Zero, Null, and Missing! Oh My!

CHRIS "MAC" MCENIRY

Chris "Mac" McEniry is a practicing sysadmin responsible for running a large e-commerce and gaming service. He's been working and developing in an operational capacity for 15 years. In his free time, he builds tools and thinks about efficiency. cmceniry@mit.edu

It's common for work settings to have multiple environments in them. These environments, e.g., Production and Development, have many similarities and some very specific differences. In the attempt to minimize cognitive load (and typing) so we can tell the differences between environments, we tend to only call out the differences. For example, Production has the prod database, and Development has the dev database, but both use the same DNS systems.

In the end, our configurations have to reflect the exact settings for each environment. But, as mentioned, we do not want to deal with all of that verbosely.

In this article, I'm going to looking at one way to simplify that verbosity. We're going to have a common/base configuration and then composite the environment-specific configurations on top of that to produce the final exact settings for each environment.

To do this, we have to take a look at how the Go `encoding` libraries work, and account for, or work around, some of the defaulting behavior in Go.

The code for these examples can be found at https://github.com/cmceniry/login in the "zeronullmissing" directory. Each directory contains a corresponding example and can be executed using `go run main.go`.

encoding Standard Library

Go has an extensive standard library with all sorts of useful functionality. One of the commonly used pieces of it is the `encoding` package, which translates Go structures to other data forms—XML, JSON, and GOB (a native Go marshaling format). The standard library pattern and interface is also used in many third party libraries for other data formats.

The `encoding` pattern relies on Go's ability to inspect Go data types via reflection. Typically, when using the `encoding` libraries, one would define a custom struct type with necessary fields. The example above might look like:

```
type Configuration struct {
   Database string
   DNS      string
}
```

While this same struct could be used for multiple formats, we're going to work with JSON and the associated `encoding/json` library. The corresponding JSON configuration data for our example environments would look like:

```
Production
  { "database": "prod", "DNS": "shared" }
Development
  { "database": "dev":, "DNS": "shared" }
```

Unmarshal is the encoding/json library function that converts the JSON structure into our Go struct. It takes a byte slice with the JSON data and a de-referenced Configuration value, and returns an error if the conversion failed.

```
d, _ := ioutil.ReadFile("conf.json")
var conf Configuration
err := json.Unmarshal(d, &conf)
```

With this in hand, we can move on to compositing the configuration together.

Overriding

On first pass, to composite together our final configuration, we can read in the base and then environment configuration, and then merge those two. In our example, we would extract the common DNS configuration into the Base and handle the databases in each environment specific. The JSON input would look like:

```
base.yaml
  { "Database": "SETME", "DNS": "common"}

development.yaml
  { "Database": "dev" }

production.yaml
  { "Database": "prod" }
```

Loading these into Go corresponds with the following Go struct values:

```
base := Configuration{
   Database: "SETME",
   DNS: "Common",
}
development := Configuration{
   Database: "dev",
   DNS: "",
}
production := Configuration{
   Database: "prod",
   DNS: "",
}
```

A point to notice is that when Database or DNS is not specified in the JSON, it is initialized with the zero value for the string type—the empty string "". In the common case, we can interpret the empty string as an unspecified value. When merging, we can take only the Database values from development and production, so those are not the empty string, and have those override the Database value in base.

But what happens if we want to clear a value or set a value to the zero value?

JSON even has a null value. If set, that will also initialize the Go variable with a zero value. We can attempt to use Go pointers to interpret this, but it really changes it from a string zero value, "", to the string pointer zero value, nil. This will help us determine the difference between null and "" in the JSON, but still does not help us with missing values versus explicit zero values.

The standard encoding libraries do not make a distinction between a zero value (including null) and a missing value. We're going to examine this zeroing quirk of Go using probably the most heavily used data formatting library, encoding/json.

Baseline

First, we're going to examine the baseline behavior of Unmarshal.

To begin, we define our custom struct. To exercise the cases, we focus on six use cases:

1. FromZero: An explicitly set empty string into a string type
2. FromNull: An explicitly set null string into string type
3. FromPtrZero: An explicitly set empty string into a string pointer type
4. FromPtrNull: An explicitly set null string into string pointer type
5. FromMissing: A missing string type
6. FromPtrMiss: A missing string pointer type

baseline/main.go: struct.

```
type Items struct {
   FromZero    string
   FromNull    string
   FromPtrZero *string
   FromPtrNull *string
   FromMissing string
   FromPtrMiss *string
}
```

We use a string var to hold the input data we're going to work with.

baseline/main.go: input.

```
var input = `{
   "fromzero": "",
   "fromnull": null,
   "fromptrzero": "",
   "fromptrnull": null
}`
```

Inside of our main, we first initialize a location to hold the output of our decoding.

baseline/main.go: output.

```
output := Items{}
```

With our input and output, we can finally call Unmarshal. To create a common interface, Unmarshal expects all inputs to be byte slices, so we cast to that. Unmarshal also does not initialize the

output, but we do want to modify it, so we pass a pointer reference already initialized output (this is the case even in the event of maps and slices). `Unmarshal` returns an error if the decode fails or `nil` if it succeeds.

baseline/main.go: unmarshal.

```
err := json.Unmarshal([]byte(input), &output)
```

To show what happens, we print out the Go value.

baseline/main.go: print.

```
fmt.Printf("%#v\n", output)
```

This output looks like the following, after being folded to fit in this column:

```
main.Items{FromZero:"",FromNull:"",FromPtrZero:(*string)
  (0xc0000860e0),FromPtrNull:(*string)(nil),
  FromMissing:"",FromPtrMiss:(*string)(nil)}
```

This confirms that we cannot tell if something is explicitly set zero or implicitly set by being missing.

Drilling Down

For us to be able to discern if there are intentionally missing keys or intentionally null values, we need to take matters into our own hands.

Go provides the very quintessentially generic type, the empty interface or `interface{}`. When the `encoding` libraries encounter the empty interface, they infer it as an indicator that you want to handle the decoding by yourself. Instead of decoding it into organized structs, they pack all that they can into the empty interface slot in as raw a format as they can.

The empty interface can be used at any point—the top level or even inside of a struct. In our example, we're using a JSON object which has key/value pairs. This equates to a Go map. We let the library decode the keys as normal string keys, but we indicate that we'll handle the values. To do that, we're going to use a map of the empty interface, `map[string]interface{}`.

Using the same data value as before, we unmarshal the same way. However, instead of using the struct, we're going to use the empty interface map.

manual/main.go: decode.

```
output := make(map[string]interface{}, 0)
err := json.Unmarshal([]byte(input), &output)
```

Since we don't have the fields of our struct as before, we're going to iterate over a list of keys that we expect to potentially be there.

manual/main.go: loop.

```
keys := []string{"fromzero", "fromnull", "fromptrzero",
     "fromptrnull", "frommissing"}
for _, k := range keys {
```

With each key, we must first check that it is there. If it is not, we `continue` to the next iteration. This detects that our `frommissing` field is not present.

manual/main.go: check.

```
v, ok := output[k]
if !ok {
      fmt.Printf(`"%s" is missing`+"\n", k)
      continue
}
```

Now, we know we have a value, we use a type switch to handle the cases of what it might be. In our example, we only care about nulls and strings, so we handle those cases and leave others to a `default`.

Note: Unlike the baseline example, we find that a null converts to the nil type, instead of a nil value of a string pointer.

manual/main.go: type.

```
switch v.(type) {
  case nil:
    fmt.Printf(`"%s" is null`+"\n", k)
  case string:
    fmt.Printf(`"%s" is present and equal to "%s"`+"\n",
k, v.(string))
  default:
    fmt.Printf(`"%s" unhandled type %T`+"\n", k, v)
}
```

Putting that all together, we can now successfully determine the difference between an explicit zero value, a null value, and a missing value.

```
$ go run manual/main.go
"fromzero" is present and equal to ""
"fromnull" is null
"fromptrzero" is present and equal to ""
"fromptrnull" is null
"frommissing" is missing
```

Conclusion

The standard library `encoding` libraries save you a lot of work and effort by decoding data formats into Go structs. It works in the majority of cases.

However, sometimes, you have cases that you need to handle differently. This can be to determine missing versus explicit values, or to allow for polymorphous structures. But if you have to work with these other use cases, you do have a bit of overhead that you have to handle yourself. Fortunately, you can still use the `encoding` libraries to handle the framing even while you're handling the Go data structures manually. I hope this example gives you options for these other use cases.

Good luck and Happy Going.