Using Docker, the short, short version

> *"Short" is not a joke, it's because I used many examples which is why this text is much longer than what you need. To make it short, find the first example that fits your needs and use it.*

Quick highlevel overview

Docker is essentially a way to run stuff in a local sandboxed environment. The environment is specified by a *docker image*, and its main component is a snapshot of all files that are needed to run in, in the form of "layers", each saved as a tar archive (and it's implemented as UnionFS).

> *Docker is not a VM, but often confused as one. Images are linux-based, and therefore Docker on Windows works by installing a tiny Hyper-V Linux VM to run on (but that is shared for all docker uses, it's not starting a VM for each run).*

When you run an image, the running sandbox is called a *container*. These container are based on the image which is the initial state (files etc), and on top of that there are any changes that the current execution created (FS changes, running process/es, etc). When container is done running, *all of that* usually disappears, making it very convenient to run random stuff without affecting your setup. (It is possible to save containers, but usually they're removed after use.)

Quick examples

You need to install docker to try the following examples. The installer itself is generally well behaved and will tell you what needs to be done to make it work (eg, turning on windows features like hyper-v or the wsl2 backend). There are also installers for macs and for linux (the latter being a system daemon rather than a tiny vm).

Once installed, you can use the `docker` command to do stuff. It has the usual `docker ⟨verb⟩ args...` format. On windows, it works fine in all forms: powershell, vscode, and even in a cmd box. The main (and possibly the only) verb you need to know about is `run`:

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

This drops you into a running `node` container. (Ctrl+D is the canonical EOF-thing in unix, use it to exit the running process and therefore the container.)

- `run`: start running a container for the specified image. The image will be pulled in on first use.
- `-it`: interactive run (short for `--interactive --tty`, the latter is a unix thing)
- `--rm`: delete the container when done (you can drop this if you want to keep the results, but usually you want to include it)
- `node`: the name of the image we want to run (there are many, enough that most random guesses for "the thing you want to try" will work)

```
docker run -it --rm node:12
```

Image names are tagged — this is similar to the above, but now I'm specifying that I want to use the `12` tag. When you don't specify a tag as in the above, you get the default of `:latest`.

> *Tags are not permanently fixed (especially not `latest`). To update a tag (eg, a new node version is published), you can use `docker pull node` to update it. Similarly, `node:12` is a tag that points at the most recent 12.x version.*

But this is still just drops you into a running `node`, what if you want to do something before starting it, like installing some suspicious package?

```
docker run -it --rm node bash
```

Here I added a `bash` at the end, overriding what the `node` image runs by default. Now I get a `bash` prompt, and I can do whatever I want: `npm install` stuff (locally or globally), `apt install` OS packages (you'll need to `apt update` first to get the package directory), and even `rm /bin/*` — it's all completely safe, and everything will disappear when the container is done.

But if you know even a little about linux, you'll recognize that this is a kind of an overkill: you start at the root of the filesystem, and as the `root` user. This could be significantly different from actual use. For example, I run into a weird new `1line-aa` npm package, with no visible information about how it's working, and I want to try it as a user.

```
docker run -it --rm node bash
$ su -l node
$ npm install 1line-aa
```

One way to do this is to run `su -l node` in the container, which starts a new shell for the `node` user (which the node image includes). Since this process is started from the first one, you'll need to Ctrl+D twice to get out of the container (or thrice if you start `node`).

```
docker run -it --rm -u node -w /home/node node bash
```

Another way to do this directly is to add:

- `-u node`: start as the `node` user
- `-w /home/node`: in its home directory

So far all of these examples left nothing behind, but what if you *want* to collect some of the resulting files?

```
docker run -it --rm -v c:\foo:/work node bash
```

- `-v c:\foo:/work`: mount the `c:\foo` directory onto `/work` in the container

This means that in the container you can `cd /work` and do whatever you want there: since it's a mounted directory, everything will actually happen in your `c:\foo` directory (which will be created if it doesn't exist). It doesn't matter that the file owner inside the container is `root`, since on a Windows host side, it's all running as your Windows user. This is *not* the case if you're using docker on linux: in that case, `root` in the container will create files that belong to `root` on the host. In any case, switching to the `node` user (as done above) is preferable.

Complicated example 1: tsserverfuzzer

This is a more involved example: running the fuzzer. First, clone the repository — the `node` image includes `git` so you can do it in the container, but you're probably more comfortable with your usual environment. You'll probably use vscode or whatever... something like

```
c:\> cd work
C:\work> git clone ...tsserverfuzzer...
C:\work> cd tsserverfuzzer
C:\work\tsserverfuzzer> docker run -it --rm -v %cd%:/fuzzer -w /fuzzer -u node node
bash
```

- `-it --rm` : interactive, dispose after use
- `-v %cd%:/fuzzer` : mount the current directory as `/fuzzer` in the container (in powershell, use `$pwd` instead of `%cd%`)
- `-w /fuzzer` : and work in there
- `-u node` : as the `node` user

```
node@...:/fuzzer$ npm install
...
node@...:/fuzzer$ npm run build
...
node@...:/fuzzer$ git status
...
node@...:/fuzzer$ node lib/Fuzzer/main.js
```

You can now do the usual things, even `git` commands (since the file format is the same — just be careful of sneaky EOL translation).

I you did all of this, the `git status` should show just a change in `package-lock.json`, and the last execution got stuck waiting for a debugger to connect. Ctrl+C to abort it, and Ctrl+D to exit the container.

```
docker run ...same... -p 9229:9242 node bash
```

It's possible to forward ports from the container to the host, and it's similar to the `-v` flag with the same meaning to the two sides of the colon: here we're saying that port 9242 in the container is exposed as port 9229 on the host. Once you do that, you can skip the building (since the built files are still in the directory on your host) and go straight to the `node` command.

... except that this won't work either. This is because the debugger listens on `127.0.0.1:9242` (and it tells you that), which means that it only accepts connections from `localhost` which is the container. We're connecting from what looks to the container like a different machine, so we need to allow that. To do this, open `C:\work\tsserverfuzzer\Fuzzer\main.ts` in your editor (outside the container!) and change `'--inspect-brk=9242'` to `'--inspect-brk=0.0.0.0:9242'` (the `0.0.0.0` tells it to listen to anyone).

The container sees the file modification, so you don't need to restart it, you can just run `npm build` again, and re-run.

```
docker run ...same... node node lib/Fuzzer/main.js
```

When you want to run the already-built fuzzer later, you can start it directly. The `node node` looks confusing, but the first one is the name of the image, and following it is the `node` command that you want to run in the container instead of starting an interactive repl.

```
docker run ...same... -e GitHubAuthenticationKey=%tok% node node lib/Fuzzer/main.js
```

At some point you'll find that it needs a GH key in the `$GitHubAuthenticationKey` environment variable. The `-e VAR=VAL` flag sets such a value, and in this case we're using a Windows `%tok%` as the value. (And something like `$env:tok` in powershell.)

Since it's running the node code directly, a Ctrl+C will stop it and exit the container immediately.

Complicated example 2: TypeScriptErrorDeltas

There's enough verbiage above to run it, but a few more useful bits that are relevant in this case:

```
docker run ... -v %USERPROFILE%\.npmrc:/home/node/.npmrc:ro ...
```

You'll find that you need an npm authentication key to be able to `npm install` this thing. Assuming that you have the key in your `C:\Users\foo\.npmrc`, you can re-use your `.npmrc` in the container. There are two new things here:

- Container mounts don't have to be directories, you can mount a single file as this is doing.
- An access mode can follow a second `:`, use `rw` (the default) for read-write or `ro` read-only. In this example using `:ro` ensures that the container cannot modify the windows `.npmrc` file.

```
C:\...> docker run -it --rm ... node bash
# apt update; apt install sudo
# node /work/index.js 1 3.3 3.4 false
```

One problem with running this code is that it requires having `sudo`, but the `node` image is based on a minimal linux so it doesn't have it. One way to do it is to fix the code to not use `sudo` if it's running as root ... but a way around it is to start the container with `bash`, and run the two `apt` commands to get `sudo` installed. (In the case of this `TypeScriptErrorDeltas` code, there is something else that is needed: see "Privileged runs" below.)

It is obviously tedious to do this installation every time you want to run it — ignoring changing the code to not require extra packages, it is pretty easy to build an image yourself. But I'll finish the quick part here.

Extras

Privileged runs

A docker container is an image running in a sandboxed environment that is restricted in several ways (like seeing its own FS and network). There are, however, cases where linux functionality is needed from the kernel — and mounting things (when you're already *in* the container) is one such case that is normally blocked. Docker has a bunch of "capabilities" that are off by default and can be turned on if needed. In cases like `TypeScriptErrorDeltas`, where you're running known non-malicious code, you can just enable all of them by adding a `--privileged` flag.

`docker build`

The `build` verb can be used with a `Dockerfile` which specifies a recipe for creating an image. For example, it's easy to make an image that is based on the `node` image, but has a few more os packages installed, and defaults to a specific directory, user, and command to run. There's lots of examples around, but in general I'll be happy to explain how to create any image that anyone might need.

`docker ps`

This is a useful command to see a list of running docker containers. Usually this should be empty, but you might press some wrong key (like Ctrl+Z, which suspends a process) and end up with a stray process. Use this to see such processes, and kill them with `docker rm -f ⟨container-id⟩`.

```
const orGUI =
  "Or, as long as you're a gui-dependent windows user,"
  + "just use the docker gui...";
```

`console.log(orGUI);`

`docker exec`

The process that gets to run on a `docker run` is not the only thing that runs. Subprocesses can run in the container, of course, but in addition to that you can start another process in the context of a running container. This is useful, for example, if you started a container as the `node` user, and you need to install some os package, but you don't want to start from scratch.

```
C:\> docker ps
... node id ...
C:\> docker exec -it 123 bash
```

To do this, use `docker ps` to find your container's id, then use `docker exec` to start a `bash` process in it. It is somewhat similar to `run` except that it expects a running container id. (Or names, since you can name running containers, because why not add features.)

`console.log(orGUI);`

`docker system prune`

You might end up with random stuff that sticks around for whatever reason. A running or a suspeded container, stray images since you played with building your own image, or whatever.

```
docker system prune -f
```

This will remove any such stuffs. The `-f` makes it just remove the unnecessary stuff rather than ask you for confirmation.

`console.log(orGUI);`