1
Backend

1.1 What is a RESTful API and why do I Need One?

An application programming interface (API) provides a client with some form of interfacing or interacting with a server. This is rather broad, and in the realm of web development there are standard HTTP (HyperText Transfer Protocol) methods—GET, HEAD, POST, PUT, PATCH, DELETE, CONNECT, OPTIONS, and TRACE—with various usages and characteristics. Perhaps the most common usage of HTTP is in REST APIs (REpresentational State Transfer) which is a stateless architecture based on a request-response interface. REST maps nicely onto basic CRUD (CREATE, READ, UPDATE, DELETE) operations common to databases. Note that REST defines an abstract framework for web services whereas CRUD defines a distinct set of operations largely in the realm of databases. With the foundations of this pairing, we can use these within our application.

We would like to have a RESTful API for our application to be able to securely wrap these CRUD functions for our database as well as having a bidirectional data flow between our frontend and our model as well as our frontend and our database. We abstract our database connection to this RESTful API as any frontend code is universally accessible; we introduce this data-layer as a means of keeping our backend private.

1.2 Anatomy of an HTTP request

As we have our database and our API in place, it is worth talking about the flow of a request. Our user will only directly interact with our frontend, which is in essence a series of forms or input/output cycles; we shall take the thesaurus page as an example. We keep around state for the word being looked up and for various aspects of the results (the response object and whether the response was empty for each of the lexicons).

To get this response, we send a GET request to API with the word as the URI/path parameter via JavaScript's fetch. This then resolves to our API where we are listening with the mux router which matches the URI to our `/thesaurus/api/v1/words/word` route. We send a reference to our database client singleton as well as our request and empty response to our route handler. Here we parse the requested word and unmarshall this request into our `entry` struct representing our database schema: should this fail, we write a `bad request` response and return. There are a few more checks along these lines that we perform before sending an `ok` response with entry from our database: we make sure that the word is not an empty string, and we make sure that the word is actually present in our database. The former is less necessary, as it would be caught by the latter, but it stems from patterns used in unsafe endpoints and thus serves to provide consistency between responses.

The latter is where the actual retrieval occurs; we access our database and collection through our passed client reference, and we attempt to find an entry with a matching `word` field. We may safely use `FindOne` as we enforced idempotency in our POST for each collection. If the word does not exist in the database, we get an error and repond with a `bad request` signifying its absence. Otherwise, we decode the resulting document into our `entry` struct and respond with a success message paired with the marshalled data of our `entry`.

Once we have written our response, we send it back as `application/json` to our frontend and interpret the result. Should we an erroneous response, we set a flag for the word not existing, and should we receive a successful reponse we send the data on to our components for response rendering.

There are other fine grained details involving Transmission Control Protocol (TCP), Transport Layer Security (TLS) / Secure Sockets Layer (SSL), and router middleware that we deal with to a certain extent. TCP underlies HTTP in the establishment of sockets, keeping track of packet loss, etc. Our SSL certificates are generated by LetsEncrypt and we use Nginx to upgrade insecure connections, serve our frontend, and to reverse proxy our containers.

1.3 Creating our API

There are several parts that go into our API as we touched on in the last section that warrants discussion. As this is, in essense, a CRUD wrapper, we work largely with the mongo go driver [`?mongoGoD17:online`]. We have to access our database from it's URI; this contains some sensitive information, as it takes the form `mongodb://<USERNAME>:<PASSWORD>@<HOST>:<PORT>`. In order to avoid having these public and accessible via our version control, we use environment variables; we store all of these in a `.env` file—which is never version controlled—and we load them with the `os` module. We have a high level config singleton in `config.go` where we have a struct with these four fields that is populated with `os.Getenv` in the beginning of our `main`. We then pass a reference to this config interface into our `app` module where we connect to the database and save the database client into a high level struct.

This `app` struct also has our mux router which is the core of our API. Due to the standard same origin policy, we would run into problems with our frontend or our model making calls to our API—even though they share a domain. As per the policy, there must be matching protocol, port, and host; as we distinguish our deployments by subdomain, the hosts do not match. To circumvent this, we use Cross Origin Resource Sharing or CORS. We wrap our router in middleware that allows `GET` and `POST` methods with `X-Requested-With`, `Content-Type`, and `Authorization` headers from all origins, thus allowing us to access our API from our other components.

Then there's a mapping of route string to function, where we must specify the HTTP method, the route—with optional parameters—and a function reference that takes along our database

client, and an `ResponseWriter` and `Request` from the `net/http` module. As all requests and responses go through these datatypes, this lends itself to utilities that abstract common behavior. Some of this is in the form of router-level middleware, but there's more at the individual route level. As we have to write the same endpoints for each collection, we would like to save as much repetition as we can by writing utility functions—this stands to cut down development time, ease developer experience, improve readability, and to have more consistency across the project.

These utility functions come primarily in two forms: helpers for our HTTP requests and responses and helpers for our database client. In terms of the HTTP helpers, we want to abstract unpacking our requests and writing our responses. For handling our requests, we have functions to unmarshal our incoming JSON, we have authentication checking by validating request headers. For writing our responses, we have some basic writers that return standardized responses like the authentication or the empty field checks, then there is a generic response function that takes the status code, some interface that is marshalled to populate the reponse data, and an optional message. This optional message then helps differentiate between generic error responses and generic success responses, where for an error we can simply wrap the function in another function that provides an error message, again furthering the level of consistency in our API.

Our database client helpers are a bit more involved, as we have to manage our database connections rather than writing responses. Each process bears similarities to one another. They all define a context, check for results existing or not existing in the database, undergo some database operation, and return an error or a result. The update and delete are remarkably similar functions. Both take in an entry and a filter, the entry corresponding to the active component, and the filter being some interface to query against; they check to ensure that a corresponding entry exists in the database, and then `update` replaces the existing entry—without upserting—and `delete` removes one entry that matches the filter. Create is a similar function, although we must check that the word does not exist and then insert a new document. Finding one is similar again, where we ensure that an entry exists, and we return the first match. The significantly different function is the batch read that returns a reduced model of the collection where all entries are

accessed at a passed key; this then involves looping over the context and appending each existing value to a list before returning.

These CRUD helpers allow for streamlined, consistent endpoints. All sets of authentication-gated endpoints are relatively homogenous, as are the non-authentication-gated endpoints. Each has a wrapper around the aforementioned utility functions, making use of individual structs, collections, and acceptance criteria. For the gated endpoints, the `read` endpoint that aggregates all words in each collection is just a matter of checking credentials before calling the utility function. On the other hand, the `create`, `update`, and `delete` all unmarshal the request body into a struct corresponding to the associated schema; each checks for empty fields in the request body—which is unique to each set of endpoints; each checks for valid admin credentials; and then each filters to the corresponding utility function. The most unique example is that of the `senselevel` endpoints, where we create a `wordlevel` field in its `create` method; this entails creating a union of unique entries across all passed word associations.

> The creation of the `wordlevel` field warranted a change in architecture to the API as a whole. Each filter was previously a copy of the struct for each collection, which worked with mongo-go-driver's `Find`, yet there were issues with the introduction of a single interface field—as opposed to a slice of interfaces. This then prompted the creation of a simple filter struct that only has a field for the word in question.

The non-gated endpoints are all simple `GET` calls. These all read the word from the path parameter, unmarshal into the corresponding struct as usual, check that the word is not empty, and retrieve the first match in the collection. The biggest distinction here is the mode of access: this is completely accessible, and comes in the form of a `GET`. Thus, to differentiate between requests we have this path parameter—as a `GET` may not have a request body—, which is not present in any of the gated endpoints.

1.4 Unmarshalling JSON into Nested Structs in Golang

So, we are thrilled that we have translated TSV to JSON... but now what? Obviously, we're in no business of `require`-ing a gigabyte of JSON, so we've got to write up some more endpoints. We figure we more-or-less know how we want things structured as it's right there in JSON, so how do we move from JSON to... JSON elsewhere?

We are using MongoDB in this project, so our database is just BSON (binary JSON). Should be a painless transfer, yes? It's not just a direct dumping though, there are a few key steps. As we can't access our database directly from our frontend, we need to have a backend to interface with our collections, in this case we are using Golang. So, let's dive into the process of transferring our local data to our database.

At a high level, we will have a little Python script that loops over all the entries in our local JSON file; the encoding/decoding is handled by such an abstracted language, so we can skip that for now, just take note that we will be loading the JSON into a dictionary and dumping that into a string. From this string, we'll make a POST request to some endpoint in our backend, let's say, for example, it's `create-sense-level`. This is an HTTP call, of course, so we'll hit some URI with headers and a body. It only makes sense that we have `Content-Type: application/json`, but since we are changing the database, we may include some sort of authorization headers, in this case `adminUsername` and `adminPassword`. As an example body, we'll use this:

```
{
  "word": "testword",
  "senselist": [
   {
    "sense": [ "lorem", "ipsum" ],
    "associations": [ "dolor", "sit" ]
   },
   {
    "sense": [ "consectetur" ],
    "associations": [ "adipiscing", "elit" ]
   }
  ]
}
```

And we'd like to get roughly the same thing back, perhaps with a message, maybe like this:

```
{
  "message": "Success!",
```

```
  "data": {
    "_id": "000000000000000000000000",
    "word": "testword",
    "senselist": [
      {
        "associations": [ "dolor", "sit" ],
        "sense": [ "lorem", "ipsum" ]
      },
      {
        "associations": [ "adipiscing", "elit" ],
        "sense": [ "consectetur" ]
      }
    ]
  }
}
```

But there are some things on the way there. `CreateSenseLevel` is defined as `CreateSenseLevel(client *mongo.Client, response http.ResponseWriter, request *http.Request)`, so we have our request and our response to worry about right now, our db-client will some soon. We first must define structs for our schema, we will have a nested struct: the outer with `word`, `_id`, and `senselist`, the inner with `associations` and `sense`. These look like this:

```
type SenseLevelEntry struct {
 ID        primitive.ObjectID `json:"_id,omitempty" bson:"_id,omitempty"`
 Word      string             `json:"word,omitempty" bson:"word,omitempty"`
 SenseList []SenseLevelData   `json:"senselist,omitempty" bson:"senselist,omitempty"`
}
type SenseLevelData struct {
 Associations []string `json:"associations,omitempty" bson:"associations,omitempty"`
 Sense        []string `json:"sense,omitempty" bson:"sense,omitempty"`
}
```

Note the `json:``___'' bson:``___''` with each field: this defines how we want to marshall our structs–we will associate the JSON "word" field with the Go `Word` string, and we will take the JSON "senselist" field to be the Go `SenseLevelData` slice (considering how this inner-struct is marshalled).

> Marshalling is the process of converting data to a byte-stream. Unmarshalling is the reverse, taking a byte-stream to it's original object (through serialization).

Let's get to the endpoint! We'll make an empty struct, and pass it to a decoder alongside our request body; this will handle our unmarshalling. We'll catch any bad unmarshalling (invalid fields and whatnot) and throw an error, and otherwise check to make sure everything else checks

out! We'll just check and make sure no fields in the request body were empty, if they are, we'll throw another error. Then we'll check our admin credentials by checking our header against valid admin data, and if we don't have the clearance, we'll throw another error! Then–for now–we'll toss back a response, assuming we haven't encountered any errors. We'll marshall our interface, wrap it in the rest of our desired response, check for any errors, and if none are present we'll write some headers and return our response!

1.5 Writing Go Modules

An early issue we ran into in this project was the disorganization of our backend. Being my first project in Go, nothing started off (and likely little currently is) pretty; I didn't have the slightest idea of how to structure a project, and what I know of the language came from building our API. I couldn't figure out how to import files from anywhere but the same directory, so in came mess—tons of files that should be abstracted floating around in one folder. Outside of itself being unpleasant to work with, it encourages a poor system of state management where we pass around globals rather than keeping more abstracted components. In comes GOMODULES:

GOMODULES was introduced in Go 1.11 as a form of dependency management, and a way to circumvent some of the issues of GOPATH. GOPATH has been an issue stemming from the opinionated nature of Golang: all packages should be centralized and reside within GOPATH.

> As of Go 1.11, the `go` command enables the use of modules when the current directory or any parent directory has a `go.mod`, provided the directory is outside `$GOPATH/src`. (Inside `$GOPATH/src`, for compatibility, the go command still runs in the old GOPATH mode, even if a go.mod is found.) the go blog

Thus, with the introduction of module mode, we are able to develop Go outside of our GOPATH. We get a bundle of versioned dependencies (respecting semantic import versioning), which allows for... modular code. There's a similar notion of reproducibility with Go modules as there is with containerization: the `go.mod` specifies the module root—everything is self contained.

2
Frontend

2.1 frontend stuff

this would be me talking about the frontend

3
Lexicons

3.1 National Research Council Canada (NRC) Emotion Lexicon

3.1.1 IMPORTANT NOTE: remember to cite as per Terms of Use in their readme

This is the lexicon that we are most interested in as it is most directly related to our project. There are two forms of this lexicon: the "word-sense" lexicon is the original annotated at the word-sense level and the "word" lexicon is a baked version which condenses all word-senses for a word.

We've made some mistakes in the past, so we want to check our restructured data; if we know that the "word-level" form was created by taking the union of the affect associations of the "sense-level" form, we can create our own version of the "word-level" and compare the two versions. As there's little sense in keeping our own version around, we may check it algorithmically.

3.1.2 Methodology

Saif M. Mohommad and Peter D. Tourney compiled this lexicon with crowdsourcing through Amazon's Mechanical Turk (an online crowdsourcing platform); they chose crowdsourcing as it is quick and inexpensive (costing them $2100 for the Turkers). As a deterrent of bad responses, they included a filtering question in each survey that asked for the best synonym for the given word, allowing them to identify either lack of word knowledge or probabilitically filtering random

responders. They selected joy, sadness, anger, fear, trust, disgust, surprise, and anticipation as per Robert Plutchik's wheel of basic emotions, as well as drawing from the present emotion lexicons WordNet Affect Lexicon, General Inquirer, and Affective Norms for English Words and both the Maquarie Thesaurus and Google's N-Gram corpus. They generated questions with the Macquarie Thesaurus with the aforementioned filtering-question followed by questions asking for alignment with the various emotions. They also included polarity (positive vs negative valence) in the lexicon, giving us 10 categories to work with.

3.2 Our Representation

We wanted to preserve their data, but bring it into our database (MongoDB). This transfer was relatively painless, as their lexicon was in consistent TSV. We borrowed a decent amount of JSON utilities and structure from our thesaurus-scraper, writing to files by first letter as we go; all that changes is the shift from making http requests and parsing HTML to loading a local file and parsing TSV. We did this for both the "word-level" and the "sense-level" forms resulting in the following schema:

3.2.1 Word Level

```
{
  "<word>": {
    "associations": [
      "<list",
      "<of>",
      "<associations>",
    ],
    "word": "<word>"
  },
  ...
}
```

3.2.2 Sense Level

```
{
  "<word>": [
    {
      "sense": [
        "<list",
        "<of>",
        "<synonyms>",
```

```
      ],
      "associations": [
        "<list",
        "<of>",
        "<associations>",
      ],
      "word": "<word>"
    },
    ...
  ],
  ...
}
```

3.2.3 Word-Sense Level

Note that the sense-level scheme consists of arrays whose entries resemble the word-level scheme along with a field representing the word-sense; this is because the word-level representation of a word is created from the union of the sense-level entries for that word.

> The original lexicon has annotations at word-sense level. Each word-sense pair was annotated by at least three annotators (most are annotated by at least five).
>
> The word-level lexicon was created by taking the union of emotions associated with all the senses of a word.
>
> — Saif M. Mohammad and Peter D. Turney (from NRC-Emotion-Lexicon-v0.92/readme.txt)

Each entry from these forms can then be easily POST-ed to our API and can be accessible!

3.3 National Research Council Canada (NRC) Colour Lexicon

3.4 National Research Council Canada (NRC) Affect-Intensity Lexicon

3.5 National Research Council Canada (NRC) VAD Lexicon

These three lexicons followed largely from the first. They shared similar formats, and we only had to change how we parsed them. These lexicons were all one-entry-per-line, so we were able to skip our finished-entry checking, and otherwise the differences were solely in the `process_line`

which had to be catered to each lexicon. As they are also single-form, we skipped difference checking for all of them.

4
Thesaurus

4.1 Why do I need a thesaurus?

A moldable thesaurus is seemingly very important for my project. I'm looking at building a model to shift the tone of a body of text while attempting to preserve semantics. Simply going off of intuition, we can figure that swapping a word for a synonym of that word may alter the tone of its enclosing sentence. Consequently, we would like to aggregate some relative relationship between some set of tones and groups of synonymous words. Upon creating such a dataset, we may interchange a word with a synonym according to a difference in tone.

4.2 Why not use an existing API?

There are a few reasons to construct or reconstruct our own thesaurus. First and foremost, we are interested in keeping additional data regarding tone that is not present within existing thesaurus APIs. It is far easier to manipulate data if it is all on hand, and it should save some server-side complexity in dealing with the decoupling of the thesaurus and the word-tone relationships. There is an issue with cost as well: APIs are rarely free past some established number of calls in a given time interval, and I would like for this software to function with minimal cost–ideally the only cost is in hosting. Finally, this is a project in the realm of software engineering. While it

is often better to rely on existing services–standing on the shoulders of those who came before you–it is also important to know how to create your own services.

4.3 How did we do it?

We created the thesaurus by web scraping, which is a large aspect of data-collection and thus is often a necessity in the sphere of machine learning. Of course, it is possible to compile a thesaurus using other means–surely one could buy a physical thesaurus and type up all the entries or even automate such menial tasks with computer vision–but we are interested in constructing a thesaurus as painlessly as possible. It is only one aspect of our project after all.

4.3.1 Choosing a site

Web scraping often comes with a give and take. There are several existing online thesaurus services, so we did looked into a few of them and landed on John Watson's Big Huge Thesaurus. We began with trying thesaurus.com, but they have protections against traditional scraping. There are many of such protections including lazy-loading content, providing fake data, services like Captcha, even automatically altering the page's HTML. It seemed as if thesaurus.com had been randomizing their CSS classes and either lazy-loading content or providing fake data. While we could likely get around this with an automated browser like Selenium, as the CSS classes are only a deterrent if scraping over a large time interval and the content would almost certainly exist within an automated browser session, we should respect that this site has practices in place to prevent scraping.

There may be protections against web scraping in place that we ought to honor, or there are often poorly laid out websites that would be a pain to use despite being open source, or the site may simply not provide all the information we would like. The latter is best exemplified by moby which we may come back to if we decide that we care not about parts of speech. John Watson's Big Huge Thesaurus, on the other hand, seems to have all that we want: synonyms by part of speech, a clean interface, and permission for use given credit is provided.

Protections against web scraping from JonasCz

4.3.2 Scraping the site

Web scraping for purposes such as gathering content from a page is a basic process: get the raw HTML of the page and retrieve what you want. Due to the ubiquity of HTTP requests and string processing, we can use just about anything we want for building our scraper. We will be using Python for its simplicity in our project, although we will create something similar in a bash script as a proof of concept for demonstration (see `thesaurus/webscrape/scrape.sh`).

We are using the `requests` module to get the HTML for each page and `bs4` for our HTML parsing. I am running Ubuntu on my computer, and thus have access to the `words` file present across Unix operating systems: this is a raw text file with a collection of words separated by line. This will be the basis of our thesaurus. We will first reduce this file by removing all entries with apostrophes with `thesaurus/webscrape/fixWords.py` whose essence is:

```
if word.find("'") == -1:
    outfile.write(word)
```

We do this as to eliminate repetition in our database to expedite searching as all nouns present in `words` have a possessive form; note that this does come with the loss of conjuntions. Now that we have the words we will use to construct our thesaurus, we may do exactly that. Each page takes the form of the same base url followed by the word: this makes for easy access. We go through word-by-word in our reduced file, make a GET request for that word, and then aggregate all the word's synonyms and antonyms by part of speech. There's just one little trick to this process: the antonyms are not in a concrete section, but rather one of several possible subsections under each part of speech. To circumvent this issue, we just have to do some checks to make sure that any present antonym section belongs to the part of speech we are considering and not a later-occuring part of speech. We allow the addition of antonyms to the synonym list and remove them prior to returning; this allows for us to have less rigorous checks in adding synonyms.

There was an earlier version of this scraper that did not deal with antonyms. This catch is attributed to Ariadne, who—when I showed her my progress—brought up the word "beautiful" which had "ugly" as the first entry under the synonym section. This prompted quite a refactoring,

and we should now be free of these bugs. The basis of this refactoring was largely tested against "beautiful" and "well" which both have antonyms in the thesaurus, and "well" had all the parts of speech.

Upon tweaking our scraper to suit our needs, we must output our results. As I only have so much RAM, and Python can be rather resource hungry, we segment our data by first letter. We will restructure our data into one object, but we will do this after collecting all of our data. We may naively write each dictionary to its corresponding JSON file and correct the result. This allows us to keep less in memory, which may otherwise present itself as a problem. This also allows us to segment our program as a failsafe; if we are to lose connection, crash, hit a request limit, or otherwise fail to run the script to completion, we may easily start again from where we have left off rather than the very beginning.

Then, upon building our thesaurus, we may move to the next part of our project. We would like to store this thesaurus in a database and create an API to interface with it.

5
Database

5.1 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. I chose DigitalOcean, a popular cloud services platform among other names like AWS and Azure. Surely we will also be using DigitalOcean for hosting the rest of this as well :)

5.2 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

5.3 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.

5.3.1 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).