Use Docker to Create a Node Development Environment

auth0.com/blog/use-docker-to-create-a-node-development-environment/

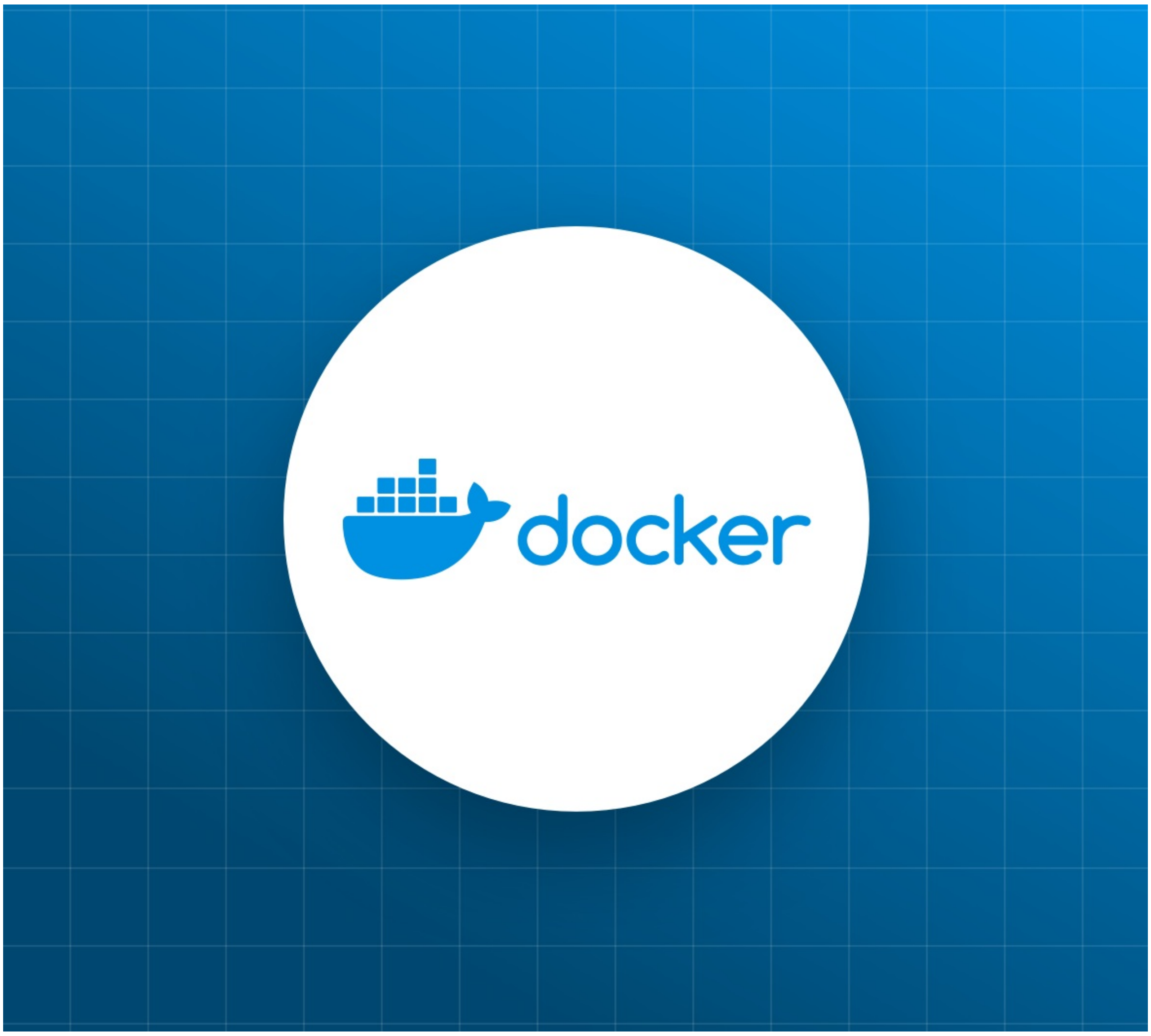
Dan
Arias

Docker
Leverage Docker images and containers to create an isolated Node development environment that runs a server

Dan Arias

R&D Content Engineer

February 26, 2019



Docker

Leverage Docker images and containers to create an isolated Node development environment that runs a server

Dan Arias

R&D Content Engineer

February 26, 2019

Table of Contents

In this tutorial, instead of creating and running a Node app locally, you'll to take advantage of the Debian Linux operating system that official Docker Node images are based on. You'll create a portable Node development environment that solves the **"But it runs on my machine"** problem that constantly trolls developers since containers are created predictably from the execution of Docker images on any platform.

Throughout this tutorial, you'll be working in two realms:

- **Local Operating System**: Using a CLI application, such as Terminal or PowerShell, you'll use a local installation of Docker to build images and run them as containers.
- **Container Operating System**: Using Docker commands, you'll access the base operating system of a running container. Within that context, you'll use a container shell to issue commands to create and run a Node app.

The container operating system runs in isolation from the local operating system. Any files created within the container won't be accessible locally. Any servers running within the container can't listen to requests made from a local web browser. **This is not ideal for local development**. To overcome these limitations, you'll bridge these two systems by doing the following:

- **Mount a local folder to the container filesystem**: Using this mount point as your container working directory, you'll persist locally any files created within the container and you'll make the container aware of any local changes made to project files.
- **Allow the host to interact with the container network**: By mapping a local port to a container port, any HTTP requests made to the local port will be redirected by Docker to the container port.

To see this Docker-based Node development strategy in action, you'll create a basic Node Express web server. Let's get started!

Removing the Burden of Installing Node

To run a simple "Hello World!" Node app, the typical tutorial asks to:

- Download and install Node
- Download and install Yarn
- To use different versions of Node, uninstall Node and install nvm
- Install NPM packages globally

Each operating system has its own quirks making the aforementioned installations non-standard. However, access to the Node ecosystem can be standardized using Docker images. **The only installation requirement for this tutorial is Docker**. If you need to install Docker, choose your operating system from this Docker installation document and follow the steps provided.

Similar to how NPM works, Docker gives us access to a large registry of Docker images called Docker Hub. From Docker Hub, you can pull and run different versions of Node as images. You can then run these images as local processes that don't overlap or conflict with each other. You can simultaneously create a cross-platform project that depends on Node 8 with NPM and another one that depends on Node 11 with Yarn.

Creating the Project Foundation

To start, anywhere in your system, create a `node-docker` folder. This is the project directory.

With the goal of running a Node Express server, under the `node-docker` project directory, create a `server.js` file, populate it as follows and save it:

```
// server.js
const express = require("express");
const app = express();

const PORT = process.env.PORT || 8080;

app.get("/", (req, res) => {
  res.send(`
    <h1>Docker + Node</h1>
    <span>A match made in the cloud</span>
  `);
});

app.listen(PORT, () => {
  console.log(`Server listening on port ${PORT}...`);
});
```

A Node project needs a `package.json` file and a `node_modules` folder. *Assuming that Node is not installed in your system*, you'll use Docker to create those files following a structured workflow.

Accessing the Container Operating System

You can gain access to the container OS with any of the following methods:

- Using a single but long `docker run` command.
- Using a `Dockerfile` combined with a short `docker run` command.
- Using a `Dockerfile` in combination with Docker Compose.

Using a single `docker run` command

Execute the following command:

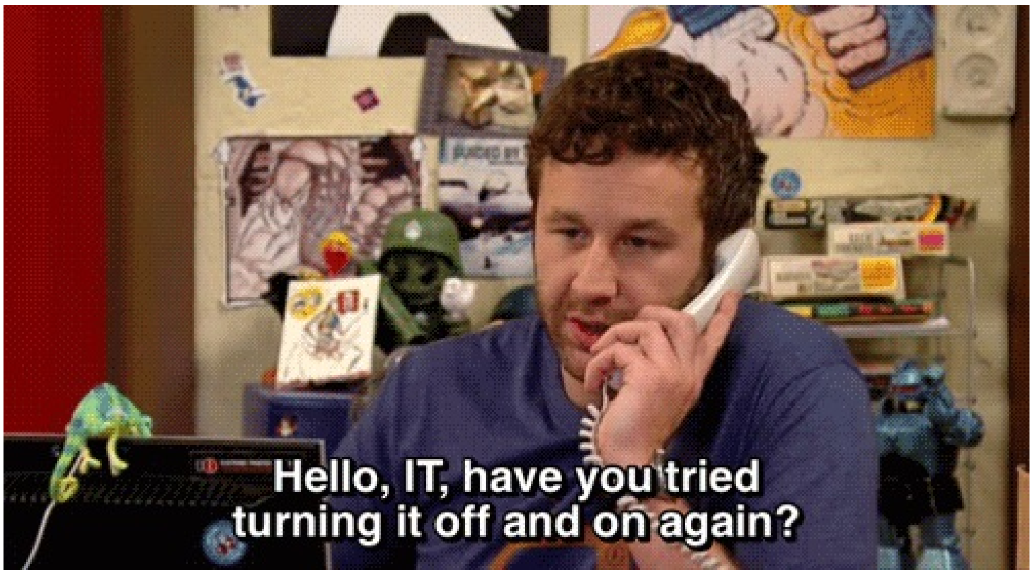
```
docker run --rm -it --name node-docker \
-v $PWD:/home/app -w /home/app \
-e "PORT=3000" -p 8080:3000  \
-u node node:latest /bin/bash
```

Let's breakdown this `docker run` command to understand how helps you access the container shell:

```
docker run --rm -it
```

`docker run` creates a new container instance. The --rm flag automatically stops and removes the container once the container exits. The combined `-i` and `-t` flags run interactive processes such as a shell. The `-i` flag keeps the STDIN (Standard Input) open while the `-t` flag lets the process pretend it is a text terminal and pass along signals.

> Think of `--rm` as "out of sight, out of mind".
>
> Without the `-it` team, you won't see anything on the screen.



```
docker run --rm -it --name node-docker
```

The --name flag assigns a friendly name to the container to easily identify it in logs and tables. For example when you run docker ps.

```
docker run --rm -it --name node-docker \
-v $PWD:/home/app -w /home/app
```

The -v flag mounts a local folder into a container folder using this mapping as its argument:

`<HOST FOLDER RELATIVE PATH>:<CONTAINER FOLDER ABSOLUTE PATH>`

An environmental variable can print the current working directory when the command is executed: `$PWD` on Mac and Linux and `$CD` on Windows. The -w flag sets the mounting point as the container working directory.

```
docker run --rm -it --name node-docker \
-v $PWD:/home/app -w /home/app \
-e "PORT=3000" -p 8080:3000
```

The -e flag sets an environmental variable `PORT` with a value of `3000`. The -p flag maps a local port `8080` to a container port `3000` to match the environmental variable `PORT` that is consumed within `server.js`:

```
const PORT = process.env.PORT || 8080;
```

```
docker run --rm -it --name node-docker \
-v $PWD:/home/app -w /home/app \
-e "PORT=3000" -p 8080:3000  \
-u node node:latest /bin/bash
```

For security and to avoid file permission problems, the -u flag sets the non-root user `node` available in the Node image as the user that runs the container processes. After setting the flags, the image to execute is specified: `node:latest`. The last argument is a command to execute inside the container once it's running. `/bin/bash` invokes the container shell.

> If the image is not present locally, Docker issues docker pull in the background to download it from Docker Hub.

Once the command executes, you'll see the container shell prompt:

```
node@<CONTAINER ID>:/home/app$
```

Before moving to the next method, exit the container terminal by typing `exit` and pressing `<ENTER>`.

Using a `Dockerfile`

The `docker run` command from the previous section is made of image build time and container runtime flags and elements:

```
docker run --rm -it --name node-docker \
-v $PWD:/home/app -w /home/app \
-e "PORT=3000" -p 8080:3000  \
-u node node:latest /bin/bash
```

Anything related to image build time can be defined as a custom image using a `Dockerfile` as follows:

- `FROM` specifies the container base image: `node:latest`
- `WORKDIR` defines `-w`
- `USER` defines `-u`
- `ENV` defines `-e`
- `ENTRYPOINT` specifies to execute `/bin/bash` once the container runs

Based on this, under the `node-docker` project directory, create a file named `Dockerfile`, populate it as follows, and save it:

```
FROM node:latest

WORKDIR /home/app
USER node
ENV PORT 3000

EXPOSE 3000

ENTRYPOINT /bin/bash
```

`EXPOSE 3000` documents the port to expose at runtime. However, container runtime flags that define container name, port mapping, and volume mounting still need to be specified with `docker run`.

The custom image defined within `Dockerfile` needs to be built using docker build before it can be run. In your local terminal, execute:

```
docker build -t node-docker .
```

`docker build` provides your image the friendly name `node-docker` using the -t flag. This is different than the container name. To verify that the image was created, run `docker images`.

With the image built, execute this shorter command to run the server:

```
docker run --rm -it --name node-docker \
-v $PWD:/home/app -p 8080:3000 \
node-docker
```

The container shell prompts comes up with the following format:

```
node@<CONTAINER ID>:/home/app$
```

Once again, before moving to the next method, exit the container terminal by typing `exit` and pressing `<ENTER>`.

Using Docker Compose

> For Linux, Docker Compose is installed separately.

Based on the `Dockerfile` and the shorter `docker run` command of the previous section, you can create a Docker Compose YAML file to define your Node development environment as a service:

`Dockerfile`:

```
FROM node:latest

WORKDIR /home/app
USER node
ENV PORT 3000

EXPOSE 3000

ENTRYPOINT /bin/bash
```

Command

```
docker run --rm -it --name node-docker \
-v $PWD:/home/app -p 8080:3000 \
node-docker
```

The only elements left to abstract from the `docker run` command are the container name, the volume mounting, and the port mapping.

Under the `node-docker` project directory, create a file named `docker-compose.yml`, populate it with the following content, and save it:

```
version: "3"
services:
  nod_dev_env:
    build: .
    container_name: node-docker
    ports:
      - 8080:3000
    volumes:
      - ./:/home/app
```

- `nod_dev_env` gives the service a name to easily identify it
- `build` specifies the path to the `Dockerfile`
- `container_name` provides a friendly name to the container
- `ports` configures host-to-container port mapping

- `volumes` defines the mounting point of a local folder into a container folder

To start and run this service, execute the following command:

```
docker-compose up
```

`up` builds its own images and containers separate from those created by the `docker run` and `docker build` commands used before. To verify this run:

```
docker image
# Notice the image named <project-folder>_nod_dev_env
docker ps -a
# Notice the container named <project-folder>_nod_dev_env_<number>
```

`up` created an image and a container but the container shell prompt didn't come up. What happened? `up` starts the full service composition defined in `docker-compose.yml`. However, it doesn't present interactive output; instead, it only presents static service logs. To get interactive output, you use `docker-compose run` instead to run `nod_dev_env` individually.

First, to clean the images and containers created by `up`, execute the following command in your local terminal:

```
docker-compose down
```

Then, execute the following command to run the service:

```
docker-compose run --rm --service-ports nod_dev_env
```

The `run` command acts like `docker run -it`; however, it doesn't map and expose any container ports to the host. In order to use the port mapping configured in the Docker Compose file, you use the `--service-ports` flag. The container shell prompt comes up once again with the following format:

`node@<CONTAINER ID>:/home/app$`

If for any reason the ports specified in the Docker Compose file are already in use, you can use the `--publish`, (`-p`) flag to manually specify a different port mapping. For example, the following command maps the host port `4000` to the container port `3000`:

```
docker-compose run --rm -p 4000:3000 nod_dev_env
```

Installing Dependencies and Running the Server

> If you don't have an active container shell, using any of the previous section methods to access it.

In the container shell, initialize the Node project and install dependencies by issuing the following commands (if you prefer, use `npm`):

```
yarn init -y
yarn add express
yarn add -D nodemon
```

Verify that `package.json` and `node_modules` are now present under your local `node-docker` project directory.

nodemon streamlines your development workflow by restarting the server automatically anytime you make changes to source code. To configure `nodemon`, update `package.json` as follows:

```
{
  // Other properties...
  "scripts": {
    "start": "nodemon server.js"
  }
}
```

In the container shell, execute `yarn start` to run the Node server.

To test the server, visit http://localhost:8080/ in your local browser. Docker intelligently redirects the request from the host port `8080` to the container port `3000`.

To test the connection of local file content and server, open `server.js` locally, update the response as follows and save the changes:

```
// server.js

// package and constant definitions...

app.get("/", (req, res) => {
  res.send(`
    <h1>Hello From Node Running Inside Docker</h1>
  `);
});

// server listening...
```

Refresh the browser window and observe the new response.

Modifying and Extending the Project

Assuming that Node is not installed in your local system, you can use the local terminal to modify project structure and file content but you can't issue any Node-related commands, such as `yarn add`. As the server runs within the container, you are also not able to make server requests to the internal container port `3000`.

In the event that you want to interact with the server within the container or modify the Node project, you need to execute commands against the running container using docker exec and the running container ID. You don't use `docker run` as that command would create a new isolated container.

Getting the running container ID is easy.

If you already have a container shell open, the container ID is present in the shell prompt:

```
node@<CONTAINER ID>:/home/app$
```

You can also get the container ID programmatically using docker ps to filter containers based on name and return the `CONTAINER ID` of any match:

```
docker ps -qf "name=node-docker"
```

> The `-f` flag filters containers by the `name=node-docker` condition. The `-q` (`--quiet`) limits the output to only display the ID of the match, effectively plugging the `CONTAINER ID` of `node-docker` into the `docker exec` command.

Once you have the container ID, you can use `docker exec` to:

Open a new instance of the running container shell:

```
docker exec -it $(docker ps -qf "name=node-docker") /bin/bash
```

Make a server request using the internal port `3000` :

```
docker exec -it $(docker ps -qf "name=node-docker") curl localhost:3000
```

Install or remove dependencies:

```
docker exec -it $(docker ps -qf "name=node-docker") yarn add body-parser
```

> One you have another active container shell, you can easily run `curl` and `yarn add` there instead.

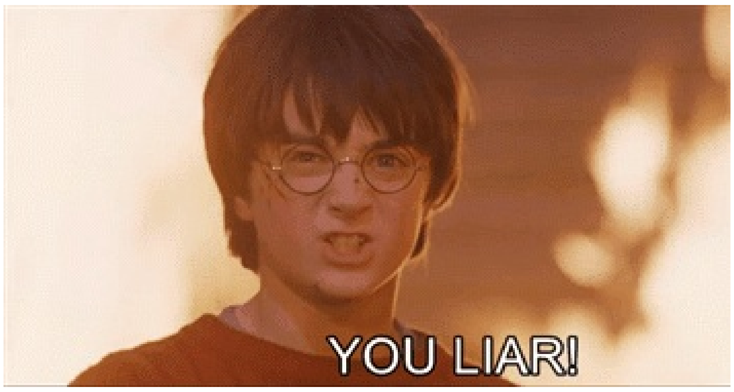
Recap... and Uncovering Little Lies

You've learned how to create an isolated Node development environment through different levels of complexity: by running a single `docker run` command, using a `Dockerfile` to build and run a custom image, and using Docker Compose to run a container as a Docker service.

Each level requires more file configuration but a shorter command to run the container. This is a worthy trade-off as encapsulating configuration in files makes the environment portable and easier to maintain. Additionally, you learned how to interact with a running container to extend your project.

You may still need to install Node locally for IDEs to provide syntax assistance, or you can use a CLI editor like `vim` within the container.

Even so, you can still benefit from this isolated development environment. If you restrict project setup, installation, and runtime steps to be executed within the container, you can standardize those steps for your team as everyone would be executing commands using the same version of Linux. Also, all the cache and hidden files created by Node tools stay within the container and don't pollute your local system. Oh, and you also get `yarn` for free!



JetBrains is starting to offer the capability to use Docker images as remote interpreters for Node and Python when running and debugging applications. In the future, we may become entirely free from downloading and installing these tools directly in our systems. Stay tuned to see what the industry provides us to make our developer environments standard and portable.