# 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!