CHAPTER 2

Essential Docker Quick Reference

It can be difficult to remember how to perform common tasks when you start working with Docker on your own ASP.NET Core MVC project. This chapter provides a quick reference for the essential Docker features, along with references to the chapters in this book that explain how the features work and demonstrate their use.

Docker Images Quick Reference

Images are the templates from which containers are created. You build on top of a base image to provide a starting point for your projects, obtaining the base image from a repository such as Docker Hub and customizing it using a Docker file. Images and Docker files are described in Chapter 4.

The Docker image used to deploy ASP.NET Core applications does not contain the .NET Core compiler, which means you must use the `dotnet publish` command to create a directory that contains the compiled code and all of the support files required to run the application. Open a new command prompt and run the following command in your project folder:

```
dotnet publish --framework netcoreapp1.1 --configuration Release --output dist
```

This command publishes the application into a folder called `dist`, which can be incorporated in the image using the `COPY` command in a Docker file. The Docker file is processed to create the image using the `docker build` command.

```
docker build . -t apress/exampleapp -f Dockerfile
```

The first argument is a period, indicating the current working directory, which sets the context directory on which the commands in the Docker file are performed. The `-t` argument specifies the name of the image, and the `-f` argument specifies the Docker file. Table 2-1 lists the essential commands for working with images.

A. Freeman, *Essential Docker for ASP.NET Core MVC*, DOI 10.1007/978-1-4842-2778-7_2

Table 2-1. *Essential Commands for Working with Images*

Command	Description
`docker build`	This command processes a Docker file and creates an image.
`docker images`	This command lists the images that are available on the local system. The `-q` argument returns a list of unique IDs that can be used with the `docker rmi` command to remove all images.
`docker pull`	This command downloads an image from a repository.
`docker push`	This command publishes an image to a repository. You may have to authenticate with the repository using the `docker login` command.
`docker tag`	This command is used to associate a name with an image.
`docker rmi`	This command removes images from the local system. The `-f` argument can be used to remove images for which containers exist.

Table 2-2 lists the essential Docker images that are used for ASP.NET Core MVC projects and the examples in this book.

Table 2-2. *The Essential Docker Images for ASP.NET Core MVC Projects*

Image	Description
`microsoft/aspnetcore:1.1.1`	This Linux image contains version 1.1.1 of the .NET Core runtime and the ASP.NET Core packages. This image is used to deploy applications.
`microsoft/dotnet:1.1.1-runtime-nanoserver`	This Windows image contains version 1.1.1 of the .NET Core runtime. This image is used to deploy applications to Windows Server.
`microsoft/aspnetcore-build:1.1.1`	This Linux image contains version 1.1.1 of the.NET Core Software Development Kit. It is used to create development environments in containers.
`mysql:8.0.0`	This Linux image contains version 8 of the MySQL database server.
`haproxy:1.7.0`	This image contains the HAProxy server, which can be used as a load balancer.
`dockercloud/haproxy:1.2.1`	This image contains the HAProxy server, configured to respond automatically to containers starting and stopping.

Images are created using Docker files, which contain a series of commands that describe the container that the image will be used to create. Table 2-3 describes the essential Docker file commands used in this book.

Table 2-3. *The Essential Docker File Commands*

Command	Description
FROM	This command specifies the base image. For ASP.NET Core MVC projects, this command is generally used to select the `microsoft/aspnetcore:1.1.1` (for deployment) or `microsoft/aspnetcore-build:1.1.1` (for development) images.
WORKDIR	This command changes the working directory for subsequent commands in the Docker file.
COPY	This command adds files so they will become part of the file system of containers that are created from the image.
RUN	This command executes a command as the Docker file is processed. It is commonly used to download additional files to include in the image or to run commands that configure the existing files.
EXPOSE	This command exposes a port so that containers created from the image can receive network requests.
ENV	This command defines environment variables that are used to configure containers created from the image.
VOLUME	This command denotes that a Docker volume should be used to provide the contents of a specific directory.
ENTRYPOINT	This command specifies the application that will be run in containers created from the image.

Docker Containers Quick Reference

Containers are created from an image and used to execute an application in isolation. A single image can be used to create multiple containers that run alongside each other, which is how applications are scaled up to cope with large workloads. You can create containers using custom images or prebuilt images from a public repository such as Docker Hub. Containers are described in Chapter 4 and used throughout the book.

Containers are created using the `docker create` command, like this:

```
docker create -p 3000:80 --name exampleApp3000 apress/exampleapp
```

Once a container has been created, it can be started using the `docker start` command.

```
docker start exampleApp3000
```

You can create and start a container in a single step using the `docker run` command.

```
docker run -p 3000:80 --name exampleApp4000 apress/exampleapp
```

The arguments for these commands are used to configure the container, which allows containers created from the same image to be configured differently. Table 2-4 describes the essential arguments for these commands.

Table 2-4. *Essential Arguments for the docker create and docker run Commands*

Argument	Description
`-e, --env`	This argument sets an environment variable.
`--name`	This argument assigns a name to the container.
`--network`	This argument connects a container to a software-defined network.
`-p, --publish`	This argument maps a host operating system port to one inside the container.
`--rm`	This argument tells Docker to remove the container when it stops.
`-v, --volume`	This argument is used to configure a volume that will provide the contents for a directory in the container's file system.

Table 2-5 lists the essential commands for working with containers.

Table 2-5. *Essential Commands for Working with Containers*

Command	Description
`docker create`	This command creates a new container.
`docker start`	This command starts a container.
`docker run`	This command creates and starts a container in a single step.
`docker stop`	This command stops a container.
`docker rm`	This command removes a container.
`docker ps`	This command lists the containers on the local system. The `-a` argument includes stopped containers. The `-q` argument returns a list of unique IDs, which can be used to operate on multiple containers with the `docker start`, `docker stop`, and `docker rm` commands.
`docker logs`	This command inspects the output generated by a container.
`docker exec`	This command executes a command in a container or starts an interactive session.

Docker Volumes Quick Reference

Volumes allow data files to be stored outside of a container, which means they are not deleted when the container is deleted or updated. Volumes are described in Chapter 5.

Volumes are defined using the `VOLUME` command in Docker files, like this:

```
...
VOLUME /var/lib/mysql
...
```

This tells Docker that the files in the `/var/lib/mysql` folder should be stored in a volume. This is useful only when a named volume is created and applied when configuring the container. Volumes are created using the `docker volume create` command, like this:

```
docker volume create --name productdata
```

The `--name` argument is used to specify a name for the volume, which is then used with the `-v` argument to the `docker create` or `docker run` command like this:

```
docker run --name mysql -v productdata:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=mysecret -e
bind-address=0.0.0.0 mysql:8.0.0
```

This command, taken from Chapter 5, tells Docker that the `productdata` volume will be used to provide the contents of the `/var/lib/mysql` directory in the container's file system. Removing the container won't remove the volume, which means that any files that are created in the `/var/lib/mysql` directory won't be deleted, allowing the result of user actions to be stored persistently.

Table 2-6 lists the essential commands for working with volumes.

Table 2-6. *Essential Commands for Working with Volumes*

Command	Description
`docker volume create`	This command creates a new volume.
`docker volume ls`	This command lists the volumes that have been created. The `-q` argument returns a list of unique IDs, which can be used to delete multiple volumes using the `docker volume rm` command.
`docker volume rm`	This command removes one or more volumes.

Docker Software-Defined Networks Quick Reference

Software-defined networks are used to connect containers together, using networks that are created and managed using Docker. Software-defined networks are described in Chapter 5.

Software-defined networks are created using the `docker network create` command, like this:

```
docker network create backend
```

This command creates a software-defined network called `backend`. Containers can be connected to the network using the `--network` argument to the `docker create` or `docker start` command, like this:

```
docker run -d --name mysql -v productdata:/var/lib/mysql --network=backend -e
MYSQL_ROOT_PASSWORD=mysecret -e bind-address=0.0.0.0 mysql:8.0.0
```

Containers can also be connected to software-defined networks using the `docker network connect` command, like this:

```
docker network connect frontend productapp1
```

This command connects the container called `productapp1` to the software-defined network called `frontend`.

Table 2-7 lists the essential commands for working with software-defined networks.

Table 2-7. *Essential Commands for Working with Software-Defined Networks*

Command	Description
`docker network create`	This command creates a new software-defined network.
`docker network connect`	This command connects a container to a software-defined network.
`docker network ls`	This command lists the software-defined networks that have been created, including the ones that Docker uses automatically. The `-q` argument returns a list of unique IDs, which can be used to delete multiple networks using the `docker network rm` command.
`docker network rm`	This command removes a software-defined network. There are some built-in networks that Docker creates and that cannot be removed.

Docker Compose Quick Reference

Docker Compose is used to describe complex applications that require multiple containers, volumes, and software-defined networks. The description of the application is written in a compose file, using the YAML format. Docker Compose and compose files are described in Chapter 6, which includes this example compose file:

```
version: "3"

volumes:
  productdata:

networks:
  frontend:
  backend:

services:

  mysql:
    image: "mysql:8.0.0"
    volumes:
      - productdata:/var/lib/mysql
    networks:
      - backend
    environment:
      - MYSQL_ROOT_PASSWORD=mysecret
      - bind-address=0.0.0.0

  dbinit:
    build:
      context: .
      dockerfile: Dockerfile
    networks:
      - backend
```

```
    environment:
      - INITDB=true
      - DBHOST=mysql
    depends_on:
      - mysql

  mvc:
    build:
      context: .
      dockerfile: Dockerfile
    networks:
      - backend
      - frontend
    environment:
      - DBHOST=mysql
    depends_on:
      - mysql
```

This compose file describes an application that contains three services (services are the descriptions from which containers are created), two software-defined networks, and a volume. For quick reference, Table 2-8 describes the configuration keywords from this example compose file.

Table 2-8. *Essential Configuration Keywords Used in Compose Files*

Keyword	Description
`version`	This keyword specifies the version of the compose file schema. At the time of writing the latest version is version 3.
`volume`	This keyword is used to list the volumes that are used by the containers defined in the compose file.
`networks`	This keyword is used to list the volumes that are used by the containers defined in the compose file. The same keyword is used to list the networks that individual containers will be connected to.
`services`	This keyword is used to denote the section of the compose file that describes containers.
`image`	This keyword is used to specify the image that should be used to create a container.
`build`	This keyword is used to denote the section that specifies how the image for a container will be created.
`context`	This keyword specifies the context directory that will be used when building the image for a container.
`dockerfile`	This keyword specifies the Docker file that will be used when building the image for a container.
`environment`	This keyword is used to define an environment variable that will be applied to a container.
`depends_on`	This keyword is used to specify dependencies between services. Docker doesn't have insight into when applications in containers are ready, so additional steps must be taken to control the startup sequence of an application (as described in Chapter 6).

Docker files are processed using the `docker-compose build` command like this:

```
docker-compose -f docker-compose.yml build
```

The containers, networks, and volumes in a compose file are created and starting using the `docker-compose up` command.

```
docker-compose up
```

Table 2-9 lists the essential commands for working with compose files.

Table 2-9. *Essential Commands for Docker Compose*

Command	Description
`docker-compose build`	This command processes the contents of the compose file and creates the images required for the services it contains.
`docker-compose up`	This command creates the containers, networks, and volumes defined in the compose file and starts the containers.
`docker-compose stop`	This command stops the containers created from the services in the compose file. The containers, networks, and volumes are left in place so they can be started again.
`docker-compose down`	This command stops the containers created from the services in the compose file and removes them, along with the networks and volumes.
`docker-compose scale`	This command changes the number of containers that are running for a service.
`docker-compose ps`	This command lists the containers that have been created for the services defined in the compose file.

Docker Swarm Quick Reference

A Docker swarm is a cluster of servers that run containers. There are worker nodes that run the containers and manager nodes that determine which containers run on individual nodes and ensure that the right number of containers are running for each service. Swarms automatically try to recover when containers or nodes fail. Docker swarms are described in Chapter 7.

A swarm is created by running the following command on a manager node:

```
docker swarm init
```

The output from this command includes instructions for setting up the worker nodes, which are configured using the `docker swarm join` command.

Services can be created manually or described using a compose file. Here is an example of a compose file that includes instructions for deployment into a swarm, taken from Chapter 7:

```
version: "3"

volumes:
  productdata:

networks:
  backend:

services:

  mysql:
    image: "mysql:8.0.0"
    volumes:
      - productdata:/var/lib/mysql
    networks:
      - backend
    environment:
      - MYSQL_ROOT_PASSWORD=mysecret
      - bind-address=0.0.0.0
    deploy:
        replicas: 1
        placement:
            constraints:
            - node.hostname == dbhost

  mvc:
    image: "apress/exampleapp:swarm-1.0"
    networks:
      - backend
    environment:
      - DBHOST=mysql
    ports:
      - 3000:80
    deploy:
        replicas: 5
        placement:
            constraints:
            - node.labels.type == mvc
```

The `deploy` keyword denotes the configuration section for deploying the services into the swarm. Table 2-10 describes the keywords used in the compose file.

Table 2-10. *The Docker Compose Keywords for Swarms*

Keyword	Description
`replicas`	This setting specifies how many instances of a container are required for a service.
`placement`	This configuration section configures the placement of the containers for the service.
`constraints`	This setting specifies the constraints for locating the containers in the swarm.

Applications that are described using a compose file are deployed using the `docker stack deploy` command, like this:

```
docker stack deploy --compose-file docker-compose-swarm.yml exampleapp
```

The final argument to this command is used as a prefix applied to the names of the containers, networks, and volumes that are created in the swarm.

Table 2-11 lists the essential commands for working with Docker swarms.

Table 2-11. *Essential Commands for Docker Swarms*

Command	Description
`docker swarm init`	This command runs on manager nodes to create a swarm.
`docker swarm join`	This command runs on worker nodes to join a swarm.
`docker node ls`	This command displays a list of the nodes in the swarm.
`docker node update`	This command changes the configuration of a node in the swarm.
`docker service create`	This command manually starts a new service on the swarm.
`docker service update`	This command changes the configuration of a service running on the swarm.
`docker service scale`	This command changes the number of containers that are running for a specific service.
`docker service ls`	This command lists the services that are running on the swarm.
`docker service ps`	This command lists the containers that are running for a specific service.
`docker service rm`	This command removes a service from the swarm.
`docker stack deploy`	This command deploys an application described in a compose file to the swarm.
`docker stack rm`	This command removes the services described in a compose file from the swarm.

Summary

This chapter provided a quick reference for the Docker features described in the rest of the book. In the next chapter, I show you how to get set up for working with Docker and create the example project that will be used throughout the rest of the book.