Credit: https://docs.docker.com

DOCKER

Part One! Don't miss next issue, subscribe on page 24!

Nextcloud and Docker containers made easy

With the help of **Chris Notley** we set sail on a voyage of discovery to install Docker and use it to build a working instance of Nextcloud.

OUR EXPERT

Chris Notley runs a technology consultancy in Surrey with an interest in open source software.

We're going to demystify and make Docker easy to use, explaining how to run it and expose services to the internet. For our case study we will use Nextcloud and a fresh installation of Ubuntu Server 18.04. The goal here is not to demonstrate a 'better' way of running Nextcloud, but rather how to use the Docker platform.

Docker is probably the best-known container system for Linux, although it is far from the only one. Containers allow developers to build all the necessary libraries, services and configuration files into a predefined package that can be recreated on-demand. They solve an age-old problem where applications work fine in development but not when they're moved into production due to various dependencies, such as OS version, libraries, etc. This may sound similar to virtualisation (*VirtualBox*, *VMware*, etc.), but containers do not emulate hardware and even use the host server's OS kernel – see the boxout (*opposite page*) for more.

Get docked

We're going to use a virtual machine (any flavour will do, *VirtualBox* will work fine) on which to install Ubuntu Server 18.04. Feel free to add Docker onto an existing Linux installation too, but you may need to change some of the settings we apply later. During the installation of Ubuntu Server accept all the default settings, and the only optional package that may be useful is the SSH server (this tutorial assumes that the username defined during installation is 'tutorial').

```
tutorial@lxf260: ~
tutorial@lxf260:~$ sudo docker run hello-world
[sudo] password for tutorial:
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:4df8ca8a7e309c256d60d7971ea14c27672fc0d10c5f303856d7bc48f8c
Status: Downloaded newer image for 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.
    (amd64)
 3. The Docker daemon created a new container from that image which runs t
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sen
    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/
```

The hello-world container image is very useful to test that your Docker installation is working.

QUICK TIP

If you want to shortcut the installation process for Docker, cheat.sh is available in the project archive at https://github.com/prel-im/lxf260. If you place the file in your home directory and execute via 'bash cheat.sh $USER', it will run all the commands to install Docker Server on Ubuntu 18.04.

```
tutorial@lxf260: ~
tutorial@lxf260:~$ sudo docker ps -a
CONTAINER ID        IMAGE               COMMAND
8600d8c3a86a        hello-world         "/hello"
tutorial@lxf260:~$
```

'docker ps' lists all running containers on your docker host. If you want to see stopped containers too make sure you add the '-a' flag.

With Ubuntu Server installed we are now ready to install Docker, and there are a number of ways to do this. The Ubuntu repositories include a version of Docker called **docker.io**; however it is quite old. While containers themselves are not a new concept, Docker is a young platform, so we'll use the official Docker repository for the Community Edition (**docker-ce**). The team at Docker maintain a full installation guide for Ubuntu at **https://docs.docker.com/install/linux/docker-ce/ubuntu**, but we will run through only the necessary steps for a fresh install of Ubuntu 18.04.

We will start by updating our repository files, and then adding the Docker repository GPG key with the following commands:

```
sudo apt-get update
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

We can now add the Docker repository to our installation with:

```
sudo add-apt-repository \
 "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
 $(lsb_release -cs) \
 stable"
```

Finally, install the various Docker components needed using the following command:

```
sudo apt-get -y install docker-ce docker-ce-cli containerd.io docker-compose
```

We should now have a working Docker installation, including all of the necessary back-end services as well as the command line tools to launch containers. To make life easier later in the tutorial we should add our username into the docker group, which will save us from having to remember to prefix every command with *sudo*. To do this, run the command:

```
sudo usermod -a -G docker $USER
```

Log out and back into the console/ssh session.

Creating containers

We can now test our installation is working correctly by using the Docker 'hello-world' container image via the command `docker run hello-world`. This container is not very exciting – all it does is display a confirmation message to show that Docker is running correctly.

Let's take a quick look at the command we have just run to understand what it does. The `docker` command provides a way to interact with the container engine running on our Docker host and allows us to launch and manage containers. We used the `run` option here to create and launch a new container using the image name 'hello-world' (a test image provided by the team at Docker Inc.). As we will see when we run the command on the Docker host, the first thing that happens is that our Docker installation cannot find that image locally, so it searches for the name on Docker Hub (see the boxout for more on Docker Hub). It then downloads the image to our server and launches a container using it.

Now let's look at some of the other Docker command line options, starting with `docker ps`, which shows the status of all running containers. If we run this command immediately after executing the hello-world example, it's surprising to see that nothing is listed at all. The reason for this is that when the hello-world container ran, it simply displayed the welcome message and then exited, so the container is no longer running. We can see all containers (running or not) with the command `docker ps -a`, which should show the stopped hello-world container. This command shows the ID of each container in the first column (this is a unique reference on each Docker installation – for this tutorial the ID was 8600d8c3a86a) along with information such as the image used to create the container, along with the current status and more.

We can delete a container as long as it is not running using the `docker rm` command. As our hello-world container is already stopped, we can delete it using the command `docker rm 8600d8c3a86a` (replace the ID with the value shown in your system when running the `docker ps -a` command).

So far we have created and listed containers as well as deleting them. However, we have seen only a small subset of the capabilities of Docker. We'll now move on to installing Nextcloud, and the first decision we need to make is what container image to use. For this tutorial we will use the excellent images from the team at **www.linuxserver.io**. Their website provides a list of the images they maintain at **https://fleet.linuxserver.io/** with a link to Docker Hub showing documentation for each image in turn.

We are going to use Docker bind mounts in this tutorial – see the boxout (*right*) for more information – and to prepare for this we will create some directories inside our home, using the following commands:

```
mkdir -p ~/nextcloud/{config,data}
mkdir -p ~/mariadb/config
```

With the directories made, manually create a new Docker container using the `docker create` command. Unlike the `docker run` command we used earlier, this command defines the container but does not launch it, and we can use the following command to do so:

```
docker create \
 --name=nextcloud \
```

```
tutorial@lxf260: ~
tutorial@lxf260:~$ sudo docker-compose up -d
[sudo] password for tutorial:
Pulling mariadb (linuxserver/mariadb:latest)...
latest: Pulling from linuxserver/mariadb
d6ba4c98da8e: Pull complete
724682e7cc32: Pull complete
d58a1db0701e: Pull complete
365fb4258aff: Pull complete
a94ffd1d50d9: Pull complete
51a3e23d7395: Pull complete
Digest: sha256:95f1ac24e8eea8dae06f4c47ac1bc48b273bc842d3f6203fc0612f0be5671cf5
Status: Downloaded newer image for linuxserver/mariadb:latest
nextcloud is up-to-date
Creating mariadb ...
Creating mariadb ... done
tutorial@lxf260:~$ sudo docker ps
CONTAINER ID        IMAGE                  COMMAND             CREATED
21f87909c3ef        linuxserver/mariadb    "/init"             12 minutes ago
8f4d6529fdaf        linuxserver/nextcloud  "/init"             45 minutes ago
tutorial@lxf260:~$
```

```
 -e PUID=1000 \
 -e PGID=1000 \
 -e TZ=Europe/London \
 -p 443:443 \
 -v ~/nextcloud/config:/config \
 -v ~/nextcloud/data:/data \
 --restart unless-stopped \
 linuxserver/nextcloud
```

Docker Compose allows you to start and stop a number of containers with a single command while also ensuring consistency.

Breaking down the command above, the first option gives the container a friendly name (this saves us from having to use the container ID and needs to be unique on your system). The next three lines define environmental variables that control the user and group IDs under which the container will run, as well as our time zone.

The next option is new in this tutorial and tells Docker to direct traffic destined to TCP port 443 on the host to this container – a bit like setting a port forwarding entry on a broadband router.

The next two lines tell Docker to mount the two directories we created earlier to **/config** and **/data** respectively, inside our new container, while the penultimate option tells Docker to restart the container

» CONTAINERS VS HYPERVISORS

Virtualisation is another way to separate applications or services – such as enabling you to easily run separate instances of applications on one physical PC. A virtual PC (whether you use *VirtualBox*, *VMware* or any other version) emulates a full hardware stack, so every virtual instance needs a full OS installation. While you can simplify this process by copying an existing virtual machine, you cannot avoid the inevitable duplication of system files. Enterprise versions often provide tools to streamline this process, but they come at a cost.

The biggest difference is that most container platforms (including Docker as used in this tutorial) provide tools to bring your container into a known state, be that versions of libraries, configuration files or exposed network ports. In contrast, with a virtual machine you need to install the OS, install packages, copy configuration files, etc. (either manually or using a tool such as *Ansible* or *SaltStack*). Another significant advantage is volume mapping, which allows us to map a directory on our container (for example **/etc/resolv.conf**) to a file contained on our host file system. This feature is not only restricted to individual files, but also whole directories.

Despite this containers are not the solution to every requirement – they tend to work best when you can break a service into multiple small pieces (e.g. web database, etc.) and coordinate them via something like Docker Compose. The industry buzzword for this approach is using micro-services.

unless we explicitly tell it to stop. And finally, we specify the name of the container image we wish to use on Docker Hub.

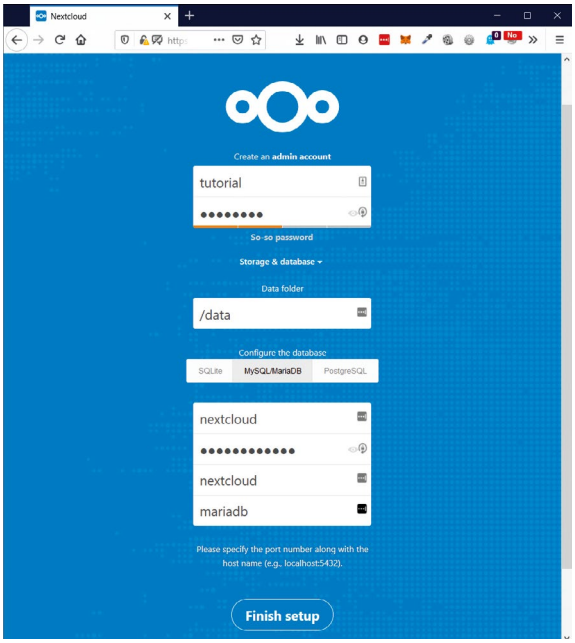
When we run the above command, the first thing that happens is that the Docker installation will realise it does not hold a local copy of the image and so will download and extract it (as an aside here, at the time of writing the size of the linuxserver/nextcloud image was 133MB, not bad when compared to the size of a virtual disk file) and will then return the full container ID when finished. We can now start our new container using the command `docker start nextcloud`, and we can test it easily enough by opening a web browser and pointing it at the IP address of our Docker host (remember to use https://). After ignoring the security error (the container uses a self-signed certificate that won't be trusted by your browser) we should see the Nextcloud setup wizard.

We now have a working instance of Nextcloud running in Docker created using a single (albeit multi-line) command, and we can start and stop it using a simple command line tool. This container has just the services and libraries needed to run the application (hence why it is only 133MB in size), but it uses the Linux kernel of our Docker host.

Creating containers this way is great when we want to test out a new application or service. However, if we want to run a service permanently, there are some big disadvantages to using `docker create`. While it is possible to inspect an existing container and determine settings like port forwarding and volume mounts, there is no record kept of the `docker create` options we used to create the container (the only exception here is if it's in the **bash_history** file).

This disadvantage becomes more obvious when we consider how we go about updating an existing container when there is an update to the image, which is the following:

› `docker pull linuxserver/nextcloud` updates the container image
› `docker stop nextcloud` stops the existing container
› `docker rm nextcloud` removes the existing container



Complete the Nextcloud setup wizard in your browser and use the database values defined in our docker-compose.yml file.

› `docker create --name=nextcloud -e PUID=1000` recreates the container as before

The above process is not particularly onerous with a single container, but if our use of Docker grows and we maintain several containers, then repeating this process for every container is going to be both monotonous and prone to errors. In addition, if we need to move containers to a new host (for instance, we test them on a laptop and then want to move them to a server), we would need to keep a record of every setting we used to create the containers.

DOCKER COMPOSE

Enter *docker-compose*, an orchestration tool that can help us manage our containers and greatly simplify activities such as upgrading, changing settings and moving containers. The tool enables multiple services to be defined as well as the dependencies between them, and relies on a single configuration file (in YAML format).

We will dive straight in and redefine our Nextcloud service this way, but to start with we need to remove the instance we created earlier using the following:

```
docker stop nextcloud
docker rm nextcloud
```

We'll now create a file named **docker-compose.yml** in our **home** directory and paste the following content in (spacing is important, and each indentation level below is made up of two spaces):

```
version: "2"
services:
  nextcloud:
    image: linuxserver/nextcloud
    container_name: nextcloud
    environment:
      - PUID=1000
      - PGID=1000
      - TZ=Europe/London
    volumes:
      - ~/nextcloud/config:/config
      - ~/nextcloud/data:/data
```

» DOCKER HUB

While Docker is not the only container solution out there, it's definitely one of the better-known ones. Anyone can create a Docker image from scratch, but the learning curve is quite steep, so being able to start with an existing container can make a big difference as we start to work with it. Docker provides a central repository of images called Docker Hub, and this is accessible to everyone immediately after Docker is installed – we don't need to register or create API keys, we can pull an image just by referencing its name.

Docker Hub contains both official images created by the team at Docker Inc. as well as a vast array of images created by the community. One such community team are the guys at **www.linuxserver.io** who maintain a significant library of images on Docker Hub (100 at the time this tutorial was written). A full list of all their images can be found at **https://fleet.linuxserver.io** with each image linking to a full documentation page at Docker Hub.

There are many other community teams and individual developers who have contributed images to Docker Hub, and the chances are very high that you will find an image for your requirement.

```
    ports:
      - 443:443
    restart: unless-stopped
```

While the above format is slightly different from the `docker create` command, the settings are the same.

With the file created we can create and launch our container using a single command: `docker-compose up -d` (the **-d** flag tells the `compose` command to start the containers in detached mode – ie running in the background). The command will create a new network and then create and start the Nextcloud container for us. We can check it is running correctly using the '`docker ps`' command we ran earlier and should see the new Nextcloud container running.

With Nextcloud defined, let's now add an additional container to run MariaDB (we would normally run Nextcloud with either MariaDB/MySQL or PostgreSQL). To add an extra container we need to define an additional service in our **docker-compose.yml** file. We can paste the content below into that file (note that the spacing is important and the first line must have two spaces in front of it):

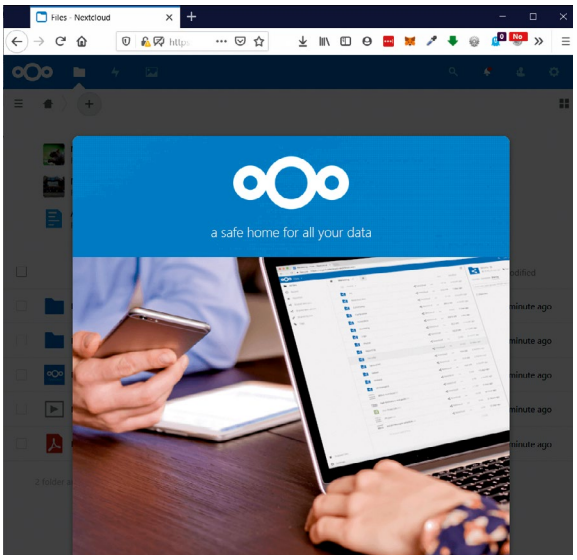
```
  mariadb:
    image: linuxserver/mariadb
    container_name: mariadb
    environment:
      - PUID=1000
      - PGID=1000
      - MYSQL_ROOT_PASSWORD=ChangeMePlease
      - TZ=Europe/London
      - MYSQL_DATABASE=nextcloud
      - MYSQL_USER=nextcloud
      - MYSQL_PASSWORD=ChangeMeAlso
    volumes:
      - ~/mariadb/config:/config
    ports:
      - 3306:3306
    restart: unless-stopped
```

Again we are using an image supplied by the team at **www.linuxserver.io**, and while the content above looks slightly different to the earlier example, it is still broken into the same sections.

For this image we can define both the root password and create an application-specific database at the point the image is created, all of which is controlled using environmental variables. This is extremely powerful, because the new container will start and already have the database, username and also the password defined in one go.

```
tutorial@lxf260: ~
tutorial@lxf260:~$ docker-compose pull
Pulling mariadb (linuxserver/mariadb:latest)...
latest: Pulling from linuxserver/mariadb
Digest: sha256:95f1ac24e8eea8dae06f4c47ac1bc48b273bc842d3f6203fc0612f0be
Status: Image is up to date for linuxserver/mariadb:latest
Pulling nextcloud (linuxserver/nextcloud:latest)...
latest: Pulling from linuxserver/nextcloud
05b6fb9d8a1c: Pull complete
4fa2725a3e0c: Pull complete
16b1397224c3: Pull complete
cc14d1c178a5: Pull complete
e82d4e8ef35f: Pull complete
75185832936d: Pull complete
ecbcd472b557: Pull complete
Digest: sha256:dda9e81be8556e101ae3826285cf67b716d6081a6c514c206d4f5cd98
Status: Downloaded newer image for linuxserver/nextcloud:latest
tutorial@lxf260:~$ docker-compose up -d
mariadb is up-to-date
Recreating nextcloud ...
Recreating nextcloud ... done
tutorial@lxf260:~$
```

Updating containers is a breeze with Docker Compose, with only two commands required, whether you are managing 2 or 10 containers.



We're done, Nextcloud and MariaDB are running in Docker, and with the wizard completed Nextcloud is working and ready for use!

With our new service defined, we can launch it using the same command as before (`docker-compose up -d`). This time, as we are using a new image for the first time, we will see `compose` pull down a copy of the latest MariaDB image from Docker Hub and then create and start our container. If you encounter any problems at this point it will probably be due to incorrect spacing/indentation of the **docker-compose.yml** file. You can pull down a complete version of the file from **https://github.com/prel-im/lxf260/blob/master/docker-compose.yml**.

We should now have two containers running on our Docker host: nextcloud and mariadb. We can now complete the Nextcloud setup wizard by pointing a web browser at the IP address of our Docker host (remember to prefix with https://) and complete the MySQL/MariaDB entries using the values defined in our **docker-compose.yml** file (change the Database host value from **localhost** to **mariadb** – this is a neat feature whereby containers can resolve each other by name).

One of the consequences of a container approach is that updates tend to be delivered quite quickly in small batches, so we'll now take a look at how to upgrade containers (when managed by `docker-compose`). We can do so by running the following commands (assuming you are in your **home** directory):

```
docker-compose pull
docker-compose up -d
```

The above commands will pull down any updates to images defined in the container files and then remove and recreate any container for which an upgraded image is available (containers with no updates will not be interrupted). Compared with the earlier example for manually created containers (four steps for each container) we can see that there is substantial scope to reduce the effort involved in managing even a small estate of Docker containers.

For now we are done. In a future edtion of this tutorial we shall look at Traefik, a reverse proxy that can automatically discover services on your Docker host. LXF

QUICK TIP

Docker allows you to map TCP or UDP ports in your containers through to your host. You cannot map a container to a port on your host that is already listening (e.g. if you run a SSH server on the host, you cannot map TCP/22 to a container).