**Docker and Containerizations**

**Overall Course Objectives**

* Understand what Docker is
* Learn how to create Docker Images
* Understand how to map ports between Docker and the Host OS
* Understand the basics of Docker networking using Docker Services
* Understand Mounts and Volumes

**Overview**

In the video, the following topics are discussed:

* What is Docker?
* How deployment was done in the past
* Virtual Machines
* What is an image?
* What is a container?
* The role of VS Code

**Goal of the course**

Goal: By the end of this course, you will know how to use Docker and Compose on your machine for better software development and testing. You will gain the skills to build development environments with your code running in containers.

**Pre-requisites**

This course assumes you know:

* VS Code
* How to build apps using Express in MVC
* ES6

**What you'll learn in the course**

In this course you'll learn about the following:

1. How to use the Docker to containerize your apps
2. How to use docker-compose.yml to create services and have each service communicate with each other.

**Objectives:**

1. Learn what Docker is
2. Learn what Virtual Machine is
3. Understand the difference between Virtual Servers and Physical Servers
4. Learn what containerization tries to solve

**Docker**

In this demo video, the following topics are discussed:

* The role of packages.json
* What is a Docker build file?
* docker-compose.yml file and ports (Don't worry about this just yet. You'll learn this later)
* Services

Docker is a software development platform that makes it easier for software developers to develop and deploy apps inside "containerised" environments. It is a tool to develop software in a more collaborative, easy, hassle-free and flexible way.

**Physical Servers**

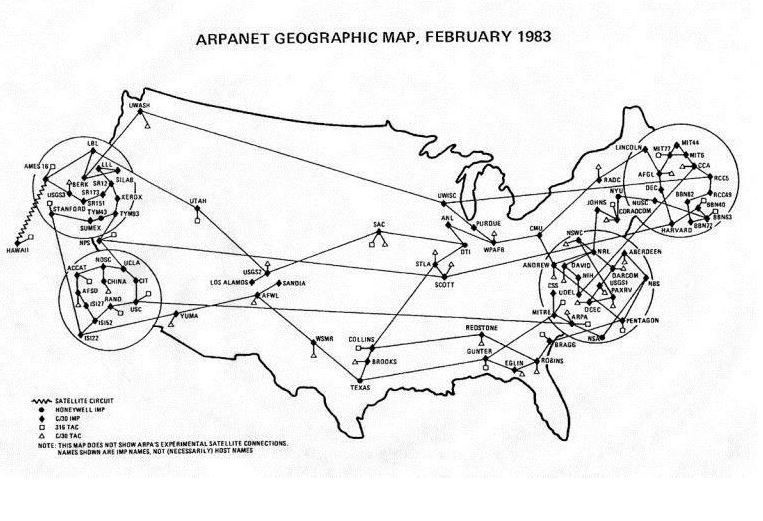


In the olden days, there were no such thing as a "virtual" server. They were all physical. Physical server is literally like a desktop computer with basic components like processor, storage, memory, OS, etc. Fundamentally they're not that different. The main difference  is that server grade hardware is much more expensive, and they are optimized for the environment in which a server will operate.

For example, a consumer and enterprise grade hardware might share much of the same technology, but a server hard drive is much more resistant to wear and tear from vibration. Something that doesn't matter for a drive that's sitting in the desktop by itself but it is critical for a drive that's surrounded on all sides by other drives that's spinning and clicking away doing their own work. And there are so many little things like this.

*Row of hard drives mounted in a rack*

In the days before Google, Facebook, and TikTok, this is what connectivity looked like:



In 1983, the world wide web was in it's infancy, but many of the core technologies that provide the building blocks for today's internet were already in place. But there's this one main problem in how things were done before: a "smart" way in managing **resources.**



Imagine you own a hotel with only a single, giant suite that most of the space goes unused when you rent it out. Usually there's only one person in there, and it's extremely rare for a group of 50 people to rent the place and fully use up the available space. A weird hotel, I know. Because a normal hotel splits the building up into smaller units so that it can be used by more guests all the time. It's efficient and it's profitable this way.

Physical servers that don't use virtualization are like the single-suite hotel... A lot of resources sit idle most of the time. It is very rare for an application to consume most of a server's computing power, it may only even happen once a year.

**Virtual Servers**

In our hotel example, virtual machine technology solved the problem of wasted resources by dividing the building into many smaller rooms. Similarly, virtualization allows you to assign a computer's resources (including processors, memory and storage) into smaller units called virtual machines (VMs). Each VM is capable of hosting a *guest*operating system and applications by **isolating system resources** from other software on the computer.

Virtual Machine technology, like VirtualBox, is a software that you install on your computer that will then enable your computer to virtualize or emulate a computer system (like Windows and Linux). This allows you to run multiple VMs on a single computer.

**Amazon Elastic Compute Cloud**is like the "hotel" for software. It is part of Amazon's cloud computing platform that allows users to to rent virtual computers on which to run their own computer applications.

But there's just this one big problem about VMs: **bloated** **resources.**

When a user rents a virtual computer from Amazon, that user will be allocated with X amount of resources (X of RAM, storage, CPU, etc.). In order to run your Node.js app, for example, you need a Linux VM. And this is one of the things about Virtual Machines, even though we're working with a very lightweight Node.js application, **we have** to put that guest OS in there and that just simply bloats it. The smallest VM with a running Node.js application is over 400MB. Whereas the Node.js runtime app itself would only be under 15MB. Why do we need 400MB to run a 15MB Node.js application, right?

Next, when we want to scale this out, we can do either of the two (or both):

1. Increase your current VM's resource allocation. Example: increase RAM from 2GB to 4GB
2. Have a separate VM to host that same app.

#2 is typically the scale option when #1 option is maxed out. With option #2, you'll notice that even though it's the exact same application, we have to use and deploy that separate guest OS and libraries every time. There's another thing that I haven't mentioned here: This Node.js application was developed on a Windows. So when you push this into Amazon's virtual computer (which is typically Linux OS), there's a big chance for it to not work because of **compatibility issues**.

**Containers**

All of the problems mentioned so far are solved when you use something like containers. Containers run a very similar setup to the VMs, but it's going to be much more lightweight because it doesn't have a *guest*OS like VMs have. When you scale out your app, and because we don't have to duplicate all of those OS dependencies and create bloated VMs, we actually will use **less resources.**

The great thing about container technology, the remaining resources becomes shared between all the running processes. In fact, another advantage: If other container processes aren't actually utilizing the CPU or memory, all of those shared resources become accessible for the other containers running within that hardware.

## Objectives

* To be able to successfully install Docker on a Mac
* To be able to successfully install the Docker extension on VS Code

# Install Docker (Mac)

Docker is easy to install. I can hand you a docker image, you can run it, and we can have the same exact container running on our machine. That means we can do collaboration a lot easier. Multiple devs can work on the same app without having to worry about dependencies, settings, configurations, etc. And Docker containers are very flexible. You can build a container with whatever you want. You decide what you want to put in it.

To install Docker, just click on this link and follow the installation steps: <https://docs.docker.com/get-docker/>

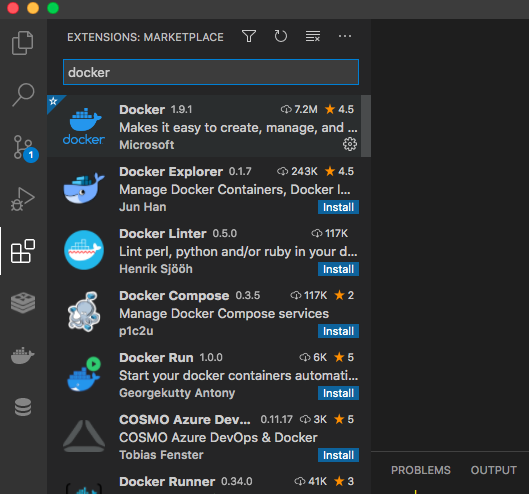
**Important:** Once Docker is installed, you should see the Docker icon in the Task Bar (for Windows) or Menu Bar (for Mac). It's important that you keep the Docker server running in order for you to create and launch Docker images and containers. If it's not running, just simply launch Docker by double clicking on the Docker icon on your Desktop.

You also NEED to sign up with Docker (if you haven't yet) and then sign in to your Docker Desktop instance. Docker Desktop will run even if you're signed out. But since we will be using images from Docker Hub, we need to be signed in.

### Docker VS Code Extension

And it's pretty easy to install the Docker extension for VS Code, so go ahead and install that extension now. Open your VS Code, go to Extensions tab, type in docker and choose the Docker extension by Microsoft. (see screenshot below):

Note: The Docker VS Code extension is just a neat extension for you to easily manage your Docker images, containers, volumes, etc. You still need to install Docker Desktop in order for you to containerize or dockerize your apps.



## Objectives

* To be able to successfully install Docker on a Windows computer
* To be able to successfully install Docker extension on VS Code

# Install Docker (Windows)

Docker is easy to install. I can hand you a docker image, you can run it, and we can have the same exact container running on our machine. That means we can do collaboration a lot easier. Multiple devs can work on the same app without having to worry about dependencies, settings, configurations, etc. And Docker containers are very flexible. You can build a container with whatever you want. You decide what you want to put in it.

To install Docker, just click on this link and follow the installation steps: <https://docs.docker.com/get-docker/>

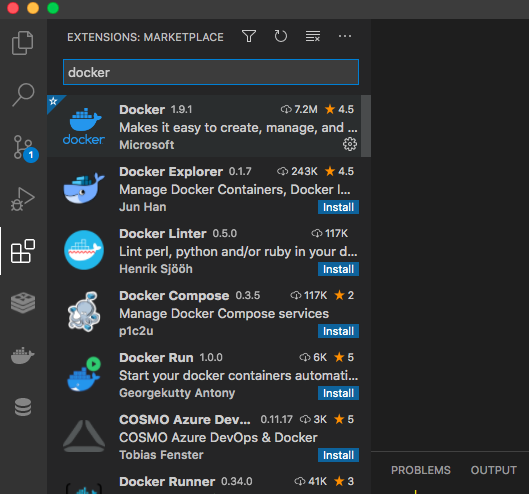
**Important**: Once Docker is installed, you should see the Docker icon in the Task Bar (for Windows) or Menu Bar (for Mac). It's important that you keep the Docker server running in order for you to create and launch Docker images and containers. If it's not running, just simply launch Docker by double clicking on the Docker icon on your Desktop.

You also **NEED** to sign up with Docker (if you haven't yet) and then sign in to your Docker Desktop instance. Docker Desktop will run even if you're signed out. But since we will be using images from Docker Hub, we need to be signed in.

### Docker VS Code Extension

And it's pretty easy to install the Docker extension for VS Code, so go ahead and install that extension now. Open your VS Code, go to Extensions tab, type in docker and choose the Docker extension by Microsoft. (see screenshot below):

Note: The Docker VS Code extension is just a neat extension for you to easily manage your Docker images, containers, volumes, etc. You still need to install Docker Desktop in order for you to containerize or dockerize your apps.



## Extra Installation Steps for Windows

If the above installation instruction didn't work for you, follow the extra installation steps below:

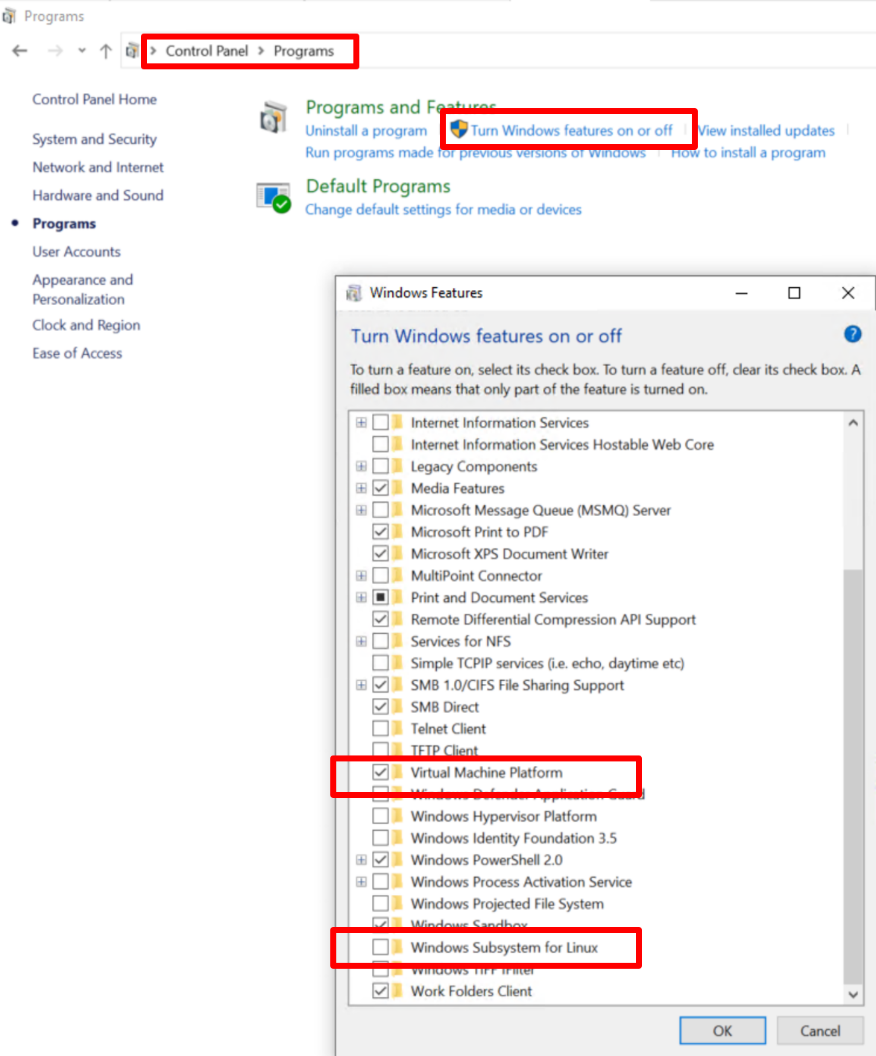
A lot of Windows users are having a hard time running Docker properly on their end, and that's mainly because of how Windows file system differs from Linux. The current official solution is for you to install Ubuntu on top of your Windows and then have Docker just basically communicate with your Ubuntu's filesystem, instead of it having to deal Windows.

For this to work, go ahead and visit this [link](https://docs.microsoft.com/en-us/windows/wsl/install-win10#manual-installation-steps) and make sure you follow **steps 4, 5 & 6**. For step 6, make sure you choose **Ubuntu 18.04**. Then make sure you **restart** your computer when you're done with the steps.

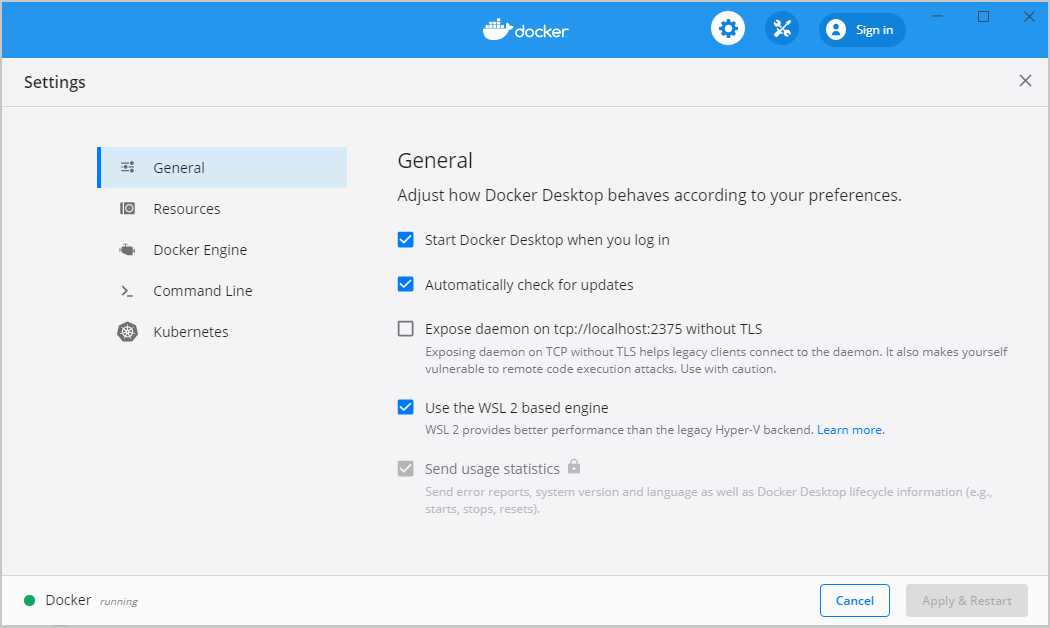
Next, open up a Power Shell in Administrator mode:

1. Click on Start, type in **Power Shell**, right click on the Power Shell Icon and choose **Run as Administrator.**
2. Run this command in power shell: **bcdedit /set hypervisorlaunchtype auto** and press Enter.
3. Restart your computer

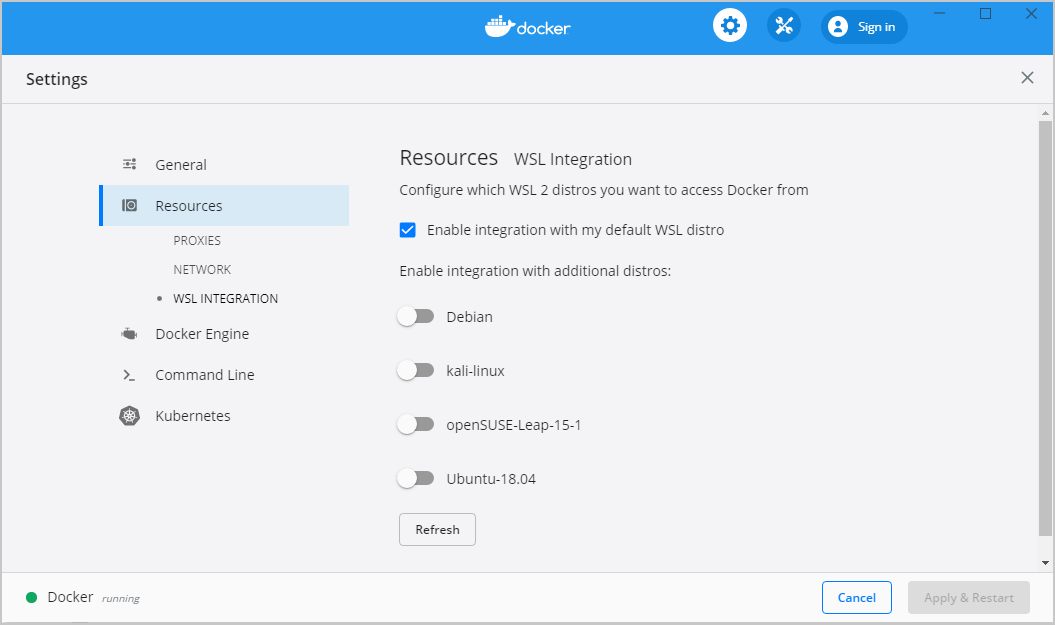
Next, open up Control Panel and select **Programs**, then **Turn Windows features on or off**, and then make sure **Virtual Machine Platform** and **Windows Subsystem for Linux** are selected (See image below). Then just restart your computer.



Next, open Docker Desktop. From the Docker menu, select **Settings > General**



Select the **Use WSL 2 based engine** check box. Then click **Apply & Restart**. When Docker Desktop restarts, go to **Settings > Resources > WSL Integration**.

Make sure **Ubuntu-18.04** is enabled and **Enable integration with my default WSL distro** is checked. Then click **Apply & Restart.**

**VERY IMPORTANT:**

Since you are using Ubuntu on top of your Windows to run Docker, you will need to have a very different way to launch your VS Code. You will also need to save your files in a specific location so that "volume mounting" will work (A concept you will learn later).

1. Make sure you save your projects in Ubuntu's **/home/[yourname]**directory. To locate Ubuntu's directory, click **Start**then type **\\wsl$**and hit enter. A new windows explorer should pop up and will contain 3 folders. Just double click on the **Ubuntu**folder. From there you will see a **home**folder, and in the home folder, you will see your **name.** This is where you should paste your projects. Paste them in **/home/[yourname]**directory. Example: **/home/oliver/demo\_docker2**
2. Finally, instead of openning VS Code directly from your Desktop, open VS Code using a command prompt
   1. Open a command prompt and type in **wsl**and press enter. Your command prompt will switch from a windows cli, to a linux terminal
   2. You will then be able to cd in to **/home/[yourname]**directory. Example: **cd /home/oliver/demo\_docker2**
   3. Then type in **code .**and press enter. The (dot) symbol is included.
   4. VS Code will open, but instead of a Windows VS Code, it will open a VS Code from Ubuntu. You will also need to install the Docker extension for this VS Code.
   5. Finally, to build and run your container, just right click on docker-compose.yml and click **Compose Up**

**Objectives**

* Understand what a Dockerfile is
* Learn the basic Docker instructions in the Dockerfile
* Learn the difference between an Image and a Container
* Learn how to build a Docker Image
* Learn how to run a Docker Container
* Learn the difference between *local*and *guest*files.

**Dockerfile**

Dockerfile is a special file name that you put in your project folder in order for you to create a **customized** docker image.

Note: **Image**is the term used to refer to the "blue print" of your container. An image is a set of instructions on how to build your container. **Container**is the term used to describe a running instance of an image. These terms are often used when you work with Docker. Just think of image like a Class in programming. A class contains code that describes what an Object is. Basically, an image is just like that. By comparison, an image is a set of instructions that describes what the container is.

By convention, a Dockerfile is saved in the root directory of your project, but you can actually save it anywhere within your project. The Dockerfile contains sets of instructions so that when you run a Docker build command, Docker will then go to each line of instructions you have in your Dockerfile and build the image according to the instructions. That Dockerfile then can be shared across other developers of the team through your Github repository. This means that other developers (that has Docker installed) can then run the Docker build command to build the identical docker image and run the identical docker container.

In this demo, let's pretend that you are building this awesome app from scratch. And you are going to use Express JS as your node framework to build this app. [Download this basic express server](https://github.com/HHOliver88/docker_demo) and we'll use it as the initial files for our app. Drag this demo app to your VS Code workspace now.

As you can see, this project has an empty Dockerfile. We want to dockerize this project so that when other developers jump in, they won't have a hard time running your app on their end. So let's go ahead and add instructions to our Dockerfile:

FROM node:14.4.0-alpine3.12

RUN mkdir -p /var/www/app

WORKDIR /var/www/app

COPY . .

RUN npm install

RUN npm install nodemon -g

EXPOSE 3000

CMD ["npm", "start"]

At the very top of our Dockerfile we have a **FROM** statement. FROM must be the first instruction in a Dockerfile. The FROM instruction sets the base image for our docker image. You can browse through other baseImages from the [hub.docker.com](http://hub.docker.com/) website, but for now just know that we are using node as our runtime that has a version of 14.4.0 that has Alpine Linux as base image baked in to it.

Next, we have the ***RUN mkdir -p /var/www/app***instruction. This just tells docker that we want to make a /var/www/app directory. Next we tell docker that the */var/www/app*should be the **WORKDIR**. WORKDIR or work directory is going to be the directory that our local express app files will be saved to in the container.

**Important:** Know that the instructions above are read by docker from top to bottom and that it happens in the "guest" container, not in the "host" computer. Example: CMD ["npm", "start"] is a command to start a node server. But it's not going to run that command in your own OS, it's going to run that command in the container itself.

**What do you mean "local" files?**

Local just means your own computer, or your own operating system. The other term for it is "host". Guest files are files that are saved in your Docker container.

Docker maps your "host" project files (Express app) to your work directory inside your container (Something that can be done when you learn docker-compose.yml). This means whatever changes you make to your host copy of your express app, it will automatically reflect those changes to the express app copy inside your docker container.

**Okay, let's continue!**

Next, we have a weird docker instruction: **COPY . .** The dot symbol on the left just tells docker that whatever files and folders we have in the host's current directory, should be copied and pasted over to the guest's work directory. The dot symbol on the right is just a shortcut name for the */var/www/app*directory (Which we specified as our WORKDIR).

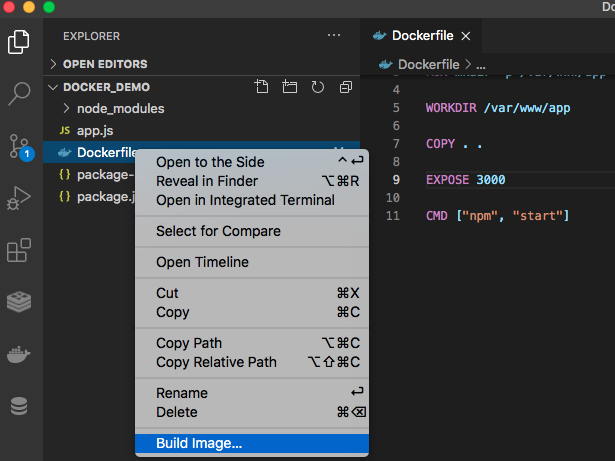
Next, we have **npm install**to install the dependencies directly in the container. Then we install **nodemon**globally (specified by the -g flag) in order for us to run nodemon command in the container.

Next, we have **EXPOSE 3000**. This just tells Docker, "This container should listen on port 3000". Which is the same port number our express app listens on.

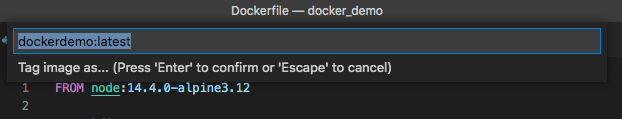
And finally, we have a Docker instruction **CMD ["npm", "start"].**This just basically runs an npm command in your container so that whenever you run your container, it runs a node server automatically as it starts up. If you look at your package.json file, you will see under *scripts* that **start**just basically runs "nodemon app.js".

Now, after you typed in the Docker instructions above to your Dockerfile, let's go ahead and build our first docker image. Just right click on the Dockerfile, and choose **Build Image...**

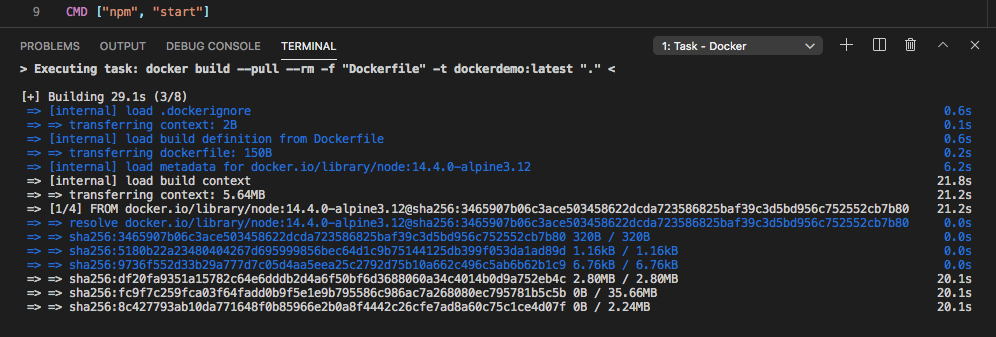
**IMPORTANT**: For **Windows Users using WSL2,** refer to **Install Docker (Windows)** to properly build and run a Dockerfile.



It will then ask you to **Tag image as...**Just **press Enter** to tag image as the default tag name (which is *latest*).

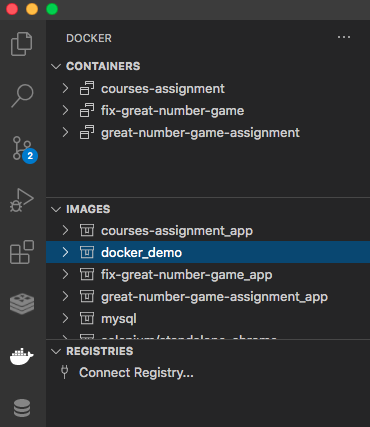


You should see a similar output from the terminal below:

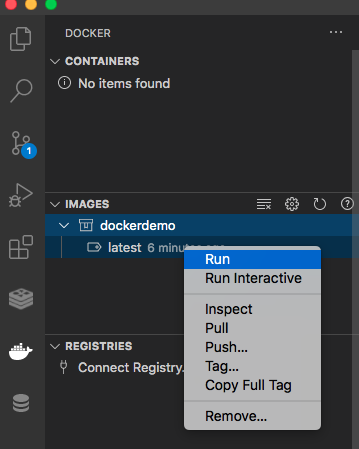


It runs a **docker build**command behind the scenes ( builds a docker image called *dockerdemo.*)It should probably take a while when you first build an image because it's going to download all the necessary files from Docker's repo. But once it's downloaded, it would take you just a few seconds when you build your image again.

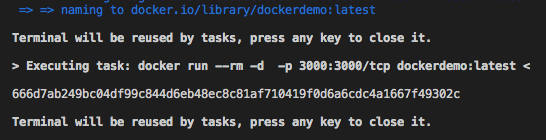
After it's done building your Docker image, go to your Docker extension in VS Code. You will see that dockerdemo is now listed as one of the available docker images in your Docker.



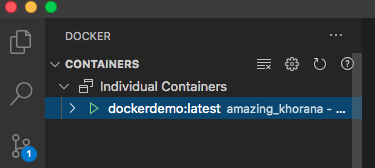
To run this docker image, you just need to expand your *dockerdemo* image, right click on the *latest* tag, and choose **Run**



You should see a similar output from the terminal below:



It executes a **docker run**command behind the scenes. When you go to your Docker extension in VS Code, you will see that **dockerdemo** is now listed as one of the running docker containers. It will have a green play icon beside the container name to tell you that it's running (And red stop icon for stopped containers).



Now, let's see our app in the browser. Visit ***localhost:3000***and you should see a welcome message saying Hello Docker. Congratulations!

Note: To stop the currently running container, just go to Docker extension, right click the container, and choose **Stop.**

**Dockerize Any Apps**

You just learned how to dockerize a node.js web application. But technically, you can dockerize apps that use Ruby, PHP, Java, C# or any other languages that you like!

Sure, some Dockerfile configurations/instructions might be different from what you will be using here, but the idea is the same:

1. You grab a preconfigured image from dockerhub (which could be of any language: PHP, Ruby, Python, etc.)
2. You customize how that image is going to be built (specify WORKDIR, expose which ports, run package manager commands, etc.)

Here are some good resources you can read through if you want to learn how to dockerize other apps:

* PHP - <https://dev.to/concatly/how-to-deploy-a-dockerize-a-basic-php-application-4756>
* Django - <https://code.visualstudio.com/docs/containers/quickstart-python>
* Ruby on Rails - <https://docs.docker.com/compose/rails/>
* ASP.net - <https://docs.docker.com/engine/examples/dotnetcore/>
* Spring MVC - <https://medium.com/@asce4s/dockerize-spring-mvc-application-a9ffbd11eadb>

**Dockerize Project**

Download the assignment files [here](https://github.com/HHOliver88/docker_project).

**Instructions:**

1. The Dockerfile provided is blank. Create a customised Docker image using that Dockerfile.
2. It should automatically run the node server when you run the **docker build** command.
3. The project should be accessible in the browser via port 8080.

**Hint:**

* Focus your attention to these files: **server.js, package.json,**and **Dockerfile**files. They are the primary files required to successfully run a simple docker container with a running node server. Some important parts in these files were **intentionally removed/altered.**Find which of those parts are missing/changed so that you can successfully build and run this project in a docker container.

START WORKING ON THIS

**Objectives**

* Understand how docker-compose file can be used
* Learn how to take advantage of both a Dockerfile and a docker-compose file
* Learn how to write basic docker-compose instructions
* Understand each line of instructions in the docker-compose
* Learn how port mapping works
* Understand why we have to arrange our folder structure in a very specific way
* Understand why we don't want node\_modules from *local*be copied over to *guest*container.
* Understand the concept of "mounting"

**docker-compose.yml**

**Important**: containers and services are terms that are used interchangeably throughout the course. Technically they are the same. It's just we use the term service in context with larger applications. Examples of services might include an HTTP server, a database, or any other type of executable program that you wish to run in a distributed environment.

How do we put multiple containers, and have them talk to each other? How can we have our node server talk to MySQL and vice versa? How can we set up our Docker container so that it "listens" to our code changes in local and apply those changes to our code in the container? That's where docker-compose comes in.

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application’s services. Then, with just a few mouse clicks, you create and start all the services from your configuration. Compose works in all environments: production, staging, development, testing, as well as Continuous Integration workflows.

Using Compose is basically a three-step process (Note: No need to do any of the steps below for now, a sample app will be provided):

1. Define your app’s environment with a Dockerfile so it can be reproduced anywhere.
2. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
3. Run docker-compose.yml using VS Code and Docker extension.

This means that Dockerfile works together with the docker-compose.yml file. Again, you can save these files anywhere in your project directory, but the convention is to save both in the root directory of your project.

A docker-compose.yml looks like this (Note: No need to copy the code below for now, sample code will be provided below):

version: "3.2"

services:

web\_app:

build: ./

restart: always

command: npm start

volumes:

- ./src:/var/www/app/src

- ./package.json:/var/www/app/package.json

ports:

- 3001:3000

Let's discuss each line of instructions.

**version: “3.2”** is just the docker compose version. Docker has changed a lot in the last few years and they keep changing how the compose files have to be written. This doesn’t relate to the docker that you have installed, this is just the version of the docker compose **file format** that you wish to write in. The latest version as of this writing is 3.8, and there’s subtle differences between each version.

**services**like **web\_app**is just a name that can be anything you want. Under **services** you can see that we have **web\_app**, but you can add other services like mysql service, redis service, selenium service, etc., and you can give it a service name to whatever you want.

**build: ./**just tells Docker that we are going to use a customized image for this web\_app service. In this case, it's going to use a **Dockerfile** that is saved in the project's current root directory in your local computer.

**restart: always**Since our node app is "public serving", we want it to always be up as much as possible. So if anything happens to our node, we just want it to automatically restart.

**command** is like the **CMD** instruction from Dockerfile. This command just runs our node server.

**volumes.**You can have as many as you want, and the format is:

**dash**, space, then the **directory of the host (or file)**, **colon**, and **where we want to mount the directory.**

**volumes**just tells docker that the work directory in the container (which is **/var/www/app/src**) should "listen" to code changes from the local computer's project's **src** directory. And it should apply those changes to the container's work directory.

This process is referred to as "mounting". Mounting a volume makes development easier because it’ll see live code changes as they happen.

**Ports.**It’s another list, like **volumes.**You can have as many as you want. Most of the apps that are going to be provided in the TDD course listen to port 3000. So to map something to port 3000, you write:

ports:

    - 3001:3000

Where 3001 could be anything. You could map it to port 3000 (doesn’t matter), but it might clash with a webserver already installed on your operating system or something else using port 3000.

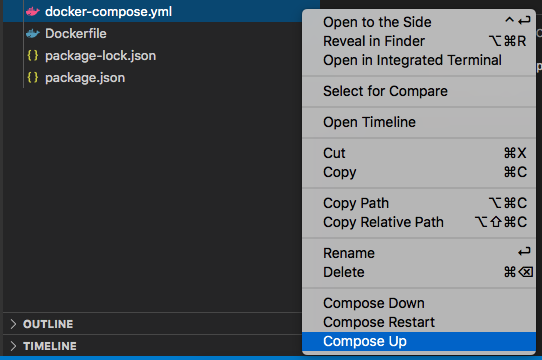
The port on the left is the host port (A public port that our **host**can access to), and the port on the right is the container’s port (A private port that's only accessible by the other services in the docker **network.**). The ports command just maps the host port to the container port. So whenever we access our app in the browser, we type in: [http://localhost:3001](http://localhost:5001/).

**Important**: Remember the **EXPOSE 3000** code we have in Dockerfile? Yeah, **remove that.**Let's have docker-compose.yml file handle port mapping from this point on.

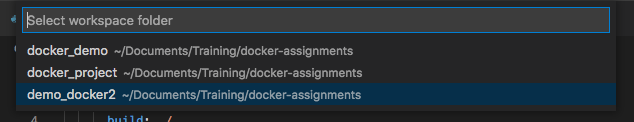
The docker configurations above are all we need.

Now that you learned what docker-compose is, download the [demo\_docker2](https://github.com/HHOliver88/docker_demo2) files, drag it to your VS Code workspace, and test it out on your end. Now, instead of right-clicking Dockerfile, we just right click the docker-compose.yml file and choose **Compose Up**:

**IMPORTANT**: For **Windows Users using WSL2**, refer to **Install Docker (Windows)** to properly build and run a docker-compose file.



It will then ask you to choose which workspace folder you want Compose Up to run, just choose the **demo\_docker2**workspace and press Enter:



To stop this docker container, right click docker-compose.yml and choose **Compose Down**

After successfully running your docker container, visit **localhost:3001**. It should show you a simple **Hello Docker**message.

While the container is still up, make changes to your express app. Change the greeting from Hello Docker to **Hello Docker Compose**, refresh your page and you should see those changes applied in real-time!

**Changes in the Folder Structure:**

You may have noticed that the **app.js**file is now saved in /src folder. You may also have noticed that the **Dockerfile** has been updated. Let's talk about why we have those changes:

# FROM node:latest

FROM node:14.4.0-alpine3.12

RUN mkdir -p /var/www/app

WORKDIR /var/www/app

COPY package\*.json ./

RUN npm install

ENV PATH=/var/www/app/node\_modules/.bin:$PATH

RUN mkdir -p /var/www/app/src

WORKDIR /var/www/app/src

COPY src .

1. We build node:14.4.0
2. We created a directory /var/www/app in the container
3. We tell docker that /var/www/app is our WORKDIR
4. We copied package.json (and package-lock.json if present) from *host* to container's WORKDIR
5. We run npm install in this WORKDIR. Meaning, the node\_modules will reside in /var/www/app folder in the container.
6. **ENV PATH=/var/www/app/node\_modules/.bin:$PATH**- This is just an extra step we need to do in order for us to still run npm commands from the **/var/www/app/src**WORKDIR
7. We created a **src**folder inside **/var/www/app**
8. We tell docker that this is now our WORKDIR
9. We copy whatever is inside the **src**folder (in *host*), and paste them over to the current WORKDIR (/var/www/app/src in the *container*)

From this point on, we will put all our front-end/back-end code in the **/src**folder and put the package.json in our project folder's root. We do this because we only want to *mount*our code, without mounting the **node\_modules**folder.

We just have to run **npm install**in the container, and not in our local computer. Running npm install in our local machine will install dependencies/binaries **THAT MAY ONLY WORK IN OUR LOCAL MACHINE**. If we don't separate our **src**code from **package.json,**then we will run in to this scenario where your **node\_modules**in *host,*will override the node\_modules in container during the *mounting*process in docker-compose.

**Objectives**

* Understand what a Docker Service is
* Learn how to have these services "talk" to each other
* Learn how to access your MySQL service using MySQL Workbench

**Services**

In the previous tab, you learned about docker-compose and how it works with a Dockerfile. In this section, you will learn how to create a docker-compose file with more than one service that we can set up to have one service communicate with the other.

Conveniently, docker compose creates a virtual **network** for all of the containers. So by default, each container can access all of the others defined in the docker compose file **using their service name.**

Let's add a mysql service in this example **(Make sure you read the comments below to learn what each line of instruction does):**

version: "3.2"

services:

db:

image: mysql:5.7 # instructs Docker that the db service is going to use a MySQL image version 5.7

restart: always

      volumes:

# db\_data is a name we set for the /var/lib/mysql path (We can change db\_data to whatever)

# that we can then use to share data between two or more services.

- db\_data:/var/lib/mysql

# this is a way for us to auto IMPORT a .sql file to the database in your MySQL container

# it auto imports the db.sql file in the mysql-dump folder

- ./mysql-dump:/docker-entrypoint-initdb.d

environment: # It creates the MySQL credentials for us, as well as a database called hh. This is the credential you'll need in order for your express app to communicate with the MySQL service.

MYSQL\_ROOT\_PASSWORD: password

MYSQL\_DATABASE: hh

ports:

         # Private port for MySQL is 3306 and Public port is 3307. To connect to our mysql container using Workbench, we use port 3307

- 3307:3306

web\_app:

depends\_on: # This will tell Docker to start services in dependency order. In this example, before starting the web\_app service, it waits for db to be started.

- db

build: ./

      restart: always

command: npm start

volumes:

        # This maps our host project files to our container's work directory. Whatever changes we make to our local copy, will reflect those changes to our work directory copy.

- ./src:/var/www/app/src

        - ./package.json:/var/www/app/package.json

      ports:

        - 3001:3000

volumes: # it means that that specified volume (in this case, db\_data) is available to all services (both app and db).

db\_data: {}

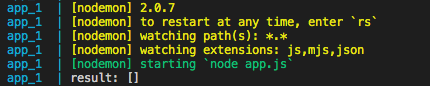
**Important:**scripts in the '/docker-entrypoint-initdb.d' folder is only evaluated the first time the container runs, and if a previous **volume** remains, it won't run the scripts. So if you've made some changes to your .sql script file, just remove the volume to rerun your .sql script. To remove volume, just go to your Docker package, under **VOLUMES,**right click the name of the volume you want to delete, and just choose **Remove...**

Download the [demo\_docker3](https://github.com/HHOliver88/demo_docker3) files and test it out on your end. Spend at max 20 minutes to understand these important files:

* src/app.js
* Dockerfile
* docker-compose.yml
* package.json
* src/config/database.js

**Note**: As you may have noticed, the demo\_docker3 app is not MVC-ready yet. This is intended because we want you to be comfortable with Docker without worrying about how your code is organised. Later, we will provide an empty express project that's MVC-ready that you can use.

If you go to Docker extension, right click on your **demo\_docker3\_app**container, and choose **View Logs**, you should see something like this in the console:



There's a blank array [] at the very bottom, with a label **result.**

That's because we have this line of code in our src/**app.js**:

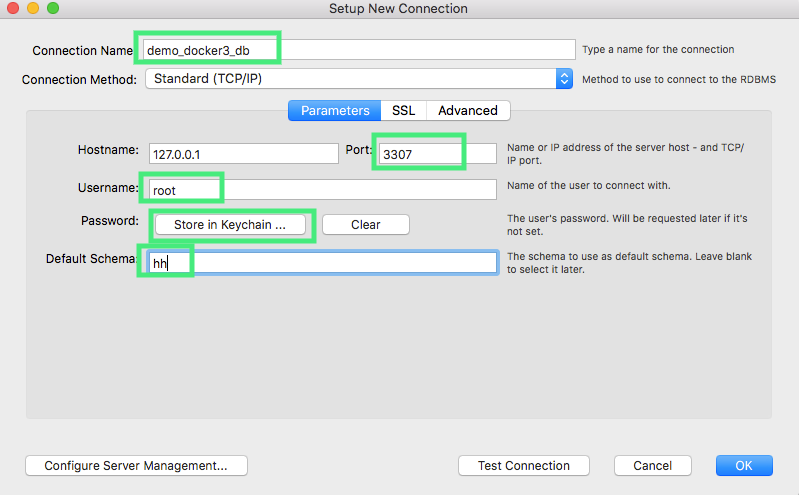
console.log("result:", result);

**Connect to our DB service using MySQL Workbench**

There are two ways we can connect to our MySQL container:

1. Through VS Code shell
2. Through MySQL Workbench

Follow the steps bellow to connect to your **db**service via MySQL Workbench:

1. Open MySQL Workbench and click the + icon.  
   
2. A new window should pop up. Copy the values in the screenshot below:  
     
   **Connection Name -**It could be whatever name  
   **Port -**The port number you specified in docker-compose.yml for your **db**service which is 3307  
   **Username -**root   
   **Password -**password  
   **Default Schema -**The database name you specified in docker-compose.yml for your **db**service which is hh  
     
   Click OK
3. You should be able to see **demo\_docker3\_db**listed in your MySQL Workbench. Click on demo\_docker3\_db connection to connect (Type in your db password if it asks for a password).
4. Now you can work on your **hh**database via MySQL Workbench. Insert a new row in the users table. And confirm that it gets printed in the console. (You may need to refresh your page to update the console.)  
   

**Objectives**

* To familiarize how to Dockerize an Express App with a MySQL Database
* To be more familiar with docker-compose and Dockerfile
* To be more comfortable with the concept of Services, ports, and mounting volumes

**Dockerize Great Number Game (with DB)**

This app is fully working. It generates a random number for you to guess. It will print out "Too high!" if your guess is too high, and print "Too low!" if your guess is too low. And it will print out a congratulation message if you guess the number correctly. It uses MySQL instead of Session (to give you an example docker project that requires MySQL as a service). It is also built using Express that's configured to work as an MVC framework.

Instructions:

1. Take this express app and dockerize it using Dockerfile and docker-compose.yml files: [dockerize\_express](https://github.com/HHOliver88/dockerize-express" \t "_blank)
2. The Dockerfile, and docker-compose.yml files provided are empty. Put in the necessary commands to deploy containers for both the node server and MySQL.
3. Upload your code to Github. Make it private but share access to **HHOliver88**
4. Update the README.md file so that it contains the following:
   1. How another user can setup and deploy your app on their computer (in other words, have setup instructions there)
   2. Include any other information you think is important for another colleague to jump into the project

The goal is to have such a good documentation that other user can set it up and get it working on their end within minutes.

START WORKING ON THIS

**Objectives**

* Learn how and when to set breakpoints
* Learn how to enable VS Code debugger for your containers
* Learn how to write the debugger configurations

**VS Code Debugger**

A Debugger is a tool to help you look at what's going on in your program. Traditionally, we do something like "print debugging", it's a way of checking things out by printing things in the console or terminal via console.log() or var\_dump() if you're familiar with PHP. There's nothing wrong with this approach specially when you're just building simple applications. But as soon as you get in to something that's complicated, where you might not know the values you actually want to look at, or there could be a lot of files, or a lot of states you want to examine - it could get really difficult to debug if you're just printing things out into the console.

Luckily we can set up a debugger in VS Code for our app that's running in a Docker container. Download [debug\_docker](https://github.com/HHOliver88/debug_docker" \t "_blank) and let's look at what specific changes are made for the VS debugger to work with containers.

**app.js**

It's just a simple express app that does addition and prints the result in the browser. It will print out a wrong answer for now, and to fix it is very easy (You don't even need to run a debugger). But let's just pretend for now that this is a complex app, and it's printing out a value you didn't expect. We'll use VS Code debugger to figure out what's wrong.

**Dockerfile**

No changes were made.

**docker-compose.yml**

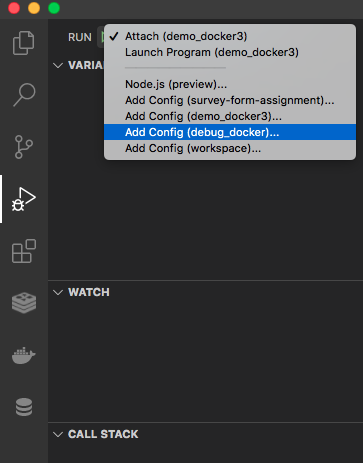
There's an additional port we added for the **web\_app**service. This port number is reserved for the debugger. Remember this port number because we are going to use it later in a new configuration file that we are going to add to our express project.

**package.json**

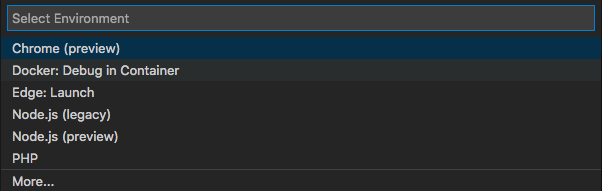
We added a **--inspect=0.0.0.0**in our **start**script. This is needed in order for our debugger to work.

Go ahead and right click on docker-compose.yml file and choose **Compose Up**to make sure everything works.

Now to activate Debugger, we go to the Debugger tab, and click on that dropdown item and make sure you choose **Add Config (debug\_docker)...**



Under *Select Environment,*choose **Docker: Debug in Container**



When you switch to File Explorer in VS Code, you should see a new folder called **.vscode**and inside it is a file called **launch.json.**launch.json is the configuration file for our debugger. Right now it just has:

{

// Use IntelliSense to learn about possible attributes.

// Hover to view descriptions of existing attributes.

// For more information, visit: <a href="https://go.microsoft.com/fwlink/?linkid=830387">https://go.microsoft.com/fwlink/?linkid=830387</a>

"version": "0.2.0",

"configurations": []

}

But we are going to update that to make our debugger work with our express app in our container. Update this file so that it will look something like below:

{

    "version": "0.2.0",

    "configurations": [

        {

            "type": "node",

            "request": "attach",

            "name": "Docker: Debug in Container",

            "remoteRoot": "/var/www/app/src",

            "port": 9229,

            "address": "localhost",

            "localRoot": "${workspaceFolder}/app.js",

            "protocol": "inspector",

            "restart": true,

            "skipFiles": [

                "node\_modules/\*\*/\*.js"

            ]

        }

    ]

}

These are basically just default values, and you don't need to worry about what each does. The only properties that probably will be changed depending on your project's folder structure is:

* **remoteRoot** - which is the WORKDIR (we specified in Dockerfile)
* **port** - The port for our debugger. You might want to change it to a different port number (depending on your project's requirements. But for now let's just keep it to port 9229).
* **localRoot** - Where our *entrypoint*file is currently located. Since our entrypoint is app.js and it's just in the root directory of our project folder, so we set it to: "${workspaceFolder}/app.js". But this can be changed depending on how you structure your project (or if your entrypoint file has a different name).
* **skipFiles** - It will skip all the files in the node\_modules directory when we do debugging. Skip means it won't show it in the call stack when we are stepping through the code.

And that's it! Switch back to VS Code Debugger, make sure that **Docker: Attach to Node (debug\_docker)**is selected in the dropdown and click on the green play icon to start the debugger. You will know that the debugger is running when the bottom part of VS Code changes from blue (your default theme) to orange

Debugger off:



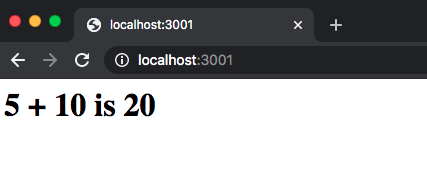
Debugger on:



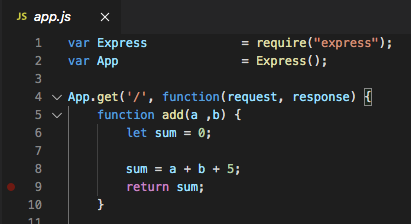
**Breakpoints**

**Breakpoints** are one of the most important **debugging** techniques in your developer's toolbox. You set **breakpoints** wherever you want to pause **debugger** execution. For example, you may want to see the state of code variables or look at the call stack at a certain **breakpoint**. It's essentially like telling debugger, "Hey, I want to stop. Pause. and have a look at what's going on at this point in time."

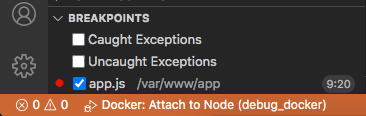
Before we set a breakpoint, let's go ahead and visit our page and see what the bug is.



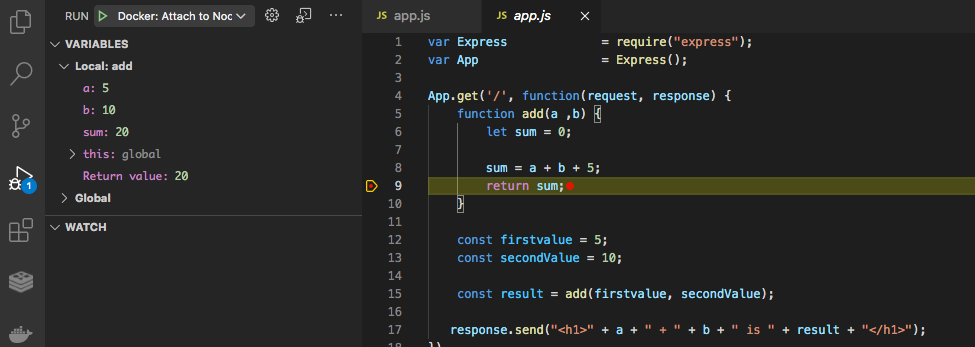
We expect it to print **5 + 10 is 15**but it's printing a different value for the sum of 5 and 10. We have a function called add() that accepts two parameters. Let's use the debugger to see what this function returns. To do that, go ahead and set a breakpoint on the return statement of the add function:



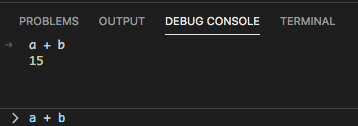
Note: Hover your mouse near the line number of the code you want to set a breakpoint to and you should see a red dot appear. Click on the red dot to set the breakpoint. You should see a checkbox appear in under BREAKPOINTS of Debugger:



To trigger the breakpoint, go ahead and visit localhost:3001. You will notice that you will instantly get redirected to your VS Code Debugger. This time, your VS Code debugger will contain various information about your app - up until the breakpoint that you set.



In the **VARIABLES**window, all the variables you used (up to line 9) and its value are shown. You can even manually perform the addition yourself in VS Code. Go ahead and click on the DEBUG CONSOLE (Located left side of TERMINAL) and type **a + b**(then press ENTER):



Okay, so by setting up a breakpoint, we now have gathered these information:

* a is 5
* b is 10
* sum is 20
* But when we run **a + b**in the **DEBUG CONSOLE,**we get a 15.

With these information we can then deduce that:

* expected values were passed to the **add()**function
* That there must be something wrong with how we add the values that were passed to the add() function because it currently returns 20.

And sure enough, there's a **+ 5**bug in our code. Removing that will fix the bug.

**Conclusion**

Use the debugger tool to get information about your app by setting breakpoints. You don't have to learn all the other fancy stuff Debugger has to offer for now. At this point, instead of using console.log, use the vs code debugger and just set breakpoints.

Tip: Just put the breakpoints on your code where you would normally put console.log() to get information on certain variables, etc.

To detach from **Debug Mode,**just click the Disconnect icon (the orange icon on the farthest right):



**Step Over (The arrow icon next to play button)**

It moves the current breakpoint that you set**to the next line of code.**

**Step Into**

It moves the current breakpoint to the **the next statement.**

**Step Out (The arrow up icon)**

Takes you out of the current scope.

**Objectives**

* To be familiar with VS Code debugger tool
* To be familiar with setting break points
* To be comfortable with VS Code's debug mode.

**Fix FirstPlusLength**

Using VS Code's debugger tool, fix the firstPlusLength app's bug. You can download the assignment files [here](https://github.com/HHOliver88/fix-first-plus-length).

Given an array, print/log the sum of the first value in the array, plus the array’s length. Assume that the array is composed of numbers.

Note: The configuration files are provided in order for you to run this app using Docker container + debugger. So no need to focus your attention to the configuration files. Just set breakpoints in **app.js**to fix the firstPlustLength bug.

# Docker Course Evaluation

Your feedback helps us to know how we can improve our training courses. Please give your honest feedback for the questions in this Evaluation form: <https://docs.google.com/forms/d/e/1FAIpQLSdUb-Tix4GK62lHofOnwcn9dZ1DtAc2CasM6sdmY4b7_QgOfQ/viewform>