Prerequisites

Work through the orientation and setup in Part 1 : Orientation and setup.

Introduction

Now that you've set up your development environment, thanks to Docker Desktop, you can begin to develop containerized applications. In general, the development workflow looks like this:

1. Create and test individual containers for each component of your application by first creating Docker images.
2. Assemble your containers and supporting infrastructure into a complete application.
3. Test, share, and deploy your complete containerized application.

In this stage of the tutorial, let's focus on step 1 of this workflow: creating the images that your containers will be based on. Remember, a Docker image captures the private filesystem that your containerized processes will run in; you need to create an image that contains just what your application needs to run.

> **Containerized development environments** are easier to set up than traditional development environments, once you learn how to build images as we'll discuss below. This is because a containerized development environment will isolate all the dependencies your app needs inside your Docker image; there's no need to install anything other than Docker on your development machine. In this way, you can easily develop applications for different stacks without changing anything on your development machine.

Set up

Let us download an example project from the Docker Samples page.

- Git
- Windows (without Git)
- Mac or Linux (without Git)

Git

If you are using Git, you can clone the example project from GitHub:

```
git clone https://github.com/dockersamples/node-bulletin-board
cd node-bulletin-board/bulletin-board-app
```

Windows (without Git)

If you are using a Windows machine and prefer to download the example project without installing Git, run the following commands in PowerShell:

```
curl.exe -LO https://github.com/dockersamples/node-bulletin-
board/archive/master.zip
```

```
tar.exe xf master.zip
cd node-bulletin-board-master\bulletin-board-app
```

Mac or Linux (without Git)

If you are using a Mac or a Linux machine and prefer to download the example project without installing Git, run the following commands in a terminal:

```
curl -LO https://github.com/dockersamples/node-bulletin-board/archive/master.zip
unzip master.zip
cd node-bulletin-board-master/bulletin-board-app
```

The `node-bulletin-board` project is a simple bulletin board application, written in Node.js. In this example, let's imagine you wrote this app, and are now trying to containerize it.

Define a container with Dockerfile

Take a look at the file called `Dockerfile` in the bulletin board application. Dockerfiles describe how to assemble a private filesystem for a container, and can also contain some metadata describing how to run a container based on this image. The bulletin board app Dockerfile looks like this:

```
# Use the official image as a parent image.
FROM node:current-slim

# Set the working directory.
WORKDIR /usr/src/app

# Copy the file from your host to your current location.
COPY package.json .

# Run the command inside your image filesystem.
RUN npm install

# Inform Docker that the container is listening on the specified port at runtime.
EXPOSE 8080

# Run the specified command within the container.
CMD [ "npm", "start" ]

# Copy the rest of your app's source code from your host to your image filesystem.
COPY . .
```

Writing a Dockerfile is the first step to containerizing an application. You can think of these Dockerfile commands as a step-by-step recipe on how to build up your image. This one takes the following steps:

- Start `FROM` the pre-existing `node:current-slim` image. This is an *official image*, built by the node.js vendors and validated by Docker to be a high-quality image containing the Node.js Long Term Support

(LTS) interpreter and basic dependencies.
- Use `WORKDIR` to specify that all subsequent actions should be taken from the directory `/usr/src/app` *in your image filesystem* (never the host's filesystem).
- `COPY` the file `package.json` from your host to the present location (`.`) in your image (so in this case, to `/usr/src/app/package.json`)
- `RUN` the command `npm install` inside your image filesystem (which will read `package.json` to determine your app's node dependencies, and install them)
- `COPY` in the rest of your app's source code from your host to your image filesystem.

You can see that these are much the same steps you might have taken to set up and install your app on your host. However, capturing these as a Dockerfile allows you to do the same thing inside a portable, isolated Docker image.

The steps above built up the filesystem of our image, but there are other lines in your Dockerfile.

The `CMD` directive is the first example of specifying some metadata in your image that describes how to run a container based on this image. In this case, it's saying that the containerized process that this image is meant to support is `npm start`.

The `EXPOSE 8080` informs Docker that the container is listening on port 8080 at runtime.

What you see above is a good way to organize a simple Dockerfile; always start with a `FROM` command, follow it with the steps to build up your private filesystem, and conclude with any metadata specifications. There are many more Dockerfile directives than just the few you see above. For a complete list, see the Dockerfile reference.

Build and test your image

Now that you have some source code and a Dockerfile, it's time to build your first image, and make sure the containers launched from it work as expected.

> **Windows users**: this example uses Linux containers. Make sure your environment is running Linux containers by right-clicking on the Docker logo in your system tray, and clicking **Switch to Linux containers** if the option appears. Don't worry - all the commands in this tutorial work the exact same way for Windows containers.

Make sure you're in the directory `node-bulletin-board/bulletin-board-app` in a terminal or PowerShell using the `cd` command. Let's build your bulletin board image:

```
docker build --tag bulletinboard:1.0 .
```

You'll see Docker step through each instruction in your Dockerfile, building up your image as it goes. If successful, the build process should end with a message `Successfully tagged bulletinboard:1.0`.

> **Windows users:** you may receive a message titled 'SECURITY WARNING' at this step, noting the read, write, and execute permissions being set for files added to your image. We aren't handling any sensitive information in this example, so feel free to disregard the warning in this example.

Run your image as a container

1. Start a container based on your new image:

   ```
   docker run --publish 8000:8080 --detach --name bb bulletinboard:1.0
   ```

   There are a couple of common flags here:

   - `--publish` asks Docker to forward traffic incoming on the host's port 8000, to the container's port 8080. Containers have their own private set of ports, so if you want to reach one from the network, you have to forward traffic to it in this way. Otherwise, firewall rules will prevent all network traffic from reaching your container, as a default security posture.
   - `--detach` asks Docker to run this container in the background.
   - `--name` specifies a name with which you can refer to your container in subsequent commands, in this case `bb`.

   Also notice, you didn't specify what process you wanted your container to run. You didn't have to, as you've used the `CMD` directive when building your Dockerfile; thanks to this, Docker knows to automatically run the process `npm start` inside the container when it starts up.

2. Visit your application in a browser at `localhost:8000`. You should see your bulletin board application up and running. At this step, you would normally do everything you could to ensure your container works the way you expected; now would be the time to run unit tests, for example.

3. Once you're satisfied that your bulletin board container works correctly, you can delete it:

   ```
   docker rm --force bb
   ```

   The `--force` option removes the running container. If you stop the container running with `docker stop bb` you do not need to use `--force`.

Conclusion

At this point, you've successfully built an image, performed a simple containerization of an application, and confirmed that your app runs successfully in its container. The next step will be to share your images on Docker Hub, so they can be easily downloaded and run on any destination machine.