**Dockers and Containers**

**Scenario 1:**

**Deploying Your First Docker Container**

In this first scenario, you'll take the role of Jane, a developer who needs to deploy a new Key-Value Store for an application she's working with. After discussions, it's been decided to use Redis, a popular KV Store.

Jane is unfamiliar with how Redis is deployed but has heard Docker makes it straightforward to deploy services into development and production.

This scenario discusses how she will complete her task and deploy Redis as a Docker Container.

Jane's development environment has access to latest version of the Docker Engine via a machine called docker. Her local dev machine has the Docker Client installed and accessible via the command line.

**What Is Docker?**

Docker describes themselves as "an open platform for developers and sysadmins to build, ship, and run distributed applications".

Docker allows you to run containers. A container is a sandboxed process running an application and its dependencies on the host operating system. The application inside the container considers itself to be the only process running on the machine while the machine can run multiple containers independently.

**Step 1: Running A Container**

The first task is to identify the name of the Docker Image which is configured to run Redis. With Docker, all containers are started based on a Docker Image. These images contain everything required to launch the process; the host doesn't require any configuration or dependencies.

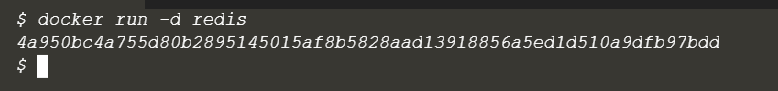
Jane can find existing images at [registry.hub.docker.com/](https://registry.hub.docker.com/) or by using the command docker search <name>. For example, to find an image for Redis, you would use

docker search redis

The Docker CLI has a command called run which will start a container based on a Docker Image. The structure is docker run <options> <image-name>.

By default, Docker will run a command in the foreground. To run in the background, the option -d needs to be specified.

docker run -d redis



By default, Docker will run the latest version available. If a particular version was required, it could be specified as a tag, for example, version 3.2 would be

docker run -d redis:3.2.

docker run -d redis:latest

**Step 2 - Finding Running Containers**

The launched container is running in the background, the docker ps command lists all running containers, the image used to start the container and uptime.

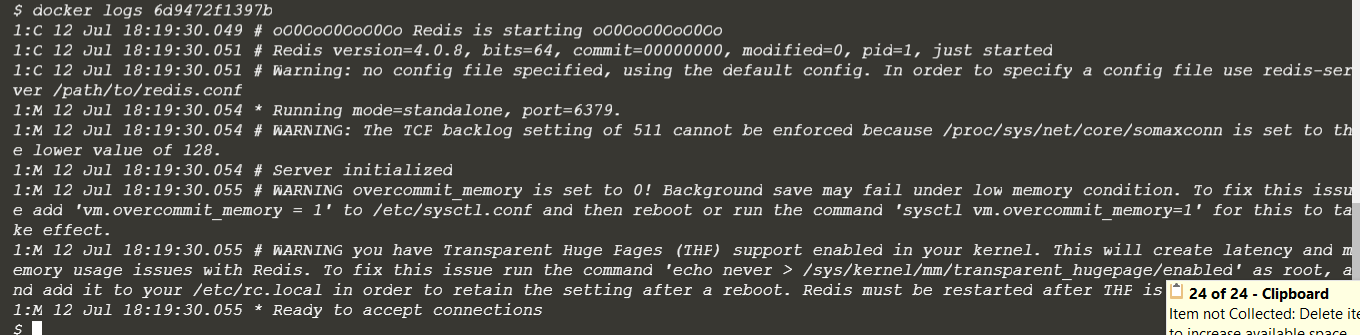


This command also displays the friendly name and ID that can be used to find out information about individual containers.

The command docker inspect <friendly-name|container-id> provides more details about a running container, such as IP address.



The command docker logs <friendly-name|container-id> will display messages the container has written to standard error or standard out.



**Step 3 - Accessing Redis**

Jane is happy that Redis is running, but is surprised that she cannot access it. The reason is that each container is sandboxed. If a service needs to be accessible by a process not running in a container, then the port needs to be exposed via the Host.

Once exposed, it is possible to access the process as if it were running on the host OS itself.

Jane knows that by default, *Redis* runs on port *6379*. She has learned that by default other applications and library expect a Redis instance to be listening on the port.

**Task**

After reading the documentation, Jane discovers that ports are bound when containers are started using -p <host-port>:<container-port> option. Jane also discovers that it's useful to define a name when starting the container, this means she doesn't have to use Bash piping or keep looking up the name when trying to access the logs.

Jane finds the best way to solve her problem of running Redis in the background, with a name of redisHostPort on port 6379 is using the following command

 docker run -d --name redisHostPort -p 6379:6379 redis:latest

**Protip**

By default, the port on the host is mapped to 0.0.0.0, which means all IP addresses. You can specify a particular IP address when you define the port mapping, for example, -p 127.0.0.1:6379:6379

**Step 4 - Accessing Redis**

The problem with running processes on a fixed port is that you can only run one instance. Jane would prefer to run multiple Redis instances and configure the application depending on which port Redis is running on.

**Task**

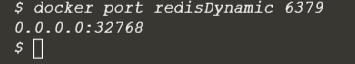
After experimenting, Jane discovers that just using the option -p 6379 enables her to expose Redis but on a randomly available port. She decides to test her theory using

 docker run -d --name redisDynamic -p 6379 redis:latest



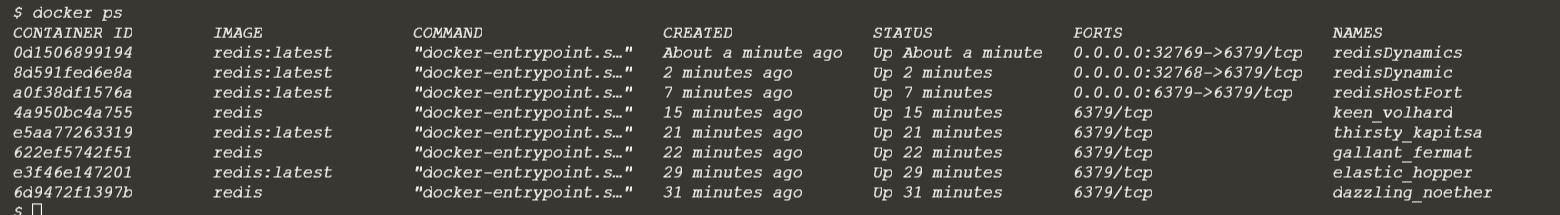
While this works, she now doesn't know which port has been assigned. Thankfully, this is discovered via

docker port redisDynamic 6379



Jane also finds that listing the containers displays the port mapping information,

 docker ps



**Step 5 - Persisting Data**

After working with containers for a few days, Jane realises that the data stored keeps being removed when she deletes and re-creates a container. Jane needs the data to be persisted and reused when she recreates a container.

Containers are designed to be stateless. Binding directories (also known as volumes) is done using the option -v <host-dir>:<container-dir>. When a directory is mounted, the files which exist in that directory on the host can be accessed by the container and any data changed/written to the directory inside the container will be stored on the host. This allows you to upgrade or change containers without losing your data.

**Task**

Using the Docker Hub documentation for [Redis](https://hub.docker.com/_/redis/), Jane has investigated that the official Redis image stores logs and data into a /data directory.

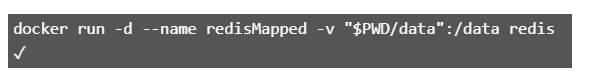
Any data which needs to be saved on the Docker Host, and not inside containers, should be stored in /opt/docker/data/redis.

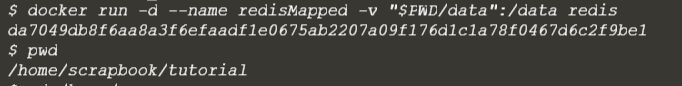
The complete command to solve the task is

 docker run -d --name redisMapped -v /opt/docker/data/redis:/data redis

**Protip**

Docker allows you to use $PWD as a placeholder for the current directory.





**Step 6 - Running A Container In The Foreground**

Jane has been working with Redis as a background process. Jane wonders how containers work with foreground processes, such as ps or bash.

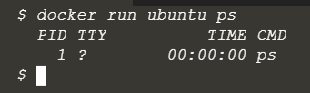
Previously, Jane used the -d to execute the container in a detached, background, state. Without specifying this, the container would run in the foreground. If Jane wanted to interact with the container (for example, to access a bash shell) she could include the options -it.

As well as defining whether the container runs in the background or foreground, certain images allow you to override the command used to launch the image. Being able to replace the default command makes it possible to have a single image that can be re-purposed in multiple ways. For example, the Ubuntu image can either run OS commands or run an interactive bash prompt using /bin/bash

#### Example

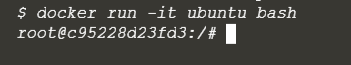
The command launches an Ubuntu container and executes the command ps to view all the processes running in a container.

 docker run ubuntu ps



Using allows Jane to get access to a bash shell inside of a container.

 docker run -it ubuntu bash



In this scenario, Jane had the requirement of running Redis. Jane decided to run this as a Docker Container to make it easier to manage. The commands Jane discovered are the cornerstone of running Docker in both development and production environments.

In the next scenario, Jane will look into how it's possible to build her own Docker Image.

**Scenario 2:**

**Deploy Static HTML Website as Container**

**Step 1 - Create Dockerfile**

Docker Images start from a base image. The base image should include the platform dependencies required by your application, for example, having the JVM or CLR installed.

This base image is defined as an instruction in the Dockerfile. Docker Images are built based on the contents of a Dockerfile. The Dockerfile is a list of instructions describing how to deploy your application.

In this example, our base image is the Alpine version of Nginx. This provides the configured web server on the Linux Alpine distribution.

**Task**

Create your Dockerfile for building your image by copying the contents below into the editor.

FROM nginx:alpine

COPY . /usr/share/nginx/html

The first line defines our base image. The second line copies the content of the current directory into a particular location inside the container.

**Step 2 - Build Docker Image**

The Dockerfile is used by the Docker CLI build command. The build command executes each instruction within the Dockerfile. The result is a built Docker Image that can be launched and run your configured app.

The build command takes in some different parameters. The format is

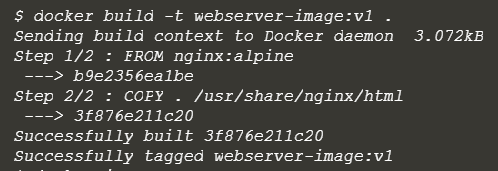
docker build -t <build-directory>.

The -t parameter allows you to specify a friendly name for the image and a tag, commonly used as a version number. This allows you to track built images and be confident about which version is being started.

**Task**

Build our static HTML image using the build command below.

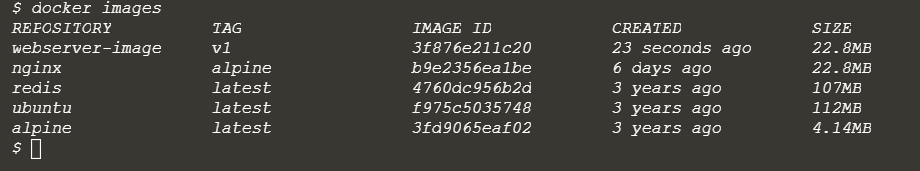
docker build -t webserver-image:v1



You can view a list of all the images on the host using

 docker images.

The built image will have the name webserver-image with a tag of v1.



**Step 3 - Run**

The built Image can be launched in a consistent way to other Docker Images. When a container launches, it's sandboxed from other processes and networks on the host. When starting a container you need to give it permission and access to what it requires.

For example, to open and bind to a network port on the host you need to provide the parameter

-p <host-port>:<container-port>.

**Task**

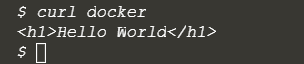
Launch our newly built image providing the friendly name and tag. As it's a web server, bind port 80 to our host using the -p parameter.

docker run -d -p 80:80 webserver-image:v1



Once started, you'll be able to access the results of port 80 via

 curl docker



To render the requests in the browser use the following links

<https://2886795312-80-elsy04.environments.katacoda.com/>

You now have a static HTML website being served by Nginx.

**Scenario 3:**

**Building Container Images**

In the first scenario, we discussed how you could start containers based on pre-existing images from the Docker Registry. 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

**FROM nginx:1.11-alpine**

**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

**COPY index.html /usr/share/nginx/html/index.html**

**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.

**EXPOSE 80**

**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.

**CMD ["nginx", "-g", "daemon off;"]**

**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.

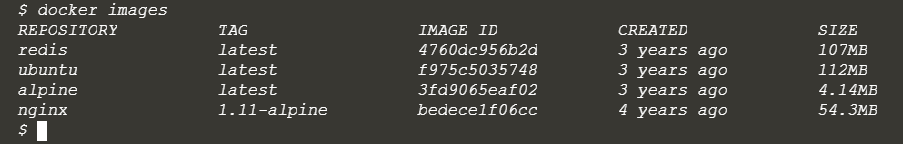
**Task**

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 below to see a list of the images on your local machine.

docker images



Now run the build command

Save this file as Dockerfile index.html

# This is your Editor pane. Write the Dockerfile here and

# use the command line to execute commands

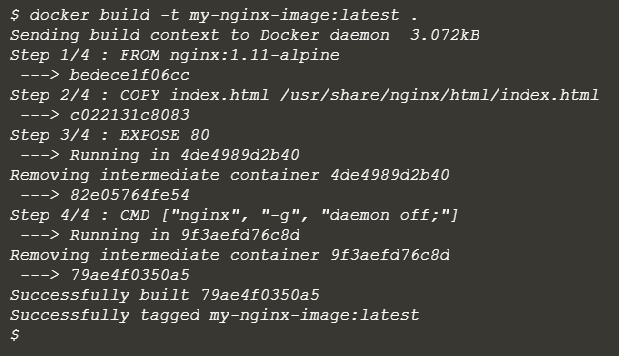
FROM nginx:1.11-alpine

COPY index.html /usr/share/nginx/html/index.html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]





**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:

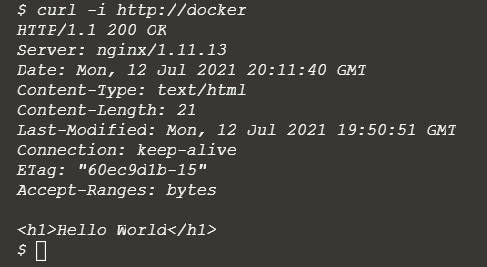
**docker run -d -p 80:80 <image-id|friendly-tag-name>**



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

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.

**Scenario 4:**

**Dockerizing Node.js applications**

This scenario continues to explore how to build and deploy your applications as a Docker container. The previous scenario covered deploying a static HTML website. This scenario explores how to deploy a Node.js application within a container.

The environment is configured with access to a personal Docker instance, and the code for a default Expressjs application is in the working directory. To view the code use ls and cat <filename> or use the editor.

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.

**Step 1 - Base Image**

As we described in the previous scenario, all images started with a base image, ideally as close to your desired configuration as possible. Node.js has pre-built images available with tags for each released version.

The image for Node 10.0 is node:10-alpine. This is an Alpine-based build which is smaller and more streamlined than the official image.

Alongside the base image, we also need to create the base directories of where the application runs from. Using the RUN <command> we can execute commands as if they're running from a command shell, by using mkdir we can create the directories where the application will execute from. In this case, an ideal directory would be /src/app as the environment user has read/write access to this directory.

We can define a working directory using WORKDIR <directory> to ensure that all future commands are executed from the directory relative to our application.

**Task: Define Base Environment**

Set the FROM <image>:<tag>, RUN <command> and WORKDIR <directory> on separate lines to configure the base environment for deploying your application

FROM node:10-alpine

RUN mkdir -p /src/app

WORKDIR /src/app

**Step 2 - NPM Install**

In the previous set, we configured the foundation of our configuration and how we want the application to be deployed. The next stage is to install the dependencies required to run the application. For Node.js this means running NPM install.

To keep build times to a minimum, Docker caches the results of executing a line in the Dockerfile for use in a future build. If something has changed, then Docker will invalidate the current and all following lines to ensure everything is up-to-date.

With NPM we only want to re-run npm install if something within our package.json file has changed. If nothing has changed then we can use the cache version to speed up deployment. By using COPY package.json <dest> we can cause the RUN npm install command to be invalidated if the package.json file has changed. If the file has not changed, then the cache will not be invalidated, and the cached results of the npm install command will be used.

**Task: Add Dockerfile Lines**

The following two lines are required in order Dockerfile to run npm install.

Copy the lines to the Dockerfile now so they can be used in the build later.

**Protip**

If you don't want to use the cache as part of the build then set the option --no-cache=true as part of the docker build command.

COPY package.json /src/app/package.json

RUN npm install

**Step 3 - Configuring Application**

After we've installed our dependencies, we want to copy over the rest of our application's source code. Splitting the installation of the dependencies and copying out source code enables us to use the cache when required.

If we copied our code before running npm install then it would run every time as our code would have changed. By copying just package.json we can be sure that the cache is invalidated only when our package contents have changed.

**Task: Deploy Application**

Create the desired steps in the Dockerfile to finish the deployment of the application.

We can copy the entire directory where our Dockerfile is using COPY . <dest dir>.

Once the source code has been copied, the ports the application requires to be accessed is defined using EXPOSE <port>.

Finally, the application needs to be started. One neat trick when using Node.js is to use the npm start command. This looks in the package.json file to know how to launch the application saving duplication of commands.

In the next step, we'll build and launch the image

COPY . /src/app

EXPOSE 3000

CMD [ "npm", "start" ]

**Step 4 - Building & Launching Container**

To launch your application inside the container you first need to build an image.

**Example: Build & Launch**

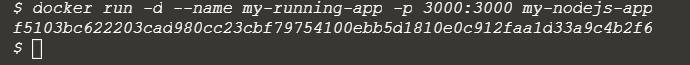
The command to build the image is

 docker build -t my-nodejs-app



The command to launch the built image is

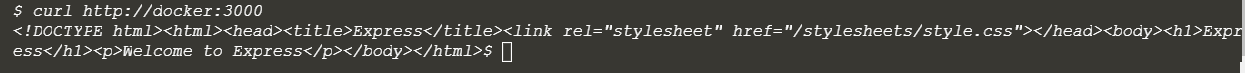
docker run -d --name my-running-app -p 3000:3000 my-nodejs-app



**Testing Container**

You can test the container is accessible using curl. If the application responds then you know that everything has correctly started.

curl http://docker:3000



**Step 5 - Environment Variables**

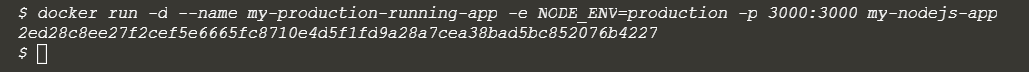
Docker images should be designed that they can be transferred from one environment to the other without making any changes or requiring to be rebuilt. By following this pattern you can be confident that if it works in one environment, such as staging, then it will work in another, such as production.

With Docker, environment variables can be defined when you launch the container. For example with Node.js applications, you should define an environment variable for NODE\_ENV when running in production.

Using -e option, you can set the name and value as -e NODE\_ENV=production

**Example**

docker run -d --name my-production-running-app -e NODE\_ENV=production -p 3000:3000 my-nodejs-app



Save this file as Dockerfile for Node.js

FROM node:10-alpine

RUN mkdir -p /src/app

WORKDIR /src/app

COPY package.json /src/app/package.json

RUN npm install

COPY . /src/app

EXPOSE 3000

CMD [ "npm", "start" ]

**Scenario 5:**

**Optimising Dockerfile with OnBuild**

In this scenario, we'll look at how you can optimise Dockerfile using the OnBuild instruction.

The environment has been configured with an example Node.js application however the approaches can be applied to any image. 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.

**Step 1 - OnBuild**

While Dockerfile's are executed in order from top to bottom, you can trigger an instruction to be executed at a later time when the image is used as the base for another image.

The result is you can delay your execution to be dependent on the application which you're building, for example the application's package.json file.

Below is the [Node.js OnBuild Dockerfile](https://github.com/docker-library/node/blob/70741d88bf688389bfac7b147573f3b761f9ede9/0.10/onbuild/Dockerfile). Unlike in our previous scenario the application specify commands have been prefixed with ONBUILD.

FROM node:7

RUN mkdir -p /usr/src/app

WORKDIR /usr/src/app

ONBUILD COPY package.json /usr/src/app/

ONBUILD RUN npm install

ONBUILD COPY . /usr/src/app

CMD [ "npm", "start" ]

The result is that we can build this image but the application specific commands won't be executed until the built image is used as a base image. They'll then be executed as part of the base image's build.

We'll see how this base image looks in the next step.

**Step 2 - Application Dockerfile**

With all of the logic to copy the code, install our dependencies and launch our application the only aspect which needs to be defined on the application level is which port(s) to expose.

The advantage of creating OnBuild images is that our Dockerfile is now much simpler and can be easily re-used across multiple projects without having to re-run the same steps improving build times.

FROM node:7-onbuild

EXPOSE 3000

This will be created in the environment for you. The steps to build and launch the Dockerfile are covered in the next step.

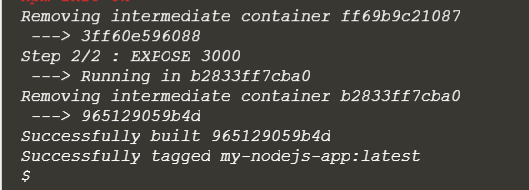
**Step 3 - Building & Launching Container**

The Dockerfile from the previous step has been created for you. Building the images based on the OnBuild docker file is the same as before. The OnBuild commands will be executed as if they were in the base Dockerfile.

**Example: Build & Launch**

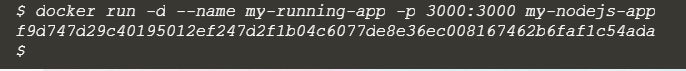
The command to build the image is

 docker build -t my-nodejs-app



The command to launch the built image is

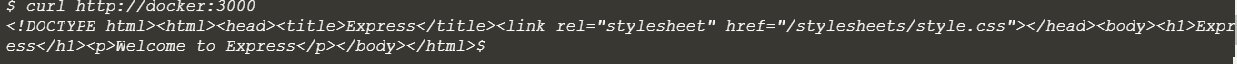
 docker run -d --name my-running-app -p 3000:3000 my-nodejs-app



**Testing Container**

You can test the container is accessible using curl. If the application responds when you know that everything has correctly started.

curl http://docker:3000



.