

This scenario explains how to build an image based on your requirements.

For this scenario, the container will be running a static HTML application using Nginx, a high-performance web server. In the future, we'll explain how to deploy other stacks such as Node.js or ASP.NET.

The machine name Docker is running on is called docker. If you want to access any of the services, then use docker instead of localhost or 0.0.0.0.

### Docker Images

Docker images are built based on a Dockerfile. A Dockerfile defines all the steps required to create a Docker image with your application configured and ready to be run as a container. The image itself contains everything, from operating system to dependencies and configuration required to run your application.

Having everything within the image allows you to migrate images between different environments and be confident that if it works in one environment, then it will work in another.

The Dockerfile allows for images to be composable, enabling users to extend existing images instead of building from scratch. By building on an existing image, you only need to define the steps to setup your application. The base images can be basic operating system installations or configured systems which simply need some additional customisations.

To help you complete the steps, an environment has been created with Docker configured. The editor allows you to write a Dockerfile which defines how to build the Docker image.

### Step 1 - Base Images

All Docker images start from a base image. A base image is the same images from the Docker Registry which are used to start containers. Along with the image name, we can also include the image tag to indicate which particular version we want, by default, this is latest.

These base images are used as the foundation for your additional changes to run your application. For example, in this scenario, we require NGINX to be configured and running on the system before we can deploy our static HTML files. As such we want to use NGINX as our base image.

Dockerfile's are simple text files with a command on each line. To define a base image we use the instruction `FROM <image-name>:<tag>`

### Task: Creating a Dockerfile

The first line of the Dockerfile should be `FROM nginx:1.11-alpine`

Make the change in the Dockerfile editor. Within the environment, a new Dockerfile will be created with the contents of the editor.

### Caution

It's tempting to use the tag `:latest` however this can result in you building your image against a version which you were not expecting. We recommend that you always use a particular version number as your tag and manage the updating yourself.

### Step 2 - Running Commands

With the base image defined, we need to run various commands to configure our image. There are many commands to help with this, the main commands two are `COPY` and `RUN`.

`RUN <command>` allows you to execute any command as you would at a command prompt, for example installing different application packages or running a build command. The results of the `RUN` are persisted to the image so it's important not to leave any unnecessary or temporary files on the disk as these will be included in the image.

`COPY <src> <dest>` allows you to copy files from the directory containing the Dockerfile to the container's image. This is extremely useful for source code and assets that you want to be deployed inside your container.

#### Task

A new `index.html` file has been created for you which we want to serve from our container. On the next line after the `FROM` command, use the `COPY` command to copy `index.html` into a directory called `/usr/share/nginx/html`

#### Protip

If you're copying a file into a directory then you need to specify the filename as part of the destination.

### Step 3 - Exposing Ports

With our files copied into our image and any dependencies downloaded, you need to define which port application needs to be accessible on.

Using the `EXPOSE <port>` command you tell Docker which ports should be open and can be bound to. You can define multiple ports on the single command, for example, `EXPOSE 80 433` or `EXPOSE 7000-8000`

#### Task

We want our web server to be accessible via port 80, add the relevant `EXPOSE` line to the Dockerfile.

### Step 4 - Default Commands

With the Docker image configured and having defined which ports we want accessible, we now need to define the command that launches the application.

The `CMD` line in a Dockerfile defines the default command to run when a container is launched. If the command requires arguments then it's recommended to use an array, for example `["cmd", "-a", "arga value", "-b", "argb-value"]`, which will be combined together and the command `cmd -a "arga value" -b argb-value` would be run.

#### Task

The command to run NGINX is `nginx -g daemon off;`. Set this as the default command in the Dockerfile.

#### Protip

An alternative approach to `CMD` is `ENTRYPOINT`. While a `CMD` can be overridden when the container starts, a `ENTRYPOINT` defines a command which can have arguments passed to it when the container launches.

In this example, NGINX would be the entrypoint with `-g daemon off;` the default command.

### Step 5 - Building Containers

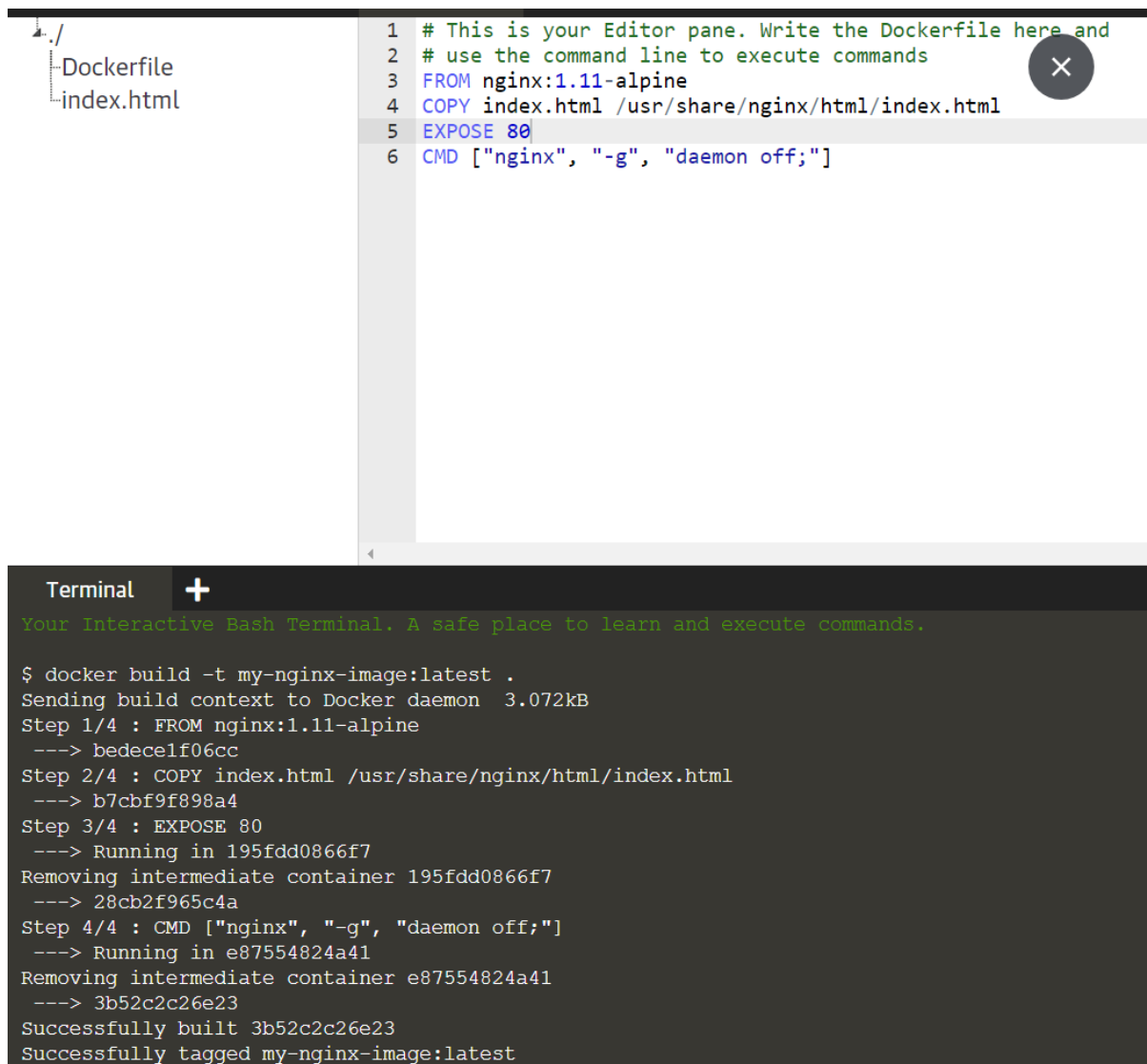
After writing your Dockerfile you need to use `docker build` to turn it into an image. The build command takes in a directory containing the Dockerfile, executes the steps and stores the image in your local Docker Engine. If one fails because of an error then the build stops.

#### Task

Using the `docker build` command to build the image. You can give the image a friendly name by using the `-t <name>` option.

#### Protip

You can use `docker images` to see a list of the images on your local machine.



The image shows a Docker IDE interface. On the left, a file explorer shows a directory with files `Dockerfile` and `index.html`. The main editor pane displays the `Dockerfile` content:

```
1 # This is your Editor pane. Write the Dockerfile here and
2 # use the command line to execute commands
3 FROM nginx:1.11-alpine
4 COPY index.html /usr/share/nginx/html/index.html
5 EXPOSE 80
6 CMD ["nginx", "-g", "daemon off;"]
```

Below the editor is a terminal window titled "Terminal" with a plus icon. It contains the following output:

```
Your Interactive Bash Terminal. A safe place to learn and execute commands.

$ docker build -t my-nginx-image:latest .
Sending build context to Docker daemon 3.072kB
Step 1/4 : FROM nginx:1.11-alpine
----> bdece1f06cc
Step 2/4 : COPY index.html /usr/share/nginx/html/index.html
----> b7cbf9f898a4
Step 3/4 : EXPOSE 80
----> Running in 195fdd0866f7
Removing intermediate container 195fdd0866f7
----> 28cb2f965c4a
Step 4/4 : CMD ["nginx", "-g", "daemon off;"]
----> Running in e87554824a41
Removing intermediate container e87554824a41
----> 3b52c2c26e23
Successfully built 3b52c2c26e23
Successfully tagged my-nginx-image:latest
```

## Step 6 - Launching New Image

With the image successfully created, you can now launch the container in the same way we described in the first scenario.

### Task

Launch an instance of your newly built image using either the ID result from the build command or the friendly name you assigned it.

NGINX is designed to run as a background service so you should include the option `-d`. To make the web server accessible, bind it to port 80 using `p 80:80`

For example:

```
docker run -d -p 80:80 <image-id|friendly-tag-name>
```

You can access the launched web server via the hostname `docker`. After launching the container, the command `curl -i http://docker` will return our index file via NGINX and the image we built.

### Protip

You can check the container is running using `docker ps`

```
Successfully tagged my-nginx-image:latest
$ docker run -d -p 80:80 my-nginx-image:latest
7e842588a920294e7453fe3f3b156fe9a8e82f83bae89a268417723b84dde0d0
$ docker ps
CONTAINER ID          IMAGE                  COMMAND
7e842588a920          my-nginx-image:latest  "nginx -g 'daemon of..."
$ curl -i http://docker
HTTP/1.1 200 OK
Server: nginx/1.11.13
Date: Mon, 29 May 2023 07:51:58 GMT
Content-Type: text/html
Content-Length: 21
Last-Modified: Mon, 29 May 2023 07:43:13 GMT
Connection: keep-alive
ETag: "64745791-15"
Accept-Ranges: bytes

<h1>Hello World</h1>
$
```

In this scenario we covered how to write a Dockerfile to define how to build and configure your image to run as a Docker container. A Dockerfile is key to creating repeatable images that can define how your applications are configured and deployed in development and production.

In future scenarios will explore how you can use Dockerfile's to build images to deploy applications based on Node.JS, ASP.NET and more.