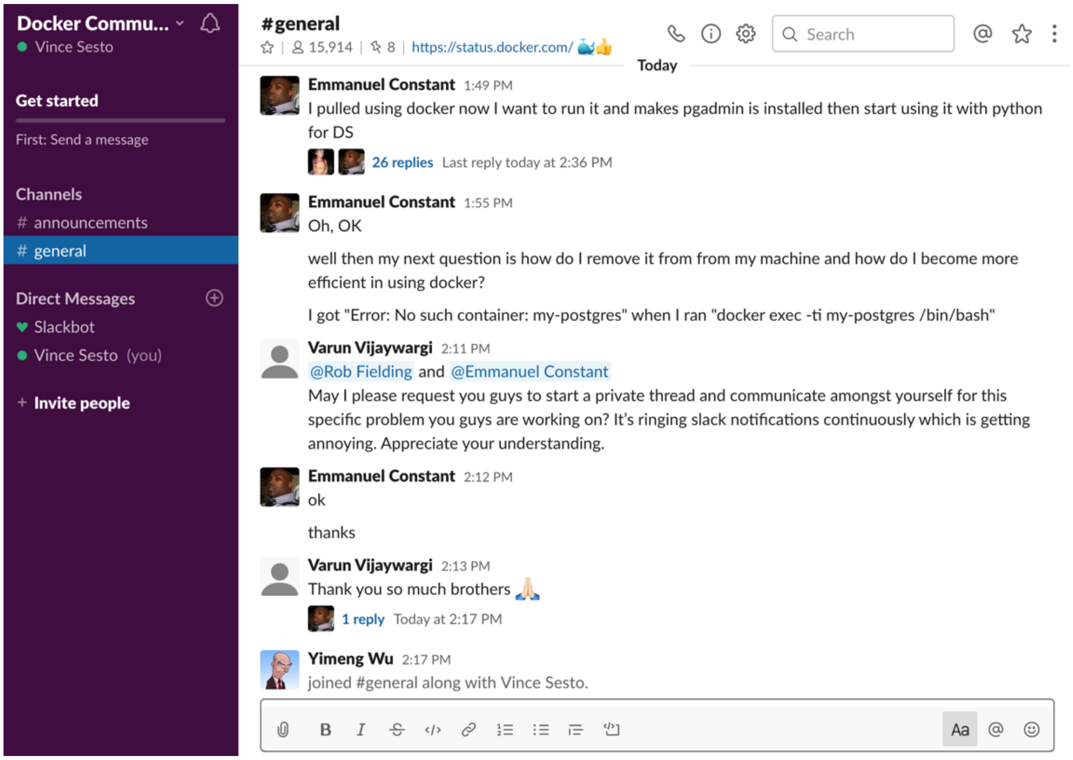

slack
Join the Slack workspace Docker Community
What is your email address?
you@example.com
Verify email
PDF

**Figure 16.3: Signing up to gain access to the Docker Slack channel**

Once you verify your email, you'll then be able to access the Docker community Slack channel and be presented with the general discussion page for the channel, as demonstrated in the following screenshot.

Feel free to introduce yourself and join in the conversation on the channel:


Docker Commu...
Vince Sesto
Get started
First: Send a message
Channels
# announcements
# general
Direct Messages
Slackbot
Vince Sesto (you)
+ Invite people

#general
15,914 | 8 | https://status.docker.com/
Today

Emmanuel Constant 1:49 PM
I pulled using docker now I want to run it and makes pgadmin is installed then start using it with python for DS
26 replies Last reply today at 2:36 PM

Emmanuel Constant 1:55 PM
Oh, OK
well then my next question is how do I remove it from from my machine and how do I become more efficient in using docker?
I got "Error: No such container: my-postgres" when I ran "docker exec -ti my-postgres /bin/bash"

Varun Vijaywargi 2:11 PM
@Rob Fielding and @Emmanuel Constant
May I please request you guys to start a private thread and communicate amongst yourself for this specific problem you guys are working on? It's ringing slack notifications continuously which is getting annoying. Appreciate your understanding.

Emmanuel Constant 2:12 PM
ok
thanks

Varun Vijaywargi 2:13 PM
Thank you so much brothers 🙏
1 reply Today at 2:17 PM

Yimeng Wu 2:17 PM
joined #general along with Vince Sesto.

**Figure 16.4: The general Slack channel for the Docker community**

If you are not able to get a definitive solution on the Docker Slack channel, you can log your issue with GitHub. There are a number of repositories listed in GitHub and there is a good chance that you will find a solution to your problem.

## LOGGING A DOCKER ISSUE WITH GITHUB

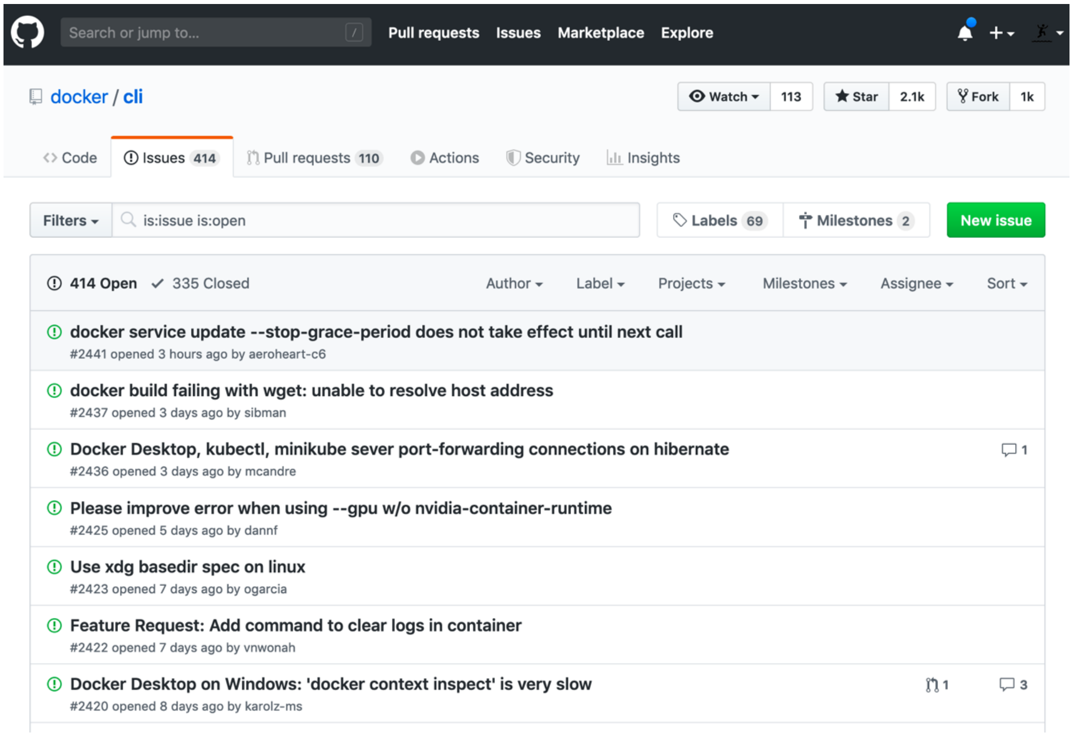
You should only log your issue with GitHub once you've exhausted all other avenues and think that your issue may be potentially caused by a bug in the code or some other dependency, such as your operating system. As Docker is an open-source project, all application code can be located on GitHub. So, as long as you have an account on GitHub, you will be able to log an issue in the relevant code repository.

The main collection of repositories for Docker is located in the following GitHub account:

https://github.com/docker.

When you get to this page, you'll be able to see all the repositories listed in GitHub that are actively being worked on. If you click on any of the repositories, you'll be provided with the source code repository of the specific piece of Docker. Before you log an issue in GitHub, click on the **Issues** tab on the page. For example, if you were having issues with the Docker CLI, you would go to https://github.com/docker/cli/issues to view the current issues with the code base:

You should see a web page such as the following with all the current issues listed for the specific repository. Selecting one of the issues will also give you an idea of how you need to log your issue in GitHub:



docker service update --stop-grace-period does not take effect until next call
#2441 opened 3 hours ago by aeroheart-c6

docker build failing with wget: unable to resolve host address
#2437 opened 3 days ago by sibman

Docker Desktop, kubectl, minikube sever port-forwarding connections on hibernate — 1
#2436 opened 3 days ago by mcandre

Please improve error when using --gpu w/o nvidia-container-runtime
#2425 opened 5 days ago by dannf

Use xdg basedir spec on linux
#2423 opened 7 days ago by ogarcia

Feature Request: Add command to clear logs in container
#2422 opened 7 days ago by vnwonah

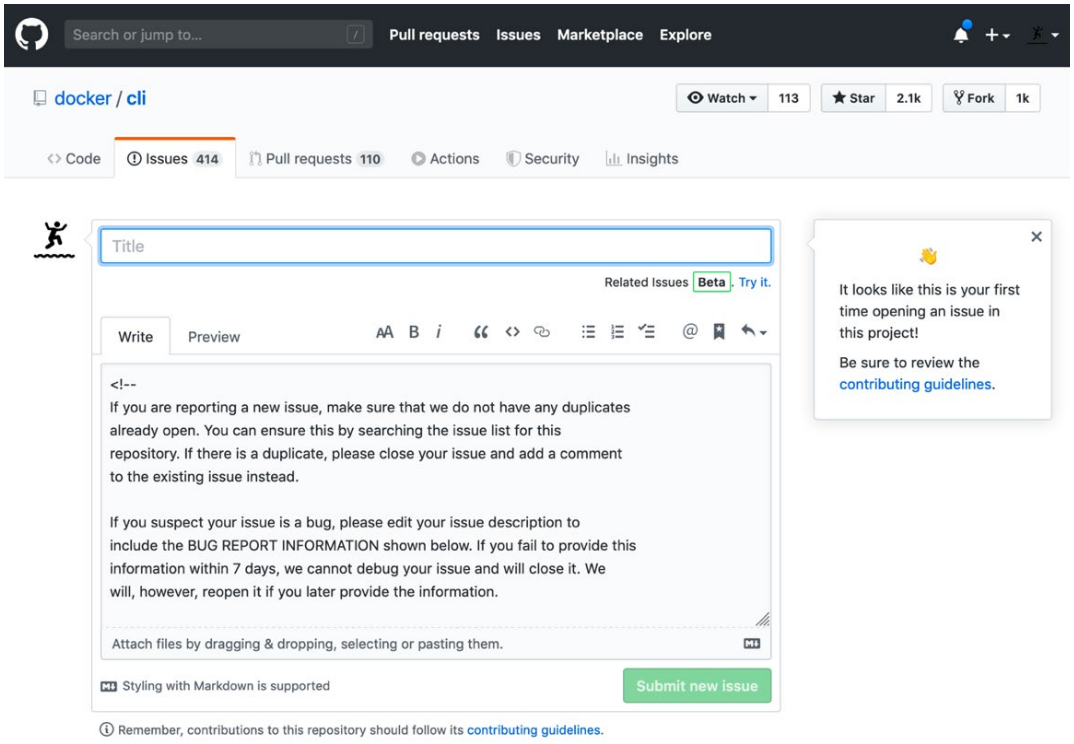
Docker Desktop on Windows: 'docker context inspect' is very slow — 1 — 3
#2420 opened 8 days ago by karolz-ms

Figure 16.5: The Issues page for the Docker CLI GitHub repository

If you would like to log an issue, click on the **New issue** button in the top-right corner of the screen. Clicking on this button will then present a screen similar to the following. You'll notice that logging an issue within GitHub is taken seriously, and administrators specifically request you to not log any duplicate requests and that you only submit a request after you have mentioned your problem in the community forums and Slack channels.

To log an issue, you will need to provide the following details:

- A description of the problem you are having in a few paragraphs
- The steps taken to reproduce the issue
- A description of the results you received
- A description of the results you expected
- The output from the **`docker version`** command
- The output from the **`docker info`** command
- Any further environment information you think may be useful in replicating this issue, such as whether the host you are running on is physical hardware or virtualized hardware:



```
<!--
If you are reporting a new issue, make sure that we do not have any duplicates
already open. You can ensure this by searching the issue list for this
repository. If there is a duplicate, please close your issue and add a comment
to the existing issue instead.

If you suspect your issue is a bug, please edit your issue description to
include the BUG REPORT INFORMATION shown below. If you fail to provide this
information within 7 days, we cannot debug your issue and will close it. We
will, however, reopen it if you later provide the information.
```

Attach files by dragging & dropping, selecting or pasting them.

Styling with Markdown is supported

It looks like this is your first time opening an issue in this project!

Be sure to review the contributing guidelines.

Remember, contributions to this repository should follow its contributing guidelines.

**Figure 16.6: Looking at a new issue in the Docker CLI GitHub repository**

When you are satisfied with the information provided in the issue, click on the **`Submit new issue`** button in the bottom-right corner of the screen.

As we mentioned earlier, creating an issue in GitHub should be the last resort, and you should exhaust all other options before logging an issue. This is mostly reserved for bugs that users have found within the code base so that developers can then correct the issue.

You can also get in touch with the Docker community, who can provide probable solutions to the issues you are facing while working with Docker. There are a lot of meetups and conferences where you can get a chance to meet the experts of the Docker community. In the next section, we will discuss various channels where you can meet the Docker community.

## Meeting the Docker Community

Getting involved with the Docker community is not limited to logging bugs and talking on chat channels. Just to finish up this section of the chapter, we have also listed some options to get some extra assistance when needed through meetups and conferences.

Docker has meetups and user groups located across the world, with groups meeting in most major cities. For an extensive list of Docker user group and meetups, refer to the following link:

https://events.docker.com/chapters/.

Even though Docker has had a short history, the Docker community has become a large part of some of the major technology conferences around the world, including **`KubeCon`** and **`DockerCon`**. For more details on conferences, check out the following URL:

https://www.docker.com/events.

If you find yourself unable to attend meetups or conferences, there is always the opportunity to attend a virtual meetup, with many planned throughout the year. Further details can be found at the following URL:

https://events.docker.com/docker-virtual-meetups/.

Hopefully, this section has provided you with some extra details and information to show you that there are a lot of options available to you when you need help. If you look for support from the community via a Slack channel or community forum, there is a good chance that an active member of the community has already been through what you are experiencing and can help you. Even though you may not have dedicated support for the version of Docker you are working with, you are far from alone when you experience an issue.

We have covered some of the ways that you can gain support when you need it. The following section will walk you through using rubber duck debugging as a way to help you troubleshoot your issues before you need to request help.

## Rubber Duck Debugging When Things Go Wrong

The term **rubber duck debugging** is a method of debugging code where, historically, a programmer would work to debug their code while narrating their problem line by line to a rubber duck.

Using rubber duck debugging is a way of getting as much information as possible by listing all the steps in detail and thinking of everything that could be the reason behind your issue before posting a bug to a forum or support channel. This may help you solve the problem you are facing. Rubber duck debugging helps to uncover the issue that is causing the problem by finding out the connection between what the user is doing and what they are supposed to be doing.

Even when using and working with Docker, you should still look to rubber duck debugging as a problem-solving technique to try and resolve the issues you are facing.

If you have ever needed to log a ticket with a product owner of GitHub, typically, you may need to perform the following steps:

- Determine the version of the application you are using.
- Determine which operating system you are running on.
- Briefly describe the problem you are experiencing.
- List the steps to reproduce the problem.
- Describe the results you received.
- Describe the results you expect.

These steps seem like a lot of work, especially if you've already been spending a great deal of time working to try and resolve your issue. The main reason why these steps help you resolve your issue is that they help you to think of everything that is involved in your issue in greater detail. Listing all the steps in detail will at least provide greater insight for those people who may be trying to resolve your issue if you get to log it with a support channel. If you stop and take the time to describe your problem as simply as possible, you force yourself to look at your issues from a different angle.

## Time for a Rubber Duck Example

The best way to look at working through these types of techniques is by using an example. In the following example, the user is trying to list the services running on the Docker swarm they are working on but is receiving an error. The following example runs through the specific questions we listed previously in this section:

**a) What version of the application are you using?**

This is a great place to start. As we are using Docker, all we need to do is use the **`version`** command on the command line and output the details. This could potentially highlight issues with software or application dependencies, or possibly known issues that could be causing the problem you are experiencing:

```
docker version
```

The output should look similar to the following, and will provide details of both the Docker client and Docker server running on your system:

```
Client: Docker Engine - Community
 Version:           19.03.8
 API version:       1.40
 Go version:        go1.12.17
 Git commit:        afacb8b
 Built:             Wed Mar 11 01:21:11 2020
 OS/Arch:           linux/amd64
 Experimental:      true

Server: Docker Engine - Community
 Engine:
  Version:          19.03.8
  API version:      1.40 (minimum version 1.12)
  Go version:       go1.12.17
  Git commit:       afacb8b
  Built:            Wed Mar 11 01:29:16 2020
  OS/Arch:          linux/amd64
  Experimental:     true
 containerd:
  Version:          v1.2.13
  GitCommit:        7ad184331fa3e55e52b890ea95e65ba581ae3429
 runc:
  Version:          1.0.0-rc10
  GitCommit:        dc9208a3303feef5b3839f4323d9beb36df0a9dd
 docker-init:
  Version:          0.18.0
  GitCommit:        fec3683
```

**b) What is the operating system that the application is running on?**

Just as the previous step provided details on possible known issues in the application version we are running on our system, so too does this step, as it may show a compatibility issue with our operating system. For a Linux-based operating system, we can output the details in the **`/etc/lsb-release`** file to provide details of the operating system that is running:

```
cat /etc/lsb-release

DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=18.04
DISTRIB_CODENAME=bionic
DISTRIB_DESCRIPTION="Ubuntu 18.04.6 LTS"
```

**c) Briefly describe the problem you are experiencing:**

This is a quick verbal explanation of the problem with no need to provide more details. If you do provide these details in a support ticket, this brief explanation may spark someone's memory of similar issues they have seen previously:

```
I am trying to view services running in docker swarm but when I
run the ls command, I get an unknown error
```

**d) List the steps to reproduce the problem you are experiencing:**

This is where you can start to go a lot more in-depth with your explanation of the steps you have performed. Try to be as detailed as possible, providing the exact commands you performed when you were doing your work. This should include details of configuration files or code you may be running:

```
I started by creating a service in the single node Docker Swarm I am
running on my system, using the docker service create command:

  docker service create --replicas 2 -p 80:80 --name web nginx

Then when I try to list the services running on my system using the
ls command, I get an unknown error:

  docker service ls -l
```

**e) Describe the results you obtained:**

A lot of the time, all you need to do is copy the output you are receiving when your command or process is failing. Note that you should be at least pasting this error into a search engine to verify the error has not been posted on a different forum and has already been answered. As you can see from the following error, it is highlighting the fact that the command we used in the previous step is using an unknown flag:

```
unknown shorthand flag: 'l' in -l
See 'docker service ls --help'.
```

**f) Describe the results you expected:**

Just as we listed in the previous step, we need to provide some sort of result that you expect to see. This could be a description of what the documentation has outlined or what you have experienced previously. As you can see in the following output, a sample of the output was produced by running the correct command, **`docker service ls`**, on a previous occasion:

```
ID              NAME     MODE           REPLICAS     IMAGE
    PORTS
<docker-id>     web      replicated     2/2          nginx:latest
    *:80->80/tcp
```

If you are reading this book, you are likely hoping to or are already working in the technology sector, so working through bugs and problems will be a major part of the work you will be doing. If you don't have a way of working through the bugs and issues you find, rubber duck debugging is a simple way to investigate and even provide support to other people's problems.

The following section will move you further into debugging with a discussion on how to work with the Docker source code and compile the Docker application from the source code to give you an option if you need to enhance or correct a bug in the software.

## WORKING WITH THE DOCKER SOURCE CODE

By now, this book has given you the knowledge to become an experienced Docker administrator and developer, but if you want to take your abilities further and start to create your own customizations of the Docker code, you will need to become familiar with working directly with the Docker source code. As your abilities progress, you may need to change a hardcoded configuration that is embedded in the source code or implement an experimental change that could enhance your environment before it is implemented into the mainstream Docker application.

> **NOTE**
>
> The source code for Docker is written in the Go programming language. To contribute to the Docker source code, you should be proficient in working with Go, which is not covered in this book. If you would like to know more about Go, the best place to start would be https://golang.org.

To be able to manipulate the Docker source code, one of the first things you will need to be familiar with is building the Docker binary from the source code. Once you are familiar with all the pieces involved, you will see that it is not too difficult or complicated to complete building the application.

Unless you have been using Docker Enterprise, you've most likely been using **Docker CE**, which stands for **Community Edition**, and this is where you would be working to make any changes as well as build the source code into a Docker binary. Docker CE itself comprises separate open-source repositories and applications:

- **The Docker CLI**: This is the client where we run our commands with the open-source GitHub repository located at https://github.com/docker/cli.
- **The Docker server**: This includes Docker Engine and `systemd` applications with the open-source GitHub repository located at https://github.com/moby/moby.

> **NOTE**
>
> You may have heard the Moby Project mentioned on a few occasions. This is an open-source project that was created to help advance the software containerization movement and drive the ecosystem further into the mainstream. Moby is designed more for system builders who want to build their container-based systems, rather than for application developers who use Docker.
>
> Docker uses the Moby Project as an open research and development lab to experiment, develop new components, and collaborate with the ecosystem on the future of container technology. This is why changes to the Docker binary are being performed via Docker CE and not the Moby Project, as Moby is a piece of the entire project. If you would like more details on Moby, go to https://mobyproject.org.

Before you compile the source code for Docker, you'll need to make sure the system on which you will be building the source code has a running version of Docker on it. The build of the Docker source code will use container images as part of the compilation process. Secondly, you will need to ensure your system has the **make** application running on it as well, which will perform the compilation of the source code. Refer to the *Preface* of this book for guidance if you are unsure of how to complete these steps.

After you have confirmed that these prerequisites are available on your system, the first step in building the source code for Docker CE is cloning the GitHub repository onto your running system. The GitHub repository is located at **docker/docker-ce** and can be cloned with the following command:

```
git clone https://github.com/docker/docker-ce
```

The repository will provide you with a **docker-ce** directory in your working directory wherein the repository has been downloaded. To move to the **docker-ce** directory, you'll need to either decide on forking the current repository if you are making changes to the code yourself, or if you want to use an experimental branch, you'll need to check out that branch before you continue building the source code with a simple **git checkout** command, as follows:

```
git checkout <branch>
```

Of course, if you simply want to build the master branch, there is no need to perform this step.

To build and compile the Docker binary, all you now need to do is run the **make** command with the **static** option to allow you to specify where you would like the packages to be created. The build takes around 10 minutes to complete, depending on the size of the processor you are using on your system:

```
make static
```

Once complete, the new binary will be created in the **components/packaging/static/build/linux** subdirectory of your **docker-ce** directory. It will have a Docker binary ready for you to run on your system, as well as a compressed and zipped package ready for you to move and install on any other system. A simple listing of the subdirectory will provide the packages created by the **make** command:

```
ls -l components/packaging/static/build/linux/docker-<branch>.tgz
```

It's only a matter of moving the new binary into your system's executable path (usually into **/usr/bin/**), so it will then be able to be used as if it was installed by the usual methods.

This sounds like an easy process, and it is a lot easier than some other source libraries that you may have compiled in the past. In the following exercise, you will create your own compiled version of the Docker binary, in case this is something you might need to do in the future.

> **NOTE**
>
> Please use **`touch`** command to create files and **`vim`** command to work on the file using vim editor.

## EXERCISE 16.01: BUILDING THE DOCKER BINARY FROM SOURCE CODE

In this exercise, you will prepare your system to compile the Docker binary from the source code, run the compile on your system, and test the new binary on a test environment, which you will then run as a container image:

1. Verify that the **`make`** command is installed on your system by running the **`make --version`** command from the command line:

```
make --version
```

   The command will return the version of the **`make`** application that is installed on your system:

```
GNU Make 3.81
...
```

> **NOTE**
>
> The version may vary depending on your system.

2. If your system does not have **`make`** installed, run the following command to install the application. If you already have the application installed, feel free to skip this step:

```
apt-get update; apt-get install build-essential -y
```

3. Clone the Docker CE repository from GitHub with the following command:

```
git clone https://github.com/docker/docker-ce
```

Depending on your internet connection, this should only take a minute or so. It also creates a **docker-ce** directory for the repository:

```
Cloning into 'docker-ce'...
remote: Enumerating objects: 717, done.
remote: Counting objects: 100% (717/717), done.
remote: Compressing objects: 100% (448/448), done.
remote: Total 421143 (delta 213), reused 483 (delta 120),
pack-reused 420426
Receiving objects: 100% (421143/421143), 164.02 MiB |
1.09 MiB/s, done.
Resolving deltas: 100% (216000/216000), done.
Checking out files: 100% (10681/10681), done.
```

4. Change to the **docker-ce** directory and get ready to start compiling your Docker binary:

```
cd docker-ce
```

**NOTE**

At the time of writing, there were only two branches that were actively being worked on – the master branch and the **19.03** branch. If there is a newer branch, feel free to check out that branch instead.

5. Perform a checkout of the **19.03** branch so that you can build the latest development branch:

```
git checkout 19.03
```

6. Run the following command using the **make static** command to compile the Docker binary. The command also specifies that you are building the Linux version with the **DOCKER_BUILD_PKGS** option set to **static-linux**:

```
make static DOCKER_BUILD_PKGS=static-linux
```

When the code is compiled with the preceding command, it will output quite a bit of data that has been reduced for clarity. It will take about 10 minutes to complete, but you should watch it as the compile pulls different container images to perform different stages of the build:

```
make VERSION=19.03.8 CLI_DIR=/tmp/docker-ce/components/cli ENGINE_
DIR=/tmp/docker-ce/components/engine -C /tmp/docker-ce/components/
packaging static
make[1]: Entering directory '/tmp/docker-ce/components/packaging'
...
done
tar -C build/linux -c -z -f build/linux/docker-rootless-
extras-19.03.8.tgz docker-rootless-extras
make[2]: Leaving directory '/tmp/docker-ce/components/
packaging/static'
make[1]: Leaving directory '/tmp/docker-ce/components/packaging'
```

7. Run the **`list`** command as follows over the **`components/packaging/static/build/linux/`** subdirectory, and you will be able to see what was created as part of the compilation we performed in the previous step:

```
ls -l components/packaging/static/build/linux/
```

The Docker directory listed at the top of the output will include all the components for Docker to run on your system now, but it is also packaged underneath (the highlighted file), ready to be moved and installed on any other system you need it to be installed on:

```
total 80500
drwxr-xr-x 2 root root     4096 Apr 11 06:41 docker
-rw-r--r-- 1 root root 63671703 Apr 11 06:42 docker-19.03.8.tgz
drwxr-xr-x 2 root root     4096 Apr 11 06:42 docker-rootless-extras
-rw-r--r-- 1 root root 18749342 Apr 11 06:42 docker-rootless-extras
```

**NOTE**

You can see from this output that there are also two extra subdirectories created as part of the compilation process. These are simply the rootless versions of the Docker binary as a directory and also a compressed and zipped package file. The rootless version means they can be run without root user access. If you are wanting to use these binaries instead, feel free to do so.

8. To test the newly compiled version of Docker you created, uninstall the current application on the system and then install your new compiled version. This will remove a lot of good work you may already have done on your system. Fortunately, there is a way to create a small virtualized system to test your new binary. Run a Docker container to test the compiled version of Docker by running the following command. This will run a container from the latest Ubuntu image, as well as mount the **`/var/run/docker.sock`** directory on your new container as this will allow us to run Docker on a Docker container:

```
docker run --rm -itd -v /var/run/docker.sock:/var/run/docker.sock
--name test-docker-source ubuntu
```

   The output similar to the following will be returned:

```
9f34f7e7aff9dc60bc1dabd4d4be8945125ac348af5aa1694359b163d2ea5fe0
```

9. Copy the newly compressed package onto the running container using the **`docker cp`** command before accessing the new container:

```
docker cp components/packaging/static/build/linux/docker-19.03.8.tgz
test-docker-source:/tmp/
```

10. Access the new container with the **`docker exec`** command to start testing our newly compiled binary, as shown:

```
docker exec -it test-docker-source /bin/bash
```

11. Access the running container you are going to test on. As it is the latest version of Ubuntu, there is a good chance it will not have Docker running on the system already. Run the following **`docker version`** command to verify that it is not already installed and running. If not, remove the application from the new container as this could cause inconsistent results with the testing of our new Docker binary:

```
docker version
```

   You should get an output such as the following if the container does not have Docker running on it:

```
bash: docker: command not found
```

12. Extract the Docker package you copied onto the new container with the **`tar`** command, as shown:

```
tar zxvf /tmp/docker-19.03.8.tgz
```

    This will unzip and uncompress it into the current directory you are working on:

```
docker/
docker/ctr
docker/dockerd
docker/containerd
docker/docker-init
docker/docker-proxy
docker/docker
docker/containerd-shim
docker/runc
```

13. Move into the Docker file that has been created as a result of extracting the new compressed package file:

```
cd /docker/
```

14. You can perform some more testing with your new binary, but for now, simply test the new version of the compiled application. Run the **`docker version`** command to verify that we can at least get a result from the command, which is always a good starting point:

```
./docker version
```

    The **`version`** command has given an output, meaning the compiled binary ran successfully. If you wanted to test the new binary further, you would run Docker commands directly from this new directory or copy the directories into the application path for the system, usually in the **`/usr/bin/`** directory:

```
Client: Docker Engine - Community
 Version:           19.03.8
 API version:       1.40
 Go version:        go1.12.17
 Git commit:
 Built:             Sat Apr 11 06:35:07 2020
 OS/Arch:           linux/amd64
 Experimental:      false

Server: Docker Engine - Community
```

```
    Engine:
     Version:          19.03.8
     API version:      1.40 (minimum version 1.12)
     Go version:       go1.12.17
     Git commit:       afacb8b
     Built:            Wed Mar 11 01:29:16 2020
     OS/Arch:          linux/amd64
     Experimental:     true
    containerd:
     Version:          v1.2.13
     GitCommit:        7ad184331fa3e55e52b890ea95e65ba581ae3429
    runc:
     Version:          1.0.0-rc10
     GitCommit:        dc9208a3303feef5b3839f4323d9beb36df0a9dd
    docker-init:
     Version:          0.18.0
     GitCommit:        fec3683
```

This exercise started with just the basics to demonstrate that building a Docker application from code is a simple task, and if you are looking to contribute to the open-source project, performing this step should not be holding you back.

## What the Future Holds for Docker

At the time of writing this book, the future of Docker is a little uncertain. Although open source applications such as Docker and Docker Compose are some of the most widely used applications running, the company underlying Docker is not financially stable, which puts into question how the future of Docker will play out.

Since Docker's introduction in 2013, it has been a major driving factor in the popularity of containerization and has done a lot of work to help drive innovation around orchestration, so with the company not sitting well financially, there is a lot of talk in the community that this could harm the continuing popularity of Docker.

In late 2019, Docker Enterprise was acquired by Mirantis, a Californian-based company that specializes in enterprise cloud solutions. The two main products that the company provides are the Mirantis Cloud Platform and the Mirantis Application Platform. The first is based on Kubernetes and OpenStack and provides an open-source solution for businesses to run their private clouds. The second, Mirantis Application Platform, focuses on cloud-native continuous delivery. The purchase of Docker Enterprise seems to fit in perfectly with the work the company already does, with the company taking over the Docker Enterprise platform, Docker Trusted Registry, Docker Unified Control Plane, and the Docker CLI. The acquisition also includes all customers and currently existing contracts.

Mirantis has made it clear that there will be no changes to pricing support or services from the purchase of Docker Enterprise. They will also be supporting Swarm for at least the next 2 years, and on announcing the purchase, Mirantis claimed that the Docker Enterprise team plans to continue to develop new capabilities for their clients.

Docker has also announced that they have secured additional investment to advance developers' workflows when building, sharing, and running their applications, which means expanding the Docker Desktop and Docker Hubs roles in the development workflow to be able to provide developers with a more deeply integrated and seamless experience for building their applications.

In the following sections, we are going to highlight some of the more promising new ideas and ways of working that will hopefully be implemented fully into Docker.

We will first introduce an experimental feature called Docker App, which expands the use of containers and Swarm clusters to allow developers to create a complete environment for their service. The second idea we will be looking at is how Docker can be used with minimal or **Internet of Things** (**IoT**) devices.

## Introducing Docker App

Docker App is a new feature and is explained as a way to define, package, and manage your applications in a single object. The goal of the Docker App is to reduce the complexity of multi-service applications and provide a simple artifact similar to a Docker container. Docker is hoping this will improve the speed at which developers are onboarded onto a project as there will be a similar workflow and commands that they would have used if they are already familiar with working with Docker. The ultimate goal is to improve productivity and collaboration between development teams.

Once this experimental feature is applied to your Docker installation, you will then have access to the **docker app** command, which will allow you to create App Package files to manage and distribute your entire application. At the time of writing this book, Docker App is available to Docker Engine Community Edition and Enterprise Edition from version 19.03 and higher.

Docker App uses **Swarm** mode to run and deploy your applications, so you will need to have Swam initialized and running on your system and your system should be running as a Docker node. As you will see in the following exercise, Docker App combines both the **docker-compose** and **Swarm** functionality and extends them further to hopefully simplify and improve the speed of our deployments.

The commands you use to run and deploy applications are similar to those we have been using throughout this book. To start working on your app, use the **docker app init** command, as we have in the following example. This example uses the **--single-file** option to create one **dockerapp** file that includes all the parts of our application broken down in the file as metadata, **docker-compose** details, and parameters. Without the **--single-file** option, we have three separate files created for each of the sections – **metadata.yml**, **docker-compose.yml**, and **parameters.yml**:

```
docker app init --single-file <application_name>
```

When the **dockerapp** file is created as a single file, it will be named as specified in the **init** command and followed by the **.dockerapp** file extension. The **dockerapp** file that is created is shown in the following code block, ready for you to add all the specific details to the file. Each section is separated by three dashes (**---**), with the **metadata** section comprising all of the relevant data for the app being created. The **docker-compose** section is similar to a **docker-compose.yml** file in its structure, except all parameters for the app are listed in the final section of the file. The **docker-compose** section then references these parameters as variables:

```
# This section contains your application metadata.
# Version of the application
version: 0.1.0
# Name of the application
name: <application_name>
# A short description of the application
description:
# List of application maintainers with name and email for each
maintainers:
  - name: vincesesto
```

```
    email:

---
# This section contains the Compose file that describes your app.
version: "3.6"
services: {}

---
# This section contains the default values for your application.
```

In the directory we are working in after we have added all the details to the **dockerapp** file, we can then verify that all of our code is correct for Docker App by using the **docker app inspect** command. This will pick up the **dockerapp** file in the current directory and verify whether the syntax is correct and ready to be deployed to our system:

```
docker app inspect
```

The **inspect command** provides a long list of details to verify the syntax and provide you with details of the services to be deployed. If you are looking for a quicker validation with less output, we also have the **validate** command available where you specify the **dockerapp** filename. You will be provided with a short, single-line output of the information stating whether the file is set up correctly:

```
docker app validate <application_name>.dockerapp
```

To install the app on your system, specify the **dockerapp** file for your application, and use the **docker app install** command as we have shown in the following code block. We also need to specify the **--name** option to name the application stack we are deploying to our system:

```
docker app install <application_name>.dockerapp --name <stack_name>
```

Docker App also provides us with options to verify that our installation has been successful and that the services in our installation are running. We have the **docker app status** command that is used with the name of your application and provides a list of details and how the services are currently running:

```
docker app status <application_name>
```

Remember that Docker App uses **Swarm** to perform orchestration of your application, so you will also have all the commands available from Swarm to verify that your applications and services are running correctly.

When we have finished working with our application, all that we need to do to stop it from running on our system is to run the **`docker app uninstall`** command, as follows:

```
docker app uninstall <stack_name>
```

> **NOTE**
>
> We are working with experimental features in this section as they provide early access to future features of the application. This means that functionality could change at any time without warning. The feature is not supported by Docker.

The Docker App feature brings extra functionality to development teams, allowing you to extend the way you work with your services and applications. Even though it is still in an experimental stage, the following exercise will show you that it is stable enough to start working with and hopefully will be brought into the main application to be taken up by more users.

## EXERCISE 16.02: CREATING AND RUNNING A BASIC DOCKER APP

In this exercise, you will set up an application using Docker App. The following instruction will give you the basic guidelines on how to enable this feature on your Docker installation and how to start working with the basic command-line commands to initiate, create, validate, and run your application. This demonstration should also give you an understanding of why this feature could be beneficial to users in the future, as well as walk you through a basic deployment of a simple web application in your environment using Docker App:

1. To enable experimental features in the system you will be working on, open the **`/etc/docker/config.json`** configuration file with your text editor and add the **`"experimental": enabled`** line to the file. If you already have configurations in the file, simply add it to the end of the file. If you don't wish to change any further configurations, add the following:

```
{
 "experimental": enabled
}
```

2. Restart your instance of Docker running on your system to activate this change. Restart Docker and wait for it to be ready before moving on to the next step.
3. Run the **`docker app version`** command to make sure you have access to the **`docker app`** command and can start using the feature:

```
docker app version
```

You should see an output similar to the following, depending on the version of Docker App you will have installed:

```
Version:                v0.8.0
Git commit:             7eea32b7
Built:                  Wed Nov 13 07:30:49 2019
OS/Arch:                linux/amd64
Experimental:           off
Renderers:              none
Invocation Base Image: docker/cnab-app-base:v0.8.0
```

> **NOTE**
>
> As we mentioned earlier, Docker App uses Swarm to deploy and run the services you create using this feature. We covered Swarm extensively in *Chapter 9*, *Docker Swarm* of this book, so it should be available already on your system.

4. Run the **`docker node ls`** command to make sure Swarm is initiated on your system:

```
docker node ls
```

You should get an output similar to the following:

```
ID        HOSTNAME          STATUS      AVAILABILITY     MANAGER
    ENGINE
j2qx…     ubuntu1804        Ready       Active           Leader
    19.03.13
```

5. Use the **docker app init** command to create a new project, using the **--single-file** option to create one **dockerapp** application file. Run the following command to create your new app called **hello-docker-workshop**:

```
docker app init --single-file hello-docker-workshop
```

The initialization of the **hello-docker-workshop** app creates a file called **hello-docker-workshop.dockerapp**:

```
Created "hello-docker-workshop.dockerapp"
```

As mentioned at the start of this section, the file is divided into three sections, separated by three dashes (**---**). Open the file with your text editor and feel free to add specific details about your application to the **metadata** section of the file. The application we are creating in the following steps is a simple web application that will use the **hashicorp/http-echo** container image and simply outputs to a web page the text forwarded to the container.

**NOTE**

You mustn't make any changes to the order that the **dockerapp** file is in. If you make any changes to the order of the details in the **dockerapp** file, your app won't be able to be installed and run on your system.

6. With the **hello-docker-workshop.dockerapp** file open in your text editor, start to add in the details for a sample app. Move into the **docker-compose** section on *line 15* and enter the details for the service you are creating. With the work you have done in this book, this section should all be clear to you, but briefly, *line 17* names the service, *line 18* provides the image details you will use for the app, and *lines 19 to 21* provide the command details to be run in the app and the ports used. Note that you are specifying variable names in *lines 19 and 21*. Add these details to the parameters section of the **dockerapp** file:

```
15 version: "3.6"
16 services:
17   web:
18     image: hashicorp/http-echo
19     command: ["-text", "${web.text}"]
20     ports:
21     - ${web.port}:5678
```

**NOTE**

As you've been working with orchestration in the previous chapters of this book, you have been learning that you need to move the configuration away from the application and instead use variables. In the example you are running, you are using the **`${web.text}`** variable and then referring to it in your parameters, including the ports you will be publishing the web service on.

7. Now, move into the **`parameters`** section of the **`dockerapp`** file. As you have added your app details to the **`docker-compose`** section of the file, the **`parameters`** section has moved further down in the file to around *line 25*. Enter the following three lines, which creates default variables for our app under the name of the **`web`**. The first in *line 26* is the port that the app will be running on, and *line 27* is the text to be displayed in our command:

```
25 web:
26   port: 8080
27   text: "Hello Docker Workshop!"
```

The complete code should look like the following:

```
# This section contains your application metadata.
# Version of the application
version: 0.1.0
# Name of the application
name: hello-docker-workshop
# A short description of the application
description:
# List of application maintainers with name and email for each
maintainers:
  - name: vincesesto
    email:

---
# This section contains the Compose file that describes your
    application services.
version: "3.6"
services:
  web:
    image: hashicorp/http-echo
    command: ["-text", "${web.text}"]
    ports:
      - ${web.port}:5678

---
# This section contains the default values for your application
    parameters.
web:
  port: 8080
  text: "Hello Docker Workshop!"
```

8. Once all the changes have been made to your **`dockerapp`** file, save the file. Now, you should be ready to validate the app and install it on your system.

9. Run the **`docker app inspect`** command to make sure your app is ready to be run:

```
docker app inspect
```

   You should see an output similar to the following, providing a breakdown of your app with all the relevant details. If you have not set something up correctly, you will see an **`inspect failed`** error printed on the screen:

```
hello-docker-workshop 0.1.0

Maintained by: vincesesto

Service (1) Replicas Ports Image
----------- -------- ----- -----
web         1        8080  hashicorp/http-echo

Parameters (2) Value
-------------- -----
web.port       8080
web.text       Hello Docker Workshop!
```

10. Docker App also comes with a **`validate`** function that allows you to perform a similar check of your **`dockerapp`** file. Run the following command:

```
docker app validate hello-docker-workshop.dockerapp
```

   You'll notice that a successful validation provides a smaller, one-line result for success:

```
Validated "hello-docker-workshop.dockerapp"
```

11. Use the **docker app install** command using the **dockerapp** file to deploy the app onto your running system. Also, use the **--name** option to provide the name of **my-docker-app**. Run the following command:

```
docker app install hello-docker-workshop.dockerapp --name my-docker-
app
```

You should see a similar output to the following, creating the network, services, and application for your app:

```
Creating network my-docker-app_default
Creating service my-docker-app_web
Application "my-docker-app" installed on context "default"
```

> **NOTE**
>
> Docker App relies on Swarm being installed on your system. This was covered in-depth in *Chapter 9*, *Docker Swarm*, but if you do not have Swarm initialized on the system you are running the following steps on, the app will not be able to be installed successfully.

12. Docker App provides the **status** command to let you see the status of your running app. Run the following command to verify that your app is running successfully on your system:

```
docker app status my-docker-app
```

The following output has been reduced as it provides a large amount of useful data, including parameters and the status of the running application:

```
INSTALLATION
------------
Name:          my-docker-app
Created:       7 minutes
Modified:     6 minutes
Revision:      01E4TA4V15QT82FHEH5YMYM8C3
Last Action:   install
Result:        SUCCESS
Orchestrator: swarm
...
```

13. As mentioned earlier, the Docker App uses Swarm as the orchestration running on our system. This means you can also use Swarm commands to verify that your system is running successfully. Run the following **`docker stack ls`** command as you did with the Swarm stacks in previous chapters to verify that the stack is running:

```
docker stack ls
```

You should see an output similar to the following:

```
NAME                SERVICES            ORCHESTRATOR
my-docker-app       1                   Swarm
```

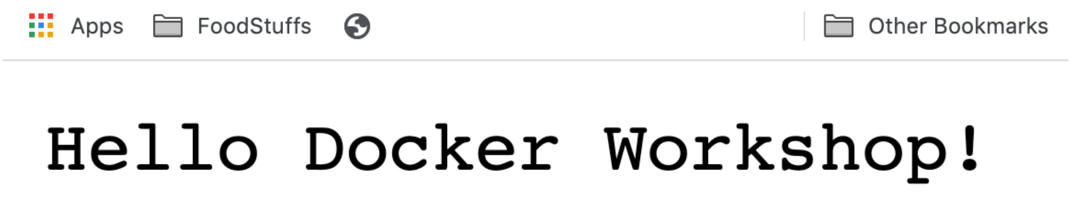
14. Before testing the new app, practice running your app using different parameters, as the ones provided in your **`dockerapp`** file are default values. Depending on your circumstances, you may need to run the app using a different port to the one specified. Run the following command to perform an upgrade to **`my-docker-app`**, changing the **`hello-docker.port`** value to **`8111`**:

```
docker app upgrade my-docker-app --set web.port=8111
```

You should see an output like the following:

```
Updating service my-docker-app_web
(id: xpirvz4gl5tkqwqv0o1e4nm53)
Application "my-docker-app" upgraded on context "default"
```

15. It's now time to test your new app running on your system. Open a web browser and enter the **`http://0.0.0.0:8111`** URL and you should hopefully see an output similar to the following:



Hello Docker Workshop!

**Figure 16.7: Displaying our Docker app in a web browser**

> **NOTE**
>
> We can also create a **docker-compose.yml** file from our **dockerapp** file using the **docker app render** command. This file can then be used as part of either **docker-compose** or **swarm** to deploy the applications. For example, we would be able to create a **docker-compose.yml** file in this exercise with the following command:
>
> **docker app render --output docker-compose.yml hello-docker-workshop.dockerapp**.

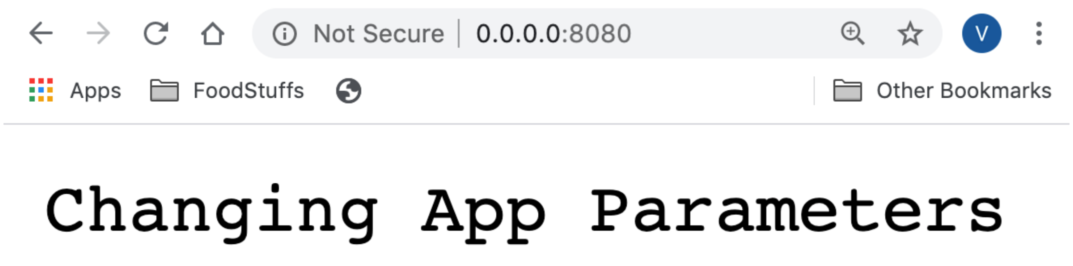
16. You have already specified parameters in your **dockerapp** file, but these can be overridden when you deploy changes as you may want to have different values for your parameters for each environment that you deploy the app to. You can override variables with the **--set** option, specifying the changes you would like. This can also be used after the app has been deployed (as in the following command), using the **docker app upgrade** command. Run the following command to change the **web.text** parameter in the **my-docker-app** Docker app:

```
docker app upgrade my-docker-app --set web.text="Changing App
Parameters"
```

This change may take a moment to update as it will need to delete the service that is currently running on the stack and create a new one displaying the following text:

```
Updating service my-docker-app_web
(id: iqp1kcgu8vl06skn0c32h5y2n)
Application "my-docker-app" upgraded on context "default"
```

17. Run the **`docker app status`** command to show the changed parameters, and if all has worked correctly, the changes should now be replicated when you load the web page, as in the following screenshot:



Changing App Parameters

**Figure 16.8: Displaying our Docker app with changed parameters**

18. Run the **`uninstall`** command to remove the **`my-docker-app`** application you have been working with from your environment:

```
docker app uninstall my-docker-app

Removing service my-docker-app_web
Removing network my-docker-app_default
Application "my-docker-app" uninstalled on context "default"
```

In this exercise, you have seen how Docker App functionality is enabled on your system and how you can start to work with this functionality in your running system. This is an exciting feature that will extend the use of applications such as Docker Compose and Swarm and allow development teams to bundle their entire applications together in one **`dockerapp`** file.

The next section of this chapter will continue this discussion on the future of Docker with a brief introduction to utilizing Docker with minimal or IoT devices.

## Docker and IoT

**IoT** is a connected network of smart devices all communicating with one another. These devices are referred to as **Things** as they are smaller electronic devices with a minimal amount of storage and processing power. They not only include household appliances such as fridges, televisions, and Wi-Fi routers but also include medical trackers and environmental and industrial sensors. The number of these devices continues to grow, and at the time of writing, it is estimated that there are over 20 billion of these connected devices around the world. Remember, this does not include PCs, laptops, smartphones, or tablets. To date, the use of Docker has been very minimal when it comes to working with IoT devices, but it seems as though this could change in the near future.

Docker can meet some of the challenges faced by developers working with IoT devices by working well with minimal hardware resources. Docker is still not being widely used in IoT development, even though a majority of these devices are running Linux as their operating system. Some issues are slowing the use of containers in these connected devices, but several projects are trying to increase the adoption of Docker running on IoT devices.

If you're interested in beginning Docker development on IoT devices, the easiest place to start is with a Raspberry Pi (https://www.raspberrypi.org/), which is a low-cost, credit-card-sized computer. There are three main ways in which you can start to integrate Docker with Raspberry Pi. The first is to purchase a Wi-Fi-enabled Raspberry Pi and a microSD card. This will allow you to install Raspbian, a Debian-based operating system for Raspberry Pi. With the Raspbian operating system available, it means you don't need to pay for a Raspberry Pi to start developing with Docker as Raspbian is also available as a virtual machine image. We will also see, in the following exercise, how we can use a Docker image of the operating system to start testing.

To perform an installation on Raspbian or other Linux-based operating systems (as long as there is an internet connection available to the system and it has the **`curl`** command installed), you run the following command. This takes the script located at get.docker.com and pipes it through the Linux shell (**`sh`**), which runs and performs the installation for you:

```
curl -sSL get.docker.com |sh
```

Without an internet connection, you would not be able to run this command as it downloads numerous other dependencies to the system before performing the installation.

Working with Docker is an easy way to get started with IoT development, as you will see in the following exercise. The most difficult part of getting started is making sure you can get Docker running on the hardware you would like to work with. Implementing Docker into IoT is still not considered a mainstream approach, but as development teams have rapidly adopted the use of Docker in their microservices, it seems only to be a matter of time until the same happens for IoT development.

In the following exercise, you will get hands-on experience working with Docker and IoT.

## Exercise 16.03: Working with Docker and IoT

In this exercise, you will emulate an IoT device on your running system to start creating and testing Docker images on these devices. You will be running the Raspbian Docker image on your system to emulate a Raspberry Pi device:

1. Because most IoT devices are running on ARM CPU architecture, you will need to install some packages to enable any standard AMD-based PC to emulate such devices. Install QEMU and some supported packages using the following command:

```
apt install qemu binfmt-support qemu-user-static
```

2. Pull the latest version of the Raspbian Docker image from Docker Hub. Run the following command to pull the latest **`stretch`** version of the operating system from the **`raspbian`** image repository:

```
docker pull raspbian/stretch
```

You should get an output similar to the following:

```
Using default tag: latest
latest: Pulling from raspbian/stretch
Digest: sha256:8c09306289628ea7ed04bc525d3a9069aa270a940e
0a347d0bc7b5f0b3768d2c
Status: Image is up to date for raspbian/stretch:latest
docker.io/raspbian/stretch:latest
```

3. You will use a Docker image to emulate an IoT device, which will then be running Docker on it. When you run the **raspbian/stretch** Docker image, make sure the image can use the **docker.sock** directory on the host system. Run the following command to start up the container with the **/var/run/docker.sock** volume mounted on the new image. Name our image **test-raspberry** in the command as well:

```
docker run --rm -itd -v /var/run/docker.sock:/var/run/docker.sock
--name test-raspberry raspbian/stretch
```

   The command will return a string of alphanumeric characters like the following:

```
ab549aa5d8452664c3fc17b9e5f2e68aa7c6623c501fe2487940c3a39befcf3b
```

4. Connect to the **test-raspberry** container using the following **docker exec** command with the container name created in the previous step:

```
docker exec -it test-raspberry /bin/bash
```

> **NOTE**
>
> If you have decided to purchase a Raspberry Pi with Raspbian installed, or use a virtual machine running Raspbian instead of using Docker, as we have, all the steps from now on will be the same. You will need to use the **ssh** command to connect to both a Raspberry Pi and a virtual machine. The default user for a Raspberry Pi running Raspbian is **pi user**, and the default password is **raspberry**. Use the following **ssh** command to connect:
>
> **ssh pi@raspberrypi.local**
>
> To run the Raspbian image on a Raspberry Pi or a virtual machine, obtain a copy of the ISO image from https://www.raspberrypi.org/downloads/raspberry-pi-desktop/.

5. Once you are successfully logged on to the host, run the **docker version** command to confirm that Docker is currently unavailable to you:

```
docker version
```

   The following output will confirm that Docker is currently unavailable to you:

```
bash: docker: command not found
```

6. Before installing Docker, first install the **curl** command, available as an **apt-get** package. Run the following command, which chains two commands together, to first update the **apt** repository and then install the **curl** command:

```
apt-get update; apt-get install curl -y
```

7. To install Docker, run the following command, which uses a **curl** command that connects to a specific URL, **get.docker.com**. This URL is an automated script that obtains all the specific libraries and applications needed to then install Docker and finishes by outputting the installed version details of Docker once it is complete:

```
curl -sSL get.docker.com |sh
```

The installation provides a long output, but we have reduced the output significantly in the following details. Please note that the following command may take around 5 to 10 minutes to complete:

```
# Executing docker install script, commit:
442e66405c304fa92af8aadaa1d9b31bf4b0ad94
…
WARNING: Adding a user to the "docker" group will grant the
ability to run containers which can be used to obtain root
privileges on the docker host.
Refer to https://docs.docker.com/engine/security/security/…
for more information.
```

8. Run the **docker version** command again to verify that Docker is installed and running on your system:

```
docker version
```

This time, you should see something better than you did previously, similar to the following output:

```
Client: Docker Engine - Community
 Version:           19.03.8
 API version:       1.40
 Go version:        go1.12.17
 Git commit:        afacb8b
 Built:             Wed Mar 11 01:34:34 2020
 OS/Arch:           linux/arm
 Experimental:      false
```

```
    Server: Docker Engine - Community
     Engine:
      Version:          19.03.8
      API version:      1.40 (minimum version 1.12)
      Go version:       go1.12.17
      Git commit:       afacb8b
      Built:            Wed Mar 11 01:29:16 2020
      OS/Arch:          linux/amd64
      Experimental:     true
     containerd:
      Version:          v1.2.13
      GitCommit:        7ad184331fa3e55e52b890ea95e65ba581ae3429
     runc:
      Version:          1.0.0-rc10
      GitCommit:        dc9208a3303feef5b3839f4323d9beb36df0a9dd
     docker-init:
      Version:          0.18.0
      GitCommit:        fec3683
```

**NOTE**

Before you run your Docker commands on the Raspbian container, you may see a similar error to the following:

**`ERRO[0000] failure getting variant error="getCPUInfo for pattern: Cpu architecture: not found"`**.

Docker is running on the container image, but the error is simply stating that it cannot access some aspects of the host operating system. Your Docker commands will still complete, though, so don't worry if you see this error in your command-line results. We have also excluded them from our output in this exercise.

9. The Docker image you are working with has been created to match the same hardware as Raspberry Pi. Run the **`uname -m`** command from the command line to verify this:

```
uname -m
```

   The output should show an ARM processor running:

```
armv7l
```

10. Run a new container on your Raspbian image to demonstrate the ability to run different containers running different virtualized hardware. The first container you are going to run on your Raspberry Pi image will be the latest Alpine image. Perform the same **`uname -m`** command as you did in the previous step to show that the image is running as an x86 processor instead of ARM:

```
docker run --rm -it alpine uname -m

Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
aad63a933944: Pull complete
Digest: sha256:b276d875eeed9c7d3f1cfa7edb06b22ed22b14219a7d6
7c52c56612330348239
Status: Downloaded newer image for alpine:latest

x86_64
```

   This may not seem like a big deal, but it shows that we are running our new container, an x86 processor, on an ARM processor, which is our IoT device.

   We could then build and run our Docker image directly on the **`raspbian/stretch`** container. It is limited in functionality as it is emulating a small IoT device. You would then be creating your applications on your system, testing, and verifying whether it works. You can then push the images to your repository that is available to then test on your IoT device or IoT container, as we have just done.

This brings us to the end of the exercises for this chapter. Follow on to the next few pages to test your knowledge with some activities to compile your changes to the Docker binary and implement the Panoramic Trekking App as a Docker App.

## Activity 16.01: Creating Your Own Development Branch for Docker

Using the source code of Docker CE, create a branch for your work, look to make a minor change to the source code, and then compile your code. Verify that your changes have been successfully created and are available in the new binary.

The steps you'll need to complete this activity are as follows:

1. Check out the source code for Docker CE.
2. Create a development branch for the source code.
3. Optionally, if you have some experience working with the code, make a minor change to the code. HINT: our answer will only make a change to the version we are building, which is located in the following directory: components/packaging/common.mk.
4. Build the code you have created as part of your development branch.
5. Verify that the code you have created is successful in a test environment.

> **NOTE**
>
> The solution to this activity can be found on page XXX.

The next activity will finish up this chapter and the book, allowing you to practice further with Docker App and deploy the Panoramic Trekking App using this new functionality.

## Activity 16.02: Deploying the Panoramic Trekking App as a Docker App

Earlier in this chapter, you learned one possible way that Docker tries to simplify the development process with the implementation of the Docker App functionality. To help embrace this new functionality, you can use all or choose some of the pieces of your Panoramic Trekking App and implement it as a Docker App.

The steps you'll need to complete this activity are as follows:

1. Verify that you are able to get started with working with your Docker app. Ensure the feature is available on your system and that you have Docker Swarm available.
2. Clone your code repository to your workspace to get it ready to be used as part of your Docker app.

3. If your Docker app is using images that are not available on Docker Hub or in a centrally available repository, make sure your images have been built and are accessible to your system.
4. Initialize your Docker app and create a **`dockerapp`** file.
5. Enter the metadata details for your new Docker app.
6. Enter the details for the **`docker-compose`** section of the **`dockerapp`** file.
7. Create default variables for your Docker app and enter them into the **`dockerapp`** file.
8. Inspect the newly created **`dockerapp`** file and validate that there are no issues when you move to install the new application.
9. Install the new Docker app onto your system and verify that it is running.
10. Test the Panoramic Trekking App's functionality and verify that it is working as expected.

> **NOTE**
>
> The solution to this activity can be found on page XXX.

## SUMMARY

In this chapter, we have gone through some of the ways in which you can find support and problem solutions within the Docker community when things go wrong. We looked at Docker Forums, the Docker Slack channel, and connecting with likeminded Docker users at meetups and conferences to discuss and resolve your issues, as well as logging them in GitHub when you think you have found a bug in the code. We then explored rubber duck debugging in more depth, focusing on its specific benefits when using Docker, as well as compiling your version of the Docker source code. The final part of this chapter considered the future of Docker as a company and some of the newer ways in which Docker is making an impact on the technology world. This includes the use of a new feature called Docker App and working with Docker when using IoT devices.

This chapter has hence provided you tips and tricks to solve the issues if you encounter while working on the Docker. It also provided the information about what the future holds for Docker.