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 Anatomy of an HTTP request

refs:

- fetch
- mux router
- idempotency
- unsafe
- LetsEncrypt
- Nginx

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.