1
Server Writeup

1.1 What's all this then

I'd like to keep all the thesaurus entries in a database, and be able to interface with that via some API. While we could do everything locally and save some of the configuration headache, we also would hardly be able to use it. So, in the pursuit of functionality, we'll need a server to put this on.

1.2 IaaS vs PaaS vs SaaS

We had some options for this project in terms of hosting, as we have decided that we aren't in the business of `Make` as a usage strategy. These options come as a slew of acronyms: IaaS, PaaS, and I suppose some bare metal solution from my pockets. The last option would consist of some computer sitting in my dorm room, running 24/7, along with some router configurations that might have BardIT coming after me; the other's are the real contenders. There's a certain balance between PaaS (Platform as a Service) and IaaS (Infrastructure as a Service), each with some pros and some cons. IaaS is a rather simple option: it is as close to the bare-metal (often called "on-premises") solution as possible, where we would pay for some virtual machine (VM) and networking. With IaaS we are leveraging server farms, and are more-or-less renting a server.

PaaS is more nuanced, as it plays a role between IaaS and SaaS (Software as a Service, like Google Apps, Dropbox, etc.), and tends to abstract desirable aspects of IaaS such as security, metrics, DNS, and so on.

We ultimately went with IaaS—for a few reasons—via DigitalOcean, a popular cloud services platform among other names like AWS and Azure. This gives us the advantage of having all our services in the same place as opposed to the spread out nature of PaaS. There are exceptions to this heuristic, such as Openshift and other CaaS (Containers as a Service) platforms, that would allow us to keep centralized services, but pricing is one of the biggest sellers for our use of IaaS. We lose some of the ease of scalability, but that is not an aspect that we are terribly concerned with while developing the core of the project. As cash is king, we will get our hands dirty with SSL, proxies, and other niceties that PaaS would shield us from.

1.3 Don't be root!

It should go without saying that it's wise to stray from doing all things as `root`. As default, we can only `ssh` into `root`, so we must add a user! We'll still need root access, so we'll be sure to add our new user to the `sudo` group. Then to finish this setup, we must patch up our `ssh` configuration: we want to `ssh` into the new user and disable `ssh` into `root`.

```
ssh root@<IP_ADDRESS>
adduser <USERNAME>
usermod -aG sudo <USERNAME>
rsync --archive --chown=<USERNAME>:<USERNAME> ~/.ssh /home/<USERNAME>
vim /etc/ssh/sshd_config    (set "PermitRootLogin" to "no")
```

Of course, we'll make a nice `ssh` config on our personal machine for our convenience. We'd like to save the hassle of keeping track of IP addresses and users and keys, so we'll make another entry to be able to simply run `ssh sproj`.

Note, we'll have to do this per system, as per this bad boy

1.4 The big whale upstairs

There is certainly a warranted section on Docker. Docker is the Kleenex of containers: it's a wildly popular open source project for working with containers. Cointainers are an industry-molding alternative to virtual machines for running system-agnostic programs. If you've written software in the past few years, you've almost certainly seen some sort of depiction of the difference between virtualization and containerization, but in case you haven't, we can go into it. First off, the picture as promised:

!Dockerization vs Containerization

While a picture is worth a thousand words, a picture with words may warrant a few extra. On each side we have a layer of apps on top of their dependencies that is running on top of some infrastructure. The apps and bins/libs are our binaries/executables/processes along with their source-code/libraries/etc that we are used to writing and running in our day-to-day as programmers. The infrastructure is just that: the silicon and bare metal that we are running on top of. That surmises the local development experience, and I'll trust you are familiar with the woes of your program working like a charm on your personal machine, but playing anything but nicely when you have to run it on another computer.

The "it worked on *my* computer" dilemma is what virtualization and containerization are here to address. If we can encapsulate these processes and dependencies and abstract them from the contents of the machine they reside in, we should have platform-independent programs. Virtual machines tackle this by building entire guest operating systems on top of a hypervisor (also known as a virtual machine monitor), which serves to channel access to hardware and thus allow for multiple operating systems to run on one host operating system.

However, operating systems are quite large, and there is a lot of waste with virtualization. With the industry going towards cloud computing, we needed smaller, faster solutions. Containerization seeks to leverage the host operating system, and rather than keep several guest-os instances, we have a single container runtime environment.

It's a similar solution to that which the JVM provided in the 1990's. And, since I couldn't help but mention the JVM, I can't stop myself from mentioning that there's great interest in scrapping it in the context of containerization: if we are running in containers, do we need to keep the JVM around? JetBrains is producing Quarkus for doing ahead-of-time compiling for making Java more container-friendly. More container-friendly is–of course–a euphimism for "Java containers are gigantic and slow": while even an amateur (read: me) can get containers down to a handful of megabytes (the beginnings of my frontend, proxy, and api are all just about 20MB), it's quite rare to see a Java image less than around 80MB.

This may not seem like a big deal, we are only talking on the order of megabytes. But in the days where serverless architecture/microservices are growing in popularity, there is an ever-increasing need for quick cold-start times (starting an idle container, should none be active and available). Serverless computing being a platform in which cloud providers provision your resources, and the customer is charged for active time rather than paying for a more traditional flat-rate server. This is attractive, as it is anything but wasteful, but if a request is made, there's always a chance that your containers are down, and they must be reinitialized to fullfill the request. If we can optimize our containers, we'll save time and money and our user-experience!

This tangent on a direction the software engineering industry is headed ought to demonstrate the appeal of containers: serverless wouldn't even be a fever dream if we only had VMs.

1.4.1 Building My Containers (Thesaurus)

With containerization, the name of the game is modularity. So we generally aspire to have per-process containers. Thus my one webapp comprised of a frontend, a backend, a database, and a server/reverse-proxy has four containers: one for each aspect. And with inter-container dependecies, one would think that things would get confusing. This is where docker-compose comes in. This is a tool that allows for the definitions/instructions for multiple containers. This is a dev-ops dream, where I can deploy my application with a single command (and preserve my volumes!). It's configured in a `yaml` file and maps nicely to standard docker cli arguments:

we specify things like `Dockerfile` location, port forwarding, networds, container dependencies, volumes, environment variables and more!

I'll touch a bit on how I made my various containers, and then configuring my compose file. It's far simpler than it sounds.

Golang compiles to binaries per OS, which is excellent news for us! If we can compile to binaries, we don't need to keep all the installation business around, so we simply copy over the source code, install the packages, and compile as a build stage. Then, we can copy over the executable to a smaller base image, expose our port, and run! It's a process similar to a writing standard shell script, and we are generally concerned with the resulting size. This was an example of a multistage Dockerfile, where we separate the compilation stage from the runner stage; this is quite common as a measure of reducing image size.

The dockerfile for the frontend is similarly simple. We copy over the source code, install all the dependencies, and build our app. Then we follow our Golang footsteps and run from a smaller nginx image. This is a good point to mention the repetition within many Dockerfiles; you may note that we are separating installing the Vue CLI tools from installing all the node dependencies. This is because docker gives intermediate image tags per layer (each line in the Dockerfile), which we may run from.

The MongoDB and Nginx containers are as easy as can be. We simply change nothing from the base images! This is part of the beauty of being a part of a vibrant open source community.

Within our docker-compose file, we naturally define a service for each previously mentioned aspect. The common bits are that each service is given the location of the Dockerfile or image, all are set to restart unless-stopped (as I will just be deploying a single instance of this), each is gicen a pseudo tty incase anything goes amuck, everything is given some sort of port mapping, and everything is added to a network. Outsude of this, we link the backend to the database, and we are good to go! From here we can build our cluster and tear it down at will.

Since this deployment is in essence a personal project (I hardly expect more than a handful of people to check it out in these stages), I'm not concerned with scaling and load-balancing.

Should the need arise, everything is fully containerized, and it is just a matter of migrating to a container orchistration platform (like Kubernetes).