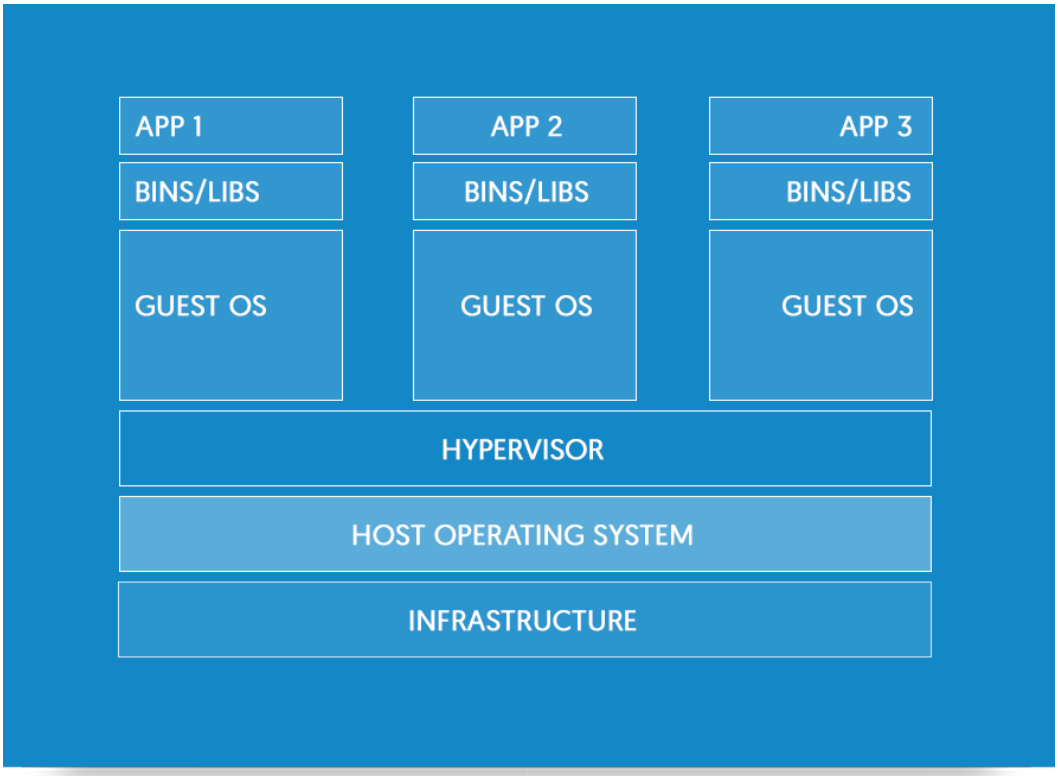
Node.js in Containers Using Docker

Container technology is one of the best options for software development and deployment. It allows you to share some of the OS resources while encapsulating the code and other concerns. You can think of containers as virtual machines but with less footprint.

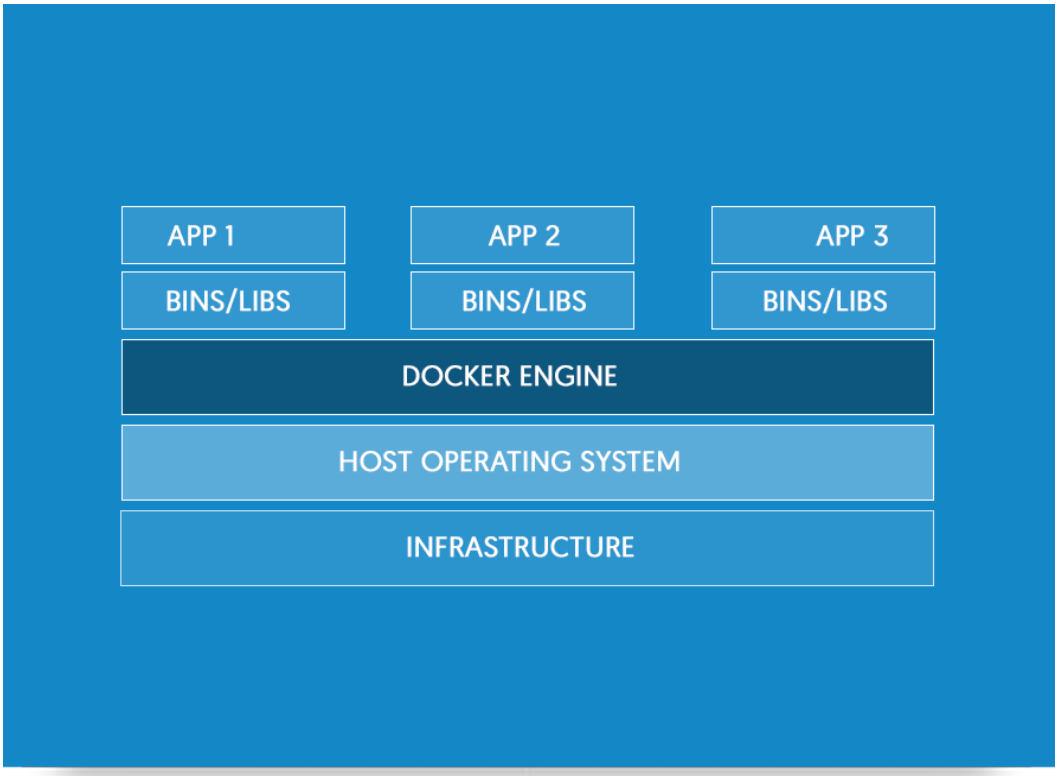
Containers are great for micro services where you replace monoliths with many services. Each of them works in isolation and communicates with other services via a well defined interface (typically REST).

Docker is one of the most popular implementations of containers. Docker's What is Docker? page has a a neat comparison of containers with VMs. In a nutshell, VMs use hypervisor and each VM has it's own OS while containers share OS and only separate libraries, bin, executables, etc.

This is a simplified diagram on how VMs work (source: docker.com).



While with containers, more things are shared. Thus, you get faster startup, execution, spin up, etc.



Here are some of the benefits of Docker containers:

- Allow for rapid application development/deployment.
- Are Extendable.
- Are Lightweight.
- Are portable across machines & environments.
- Are fast with a lightweight footprint.
- Are easy to use.
- Allow consistent behavior from dev to production.
- Can be versioned and components can be reused.
- Allow for community/collaboration.
- Are easily maintained.

Before we can start working with Docker, we should define commonly used terms, i.e., terminology that is used frequently in the Docker ecosystem:

Images - The blueprints of our application which form the basis of containers. We will use the `docker pull` command to download the specified image.

Containers - Created from Docker images and run the actual application. We create a container using `docker run`. A list of running containers can be seen using the `docker ps` command.

Docker Daemon - The background service running on the host that manages building, running and distributing Docker containers. The daemon is the process that runs in the operation system to which clients talk to. It is what makes Docker Engine work.

Docker Client - The command line tool that allows the user to interact with the daemon. There can be other forms of clients - such as Kitematic which provides a GUI.

Docker Hub - A registry of Docker images. You can think of the registry as a directory of all available Docker images. If required, one can host their own Docker registries and can use them for pulling images.

Dockerfile - A recipe from which you can create an image. Dockerfile has the base image, instructions to add or copy files, commands to execute, ports to expose and other information. `Dockerfile` is case sensitive.

Docker Compose - A mechanism to orchestrate multiple containers needed for a service(s) from a single configuration file `docker-compose.yml`.

Host - Your computer which hosts docker daemon or a remote machine which hosts docker deamon/engine.

Node.js is one of the fastest growing platforms. It's great for web applications and API, especially for microservices. Let's take a look how you can get started with Node and Docker in these steps:

- Installing Docker
- Docker Basics
- Creating Node images
- Working with multiple containers: Node and MongoDB

Installing Docker

First of all, you would need to get the Docker deamon. If you are a macOS user like I'm, then the easiest way is to just go to the official Docker website https://docs.docker.com/docker-for-mac.

If you are not a macOS user, then you can select one of the options from this page: https://docs.docker.com/engine/installation.

Once install is complete, test your Docker installation by running:

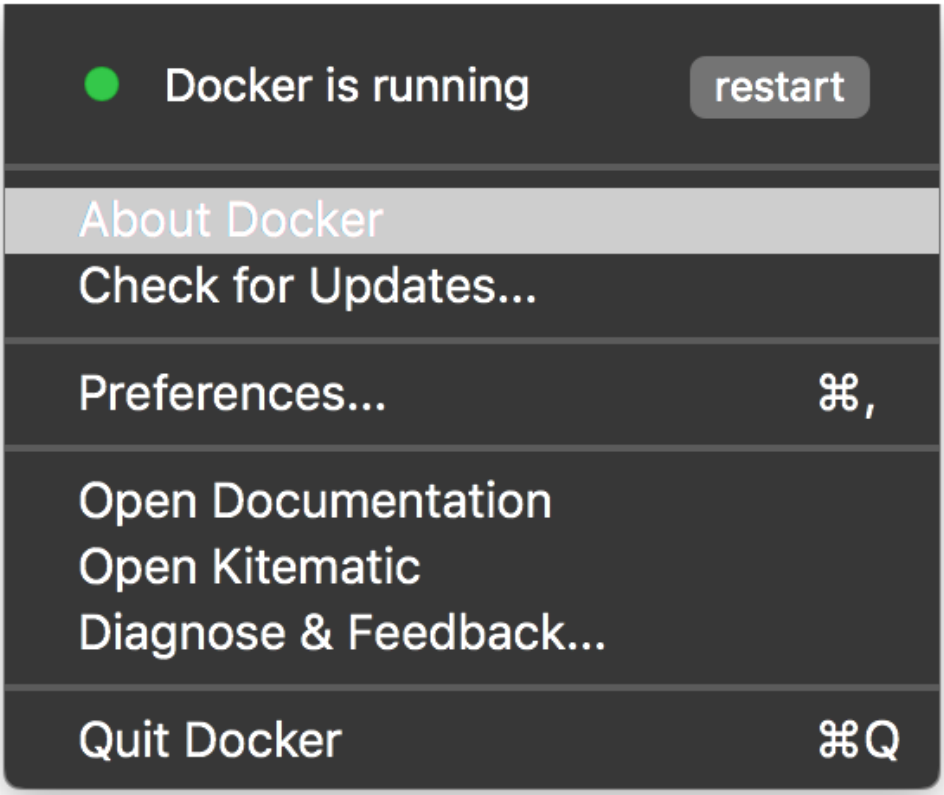
```
$ docker run hello-world
```

If you see a message like this most likely you didn't start Docker:

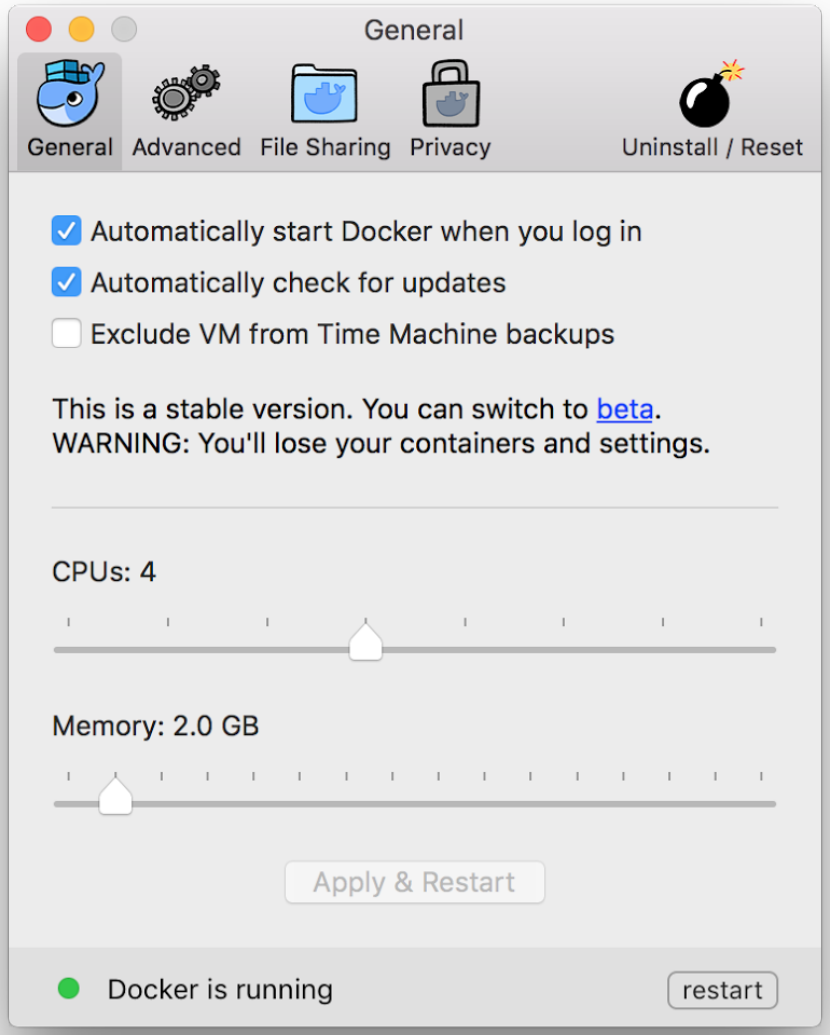
```
Cannot connect to the Docker daemon. Is the docker daemon
running on this host?
```

Start Docker. If you used macOS, you can utilize the GUI app. Otherwise, CLI.

This is how running Docker deamon looks on my macOS:



I can even configure how much memory it takes, whether it updates automatically or starts itself on the log in.



title

On the contrary, if you see a message like the one below, then deamon is running and you are ready to work with Docker!

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world

c04b14da8d14: Pull complete
Digest:
sha256:0256e8a36e2070f7bf2d0b0763dbabdd67798512411de4cdcf9431a

Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be
working correctly.

...
```

Next, we will download a lightweight version of Linux as an image. It's called Alpine. We will get this Alpine image from Docker Hub.

```
$ docker pull alpine
```

Wait for the image to download. I hope you have a fast internet speed. The good thing is that you need to download image just once. It will be stored on your computer for future usage. Let's actually go ahead and check that the image is there by running:

```
$ docker images
```

It will show you Alpine, hello world and probably nothing else. It's okay, because you just started working with Docker. Let's learn Docker basics first.

Docker Basics

First, to install an image (pull) from Docker Hub, there's a `docker pull {name}` command. You already used it for Alpine:

```
$ docker pull alpine
```

Some other images by names from Docker Hub are:

- `boron`: Node.js v6 based on Debian Jessie
- `argon`: Node.js v4 based on Debian Jessie
- `ubuntu`: Ubuntu
- `redis`: Redis based on Debian Jessie
- `mongodb`: MongoDB based on Debian Wheezy

Once you download an image, you can run it with `docker run {name}`, e.g.,

```
$ docker run alpine
```

But then **nothing** happened! That's because when you call `$ docker run {name}`, the Docker client (CLI):

- Finds the image (`alpine` in this case)

- Loads up the container
- Runs commands (if any) in the container

When we run `$ docker run alpine`, we didn't provide any commands, so the container booted up, ran an empty command and then exited.

Let's try a better command which will print `hello world from alpine`:

```
$ docker run alpine echo "hello from alpine"
```

As a result, Docker runs the echo command in our alpine container and then exits.

Imagine how long it could have taken to boot up a virtual machine, run a command in it and then killing it! Much longer then a container. That's a benefit of containers.

If it was the case that containers can only run one echo command and exit, they would be very useless. Luckily, containers can perform long running processes, i.e., they are running without exiting. To see all containers that are currently running, use this command:

```
$ docker ps
```

The `ps` will show you a list of all containers that we've run on this computer

(called host):

```
$ docker ps -a
```

To stop a detached container, run `$ docker stop {ID}` by giving the container ID.

Some useful options to the `doker run` command are:

- `-d` will detach our terminal (bg/daemon).
- `-rm` will remove the container after running.
- `-it` attaches an interactive tty in the container.
- `-p` will publish/expose ports for our container.
- `--name` a name for our container.
- `-v` mounts a volume to share between host and container.
- `-e` provides environment vars to the container.
- `docker run --help` for all flags

Creating Docker Images

If you remember from definition, there's such a thing as Dockerfile. It's how we can create new images. In fact, each image on Docker Hub has Dockerfile. A Dockerfile is just a text file that contains a list of commands that the Docker client calls while building an image.

You can include the following instructions in the Dockerfile:

- **FROM**: (required as the first instruction in the file) Specifies the base image from which to build the Docker container and against which the subsequent Dockerfile instructions are run. The image can be hosted in a public repository, a private repository hosted by a third-party registry, or a repository that you run on EC2.
- **EXPOSE**: Lists the ports to expose on the container.
- **ADD**: adds specified files to a location on the container
- **WORKDIR**: sets the current working directory to run commands in the container.
- **VOLUME**: marks a mount point as externally available to the host (or other containers).
- **CMD**: Specifies an executable and default parameters, which are combined into the command that the container runs at launch. Use the following format:

```
CMD ["executable","param1","param2"]
```

CMD can also be used to provide default parameters for an **ENTRYPOINT** command by omitting the executable argument. An executable must be specified in either a **CMD** or an **ENTRYPOINT**, but not both. For basic scenarios, use a **CMD** and omit the **ENTRYPOINT**.

ENTRYPOINT: Uses the same JSON format as **CMD** and, like **CMD**, specifies a command to run when the container is launched. Also allows a container to be run as an executable with docker run.

If you define an **ENTRYPOINT**, you can use a CMD as well to specify default parameters that can be overridden with docker run's -d option. The command defined by an **ENTRYPOINT** (including any parameters) is combined with parameters from **CMD **or docker run when the container is run.

RUN: Specifies one or more commands that install packages and configure your web application inside the image.

ENV - sets the environment variable `{key}` to the value `{value}` using `{key}={value}`. Syntax example:

```
ENV myName="John Doe" myDog=Rex The Dog myCat=fluffy
```

For more information about instructions you can include in the `Dockerfile`, go to Dockerfile Reference: http://docs.docker.io/reference/builder. For Dockerfile tips and best practices: https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices

Volumes can share code between host (your computer) and the container. In other words, a Docker volume is a wormhole between an ephemeral Docker container and the host. It's great for development or persistent data. The following command will mount a volume from a current working directory (pwd) on host. The files will be available in `/www/` in the container. The way to remember the options is left to right, i.e., `host:contaner`.

```
$ docker run -v $(pwd)/:/www/ -it ubuntu
```

Once the command is run, you will be inside of the container thanks to the `-it`. There you can navigate to the `/www` with `cd /www`. What do you see (use `ls`)? Your files! Now here's the magic. If you modify, remove add files to your host folder, those changes will be automatically in the container!

More over, even if the container is stopped, the persistent data still exists on the Docker host and will be accessible.

Creating Docker Node Images

Now, when it comes to Node, you have an option to get one of the official Node images from Docker Hub. The current versions are Boron and Argon, but there are also version 7 and nightly releases.

Another option is to build Node image from Debian or Ubuntu base. You don't even have to compose the Dockerfile all by your self. You can borrow a few lines from official images and add/remove as necessary.

We will proceed with the first option since it's the simplest way. So we create `Dockerfile` in our Node.js project folder right where you have `package.json` and `node_modules`, i.e., the root of the project. Each project is typically a folder or even a separate Git repository. Then, write in Dockerfile instructions:

```
FROM node:argon

# Create app directory
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# Install app dependencies
COPY package.json /usr/src/app/
RUN npm install

# Bundle app source
COPY . /usr/src/app

EXPOSE 3000
CMD [ "npm", "start" ]
```

We are starting from Argon image, creating a folder for your application code. Then, we copy the source code files from the current folder (project root). Finally we expose the app port (for some strange reason it's almost alway 3000) and boot up the server with `npm start` assuming you have that npm script defined in your `package.json`. If you are not a fan of npm scripts like `npm start`, then simply use `node app.js` or `node server.js` depending on your file name.

To build the Node.js app image, run `$ docker build .` It might take longer the first time you run it if you don't have agron already. Next time, it'll be faster. Once the build is over, you can run your app container like any other images:

```
$ docker run {name}
```

Here's a catch. You might have noticed your new image has no name if you just used `docker build`. And most likely, you already have or will have multiple images. Therefore, it's better to name and tag images when you build them. Use `-t` flag and `nam:tag` format. For example,

```
$ docker build -t {your-name}/{your-app-name}:{tag} .
```

Containers are fast but it's not very cool to build new images each time you make a change to your source code. So for development, we can mount source

code as a volume and use something like `forever` or `nodemon` or `node-dev` to listen for file changes and restart the server anytime we make press save. In the case of a volume, there's no need to copy source code since it'll be mounted from the volume.

```
FROM node:argon

WORKDIR /usr/src/app
RUN npm install

EXPOSE 3000
CMD [ "nodemon", "app.js" ]
```

The command to run this image will be slightly more advance since now we need to mount the volume:

```
$ docker run -v ./:/usr/src/app -it {name}
```

Now, the changes you make will be passed to container, the server will restart and you can develop in your host environment while running code in container. Best of both world! (It's great because the container environment will be *exactly* the same in production as the one you have now.) But apps don't function by themselves. You need some persistence and other services.

Working with Multiple

Containers: Node, and MongoDB

```
version: '2'
services:

  mongo:
    image: mongo
    command: mongod --smallfiles
    networks:
      - all

  web:
    image: node:argon
    volumes:
      - ./:/usr/src/app
    working_dir: /usr/src/app
    command: sh -c 'npm install; npm run seed; npm start'
    ports:
      - "3000:8080"
    depends_on:
      - mongo
    networks:
      - all
    environment:
      MONGODB_URI: "mongodb://mongo:27017/accounts"

networks:
  all:
```

Let inspect this ymp file line by line. We start with a list of services. Name of a

service, i.e., `mongodb` will be available in other containers so we can connect to MongoDB with `mongodb://mongo:27017/accounts`. You don't have to pass this connection string in an environmental variable. I just did it to show that you can do it.

The image, volumes, ports and other fields mimic the Dockerfile instructions. The key distinction is that we use `depends_on`. This will tell the `web` service to use the `mongo` service.

To run the Docker compose, simply execute this terminal command (assuming deamon is running):

```
$ docker-compose up
```

You can look at the full working example of a MERN (MongoDB, Express, React and Node) app at https://github.com/azat-co/mern/blob/master/code. Docker compose is a brilliant and easy way to startup multi-container environment.

Wrap-up

Containers are great to getting your code safely in multiple environment with very little overhead. This allows you to minimize any discrepancies. The basic idea is that by developing in an environment identical to the production, you will eliminate any potential issues related to differences between dev and prod.

Moreover, by getting encapsulation cheaper than with VMs, we can split our apps into more granular services. They can divided be not just into app, database, cache, web server, but even further. We can split web apps into containers by resources, e.g., endpoints for `/accounts` in one container, endpoints for `/users` in another, etc…. but this is a topic for another post.

Further Reading and Docker Resources

Learning never stops! Here's some reading on Docker along with resources.

- **Awesome Docker:** https://github.com/veggiemonk/awesome-docker
- **Hello Docker Workshop:** http://docker.atbaker.me
- **Why Docker:** https://blog.codeship.com/why-docker
- **Docker Weekly and archives:** https://blog.docker.com/docker-weekly-archives
- **Codeship Blog:** https://blog.codeship.com