

# FE - Docker Containers and Micro-services

This page serves as documentation on Docker and how to implement it with a React application in a Micro-services type of architecture.

## Docker Installation Instructions:

**NOTE:** You will have to create a docker account in order to download and use the local docker application on your machine.

- Docker for Mac OSX: [Docker Desktop on Mac](#)
- Docker for Windows: [Docker Desktop on Windows](#)

## TLDR; Docker Workflow Process:

### Create Application Directory:

- Create application directory via create-react-app or manually

### Create Dockerfile:

- DEV:
  - Create a file called 'dockerfile.dev' in the root of the application project directory
  - Example File Contents:

```
FROM node:alpine

WORKDIR '/app'

COPY package.json .
RUN npm install
COPY . .

CMD [ "npm", "run", "start" ]
```

- PROD:
  - Create a file called 'dockerfile' in the root of the application project directory
  - Example File Contents (**TBD**)

### Create docker-compose.yml file:

- **TBD**

### Commands to run tests:

- Create a new build via **Option 1 (preferred method)** in Testing section that does NOT use a separate 'tests' service in the docker-compose.yml file:
  - Example docker-compose.yml file:

```

• version: '3'
  services:
    web:
      build:
        context: . # Specifies where we want all files
        and folders to be pulled from
        dockerfile: Dockerfile.dev # Name of
        dockerfile to be used to build and run container
      ports:
        - "3000:3000"
      volumes:
        - /app/node_modules
        - ./app

```

- Open a terminal session and type 'docker-compose up --build'
- Open another terminal session and run 'docker ps' get container id run 'docker exec -it <container id> npm/yarn run test' to run interactive test runner in this terminal session window

#### Build Production Version of Container:

- TBD

#### Docker CLI Commands:

	Description	Command	Example
1	Create and run an image and container(s)	docker run	docker run hello-world
2	Show all Containers as list (current and not running)	docker ps --all	docker ps --all
3	Build and run a container	docker build .	docker build .
4	Build and run a container from a <b>custom</b> Dockerfile (like running a dev version of the Dockerfile named ' <b>Dockerfile.dev</b> ')	docker build -f <custom file name> .	docker build -f Dockerfile.dev .
5	Show running containers	docker ps	docker ps
6	Restarting stopped containers ('-a' flag shows the output of the container)	docker start -a <container id>	docker start a98qu49q6r87q34
7	Remove stopped containers	docker system prune	docker system prune
8	Retrieve log outputs	docker logs <container id>	docker logs 9847r7969fas678
9	Stopping containers (stops but allows cleanup, saving, etc. before killing the container) - <b>NOTE</b> : After about 10 seconds, it will automatically run 'docker kill' and automatically kill the container if it hasn't stopped by then.	docker stop <container id>	docker stop 2348768sd786swa5
10	Killing containers (immediately kills container without cleanup or save opportunity)	docker kill <container id>	docker kill 34987faaf78689ag67
11	Execute Commands in Running Containers ( <b>NOTE</b> : '-it' flag just allows for input and info to be displayed back)	docker exec -it <container id> <command to execute in running container>	docker exec -it 20d98a591d80 redis-cli
12	Accessing a CMD Prompt in a Container via 'exec'	docker exec -it <container id> sh	docker exec -it fss976af8a72jkaf sh

13	Starting with a shell (NOTE: this blocks any other process, but is good to be able to initially look into things on the image)...type "exit" or hit 'ctrl +d' to get out of the shell session	docker run -it <image> sh	docker run -it busybox sh
14	Execute a container in the background and give back access to terminal	docker run -d <container id>	docker run -d busybox
15	Port Mapping (forwards localhost traffic into container)	docker run -p localhostPort:containerPort <image name>	docker run -p 8080:8080 oe/simpleweb
16	Specifying a Working Directory (to avoid adding new files to the root of the container)	WORKDIR /usr/app (in a linux-based OS, the 'usr' directory is a safe place to specify all user generated content and is ideal)	WORKDIR /usr/ app
17	How to forward input from a terminal directly to a running container ( <b>NOTE:</b> This only gives you a handle to the <b>PRIMARY</b> process and not any subsequent processes that you may want to hook into!!!)	docker attach <container id>	docker- attach <container id>
18	Docker Volumes to create directory mappings between container and local source code	docker run -p 3000:3000 -v /app/node_modules -v "\$(pwd)/:app" <image id>	docker run -p 3000:3000 -v /app/node_modules -v "\$(pwd)/:app" oe/simpleweb
19	Live Updating Tests by attaching to a running volumed container	docker exec -it <container id> npm/yarn run test	docker exec -it af08qeqja98a77 npm/yarn run test
20			
21			

## Docker Compose Commands:

	Description	Command	Example
1	How to start multiple Docker containers at the same time and also how keep from writing tons of repetitive commands with the Docker CLI	docker-compose	docker-compose
2	Run a docker image	docker-compose up	docker-compose up
3	Build and then run a docker image so that you have latest changes	docker-compose up --build	docker-compose up --build
4	Launch containers in background and give back terminal access	docker-compose up -d	docker-compose up -d
5	Stop docker containers with a single command	docker-compose down	docker-compose down
6			
7			

## How To Create A Docker Image:

- Building a **Dockerfile**:
  - This is a plain text file that will contain configurations for how our container behaves, specifically what it programs it contains and what it does as it starts up as a container and almost always follows this flow:
    - Specify a base image
    - Run some commands to install additional programs
    - Specify a command to run on container startup
  - Steps to build the Dockerfile:
    - Create a directory where you want the Dockerfile to reside on your local machine and create a file called '**Dockerfile**' without any extensions. Docker will pick this file name up by default.
      - NOTE:** It is possible to use a **custom dockerfile name**, most likely for development purposes like '**Dockerfile.dev**' for example.
        - To get docker to use this custom file rather than the default, you have to use the '**docker build -f <custom file name> .**' command
    - Open the Dockerfile in your text editor and add comments for what we'll need to do:
      - '# Use an existing docker image as a base'
      - '# Download and install a dependency;
      - '# Tell the image what to do when it starts as a container'
    - Example of #2 for a redis-server:

```

a. # Use an existing docker image as a base
FROM alpine

# Download and install a dependency
RUN apk add --update redis

# Tell the image what to do when it starts
# as a container
CMD ["redis-server"]

```

- Run Docker command in the CLI to build the Dockerfile:
  - From with directory that contains the Dockerfile, run '**docker build .**' (be sure to include the '.' at the end)
    - Running this command feeds the Dockerfile to the **Docker CLI (Docker Client)** which feeds it to the **Docker Server** which then builds a **Usable Image**
  - After it builds, it will display an id at the end. Copy this id and then run '**docker run <copied id>**' and the expected behavior of the container should work.
- Docker will automatically implement caching **only as long as the order of the operations previously used in the build have not changed** and only new operations are added at the end of the file. Changing these operations around will cause a complete rebuild of the image, thus taking longer when needing to add new things.
- Tagging an Image:
  - To avoid having to constantly copy/paste the id of an image when trying to run them in docker, you can create a custom tag to use instead by doing the following:
    - Based off the convention of **<'your docker id' / 'repo/project name' / 'version'>**, i.e., running '**docker build -t oe/redis:latest .**' will now allow you to run an image using the tag called 'oe/redis'. So you can now run it by typing '**docker run oe/redis**' rather than '**docker run afs98a8f67a1**'

### How to Resolve Errors and Base Image Issues:

Sometimes when using a base docker image, you will encounter errors such as "npm: not found", etc. (in this example where a NodeJS server is being used and has 'express' as a dependency, but 'alpine' was used as the base) and this is due to the base image being used in the Dockerfile not actually having a specific program installed on it, Node/NPM in this case, and the RUN command in our Dockerfile fails because it can't find 'npm' to run the 'npm install' command.

Options to resolve Dockerfile build errors are as follows:

1. Install another base image that does contain the programs you need in your image (NodeJS in this example):

```

# Use an existing docker image as a base
FROM alpine

# Download and install a dependency
RUN npm install

# Tell the image what to do when it starts
# as a container
CMD [ "npm", "start" ]

```

- a. Reference [docker hub](#) to search for a base that does include the programs you for sure will need.
- b. We can change our 'FROM alpine' in the docker file to be 'FROM node:alpine' to resolve the 'npm: not found' issue because Node contains NPM.

```
# Use an existing docker image as a base
FROM node:alpine

# Download and install a dependency
RUN npm install

# Tell the image what to do when it starts
# as a container
CMD [ "npm", "start" ]
```

- i. **NOTE:** 'alpine' is a term for a version of an image that is as small and compact as possible, meaning it won't contain a bunch of additional pre-installed programs.

1. 'node:alpine' is using the 'alpine' tag specified as available on the NodeJS official image in the docker hub.
2. Running 'docker build .' at this point will resolve the error, but now shows another commonly seen when doing an 'npm install' where it displays npm WARN messages and ENOENT errors that it can't find '/package.json', etc. (See Copying Build Files section below)

2. Manually install dependencies/programs into the image with multiple RUN commands.

### Copying Build files:

To resolve issues where files/directories are not found during the 'docker build .' process that exist in the local directory of your project, you must copy the files using the 'COPY' command, i.e., 'COPY ./ .' (**IMPORTANT:** These paths are relative to the current build context). The updated Dockerfile should look like this:

```
# Use an existing docker image as a base
FROM node:alpine

# Download and install a dependency
COPY ./ ./
RUN npm install

# Tell the image what to do when it starts
# as a container
CMD [ "npm", "start" ]
```

- Run the '**docker build -t oe/simpleweb .**' command to rebuild everything correctly and with a tag ('simpleweb', the demo project name for the example) for easy reference.
- Test the image by starting it with '**docker run oe/simpleweb**' and it should run correctly.

### Container Port Mapping:

When following the previous steps and then trying to open a browser to '<https://localhost:8080>' for example, you'll find that the page will not work or the 'connection refused' error is on the page. This is because the localhost network is different from the network found inside of the container and traffic from the localhost has to be routed to a port inside of the container. **NOTE:** The container can communicate with the outside world by default without any special instructions, but steps need to be taken to route traffic into the container.

- The solution is a runtime only command:
  - '**docker run -p 8080:8080 oe/webserver**' will now allow you to implement the '-p localhostPort:ContainerPort' flag to route the traffic appropriate to the container. Refreshing the page after running this command will allow it to render correctly without any errors.
  - The ports used do **NOT** have to match, but whatever the port gets changed to that's used by the container needs to also have the port updated in the server file, index.js, to also use the same port.

### Specifying a Working Directory:

Without specifying a working directory, copying and adding new files to your container will place the files in the root of the container and could cause potential naming conflicts or other issues when some programs generate files and directories. To avoid this, it's always advised to specify a working directory in the Dockerfile:

```
# Use an existing docker image as a base
FROM node:alpine

WORKDIR /usr/app

# Download and install a dependency
COPY ./ ./
RUN npm install

# Tell the image what to do when it starts
# as a container
CMD [ "npm", "start" ]
```

**NOTE:** in 'WORKDIR /usr/app', if the 'app' directory doesn't exist, it will be created for you automatically.

- Running a new build and starting the container shell will now automatically start you from the workdir specified.

#### Minimizing Cache Busting and Rebuilds:

Making a change of any size to the project files that were built into the container will not reflect those changes on refresh in the web browser. In order to see the changes, another build would need to be initiated since the last build was a snapshot of the project and rebuilding and copying the same files repeatedly just to see changes is not good. We can fix this by splitting the COPY operation in the Dockerfile into 2 (or more, depending on your needs) steps in order to avoid unnecessary rebuilds:

```
# Use an existing docker image as a base
FROM node:alpine

WORKDIR /usr/app

# Download and install a dependency
COPY ./package.json ./
RUN npm install
COPY ./ ./

# Tell the image what to do when it starts
# as a container
CMD [ "npm", "start" ]
```

- After running a new build, we are now specifying that we only want to copy changes to the 'package.json' file and then run 'npm install' **only when there are changes to those files**. NOTE: At this point the container run command w/port mapping must still be rerun to pick up changes (this will be addressed later- add update here).

#### Docker Compose

When running multiple containers at the same time that actually depend on processes being run in another container, there are 2 options available.

1. Use **Docker CLI's Networking Features** (as have been the case so far in this documentation), but this becomes cumbersome for any

real application. Although it's possible, it's is never actually used as a a solution or standard in a real production application.

## 2. Use **Docker Compose**!

- a. Docker Compose is a separate CLI tool that gets installed along with Docker and is used to startup multiple Docker containers at the same time that automatically connects them together with networking and also to keep from writing lots of repetitive commands with Docker CLI.
  - i. Type '**docker-compose**' to see a list of available options
- Docker Compose will make use of a '**docker-compose.yml**' file that will be run by the docker-compose CLI.
  - This file essentially will combine the commands we've been running in the docker CLI into one, i.e. 'docker build -t oe/simpleweb' and 'docker run -p 8080:8080 oe/simpleweb'
  - Example docker-compose.yml file that runs separate redis and a node js app containers, w/comments at the '#' symbols:

```
version: '3' # Specifies the version of docker compose we want to use to
put together this file
services: # 'service' basically means a type of container
  redis-server: # the name of the service you want to create (could be
anything)
    image: 'redis' # the image you want to use to create this
service/container
  node-app:
    build: . # look in the current directory for a dockerfile and
build from there
    ports:
      - "4001:8081" # ports specified that we want opened up on
this container solely for access on local machine
```

### Docker Compose Commands:

- '**docker-compose up**' will run the image
- '**docker-compose up --build**' will build and then run an image with latest changes

### Docker Compose - Automatic Container Restarts

In the event of a crash or other error on the server, we want to give Docker the ability to automatically restart the container. You can do this by specifying a '**Restart Policy**' inside of the **docker-compose.yml** file.

- Restart Policies:

1	"no"	Never attempt to restart this "." container if it stops or crashes
2	always	If this container stops *for any reason*, always attempt to restart it
3	on-failure	Only restart if the container stops with an error code
4	unless-stopped	Always restart unless we (the developers) forcibly stop it

- Updated example docker-compose.yml file implementing restart policies:

```

version: '3'
services:
  redis-server:
    image: 'redis'
  node-app:
    restart: always
    build: .
    ports:
      - "4001:8081"

```

- **NOTE:** The “no” restart policy must be surrounded by quotes (single or double) in the docker-compose.yml file, otherwise the file will interpret it as the value of false.

### Creating a Production-Grade Workflow:

In the full context of developing and making updates to an application, there has to be a set workflow with is commonly implemented. Docker is NOT necessary, but it's a tool that makes this process and workflow much easier if implemented. This section will detail how to add Docker into your Production workflow as well as provide tips to make the process as efficient as possible.

**NOTE:** These steps are from the perspective of using the default CRA (create-react-app) template as the FE application that needs to have Docker implemented and put into the daily workflow of Git, Testing, Deploying, etc.

### React/NPM/Yarn Necessary Commands for Running the FE Application

- **Run The App:** 'npm run start' or 'yarn run start'
- **Test The App:** 'npm run test' or 'yarn run test'
- **Build The App:** 'npm run build' or 'yarn run build'

**NOTE: \*\*\*\*IMPORTANT\*\*\*\*** To avoid duplicating dependencies when running a docker build, it is safe to **REMOVE the local node\_modules file from the React application** as the dependencies are actually on the container as specified by the 'RUN npm install' procedure in the file. If you do NOT remove the local node\_modules folder, then the first thing you will see upon running a docker build is a large amount of data that was 'Sending to build context Daemon .... 150+mb'. This data already lives within the container and can safely be removed.

### Creating the Dev Dockerfile

- Docker picks up the name 'Dockerfile' by default within the project structure to use to build the project and due to this, you will need to create a different version of the Dockerfile to use while in development, each using a slightly different set of commands.
- Create a custom dockerfile by creating a file named 'Dockerfile.dev', in the same manner and place in the project structure you would normally create the dockerfile. Add the usual content that you want to use that's required for the dockerfile. Next, you will need to use the docker command that will allow docker to use the custom file rather than the default one by using this command:
  - **'docker build -f <custom file name> .', i.e. 'docker build -f Dockerfile.dev .'**

### Starting the Container

- Remember to add the '-p' flag along with the port mappings followed by the container id to get it to render from the container in the browser.
  - **docker run -p 3000:3000 <container id> .....**This will build and run the container with your CRA and set your port mappings so that the container actually renders the app when you go to 'https://localhost:3000'

### Getting Source Code Changes to Appear with Stopping and Rebuilding the Container

The previous section of 'Starting the Container' has one issue, and that's that while the app is running, any changes to the source code of the CRA aren't reflected unless you stop and rebuild the container and then run it again. This can be resolved by implementing Docker Volumes. Docker Volumes are essentially mappings from a folder inside the container to a folder outside of the container (folders on your local machine in the CRA source code). You can think of it as port mappings but for folders so that updates can be fed to the container when made.

- Docker Volumes CLI Command: **'docker run -p 3000:3000 -v /app/node\_modules -v "\$(pwd)"/app' <image id>'**
  - This volume mapping says to map create volumes for the node\_modules folder in the container '/app' directory and the '\$(pwd)' local '/app' directory
  - Doing this will now allow for the **Hot Reloading** feature of CRA.



## Using Docker Compose Shorthand

Rather than using the manual Docker CLI commands such as `'docker run -p 3000:3000 -v /app/node_modules -v "$(pwd):/app" <image id>'` in the previous section, we can again avoid this by using Docker Compose like so:

- Create a **'docker-compose.yml'** file in the same location as the `Dockerfile.dev` file and it should look like this:

```
version: '3'
services:
  web:
    build:
      context: . # Specifies where we want all files and
      dockerfile: Dockerfile.dev # Name of dockerfile to be
      used to build and run container
    ports:
      - "3000:3000"
    volumes:
      - /app/node_modules
      - ./app
```

- Running **'docker-compose up'** will now execute the same run commands using volumes as discussed in the previous section!

## Executing Tests

You can easily run your unit tests using the commands mentioned previously in this documentation by doing the following:

- Run a new build and get the tag or container id
- Run **'docker run -it <container id> npm/yarn run test'**
  - **NOTE 1:** Make use of the `'-it'` flag in order to get the normal terminal UI that you can interact with, otherwise it will just run the tests one time and be static.

## Running Tests that autorun/update on changes:

Tests won't rerun or update via this alone so you so you have some options to do this:

1. **Attaching to an existing container:** Build and Run your volumed container with docker-compose running in one terminal and in another one, get the container id by running **'docker ps'** and then running **'docker exec -it <container id> npm/yarn run test'**
  - a. Making changes to test files will now autorun and update them in the terminal session.
  - b. **NOTE: I prefer this method as I prefer having access to the test suite commands in the terminal and that's only possible w/this approach.**
2. **Adding a new service to the docker-compose.yml file:** You can update your docker-compose.yml file to include another service for testing, 'tests' in this example and then run a new build and run command with **'docker-compose up --build'**:

```

a. version: '3'
  services:
    web:
      build:
        context: . # Specifies where we want all files and
        folders to be pulled from
        dockerfile: Dockerfile.dev # Name of dockerfile to
        be used to build and run container
      ports:
        - "3000:3000"
      volumes:
        - /app/node_modules
        - ./app
      tests:
        build:
          context: .
          dockerfile: Dockerfile.dev
        volumes:
          - /app/node_modules
          - ./app
        command: [ "npm", "run", "test" ] # Override our
        default "npm run start" starting command to use this instead

```

- b. **NOTE:** It's recommended to run a new build at this point as sometimes running a container after adding a new service is finicky.
- c. Unfortunately, this is as good as it gets with docker-compose with testing. You won't ever be able to access the secondary processes in the container via '**docker attach**' or anything else and are just left with the static test output in the terminal that docker-compose is running it. **NOTE:** It still refreshes and autoupdates when there are changes to tests, but you won't be able to use the terminal to run the test suite w/Jest or another test runner commands.

## TESTING TLDR;

1. Using option 1 above:
  - a. You just need the 'web' service in your docker-compose.yml file:

```

i. version: '3'
  services:
    web:
      build:
        context: . # Specifies where we want all files
        and folders to be pulled from
        dockerfile: Dockerfile.dev # Name of
        dockerfile to be used to build and run container
      ports:
        - "3000:3000"
      volumes:
        - /app/node_modules
        - ./app

```

- b. Run '**docker compose up**' in one terminal session
  - c. Open another terminal session and run '**docker ps**' get the **container id** run '**docker exec -it <container id> npm/yarn run test**'
    - i. This will run the Jest/Specific test runner suite in the terminal with full access to it's commands
2. Using option 2 above:
  - a. You need to create a service for 'tests' in your docker-compose.yml file:

```
i. version: '3'
  services:
    web:
      build:
        context: . # Specifies where we want all files
        and folders to be pulled from
        dockerfile: Dockerfile.dev # Name of
        dockerfile to be used to build and run container
      ports:
        - "3000:3000"
      volumes:
        - /app/node_modules
        - ./app
    tests:
      build:
        context: .
        dockerfile: Dockerfile.dev
      volumes:
        - /app/node_modules
        - ./app
      command: [ "npm", "run", "test" ] # Override our
      default "npm run start" starting command to use this
      instead
```

- b. Run '**docker compose up --build**' to make sure the build is clean since you added a new service in a terminal session
    - i. This will run build and run **EVERYTHING** in this terminal session, including test suite results, along with auto-retests on test file changes, but **NOTE: that you won't be able to directly interact with the test suite using this option**

#### Docker Links and Resources:

- [Docker Official Docs](#)
- [Docker Labs Cheatsheet](#)
- [Docker Hub](#)