KIT304 Server Administration and Security Assurance

Tutorial 14

Goal

In this tutorial you will learn about using containers through the Docker container system. This includes using containers to replicate what you've done earlier with stand-alone web server and database server installations.

Introduction

In this tutorial you will explore setting up a web server, database server, and content management system using containers. You'll be doing this on the CentOS virtual machine, using Docker. While this is a Unix-based tutorial, everything you do here would work almost identically if you were using a Docker installation running on Windows or macOS, and that reflects one of the strengths of Docker – it's easy to use almost any environment to quickly build containers, and it's very easy to move containers from one platform to another.

Getting Started with Docker

1. Launch CentOS as you usually would, and log in as the **Student** user. You may want to maximise the CentOS window, since you'll be doing the bulk of the work in this tutorial in this VM.

2. To complete this tutorial, you're going to need access to the public internet – specifically, you'll be connecting to the Docker Hub to download Docker images for the various services that you want to run. Docker itself has already been installed on the CentOS virtual machine, but no images or containers are yet available.

   The CentOS VM has two Ethernet interfaces. You may recall from earlier tutorials that you've only ever configured the **ens33** interface. This interface provides a private, internal connection to the other VMs in the cluster of VMs that you use for this unit. The second (unused to-date) interface is **ens38**. This interface, when enabled, creates a connection to the external internet via NAT (much like your home router provides NAT from your internal systems to the external internet). To enable the **ens38** interface, open a terminal window, and enter the following command:

   ```
   nmcli connection up ens38
   ```

After a short pause, you'll see a "connected network" icon appear on the status bar at the upper-right of the screen, indicating that the connection is now active. This VM can now connect to the Docker Hub via the Mac's Ethernet port and a NAT translation layer service running between the Mac and the CentOS VM. You don't need to provide an IP address for this VM either, since that's provided by a DHCP server that's running on the Mac.

3. Although Docker is installed, it is not actually running at this point. Start Docker with the following command (note that you need to become the root user first, the password is **toor**):

```
su -
service docker start
```

4. Once Docker is running, you can test it with the following command (in the same terminal window as the previous command, and still as the root user):

```
docker run hello-world
```

This instructs Docker to make a running container from the **hello-world** image. Since there's no **hello-world** image on the machine, Docker connects to the Docker Hub and searches for an image of that name. When it finds it, it will download that image, and then spin it up into a running container. This includes downloading any binary files required by the image. For very large images, this can take a minute or more, depending on the speed of your internet connection.

This particular container doesn't do anything particularly useful – it just outputs a message and terminates – its purpose is merely to prove that Docker is installed and running, and that it can connect to the Hub, download an image, and spin it up into a container.

> ➢ It's important to understand that this container terminates when it's done. It is no longer running, and can't be interacted with. This is the default behaviour for containers – when the program they run at start-up ends, the entire container terminates. If you need a container to continue running, you need to ensure that the start-up process never terminates, or can restart itself should it crash unexpectedly.

5. Since we don't want to be continually running all of our Docker containers as the **root** user, let's configure the system to allow the **student** user to also run Docker containers. When Docker was installed, it added a new group to the system, named **docker**. Anyone who is a member of this group has the permission to run Docker containers. To add the **student** user to the **docker** group enter the following command (still as the root user):

```
usermod -aG docker student
```

You should recognise this command from an earlier tutorial in the Unix module – it *appends* the group **docker** to the list of groups that the **student** user is currently a member of. To have this change be recognised by the operating system, you need to log out and log in again. To do that, go to the top right-hand corner of the CentOS screen, and click on the "power" icon. In the menu that pops down, click **student** and then select **Log Out**, and then click the **Log Out** button in the confirmation dialog that is displayed.

Now log back in again as the **student** user, open a new terminal window, and enter the following command:

```
docker run hello-world
```

You should see that while you get the same output as before, the entire process was quicker, and didn't involve downloading any content from the Docker Hub before the container was run. That's because Docker caches image content that hasn't changed, making it faster to re-run something that you've already set up in the past. Docker will only rebuild or redownload a container if an image that it depends on has been modified.

Building a Docker-based Web Server

Your next task will be to use Docker to spin up a web server container that's built on Apache and that already has PHP pre-installed. The Docker Hub hosts official PHP Docker container images for this purpose – all you really need to do is create the content for a very simple web site, and a small configuration file that tells Docker which specific container image you want to use. Do that as follows:

6. In the terminal, create a directory for your project called **DockerDemo**, and move into it:

```
mkdir DockerDemo
cd DockerDemo
```

7. Create a subdirectory for your web site source code, and get ready to create some content in it:

```
mkdir src
cd src
```

8. In your `src` directory, create the file `index.php`, with the following content:

```
<?php
echo "Hello, World!";
?>
```

9. Move back up in the folder hierarchy one level (to the **DockerDemo**) folder, and create a configuration file for Docker named `Dockerfile` with the following content:

```
FROM  php:8.0-apache
COPY  src/  /var/www/html
EXPOSE  80
```

The first command tells Docker to pull down (from the Docker Hub) the official PHP 8.0 image with Apache support. The second tells Docker to copy the **src** folder (that sits alongside your Dockerfile, and that contains your ultra-simple web site) into the **/var/www/html** directory in the new image. You should recall that this directory is the default area from which Apache serves web content. Finally, the last command tells Docker to expose port 80 in the running container to the local machine – allowing the Apache server to listen for incoming HTTP requests on port 80. Due to the sandboxing that Docker uses, this communication through port 80 is the only communication that can happen between the container and the host system.

10. To build your image enter the following command:

```
docker build -t hello-world .
```

This tells Docker to **build** the image using the Dockerfile in the current directory (**.**) and that the name of the new image should be **hello-world**.

The first time you run this build process it will take about 30 seconds to complete, since it has to pull down a number of component parts of the base PHP image from the Docker Hub. The command displays its progress as a series of steps, one step for each command in your Dockerfile. After the new image is built, you should see the last two lines of output are as follows:

```
Successfully built someUniqueID
Successfully tagged hello-world:latest
```

This means your image is ready to run. If you don't see similar output, check your work, and ask the tutor for assistance if you can't find and resolve the problem.

> ➢ That unique ID you see in the output (and the others above it output by the build process) are portions of hashes of the image state at each phase of the build process. The hashes are what Docker uses to know whether a part of the build chain has changed, and therefore needs to be rebuilt, or whether the previously built and still cached version can be used.

11. You're now ready to run your new image. To do that, enter the following command:

```
docker run -p 80:80 hello-world
```

This tells Docker to **run** the image named **hello-world** and bind port 80 on the CentOS system to port 80 (the one you exposed in the Dockerfile) in the running container. So any incoming traffic to port 80 on CentOS will be forwarded into the container, which happens to be running the Apache web service.

When you run this command, your console will effectively become the logging output of the Apache web service – any incoming requests will be forwarded to the container, and those requests will also be logged onto the console.

12. Open the CentOS Firefox browser, and, in the address bar, enter the URL **localhost** and press the **return** key. Since the browser defaults to using port 80, and you've told Docker to bind port 80 to port 80 in your running container, this request will be passed to the Apache server in the container, which will execute your PHP script, and return the result to the browser. You'll even see the log of the request in the terminal window where you're running the container.

13. Open a new terminal window (so as not to disturb the running Apache/PHP container in the original window), change into your **DockerDemo/src** directory, and modify the **index.php** file so it outputs a different message, then reload the page in the browser. Did your new message appear?

    It shouldn't since changes you make in the **src** directory aren't reflected in the running container. That's because the image that you built back in steps 9 and 10 used the **COPY** line in your Dockerfile to copy the **src** directory *at that point in time* into the new image. Changes you make now to the **src** directory have no effect, since they're outside the container's sandbox.

    You can fix this by stopping the current container (press control-C in the Apache logging window), rerunning your **docker build** command, and then rerunning your **docker run** command.

14. Sealing the content of a web site (like the contents of your **src** folder) into a container is generally only done in production environments. In development environments, you typically want to be able to make changes to the web site source code, and have them reflected instantly in your running service. You can do that in Docker with the following tweaks:

    - Stop the Apache container running (by typing a control-C in the window where it's logging).
    - Edit the **Dockerfile** that defines your image, and remove the second line that copies the **src** directory into **/var/www/html**.
    - Rebuild the image by re-entering your earlier **docker build** command.
    - Spin your updated image up into a running container with the following command:

    ```
    docker run -p 80:80 -v /home/student/DockerDemo/src:/var/www/html hello-world
    ```

This longer command includes a new **-v** option which specifies a *volume mount*. In this case, you're telling Docker to mount the directory **/home/student/DockerDemo/src** on the host OS (CentOS) onto the directory **/var/www/html** inside the running container. This is a *live* mount – any changes you make in the **src** directory are immediately visible inside the container when it is running, and any changes the container might make in **/var/www/html** are immediately reflected in the **src** directory on CentOS.

> ➢ This **-v** option effectively opens up another hole in the isolation that exists between the host and the container (on top of the port 80 hole the **-p** option already opens).

15. Verify that you can now make changes to the **index.php** file, and when you reload the page in the browser, those changes are instantly visible.

    Before moving on, enter a control-C in the terminal running the web server container so that the container shuts down.

Taking Stock

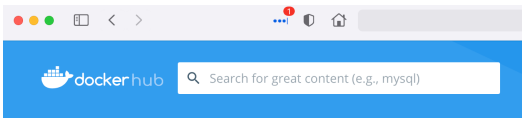
At this point, you've not done anything that can't be done on a stand-alone Unix system with Apache and PHP installed – but what is different here is that you've done this with a very small configuration file (the **Dockerfile**) and your web site content.

The beauty of Docker is that you could move those few files onto a totally different piece of infrastructure (for example, an Ubuntu Linux system, or a Mac or a Windows-based PC), and as long as Docker was installed, you could run the same commands to build the image and run the container, and it would behave in exactly the same way as it does on the CentOS system, without you having to manage the installation of Apache or PHP in any way – Docker has abstracted those processes away into that single **FROM** command at the first line of your configuration file.

Exploring Docker Hub

Docker Hub has been mentioned several times in this tutorial already. Docker Hub is a repository for Docker images. It contains thousands of official and unofficial images that you can use freely, and offers a paid service that allows you to host your own images in your own private repository.

16. To explore Docker Hub, open https://hub.docker.com in your preferred web browser. In the upper left corner of the web page you'll find a search field. Click in this field, enter the text **php**, and press return.



17. The first of the thousands of results that are returned is for the official PHP image collection. Click on the main **php** text title in the first search result, and you'll be taken to the PHP Docker Official Images page. Scroll down this page to the **Supported tags and respective Dockerfile** links section.

    Tags are unique references to specific Docker images in the Docker Hub repository. They are usually presented as a version number, followed by some descriptive text. You'll see multiple PHP tags which represent different versions of PHP, built on different base Linux distributions.

    At the time this tutorial was written, the first row of tags looked like this:

    - `8.0.3-cli-buster`, `8.0-cli-buster`, `8-cli-buster`, `cli-buster`, `8.0.3-buster`, `8.0-buster`, `8-buster`, `buster`, `8.0.3-cli`, `8.0-cli`, `8-cli`, `cli`, `8.0.3`, `8.0`, `8`, `latest`

    These tags all represent images that provide a command-line version of PHP, built on a slimmed-down version of the Debian **buster** release (which is version 10 of the Debian OS). Notice how the version numbers get less precise as you move to the right. You may not realise that while PHP is commonly used to develop web sites, it is also a general purpose language, and can be used from the command line as a general programming language, much like Perl or Python – so these images (with the **`cli`** designation) are useful for developers wanting to write and run PHP command-line programs via containers.

    The second row of tags looks like this:

    - `8.0.3-apache-buster`, `8.0-apache-buster`, `8-apache-buster`, `apache-buster`, `8.0.3-apache`, `8.0-apache`, `8-apache`, `apache`

    These tags all represent images that provide a version of PHP integrated with the Apache web server – again all built on a slimmed-down **buster** release. In fact, the third last tag on this line is the one that you used in your Dockerfile earlier, which looked as follows:

    ```
    FROM php:8.0-apache
    ```

    As you look further down the page, you can see tags for images that are built on **Alpine** rather than Debian buster. Alpine is a very minimal Linux distribution that is sometimes used in image builds to keep the running container footprint as small as possible. The images that include **fpm** in the tag name have the **FastCGI Process Manager** built into the image – this is an alternative PHP FastCGI implementation sometimes used on heavy-loaded sites. The images that include **zts** in the tag name include the **Zend Thread Safety** package, which is required in PHP scripts that use **pthreads**.

    If you continue to scroll further down the page, you'll see additional sections that describe further features of the images, and how to use them in various ways.

18. Click on one of the tags that are listed for any of the PHP images. You'll be taken to a new page that shows the content of the actual Dockerfile that was used to build the image for the specific tag that you clicked on. At the top, you'll see a FROM line that specifies the base Linux image that this image depends on, and below that, you'll see multiple RUN, COPY and other commands that are executed during the build phase. This is in exactly the same format as the Dockerfile you created, and the sorts of things this Dockerfile does can all be done in your own Dockerfiles. If you choose to use Docker for real projects, you'll almost certainly be creating customised containers that feature your own inclusions like this.

19. Go back to the previous page (the page that lists all of the PHP tags) and scroll back to the top. In the search field at the top left, search for some other images – such as **MariaDB** and **Wordpress**, and explore the tags and features for each. While you're searching, see what other tools that you use might be available in Docker container format. You may be surprised to see that you can even get images for full Linux distributions, such as Ubuntu and CentOS.

Managing Multiple Containers with Docker Compose

So far, you've built a single running container with Docker. You can also run multiple containers simultaneously, and create links and dependencies between them. Rather than do this with multiple **`docker build`** and **`docker run`** commands, the **`docker-compose`** tool is frequently used instead.

Docker's **compose** tool is used to define and run multi-container Docker applications. It can still use a basic Dockerfile to describe each image, but also uses a **YAML** file to declare how the services relate to each other, what volume(s) they mount, and what port(s) they bind. You then use a single command to build them, to start up all of the services at the one time, or shut them all down.

> ➢ YAML files are text files structured in a defined way that are often used as configuration files and for simple data storage. Comparable formats are JSON and XML.

Docker even sets up named host mappings for each container created via compose, and allows the various containers to use each other's names as fully qualified domain names – so anywhere in a container that you need to refer to another host by name (for example – web sites often need to know the hostname of their database server) you can use the container name as that host name.

In this section, you'll use docker and docker-compose to build a blogging application that uses two containers that communicate with each other. One will host your database process, and the other will host your web server/blogging engine.

20. Start by creating the following directory and file structure in the student home directory (not the DockerDemo directory):

```
blog/                          (a subdirectory)
     docker-compose.yml        (a text file – see below for the content)
     web/                      (a subdirectory)
          Dockerfile           (a text file – see below for the content)
          src/                 (a subdirectory for the WordPress source code)
     database/                 (a subdirectory)
          Dockerfile           (a text file – see below for the content)
          db/                  (a subdirectory for the MariaDB databases)
```

21. Set the content of **blog/web/Dockerfile** to the following:

```
FROM wordpress:latest
EXPOSE 80
```

22. Set the content of **blog/database/Dockerfile** to the following:

```
FROM mariadb:latest
RUN echo "Australia/Hobart" > /etc/timezone
```

23. Check that both containers can be built manually. The following commands assume that you are currently inside the **blog** directory – not lower down in any of its children:

```
docker build -t apache web/
docker build -t mariadb database/
```

Assuming you get no errors, you're ready to proceed with the next step. Otherwise, work out where the problem is, or ask your tutor for help.

24. Now create the file **docker-compose.yml** (inside the **blog** directory) and populate it with the following content. Note as you enter the content into this file that it has specific levels of indentation that must be preserved. Typically, the default indentation in YAML files is to use two spaces for each indentation level. In addition, replace the green values with values of your own that make sense.

```
version: '3'

services:

  database:
    build: ./database
    container_name: mariadb
    volumes:
      - ./database/db:/var/lib/mysql
```

```
    environment:
      - MYSQL_DATABASE=root
      - MYSQL_ROOT_PASSWORD=mySecretPassword
    ports:
      - "127.0.0.1:3306:3306"
    user: "1000:1000"

  web:
    build: ./web
    container_name: wordpress
    environment:
      - WORDPRESS_DB_HOST=mariadb
      - WORDPRESS_DB_USER=wpDatabaseUsername
      - WORDPRESS_DB_PASSWORD=wpUserPassword
      - WORDPRESS_DB_NAME=wpDatabaseName
    volumes:
      - ./web/src:/var/www/html
    ports:
      - "80:80"
    user: "1000:1000"
    depends_on:
      - database
```

The first line declares that this compose file follows version 3 of the docker-compose layout standard.

The next line declares that a nested series of services will be defined. One of these services is labelled "**database**", and the other is labelled "**web**". These names can be used internally within the YAML file when one service needs to refer to another (for example, we've declared that the **web** service **depends_on** the **database** service being up before it can be run).

Within each service definition, you can see the location of the **`build`** directory relative to the YAML file, the name of the container that each service will build (**`mariadb`**, **`apache`**), the volume mount(s) that must be in place when the container is running, and which port(s) are mapped between the container and the host OS. Some of those values (such as the container name, volume mounts, and mapped ports) serve the same role as if they were entered on the command line as part of the **`docker build`** and **`docker run`** commands.

Both the **database** and **web** services include environment variable definitions – for example, the database service includes one named **`MYSQL_ROOT_PASSWORD`** with some value you've specified. The first time the MariaDB container is run, it will create an empty set of databases, and set the password for the **root** user to whatever you assigned to this environment variable. Passing variable values into containers in this way is quite common in Docker images.

The web service includes multiple environment variables, which are used on a new container to create a database for WordPress to use, along with a username and password to access it.

Finally, we've specified that the user ID and group ID of the processes that run inside the container should be the same as those of the CentOS **student** user (1000, 1000). This isn't always done in Docker containers, and if it is specified, it's typically specified via environment variables rather than hard-coded values like this. Today, we're doing this as a short-cut. This setting ensures that files created in the CentOS file system by the running containers are all owned by the student user, which makes it easier for you to read and modify them from the CentOS environment. (Since containers are as light as possible, they don't usually include tools like **vi** for editing files, so making files editable from the hosting OS can be very useful).

25. With your **docker-compose.yml** file in place, you can now use the **docker-compose** command to build everything:

    ```
    docker-compose build
    ```

    If everything builds correctly, you're ready to run your database and web containers with the following command:

    ```
    docker-compose up
    ```

    This brings up both running containers and leaves them running and attached to the console – you will be able to see log messages from both Apache and MariaDB as the two containers run. Docker also creates a virtual network connection between the containers that allows them to communicate with each other, using the container name as a host name.

26. As this is the first time that MariaDB has run, it will have created an empty set of databases, and set the root user password to the value you specified in the **MYSQL_ROOT_PASSWORD** environment variable in the docker-compose YAML file. But there's still some work you need to do with the database in order to prepare it for use with WordPress. You should recall these tasks from tutorial 9 – they need to be entered on the command line, but your MariaDB server is now running inside a container. How do you access its command line? Docker provides a tool that allows you to run binaries that exist inside containers, as follows:

    ```
    docker exec -it container binary
    ```

    This executes (**exec**) the executable *binary* in the specified *container*. The **-it** options tell Docker to run this command interactively and allocate a pseudo-TTY for the purpose – which is always required when you're running an interactive process like a shell.

    Open a new terminal window, and enter the following command to run a bash shell in the MariaDB container:

    ```
    docker exec -it mariadb bash
    ```

You'll get a groups warning message (which you can ignore), followed by the shell prompt, from a bash process running inside the container. It will look something like this:

```
I have no name!@b3ab39f98db1:/#
```

This is much like a **telnet** or **ssh** session to another host. Commands you enter at this prompt will be executed inside the container, not on the CentOS host.

27. Since you're building a WordPress CMS, you need to create a database and username/password pair that WordPress can use to access it (just as you did in tutorial 9). While still in the MariaDB container's command line, open MariaDB using **mysql -u root -p** and when prompted for the password, enter the root user password from the **docker-compose.yml** file.

    Create a database user for WordPress with the following commands, using the same values for the database name, the username, and the password that you specified in the **web** service section of the **docker-compose.yml** file:

    ```
    CREATE USER userName IDENTIFIED BY 'password';
    ```

    Next, create a database for Wordpress. Again, choose any name you want here for the database. If you can't think of one, just use **wp** (for WordPress):

    ```
    CREATE DATABASE databaseName;
    ```

    Next, grant access to the user you just created to the database:

    ```
    GRANT ALL ON databaseName.* TO userName@'%' IDENTIFIED BY 'password';
    ```

    Finally, tell the MariaDB to clear its cached privileges (so the new ones you've created can be used) and quit the command-line tool:

    ```
    FLUSH PRIVILEGES;
    EXIT;
    ```

    The last command above will return you to the container's bash shell.

    Type the control-D character, or enter the command **exit**, to exit this shell, leaving the container environment and returning to the CentOS shell.

28. Return to the Firefox browser, and enter **localhost** in the address bar. It should redirect to a page called **install.php** asking you to choose your install language. Select the language you want, and click the **Continue** button. You'll be taken next to the same "Welcome" page that you saw in

tutorial 9. If instead it asks you about database information you have not completed the previous steps correctly.

Complete the form as required (noting what username and password you use, as these are required to administer the site afterwards) and then press the **Install WordPress** button. Once the installation process has finished, and you see the "Success!" response, click the **Log In** button to log into the site.

Once again, you'll be back in the same basic WordPress interface you saw in tutorial 9. Note that this is the latest version of WordPress, so it will look visually different to the version you were experimenting with in that earlier tutorial.

Create some new posts and pages, customise the web site theme, experiment with the site, and be sure to test the results in a browser that isn't logged in as the WordPress admin user so you can see what the site looks like to regular users.

Some Additional Docker Commands

At this point, you've created two running containers – one named **mariadb** running a database, the other named **wordpress** running the Apache web server with PHP and the WordPress blogging engine. These two containers are communicating with each other over a private internal network connection.

In the **`docker-compose.yml`** file, the Apache container exposes port 80 to CentOS port 80, so any connections coming from outside the CentOS host on that port are channelled into the container.

Similarly, the MariaDB container exposes port 3306 to CentOS port 3306, but it's not exposed to the outside world. In the **`docker-compose.yml`** file, the mapping is expressed as **`127.0.0.1:3306:3306`**. That leading localhost IP address restricts all access to port 3306 to processes running on the CentOS host, and prevents external hosts from accessing the database.

> ➢ Note: Misconfiguring port mappings in Docker can cause serious security issues. If you're running containers on public internet hosts, be very careful not to expose containers to the internet except where necessary. The MariaDB container in this case doesn't need to be publicly exposed, and so we secure it by only exposing it to the localhost.

At this point, your two containers should still be running with their console output being sent to the terminal window in which you ran the **`docker-compose up`** command. Just as you can run docker containers and put them into "daemon" (or background) mode, you can run multiple containers with docker-compose in the same way.

29. In the terminal window running the two containers, type control-C to terminate both containers. Now, rerun them with the **-d** option:

    docker-compose up -d

    Now your two containers are running in the background - the logging output is not sent to the terminal and you can enter other commands into the window. This is the usual way that docker containers are run with docker-compose once you've ironed out any issues.

30. Get a list of the docker containers that are running by entering the following command:

    docker ps

    The output will show you the names of the containers and their images, how long they've been running, and what port mappings they have in place.

31. Take all of the running containers from the current docker-compose project down again with the following command:

    docker-compose down

    This shuts down all of the services listed in the **docker-compose.yml** file, in a clean and coordinated way.

32. With your containers now shut down, take a look at the contents of the **blog/web/src** and **blog/database/db** folders. You should find that even though they were empty directories when you created this blogging project, they are now populated with files. The **web/src** directory contains the entire content of the Wordpress installation. The **database/db** directory contains the raw MariaDB database files.

    If you wanted to bring this project up on a different system with its own Docker installation, you just need to move the **blog** directory, and everything inside it, to that new machine, and use the **docker-compose build** command to rebuild it. When you run those newly-built containers, the existing content of the **web/src** and **database/db** directories will be used to power the new site. Moving the content of a web site and database to another host, when you're not using containers, can be a complex process. With Docker, it's very simple.

33. To completely remove all of your docker containers, including all of the binary data that Docker has downloaded to build them, enter the following command:

    docker system prune -a

You'll be asked to confirm this action – and you may be surprised to see that when you do so, that it frees up around 1 gigabyte of space, which represents all of the downloaded image data, plus the intermediate image phases that were cached during each of the build steps. Removing all of your content in this way is a good idea when you're no longer using Docker on a given system. It can also useful to run this if you're experiencing some types of issues that prevent containers from being built or run.

Note that this command doesn't touch your **`docker-compose.yml`** file, or the contents of your blog project – that's still available if you want to rebuild and rerun your containers.

Conclusion

Today, you've learned the basic principles of the Docker container system. Docker and similar container services are a powerful way to abstract away much of the complexity involved when installing and configuring applications and their dependencies, using simple configuration files. Containers, and container orchestration services like Docker Swarm and Kubernetes are now a mainstay of modern online applications such as web and application servers.

Skills to Master

To successfully complete this tutorial, you must have mastered the following skills:

- create basic **Dockerfile**s that define containers and configure them appropriately
- orchestrate the creation and coordinated execution of several Docker containers at the same time with **docker-compose**
- understand and be able to use the basic **`docker`** subcommands, such as **`build`**, **`run`**, **`ps`**, and **`system prune`**.

You could be required to demonstrate any of these skills in this module's practical exam.

Appendix – installing Docker on your own infrastructure

If you're running your own CentOS 8 system or virtual machine, you can install Docker as follows. In the CentOS terminal, as root, run these commands:

```
yum install -y yum-utils
yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo
yum erase podman buildah
yum install docker-ce docker-ce-cli containerd.io
```

As with other installs you've done, this doesn't actually start Docker – to do that:

```
systemctl start docker
```

The process is similar for Debian-derived systems (such as Ubuntu):

```
sudo apt-get update
sudo apt-get install apt-transport-https ca-certificates curl gnupg-agent software-
properties-common
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -
sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/debian $(lsb_release -cs) stable"
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

(above requires docker be run as root… see https://docs.docker.com/engine/install/debian/ for more tips including how to run as non-root user)

Docker-compose is not installed automatically on Linux systems – instead, you must manually install it as follows (enter this as a single line, with no space between "docker-" and "compose"):

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.29.0/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

Then, you need to make the binary executable, as follows:

```
sudo chmod +x /usr/local/bin/docker-compose
```