The Many Faces of Struct Tags

Kyle Erf and Sam Helman

Why are we here?

- To showcase and explain an underused feature of the language
- To galvanize you to consider struct tags as a potential solution to problems you may encounter

What we’ve built

Evergreen - a continuous integration tool built on MongoDB (github.com/evergreen-ci/evergreen)
over the last two years.

MongoDB Tools (backups, restoration, monitoring…)

Both rely heavily on struct tags for communication and configuration.

What is a struct tag?

From the official spec

“A field declaration may be followed by an optional **string literal** *tag*, which becomes an attribute for all the fields in the corresponding field declaration. The tags are made visible through a reflection interface and take part in **type identity for structs** but are **otherwise ignored**.”

In other words

An arbitrary unicode string value attached to a struct field.

```
type Candidate struct {
     Name       string   `name`
     Priorities []string `priorities`
}
```

```
type VocalCandidate struct {
     Name       string   `name`
     Priorities []string `stated_priorities`
}
```

```
x := Candidate{}
y := VocalCandidate(x) // <-- not allowed
```

Why?

- Attach arbitrary values to your fields
- Provided without a specific use case

Namespacing

What is it?

A mechanism for allowing multiple tags to be associated with the same struct field.

What are its origins?

https://groups.google.com/forum/#!topic/golang-nuts/HbJIUfUc58E

```
type Candidate struct {
     Id   int    `json:"id" bson:"_id"`
     Name string `json:"full_name"`
     Age  int
}
```

Namespacing

Remember the spec? *Namespacing isn't there.*

But it's so useful it's in the official reflect package. Use `go vet` to catch errors.

http://golang.org/pkg/reflect/#StructTag.Get

```
type Candidate struct {
      Id   int    `json:"id" bson:"_id"`
      Name string `json:"full_name"`
      Age  int
}
```

```
candidate := Candidate{}
st := reflect.TypeOf(candidate)
field := st.Field(0)

fmt.Println(field.Tag.Get("json")) // "id"
fmt.Println(field.Tag.Get("bson")) // "_id"
```

Serialization

Serializing to and from json

Used by the `encoding/json` package to determine field names when marshaling and unmarshaling structs

Other data formats

Used for XML, BSON, SQL ORMS...

```go
type Candidate struct {
     Id   string `json:"id,omitempty"`
     Name string `json:"full_name"`
     Age  int
}
```

```go
gopher := &Candidate{
     Name: "Gopher Cleveland",
     Age:  40,
}
asBytes, _ := json.Marshal(gopher)
fmt.Println(string(asBytes))

// "{"full_name":"Gopher Cleveland","Age":40}"
```

Tag Safety

<u>Problem</u>

What if you need to reference the content of a tag? Useful for MongoDB, ORMs, etc.

<u>Workaround</u>

Check tags at program initialization, similar to regex.MustCompile

```
type Donor struct {
     Name string `bson:"donor_name"`
     Id   string `bson:"_id"`
}

collection.Find(
  bson.M{"donor_name":"Selena Gomez"}))
```

```
type Donor struct {
     Name string `bson:"name"`
     Id string   `bson:"_id"`
}

collection.Find(
  bson.M{"donor_name":"Selena Gomez"}))
// nothing is returned
```

MustHaveTag

https://github.com/evergreen-ci/tags

```
type Donor struct {
    Name string `bson:"name"`
    Id   string `bson:"_id"`
}

// get the BSON tag for Donor.Name; panic if it does not exist
NameTag := tags.MustHaveBSONTag(Donor{}, "Name")

collection.Find(bson.M{NameTag: "Selena Gomez"}) // safe!
```

A More Unusual Mapping Case

github.com/mitchellh/mapstructure

Library for converting between structs and Go's builtin `map`.

Why?
"If you have configuration or an encoding that changes slightly depending on specific fields"

In other words
If it's easier to decode some data into a `map` first, then pull it into a struct based on one of its values.

```go
type PersonalDonation struct {
	Amount int    `mapstructure:"amt"`
	Donor  string `mapstructure:"donor"`
}

asMap := map[string]interface{}{
	"type":  "personal",
	"amt":   500000,
	"donor": "Hugo Weaving",
}

if asMap["type"] == "personal" {
	var donation PersonalDonation
	Decode(asMap, &donation)
	fmt.Println(donation.Donor) // "Hugo Weaving"
} else {
	...
}
```

Command-line Configuration

github.com/jessevdk/go-flags

What is it?
Alternative to Go's builtin flag package.
Uses struct tags to specify *everything*.

Why?
Convenience, personal preference

```
type DonationOpts struct {
      Amount    int  `short:"a" long:"donation_amount" description:"how much to donate"`
      Anonymous bool `short:"n" long:"anonymous" description:"make the donation anonymous"`
}

var opts DonationOpts
flags.Parse(&opts) // populates opts based on the command-line flags used
```

Default Values

http://github.com/mcuadros/go-defaults

Library for setting non-zero default values of a type.

Benefits

Simplify your initializers. Lets your initialization logic live inside its type definition.

Drawbacks

Loss of compile safety. Most use cases are too complex for this method. Go’s struct literal syntax is already pretty slick.

```
type Volunteer struct {
    PartTime bool   `default:"true"`
    Events   int    `default:"1"`
}

func NewVolunteer() *Volunteer {
    example := new(Volunteer)
    defaults.SetDefaults(example)
    return example
}
```

Validation

https://github.com/asaskevich/govalidator

What is it?
Use struct tags to specify validators to be applied to a field.

Why?
We often need validation above and beyond type. Specify it declaratively in the struct tags.

```
type Donor struct {
      Email                string `valid:"email"`
      CreditCardNumber string `valid:"creditcard"`
      DonorId              string `valid:"hexadecimal"`
}

donor := &Donor{
      Email:               "ernest@hemingway.biz",
      CreditCardNumber: "0000-0000-0000-0000",
      DonorId:             "a8392ac7",
}
result, _ := validator.ValidateStruct(donor)
```

How About Type Validation?

<u>interface{} with type bounds?</u>

Something like type polymorphism. We can use struct tags to tell our program what types a field can be.

<u>Should you do this?</u>

Nah.

```
type APIResult struct {
     Status  interface{} `types:"int,error"`
     ZipCode interface{} `types:"uint,string"`
     Address interface{} `types:"Home,Office"`
}


types.Validate(APIResult{
     Status:  200,
     ZipCode: 10037,
     Address: Residential{...},
})
```

Foreign Languages

Huh?

Use struct tags to specify the translations of field names in various languages.

Why?

For printing / serializing.

Does this library exist?

Not as far as we know.

```
type Candidate struct {
	Name string `lang_es:"nombre" lang_gr:"όνομα"`
	Age  int    `lang_es:"edad" lang_gr:"ηλικία"`
}

gopher := &Candidate{
	Name: "Gopher Cleveland",
	Age:  40,
}
asBytes, _ := langMarshaler.Marshal(gopher, "es")
fmt.Println(string(asBytes))

// "{"nombre":"Gopher Cleveland","edad":40}"
```

The Future: Code Generation?

Right Now

Code Generation is usually configured through comments, which is powerful and workable, but not always clean to read.

Struct Tags?

More idiomatic for tagging things than comments are, when you can use them. An extension of their original purpose.

```
type Voter struct {
      email   string `generate:"get,validate-email"`
      zipCode string `generate:"set,get,validate-zip"`
}

// === Generated Code ===
func (v *Voter) GetEmail() string           { return v.email }
func (v *Voter) GetZipCode() string         { return v.zipCode }
func (v *Voter) SetZipCode(zipCode string) { v.zipCode = zipCode }
func (v *Voter) Validate() error {
      if err := validate.Email(v.email); err != nil {
            return err
      }
      if err := validate.Zip(v.zipCode); err != nil {
            return err
      }
      return nil
}
```

Questions?