Welcome aboard!

some introductory text

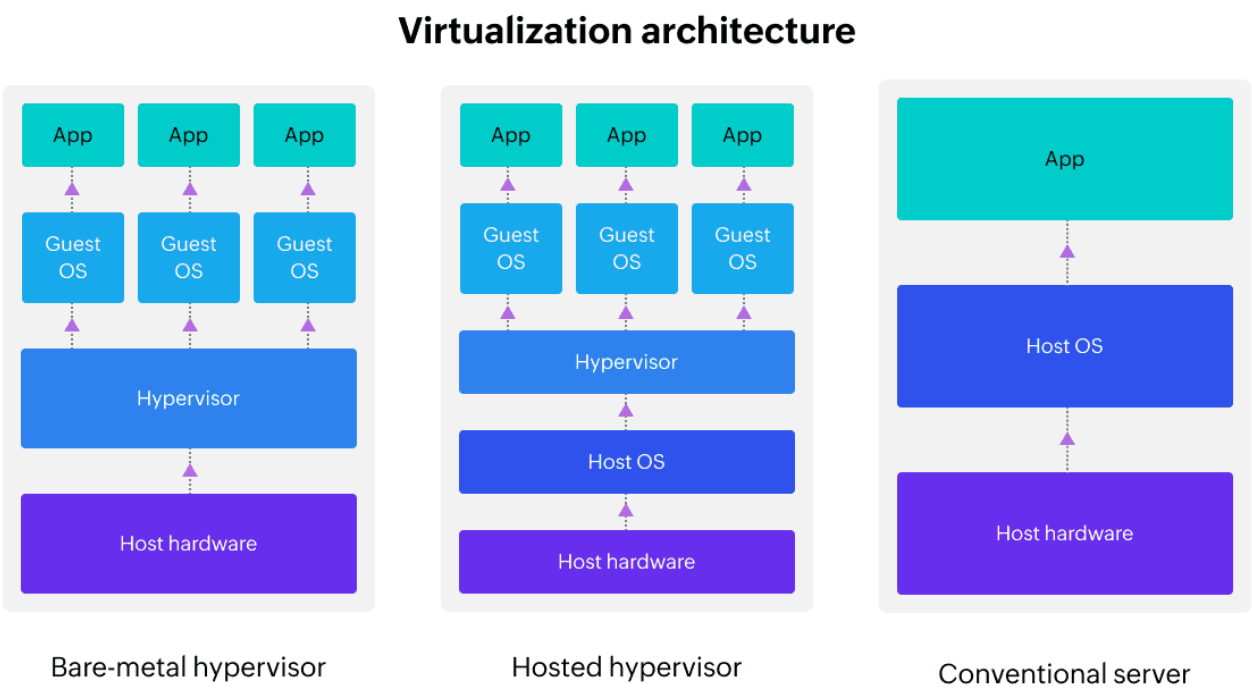
Prerequisites

1. Install Docker desktop and engine

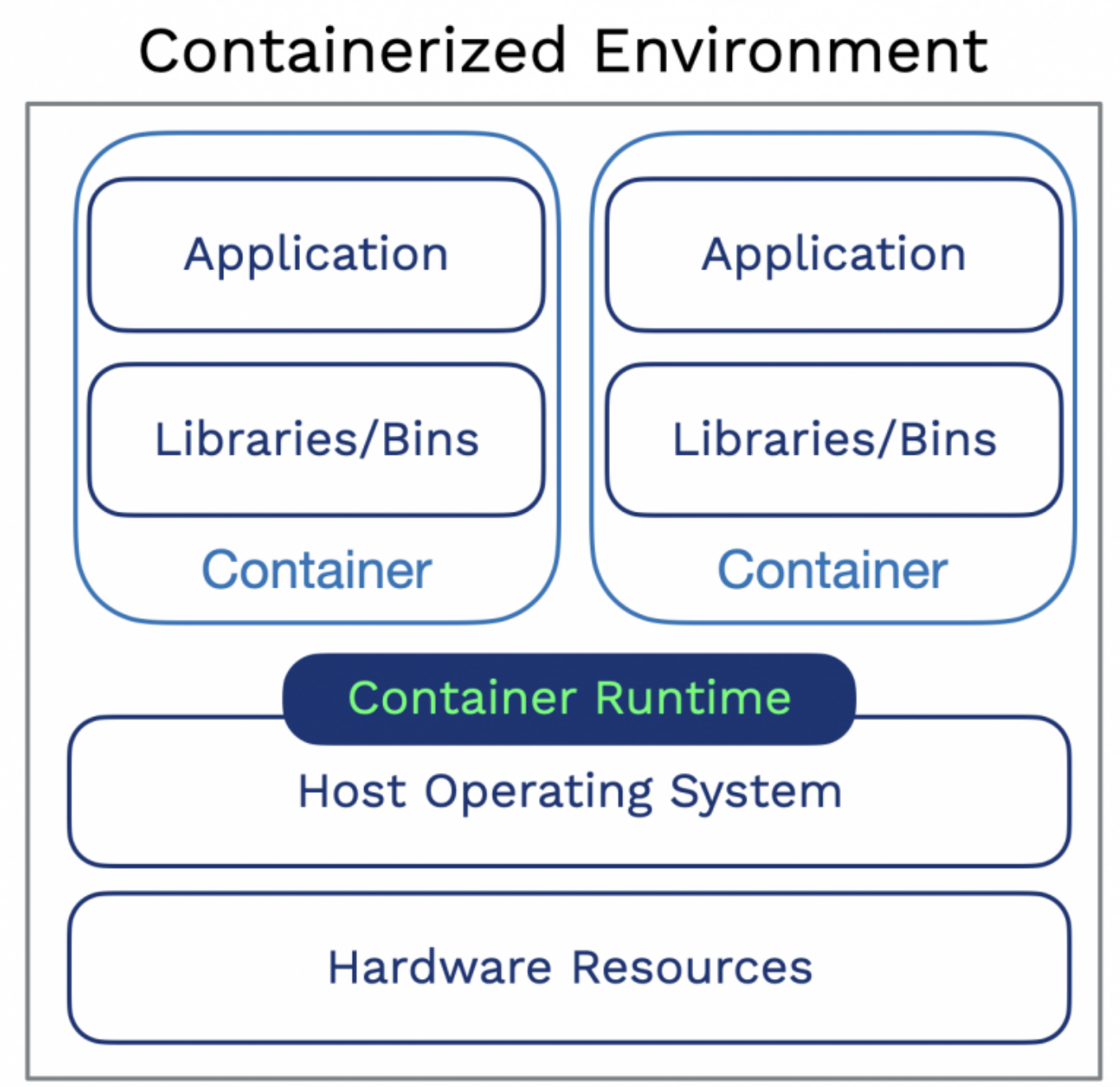
- Install for Windows
- Install for Linux
- Install for Mac

A note for Windows users: To run Linux Docker containers on Windows 10 or 11 64-bit Home version, you will need to install and setup wsl version 2 with your preferred Linux distribution of choice and enable the ...

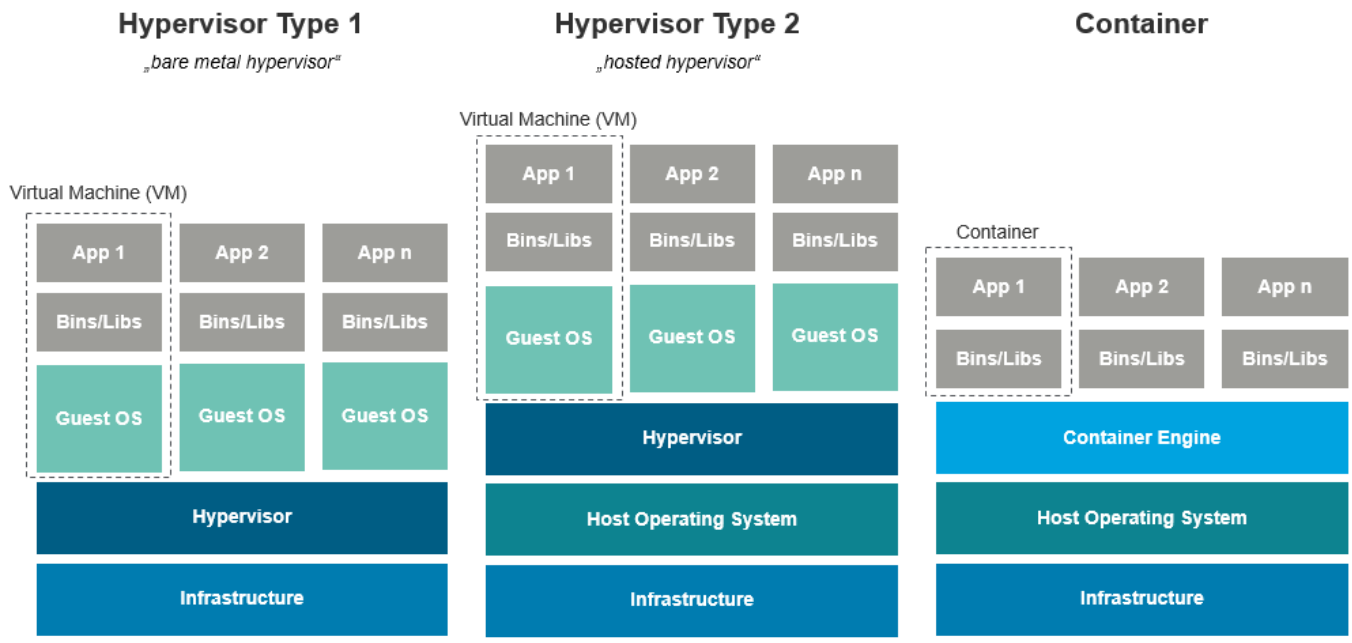
Virtualisation



Containerisation



VMs vs Containers



So what exactly is Docker?

Docker images vs containers

Let us try and understand this difference with the example of a Panner Sandwich from the cafeteria on the fourth floor:

- We shall assume that the cooks at the cafeteria follow a recipe with a set of instructions that must be followed in order, and the list of ingredients required to prepare that tasty sandwich for us.
- Based on this information, the cafetaria staff put together all the ingredients required to prepare the sandwich in the kitchen. They first check if the ingredients are already present in their kitchen and proceed to buy those that aren't available.
- The cooks working at the cafetaria then follow the instructions, one by one to prepare a product one layer a time, in our case the tasty Paneer Sandwich, that can be eaten and savoured by us.
- But wait! This sandwich is still lying in the kitchen and is yet to be served to us. We cannot touch or eat it and even though it has been prepared and looks super delicious, it is just on display.
- So we ask one of the friendly staff there to serve it to us on one of the cardboard plates. Once it is served, we then have the option of adding some extra mayonnaise or removing a vegetable we do not like.

The recipe in this example is a Dockerfile (more on this later), which is followed, layer by layer, to prepare or build the Paneer Sanwich, which is analogous to a Docker image. When we ask the staff to serve the sandwich, they put it on a carboard plate and serve it to us following which we can add our own touch ups and savour it. The sandwich in a plate which is served when asked to so that we can finally enjoy our sandwich, is analogous to a Docker container, which is an isolated running instance of a Docker image. Hence we have multiple plates of Paneer Sandwiches of the same kind and template prepared using the same recipe.

If that analogy didn't do the trick for you, you can think of images as classes from an object oriented language, where as a container is an instance of that class and we can create many such instances based off of the same class.

Now, lets take a look at a more formal and elaborate explanation:

A Docker image is a read-only, executable template containing a set of instructions, libraries, dependencies, configurations, files and source code that are required to create a runtime instance or a container environment where they are packaged together. It can be thought of as a snapshot that references a set of read-only or immutable layers that are stacked on top of each other to form the base for a container's root file system. Taking the example of our Paneer Sandwich, the layers of bread and the Paneer inside of it can be thought of as layers of the image that are put together one by one when it is prepared. Docker images can be shared and stored in multiple locations and can be pulled from/ pushed to a public or private repository (The Docker Hub Registry for example).

A Docker container is a lightweight, isolated, running instance of an image. Docker images become containers when they run on the Docker engine. The instructions and code, dependencies and libraries that were packaged up as read-only layers during the build process of the image form the base of the containers file system as mentioned earlier. Docker containers use these images and create a thin, top writeable layer to perform modifications to the file system and data. The addition of extra mayonnaise on the side of the cardboard plate for our Paneer Sandwich could be thought of as an action in the writeable layer of the container. Now we can run commands in this container, assign and bind it to a port and perform other actions as opposed to a Docker image which is just a snapshot.

Getting our hands dirty

Let us explore the Docker Hub Registry and pull our first Docker image.

- Go to Docker Hub. This is a popular cloud based container registry where developers can find and share container images.
- Search for 'Ubuntu' in the search bar and click on the Ubuntu image.
- The `tags` field specifies human readable identifiers for specific version or variants of an image.
- Following the command provided on the page, let us run `docker pull ubuntu` on our host.
- We can verify that the image was build successfully by using the `docker images` command to list all the Docker images on our local respository.
- Next, we spin up a container from this Ubuntu image by running `docker run -it --name ubuntu_container ubuntu`. The -it flags combine the "interactive" and "pesudo-tty" options to provide us with an interactive terminal emulation within the container which will then allow us to interact with its processes.
- That was so simple wasn't it? We were able to set up an Ubuntu environment on our host machine in just a few minutes.
- Now, exit the container and run `docker stop <container_id_or_container_name>` to stop the container and `docker start` to start it again. Note that in the case of no-name containers, providing just the first few unique characters of the container ID will suffice.
- Run the `docker ps -a` command to list all containers on your machine regardless of their state.
- We can now easily spin up multiple containers from the Ubuntu image using the `docker run` command.
- We can also use the `docker exec <container_name_or_id>` command to run a new command inside the specified container.
- We can also just run `docker run` without running `docker pull` and Docker will pull the image from the registry service as it will not be able to find the image on the local respository.

Can we build our own images?

YES! We build or own images and create containers based off of them. We are going to explore this now with a simple NodeJS example

To start off, create a directory for this example with the `mkdir` command and switch directories using the `cd` command.

Once you are inside the project directory, run npm init -y to initialize a new project and create the package.json file. Then run the following command `npm install express`

Now open the project on your preferred editor of choice and create a file called `index.js`.

Type in the following code inside `index.js`:

```
const app = require("express")();

app.get("/", (req, res) => {
  res.json({ Page: "This is the homepage" });
});

app.get("/first", (req, res) => {
  res.json({ Page: "This is the first page." });
});

app.get("/second", (req, res) => {
  res.json({ Page: "This is the second page." });
});

app.all("*", (req, res) => {
  res.status(404).send("Oops! Cannot find that page");
});

app.listen(3000, () => {
  console.log("Listening on port 3000");
});
```

Now, to make sure this works locally, type in `node index.js` in the terminal inside the project directory. Then either visit `localhost:3000` on a browser or run `curl localhost:3000` on the command line.

Once we have confirmed that the app works fine locally, we will proceed towards containerizing this app by creating our own custom image. To do this, go ahead and create a file called `Dockerfile` in the project directory.

A Dockerfile is a text file inside which we list instructions for the docker daemon to follow when building Docker images. Each layer of a Docker image can be thought of as the byproduct of the execution of an instruction inside this file. This is basically the recipe the cooks working in the cafeteria use to prepare our Paneer Sandwich.

Inside the `Dockerfile`, type in the following:

```
FROM node:latest
WORKDIR /app
COPY . /app
RUN npm install
CMD ["node","index.js"]
EXPOSE 3000
```

Let's talk about what each line in this code does:

- FROM `<base_image:tag>`: This line specifies a base image to build our custom image from. It does the work of building the node environment for us.
- WORKDIR `<directory>`: This directive specifies the working directory for the subsequent instructions in the Dockerfile and inside the containers that we spin off of the image built.
- COPY `<path_to_copy_from> <path_to_copy_to>`: This line copies the contents of directory to the specified directory while building the image and inside the container. In this case we copy everything from our current directory to the working directory `/app` inside our container.
- RUN `<command>`: Here we specify the command to run while building the image and the results of this command will be reflected inside of any container we spin up from this image.
- CMD `<command>`: The default command to run when we start ant container that we spin up from our custom image.
- EXPOSE `<port>`: This line informs docker that all derived containers listen on the specified network port at runtime.

Then, create a `.dockerignore` file in the same directory that excludes certain files/directories from the built image and its containers. Before the build context for an image is sent to the Docker daemon, it searches for this file in the root directory of the build context so it knows which paths to ignore during the build process. This helps in reducing the size of containers. It is pretty similar to the `.gitignore` file for Git.

Inside the `.dockerignore` file, type in the following:

```
node_modules
Dockerfile
```

We do this to ensure `node_modules` is not unnecessarily copied each time we build the image and so that we can hide the Dockerfile from others building our image.

Once that is done, run the following command in the project directory: `docker build -t <image_name>:<image_tag> .`

Where we specify the image tag after the `-t` flag and the directory inside which our Dockerfile resides to build our custom image.

We can then check if the image was created by running `docker images` to list all images present on our machine or the specified repository.

To spin up a container form this image, we need to run the command `docker run --name <container_name> -p <host_port>:<container_port> <image_name>:<image_tag>` which tells docker

what the container name should be and which network port of the host machine should point to the specified network port of the container and what image should be used to spin up the container.

Then either visit `localhost:<host_port>` on a browser or run `curl localhost:<host_port>` on the command line and verify that you have successfully containerized and run your own application.

But let us try something else here. In the index.js file add another route with the following code:

```
app.get("/third", (req, res) => {
  res.json({ Page: "This is the third page." });
});
```

Now, check the result on `localhost:<host_port>`. You will notice that there is no change and we get the 404 message when we try to view `localhost:3000/third`. This is because Docker images are read-only and once built they are locked in and their contents cannot be changed. So to view these changes, build the image again using the same command as before with a new name. We can see that the image is being built with the exact same steps as before and `npm install` runs again to install packages even though we did not install any new packages. This is unnecessary and inefficient.

All or most instructions in a Dockerfile create a layer in the build process of an image and these layers are shared and cacheable. This means that if nothing has been changed in the context of a certain instruction, that layer does not have to rebuilt and the cached result of previous build can be reused. The instructions in a Dockerfile are executed sequentially from top to bottom and this is the order in which layers are built. Therefore, only the instruction for which the context has been modified, and subsequent instructions, need to be executed again and hence only those layers need to rebuilt. The layers before this point can reuse their respective caches.

To exploit this feature and optimize our build, we can rewrite our Dockerfile like this:

```
FROM node:21-alpine3.19
WORKDIR /app
COPY package*.json /app
RUN npm install
COPY . /app
CMD ["node","index.js"]
EXPOSE 3000
```

This ensures that we do not unnecessarily reinstall packages each time we build the image making the build process quicker and more efficient. We also specifiy a tag/version for our node environment instead of the "latest" tag to ensure that our app remains stable regardless of any updates to the base image.