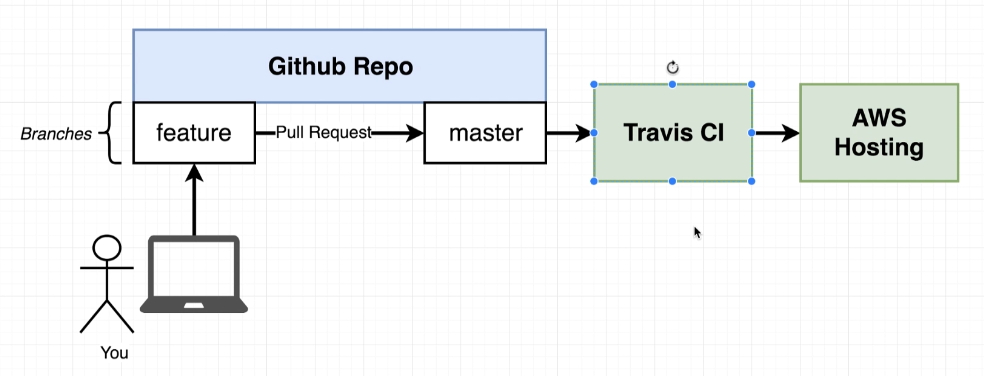
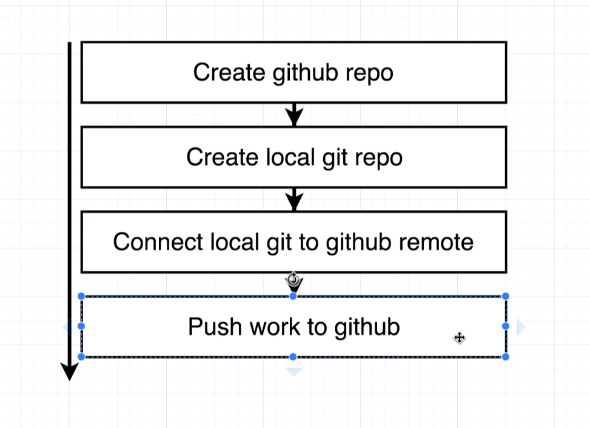
**Section 7: Continuous Integration and Deployment with AWS**

Up to this point, we’ve set up everything we need on the Docker side of things. We now need to figure out how to use these containers that we put together to integrate with GitHub, integrate with the CI tool Travis CI, and deploy to a hosting site (AWS).



First, let’s integrate with GitHub



Log-in to GitHub, press + button in top right to create new repository and create one. They will then provide you with a link to that recently-created GitHub repository and follow their instructions on making a local git repository via your terminal (git init, git add, git commit, git remote, git push).

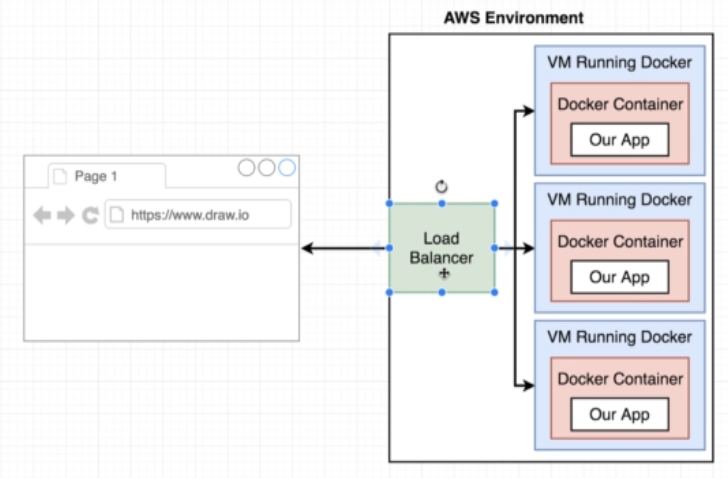
Anytime we push code to GitHub, Travis CI pulls it down and does some work. This work DOESN’T HAVE TO BE JUST TESTING. We can do anything…we will be using it for testing and then deploying on AWS.

To set it up, watch video 3-6 (#88-91) in Section 7 of this course.

[www.Travis-ci.org](http://www.Travis-ci.org)

To deploy our project, we will be using AWS Elastic Beanstalk (used for running and managing web apps). He said Elastic Beanstalk is the easiest way to get started with production docker instances. It’s most appropriate when we are starting up 1 container at a time. Can be multiple copies of this 1 container.

Elastic Beanstalk automatically scales. It has a load balancer included. The load balancer will route requests to a VM that is running docker and has our container running inside it. If there is a lot of traffic, Elastic Beanstalk automatically adds additional virtual machines with our container inside as needed.



I skipped most of this section. A lot of set up. He made his Travis CI automatically deploy to Elastic Beanstalk. When it goes to deploy to Elastic Beanstalk, Travis will zip up all files in our GitHub repo and dump them in an S3 bucket (bucket\_name below). The Elastic Beanstalk will use this to deploy app.

He needed to make a .travis.yml file for setting up testing and deployment with Travis CI.

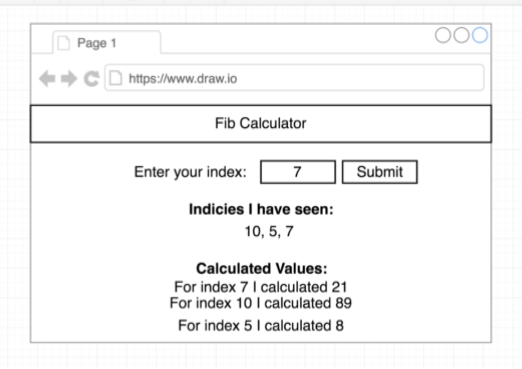
Issues with this deployment we just did:

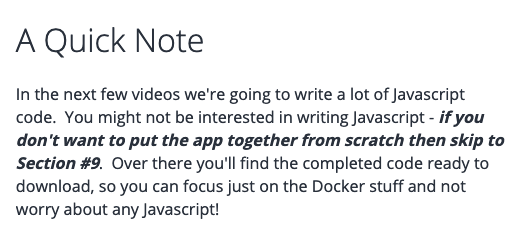
-The app was simple, no outside dependencies. No database connection was used/needed

-Our image was built multiple times…we built it on Travis CI and then we built it again on AWS

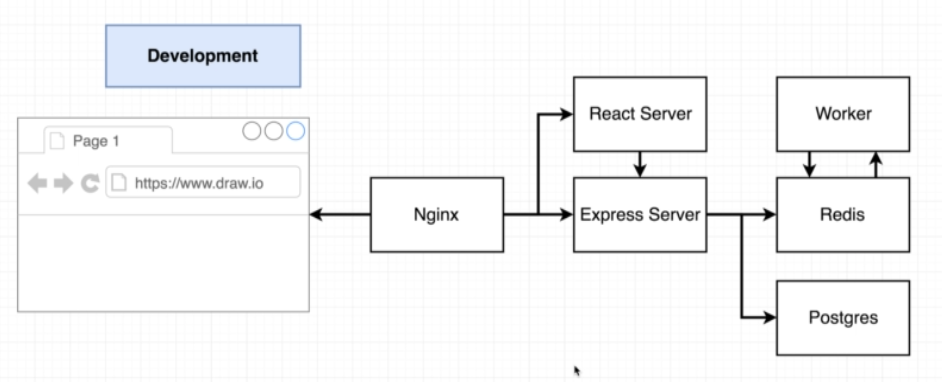
**Section 8: Building a ­Multi-Container Application**

In this section, we will be building a multi-container application. It will be a Fibonacci sequence calculator that is purposefully over-the-top complex so that we can learn multi-container apps.





Back-end Dev architecture:

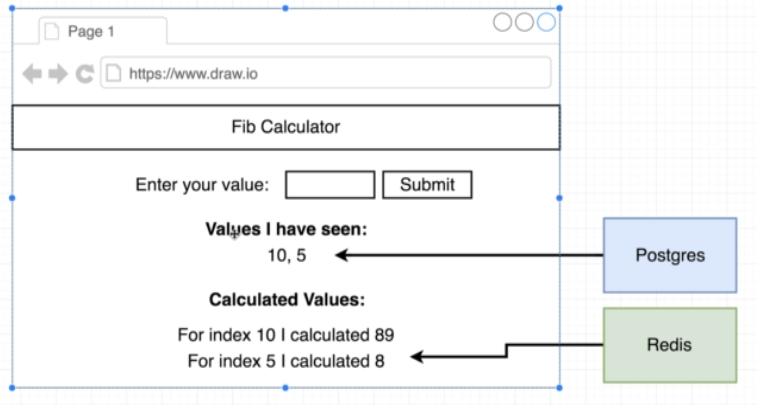


^Again, this is way more complicated than it needs to be for this particular project.

^When a user accesses our app through the browser, the Nginx server needs to decide if the browser is trying to access our React application (to get some front-end assets like the HTML files/JavaScript files needed to build the front-end app) or the Express Server (for some back-end API (actually AJAX?) that is used for submitting numbers and doing the calculations and retrieving values for the “For index 7 I calculated 21” lines)

^Redis is an in-memory database and it is very commonly used for housing temporary (cached) values

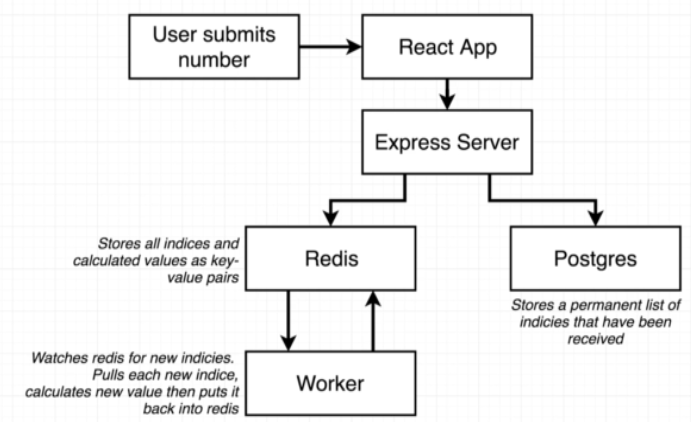
^Postgres is a database similar to mySQL



^All of the numbers for the “Values I have seen:” section are going to be permenantly stored

^The calculated values of 89 and 8 are going to be temporarily stored via the Redis database

^Doesn’t have to be this way but showing us this so we learn how different databases can be used in an app



^When user submits a number, the react app will make an AJAX request to the back-end Express Server. The Express Server, once it receives this number, will store it in our Postgres database for a permanent store. It will also take that number and put it into the Redis database. When a new number shows up in our Redis database, it will automatically start a separate back-end process we’ll call the “Worker” which will compute the fibonaci value and put that calculated value back into Redis so that it can be eventually requested by our React App and show up on the screen.

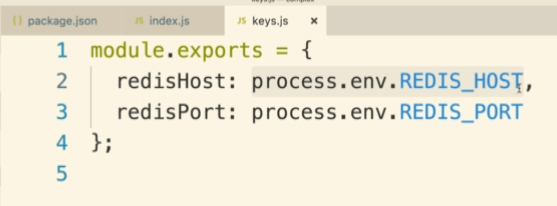
AJAX vs API:

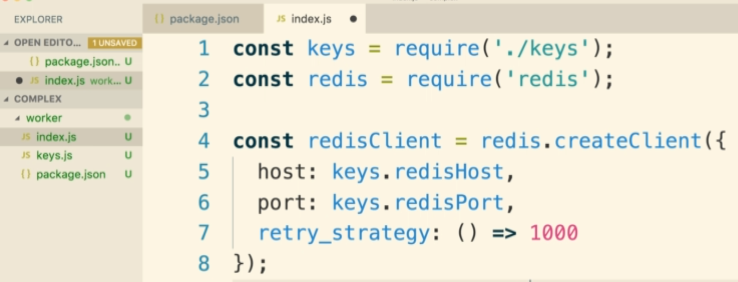
https://www.youtube.com/watch?v=EXMnwMYeWWE

Setting up the Worker: Video #108 Worker Process Setup



Keys file for exporting our redis keys:







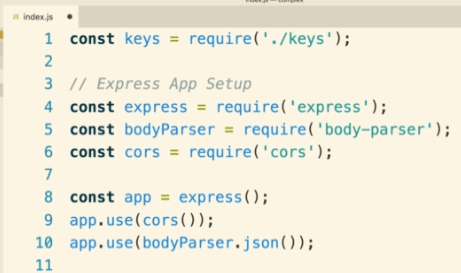
Setting up the Express Server (API Layer that communicates with Redis and Postgres for when our React App calls it):

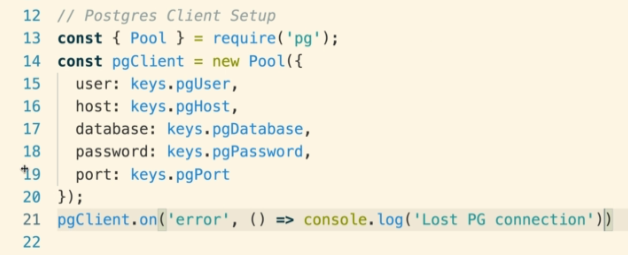


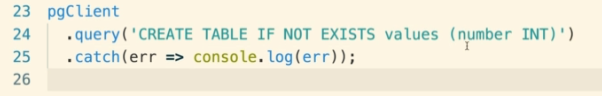
^pg is postgress

Keys file for exporting our redis and postgres keys:









^CORS stands for Cross-Origin Resource Sharing. It allows us to make requests from one domain/port (the one that the react app is going to be running on) to a different domain/port (the one that the express api is hosted on)

^The line “app.use(bodyParser.json()) is going to parse incoming requests from the react application and turn the body of the post request into a JSON value that our Express API can work with

^Starting at line 23, he is creating a table which store the values already submitted

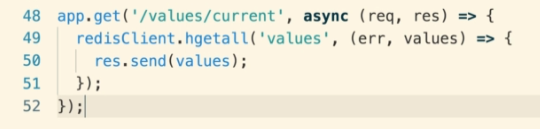
^So with this code, the Express Server will be connected to a Postgres database and will create a new table inside the database if it doesn’t already exist for collecting submitted values to the React App

Now, we will add code to this file to make it connect to a Redis database and have the number submitted from the React App be pushed to this database:



^We make duplicate connections because according to the Redis documentation, if we ever have a client/connectino that is listening or publishing or subscript to information from another database it cannot be used for other purposes.

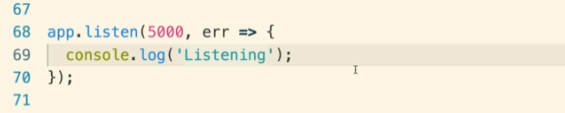




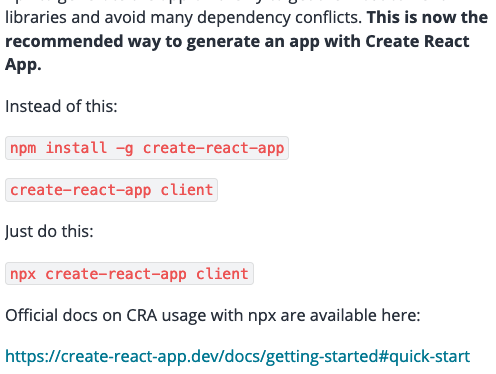
^Line 42-46 will be used to query our running Postgres instance and retrieve all of the different values that have ever been submitted to Postgres. Looks at “values” table in a SQL query statement and returns all rows from this table back

^Line 48-52 will look at all info from the “values” hash table from Redis





^The last route receives new values from our React App (receives POST requests from our Front-end React App)…The Worker will come into the Hash later and replace the “Nothing yet!” value with the calculated Fibonacci number. The .publish line is what sends a message to the Worker process/notifies it to do the calculation. Line 63 permenantly adds the number to our Postgres client  
^working: true is just some arbitrary response to let the user know work is getting done



He creates the React app in lectures 113-117

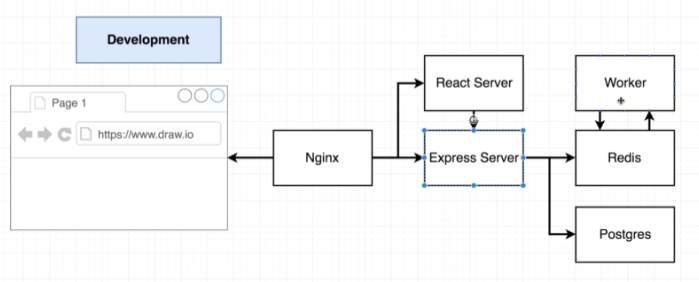
**Section 9: “Dockerizing” Multiple Services**

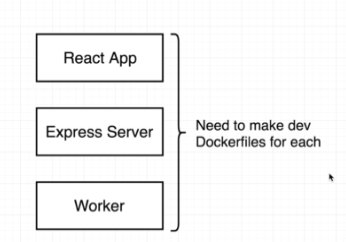
We will be using the app that we made in the last section. Downloading the zip folder, we should have 3 directories from his app he made:

/client (React Server)

/server (Express API Server)

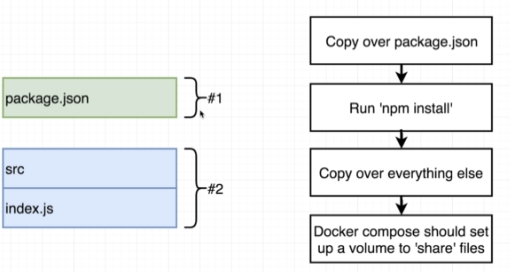
/worker (Worker)



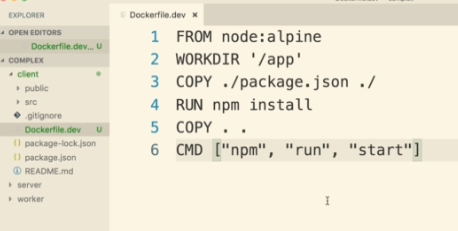


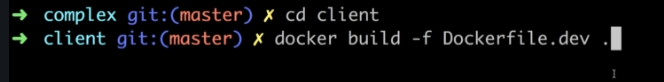
^We’ll be making dev containers for each

How?

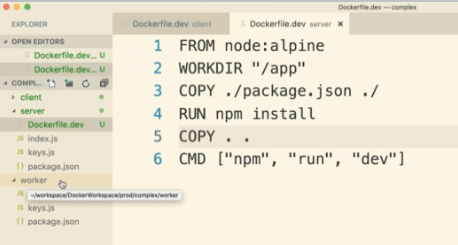


Inside client/ directory he made Dockerfile.dev:





Inside server/ directory he made Dockerfile.dev:



^This is slightly different from the client/ Dockerfile.dev because the server/ and worker/ directories’ package.json makes use of nodemon, which is a command-line tool to automatically restart your project whenever any of the source code has changed. So when a volume changes, it will automatically restart. Here is their package.json with nodemon:

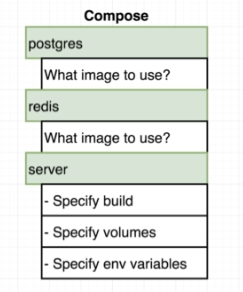






Now let’s create a Docker Compose file. Reminder: Docker Compose makes it easy to set up connections between different containers.

We’ll first set up the Express server with the Redis and Postgres databases. Then add React server and Worker and nginx server to our build.



^The env variables are the keys file that we made for the connections to Redis and Postgres

Inside of the root project directory, he made docker-compose.yml:



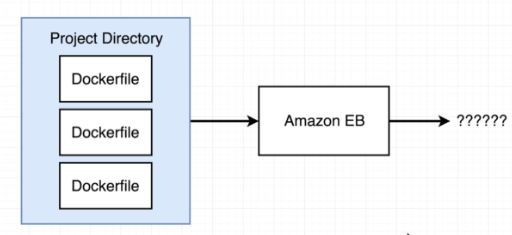
^He found a postgres and redis image on the Docker Hub (hub.docker.com/explore/)…the “latest” was the tag for the ones he wanted to use. “Lastest” seems like a common tag to use.

^The - /app/node\_modules inside of “server:volumes:” makes it so it will not try to override this folder inside of the container. The next line makes it so that everything (else) in the server/ directory gets copied over to the app/ folder of our container. So these two lines together make it so that anytime our applications tries to access anything inside of the /app directory of our new container (aside from /app/node\_modules), it will essentially get redirected back to the server/ directory of our local current project folder. Thus, any change in the ./server folder of our local current project folder, it will get automatically reflected inside our running container. Hence we don’t have to rebuild our image when we change our source code.

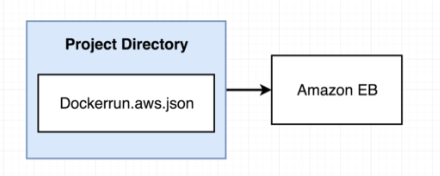
Stopped taking notes lecture 124: Environment Variables with Docker Compose

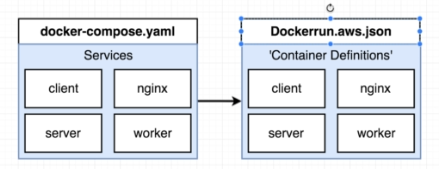
**Section 10: A Continuous Integration Workflow for Multiple Images**

**Section 11: Multi-Container Deployments to AWS**



Note: previously when we were using Amazon Elastic Beanstalk, it was automatically building our container. But this was because we were only giving it a single Dockerfile to build. When you hand it multiple Dockerfiles (and hence interested in building multiple, separate containers), we need to build the Dockerrun.aws.json file for Elastic Beanstalk.

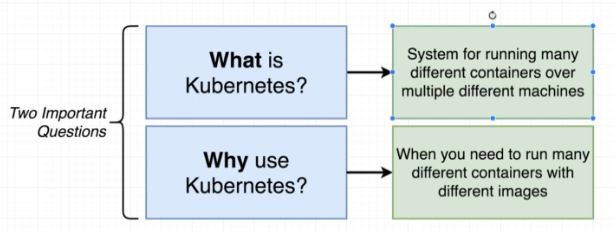
D



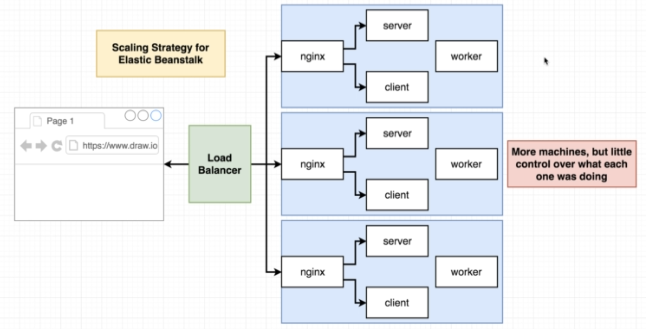
^Setting up AWS to deploy containers (which is what we want to do for our production environment) through this Dockerrun.aws.json file is very similar to a docker-compose.yml (we use docker compose in a dev environment generally). Instead of “Services”, we call them “Container Definitions”. Also, docker-compose.yml files build images based off of dockerfile(s) that you specify for each container you are building, whereas Dockerrun.aws.json will build containers based off of a images on DockerHub (previously he put his images on DockerHub).

Side Note: AWS Elastic Beanstalk doesn’t actually know how to run containers. It passes off this task to something called Amazon ECS (Amazon Elastic Container Service). If you want to figure out how to customize your Dockerrun file, you’ll need to look at the ECS “task definition” documentation: <https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_definition_parameters.html#container_definitions>

**Section 12: Onwards to Kubernetes!**



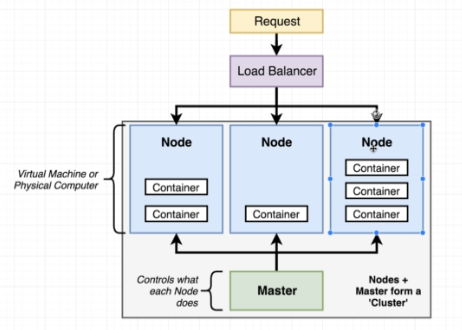
Why Kubernetes? Watch video 166: The Why’s and What’s of Kubernetes, where he explains that Elastic Beanstalk would scale our app like this (scales the entire set of containers):



Side Note: He said the Redis and Postgres databases were not containers, they were outside services

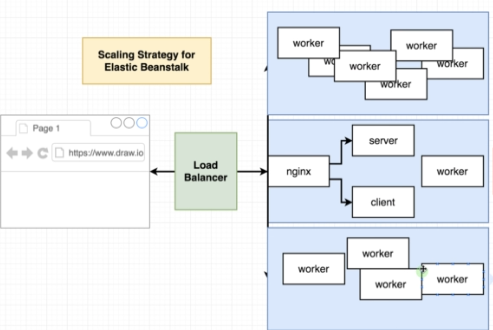
…whereas Kubernetes could scale out just the necessary portion needed (the heavy-processing “Worker” container).

A “Cluster” in Kubernetes is an assembly of a Master and 1 or more nodes (a node is a VM or a physical computer, and are used to run 1 or more containers (can be different containers)).



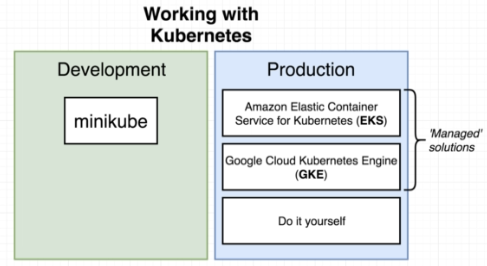
^Outside of our cluster, we have a load balancer which will take traffic (network requests) and relay these requests into our different nodes.

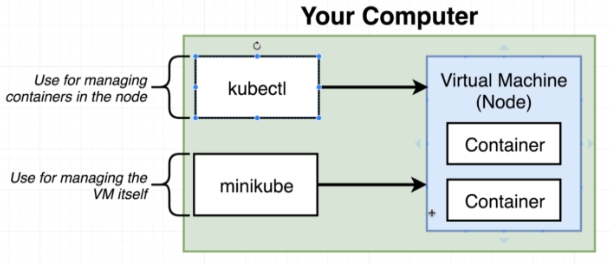
So, in Kubernetes, we could have something more similar to what we need such as something like:



We interact with a Kubernetes cluster by reaching out to the cluster’s Master.

There is a large distinction between working with Kubernetes in a Dev vs. Prod environment. In a Dev environment (such as on our local computer), we can use of Kubernetes using a program called minikube (a command-line tool which sets up a Kubernetes cluster on our local machine). For using Kubernetes in a production environment, we make use of something called “managed” solutions. Managed solutions are references to outside Cloud providers such as Google Cloud or AWS that will set up a Kubernetes cluster for you, and also will take care of many low-level tasks that we wouldn’t expect. You could also set it up on your own instead of using managed solutions but it is tough.





^Minikube will create a Kubernetes cluster on your machine. It creates a VM whose sole purpose is to run some number of containers. In order to interact with this node’s (VM’s) containers, we will be using a program called kubectl (this program is used to interact with Kubernetes clusters in general, doesn’t have to be in Dev environment).

^ONLY MAKE USE OF MINIKUBE IN A DEV/LOCAL ENVIRONMENT…Kubectl will be used in both LOCAL AND PROD ENVIRONMENT

To get setup with Kubernetes on our local machine:

A screenshot of a cell phone

Description automatically generated

^Install Kubectl, VirtualBox, and minikube

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated