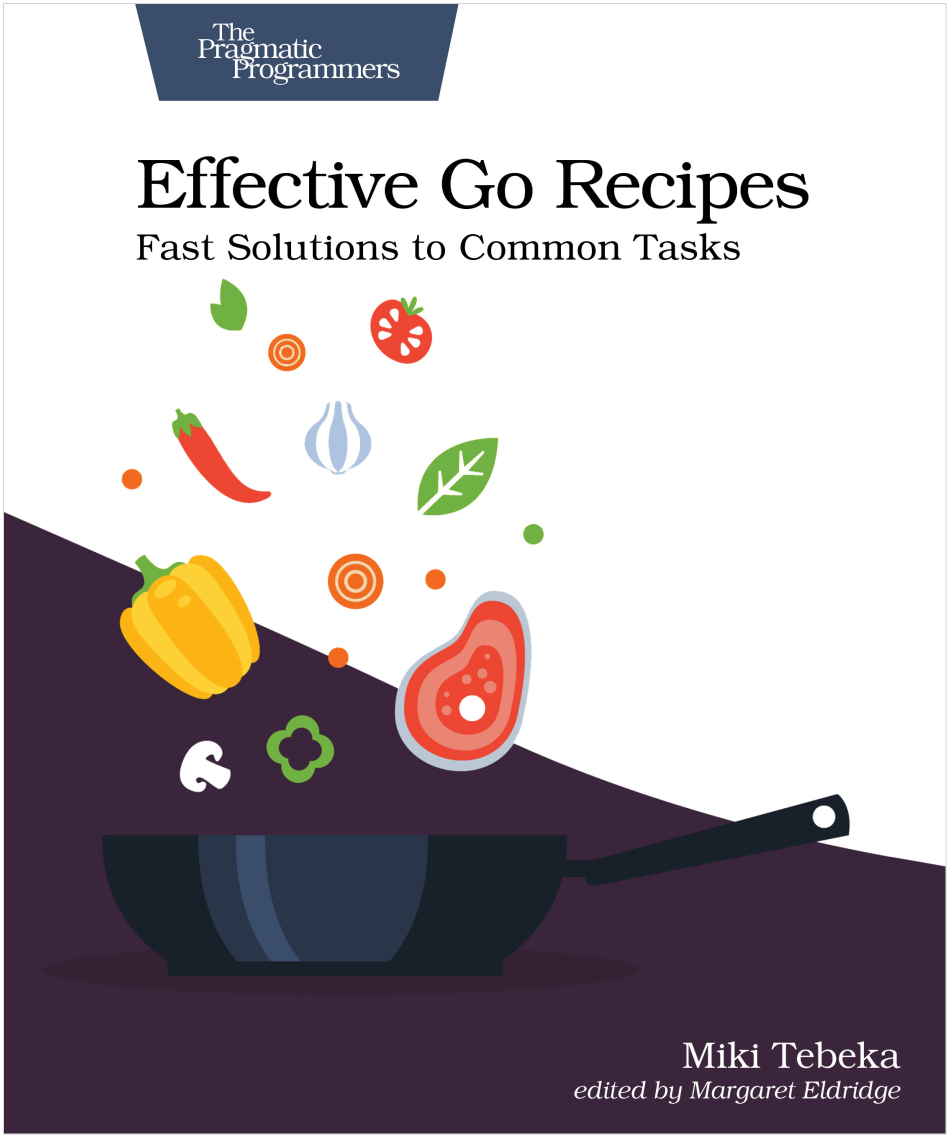

The Pragmatic Programmers

Effective Go Recipes

Fast Solutions to Common Tasks

Miki Tebeka

edited by Margaret Eldridge

Recipe 14

Parsing Struct Tags

Task

You’re building an ORM (object-relational mapper) for your company’s proprietary database. An ORM simplifies working with the database—you store and retrieve Go structs directly instead of using []any.

Your first task is to parse a user struct and return the mapping between the field names and the database columns. Take the following struct, for example:

serialize/orm/orm.go
```go
// Log is a log structure.
type Log struct {
    Time  time.Time `db:"ts"`
    Level int       `db:"level"`
    Text  string    `db:"message"`
}
```

You should return the mapping of the following:

- Time → ts
- Level → level
- Text → message

Solution

You start by defining the parsing function signature:

serialize/orm/orm.go
```go
func parseStructTags(s any) (map[string]string, error) {
```

First, you use the reflect package to get the type of the parameter:

serialize/orm/orm.go
```go
typ := reflect.TypeOf(s)
if typ.Kind() != reflect.Struct {
    return nil, fmt.Errorf("%s is not a struct", typ)
}
```

Then you iterate over the struct fields and extract the db key from the field tag:

serialize/orm/orm.go

```
m := make(map[string]string)
for i := 0; i < typ.NumField(); i++ {
    fld := typ.Field(i)
    if dbName := fld.Tag.Get("db"); dbName != "" {
        m[fld.Name] = dbName
    }
}
return m, nil
```

Discussion

Since parseStructTags should accept any value, the type of its parameter is any. Most of the time, using any is a "code smell" and you should use concrete types or an interface.

However, in this case, you can't use a concrete type or an interface, so you're left with using any.

(Note: the term *code smell* was coined by Kent Beck and means a surface indication that usually corresponds to a deeper problem in the system.)

Struct tags are used by many serialization packages; the built-in encoding/json, encoding/xml, and many other external packages, such as yaml, use them.

Struct tags allow you to add extra information about a field. They have a known format: key:"value" key:"value" .... The Field type in the reflect package has methods to extract a specific key.

In some cases, the value can be more than just a name. Commonly, it's either a list of values separated by a , or a name=value. Here's an example from a struct generated by Google's protoc tool:

```
Value float64 `protobuf:"fixed64,1,opt,name=value,proto3" \
    json:"value,omitempty"`
```

The struct tag has two keys: protobuf and json, and both have complex values.

If you decide you need complex values (like protobuf in our example), you need to design the format used in these tags since you're basically implementing your own serialization format.

It's best to copy what encoding/json or protobuf are doing instead of inventing your own.