1
Backend Writeup

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 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!

1.3 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.