Advanced JSON handling in Go

19:40 05 Mar 2020

Jonathan Hall
DevOps Evangelist / Go Developer / Clean Coder / Salsa Dancer

About me

- Open Source contributor; CouchDB PMC, author of Kivik
- Core Tech Lead for Lana
- Former eCommerce Dev Manager at Bugaboo
- Former backend developer at Teamwork.com
- Former backend developer at Booking.com
- Former tech lead at eFolder/DoubleCheck

Show of hands

Who has...

- ...used JSON in a Go program?
- ...been frustrated by Go's strict typing when dealing with JSON?
- ...felt limited by Go's standard JSON handling?

What have been your biggest frustrations?

Today's Topics

- *Very* brief intro to JSON in Go
- Basic use of maps and structs
- Handling inputs of unknown type
- Handling data with some unknown fields

A brief intro to JSON

- JavaScript Object Notation, defined by RFC 8259
- Human-readable, textual representation of arbitrary data
- Limted types: null, Number, String, Boolean, Array, Object
- Broad applications: Config files, data interchange, simple messaging

Alternatives to JSON

- YAML, TOML, INI
- BSON, MessagePack, CBOR, Smile
- XML
- ProtoBuf
- Custom/proprietary formats

Many principles discussed in this presentation apply to any of the above formats.

Marshaling JSON

Creating JSON from a Go object is (usually) very straight forward:

```
func main() {
    x := map[string]string{
        "foo": "bar",
    }
    data, _ := json.Marshal(x)
    fmt.Println(string(data))
}
```

Run

Marshaling JSON, #2

Creating JSON from a Go object is (usually) very straight forward:

```
func main() {
    type person struct {
        Name        string `json:"name"`
        Age         int    `json:"age"`
        Description string `json:"descr,omitempty"`
        secret      string // Unexported fields are never (un)marshaled
    }
    x := person{
        Name:   "Bob",
        Age:    32,
        secret: "Shhh!",
    }
    data, _ := json.Marshal(x)
    fmt.Println(string(data))
}
```

Run

Unmarshaling JSON

Unmarshaling JSON is often a bit trickier.

```
func main() {
    data := []byte(`{"foo":"bar"}`)
    var x interface{}
    _ = json.Unmarshal(data, &x)
    spew.Dump(x)
}
```

Run

Unmarshaling JSON, #2

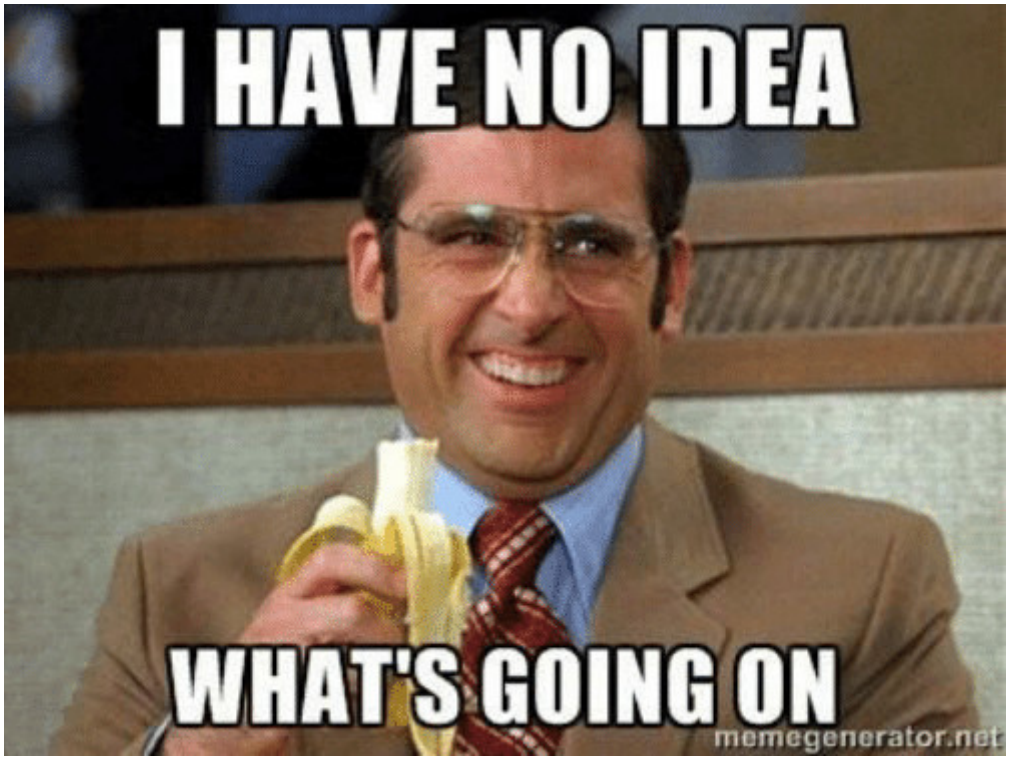
Avoid a map whenever possible. interface{} says nothing (https://www.youtube.com/watch?v=PAAkCSZUG1c&t=7m36s)

```
func main() {
    type person struct {
        Name        string `json:"name"`
        Age         int    `json:"age"`
        Description string `json:"descr,omitempty"`
        secret      string // Unexported fields are never (un)marshaled
    }
    data := []byte(`{"name":"Bob","age":32,"secret":"Shhh!"}`)
    var x person
    _ = json.Unmarshal(data, &x)
    spew.Dump(x)
}
```

Run

Unknown types

Structs are nice, but what happens when you don't know the data type prior to unmarshaling?



Cases of "unknown input"

- Input may be string or Number

```
123
"123"
```

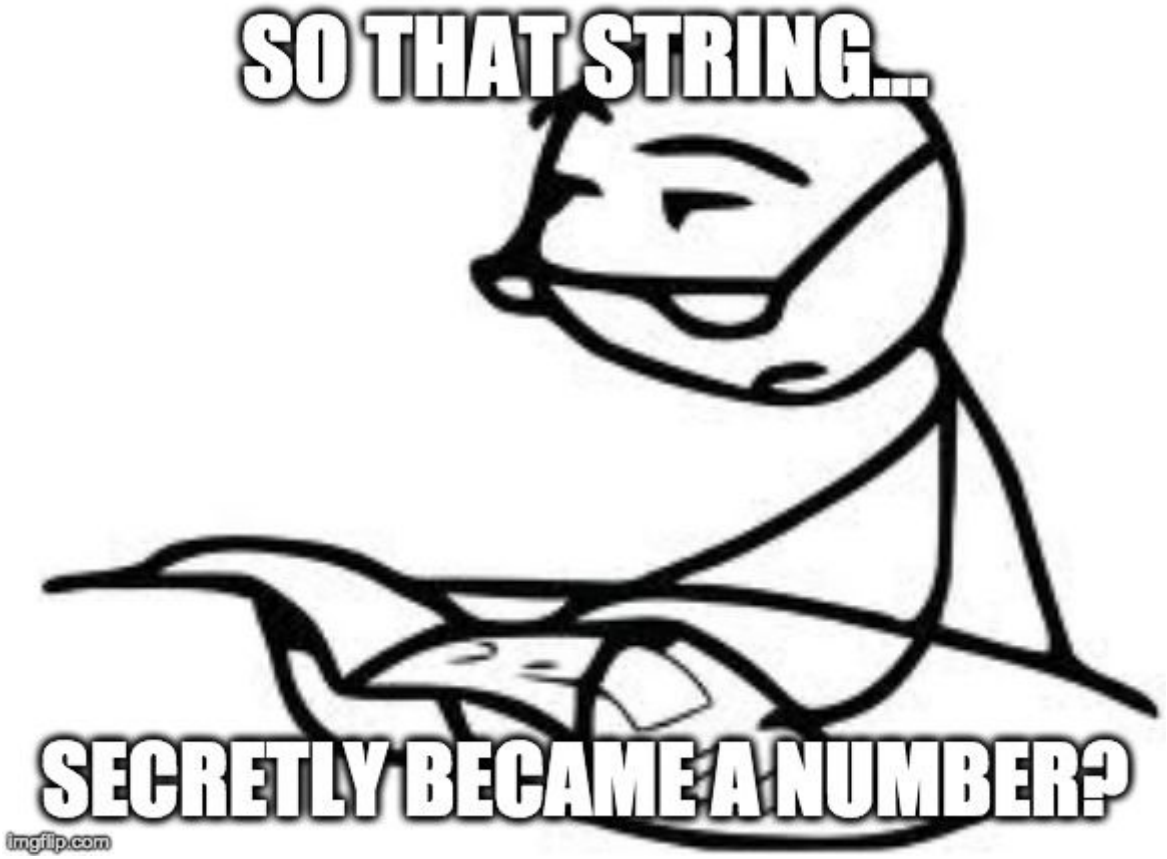
- Input may be object, or array of objects

```
{...}
[{...},{...}]
```

- Input may be success, or an error

```
{"success":true,"results":[...]}
{"success":false,"error":"..."}
```

Number literal, or string to number



Number literal, or string to number

Perhaps you have a sloppy API, that sometimes returns strings, and sometimes numbers. (True story at Lana)

Sometimes:

```
{"id": 123456}
```

Other times:

```
{"id": "123456"}
```

Number literal, or string to number

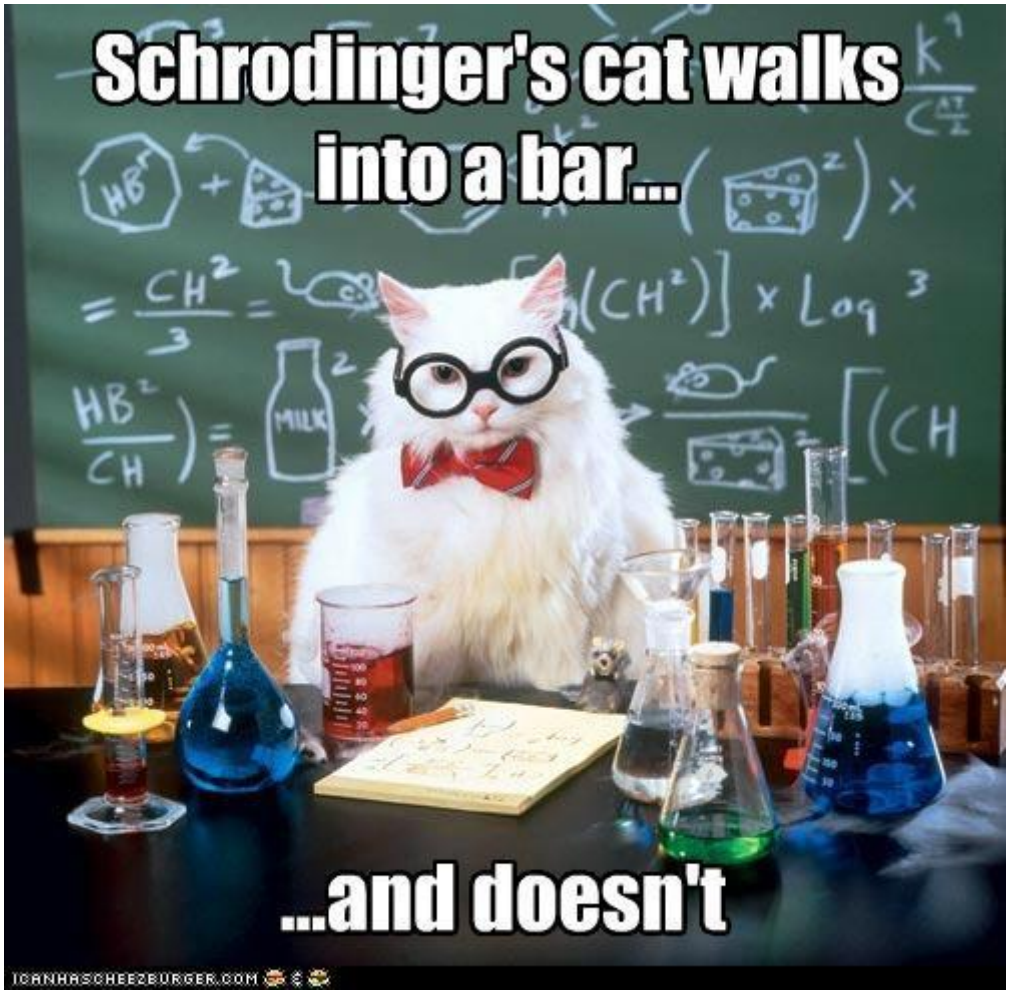
```
// IntOrString is an int that may be unmarshaled from either a JSON number
// literal, or a JSON string.
type IntOrString int

func (i *IntOrString) UnmarshalJSON(d []byte) error {
    var v int
    err := json.Unmarshal(bytes.Trim(d, `"`), &v)
    *i = IntOrString(v)
    return err
}
```

Number literal, or string to number

```
func main() {
    data := []byte(`[123,"321"]`)
    x := make([]IntOrString, 0)
    if err := json.Unmarshal(data, &x); err != nil {
        panic(err)
    }
    spew.Dump(x)
}
```

Run

Array or single element



Array or single element

Some sloppy APIs return either a single element, or an array of elements.

One result:

```
{"results": {"id": 1, ...} }
```

Multiple results:

```
{"results": [{"id": 1, ...}, {"id": 2, ...}] }
```

Array or single element

```
// SliceOrString is a slice of strings that may be unmarshaled from either
// a JSON array of strings, or a single JSON string.
type SliceOrString []string

func (s *SliceOrString) UnmarshalJSON(d []byte) error {
    if d[0] == '"' {
        var v string
        err := json.Unmarshal(d, &v)
        *s = SliceOrString{v}
        return err
    }
    var v []string
    err := json.Unmarshal(d, &v)
    *s = SliceOrString(v)
    return err
}
```

Array or single element

```
func main() {
    data := []byte(`["one", ["two","elements"]]`)
    x := make([]SliceOrString, 0)
    if err := json.Unmarshal(data, &x); err != nil {
        panic(err)
    }
    spew.Dump(x)
}
```

Run

Unknown types

Unknown types

Many APIs will return either a successful response, or an error. Some use common fields (i.e. `status`) across both types of responses, some don't.

Success:

```
{"results": [ ... ]}
```

Failure:

```
{"error":"not found", "reason":"The requested object does not exist"}
```

There are a number of approaches to this problem.

Unknown types

First, create our distinct success and error types...

```
type Success struct {
    Results []string `json:"results"`
}
```

```
type Error struct {
    Error  string `json:"error"`
    Reason string `json:"reason"`
}
```

Unknown types

In this simple example, because no fields are shared between the types, we can simply embed both types in a wrapper.

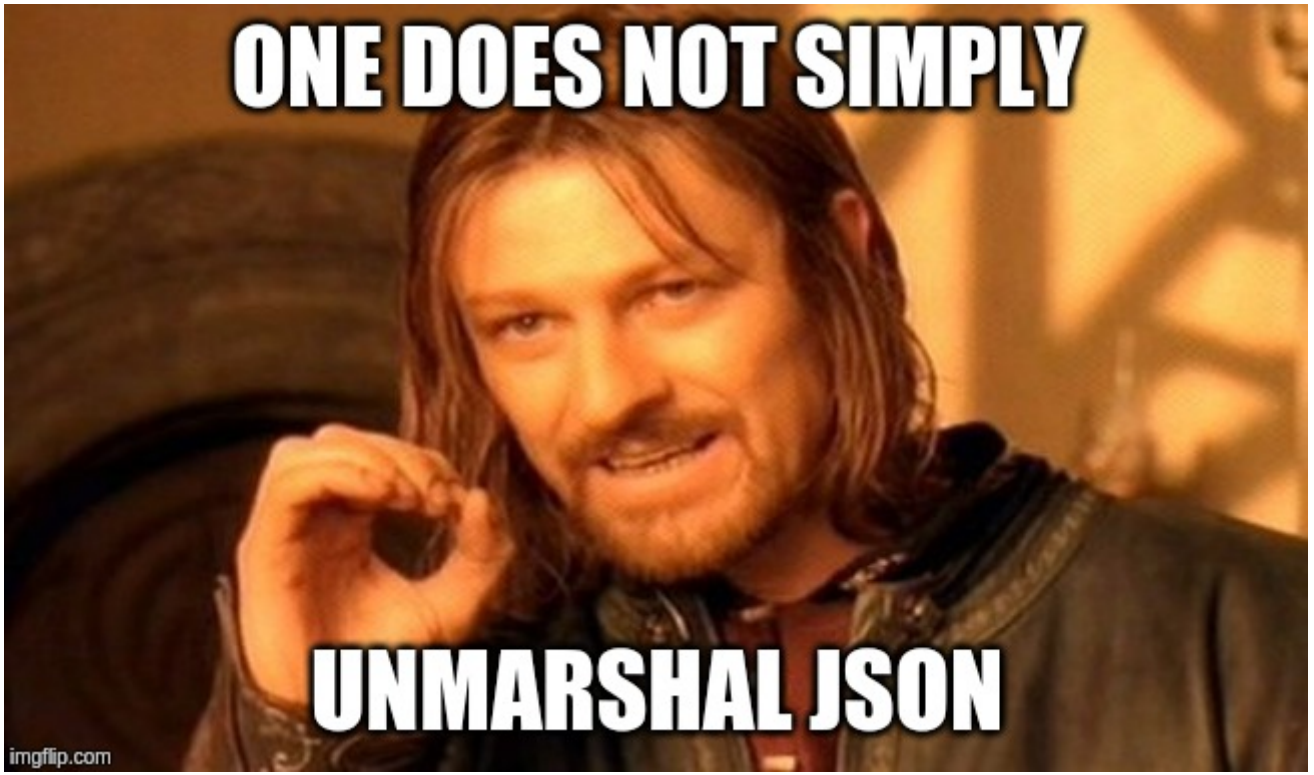
```
type Response struct {
    Success
    Error
}
```

Unknown types

```
func main() {
    data := []byte(`[
        {"results": ["one", "two"]},
        {"error":"not found", "reason": "The requested object does not exist"}
    ]`)
    x := make([]Response, 0)
    if err := json.Unmarshal(data, &x); err != nil {
        panic(err)
    }
    spew.Dump(x)
}
```

Run

Unknown types, #2



Unknown types, #2

Let's imagine a less straight-forward situation:

Success:

```
{"status":"ok", "results": [ ... ]}
```

Failure:

```
{"status":"not found", "reason":"The requested object does not exist"}
```

Unknown types, #2

First, create our distinct success and error types...

```
type Success struct {
    Status  string   `json:"status"`
    Results []string `json:"results"`
}
```

```
type Error struct {
    Status string `json:"status"`
    Reason string `json:"reason"`
}
```

Unknown types, #2

Because the embedded fields conflict, we must be more explicit.

```
type Response struct {
    Success
    Error
}

func (r *Response) UnmarshalJSON(d []byte) error {
    if err := json.Unmarshal(d, &r.Success); err != nil {
        return err
    }
    if r.Success.Status == "ok" {
        return nil
    }
    r.Success = Success{}
    return json.Unmarshal(d, &r.Error)
}
```

Unknown types, #2

```
func main() {
    data := []byte(`[
        {"status":"ok","results": ["one", "two"]},
        {"status":"not found", "reason": "The requested object does not exist"}
    ]`)
    x := make([]Response, 0)
    if err := json.Unmarshal(data, &x); err != nil {
        panic(err)
    }
    spew.Dump(x)
}
```

Run

Using a container

Using a container

Bundling success and error objects like this can be cumbersome. We can use a different container type.

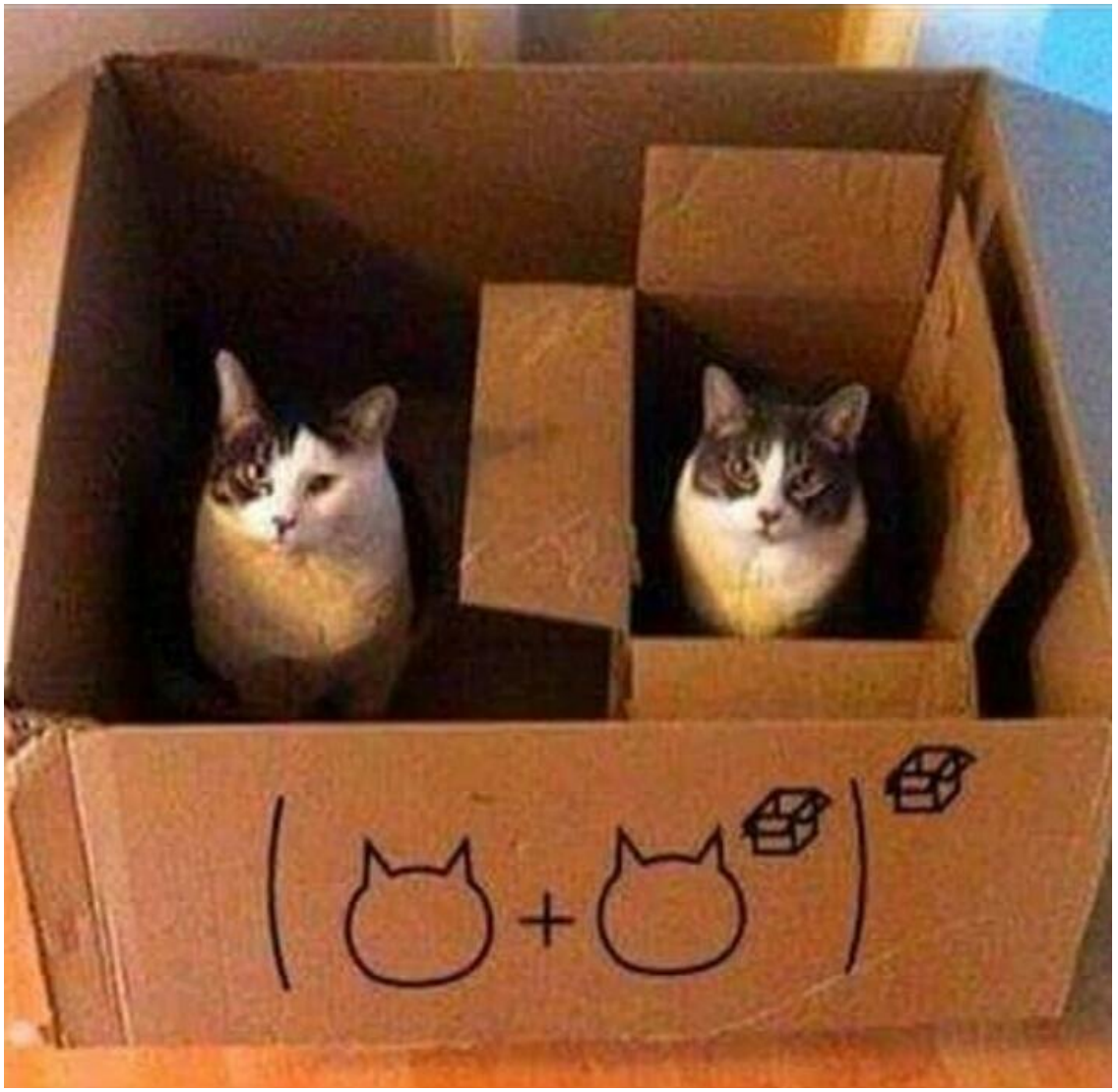
```
type Response struct {
    Result interface{}
}

func (r *Response) UnmarshalJSON(d []byte) error {
    var success Success
    if err := json.Unmarshal(d, &success); err != nil {
        return err
    }
    if success.Status == "ok" {
        r.Result = success
        return nil
    }
    var fail Error
    if err := json.Unmarshal(d, &fail); err != nil {
        return err
    }
    r.Result = fail
    return nil
}
```

Using a container

```
func main() {
    data := []byte(`[
        {"status":"ok","results": ["one", "two"]},
        {"status":"not found", "reason": "The requested object does not exist"}
    ]`)
    x := make([]Response, 0)
    if err := json.Unmarshal(data, &x); err != nil {
        panic(err)
    }
    spew.Dump(x)
}
```

Run

Using a container, #2



Using a container, #2

The container can also be a slice (or map) of objects, rather than a single object.

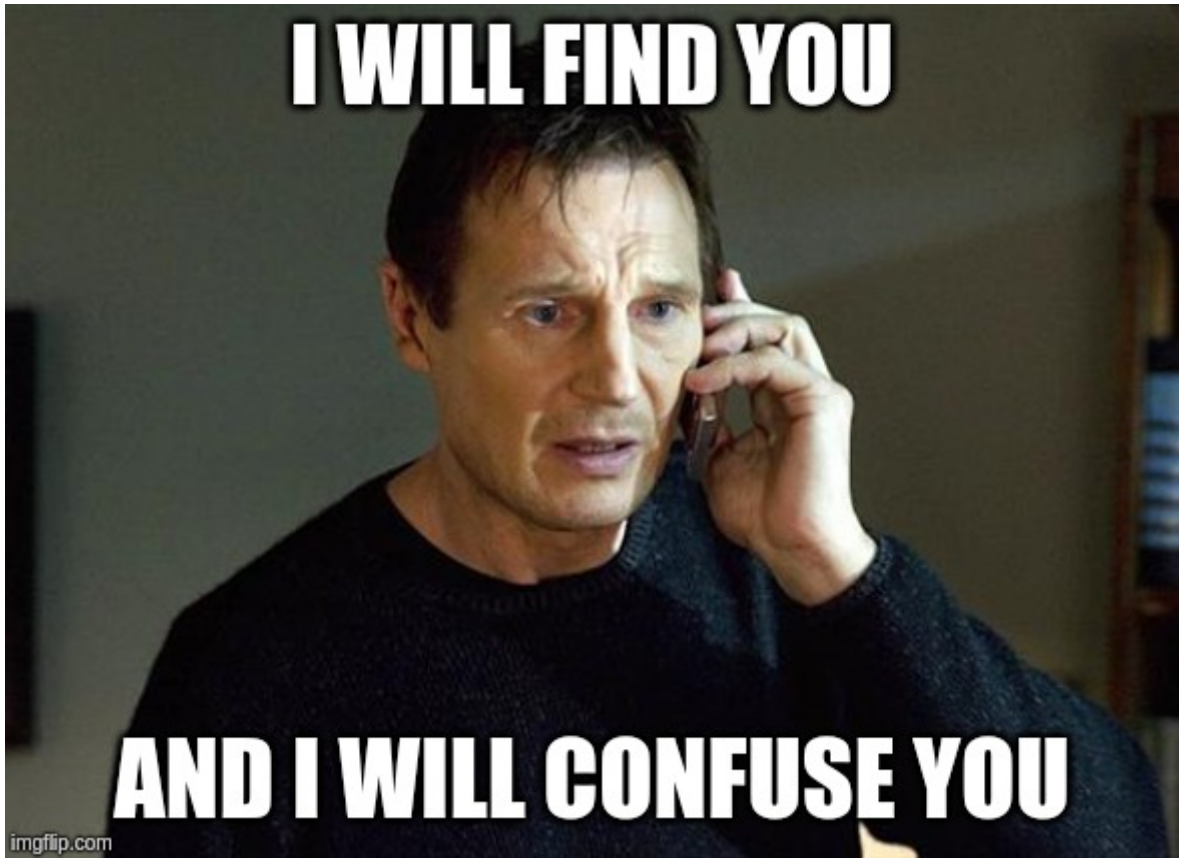
```
type Responses []interface{}

func (r *Responses) UnmarshalJSON(d []byte) error {
    var raw []json.RawMessage
    if err := json.Unmarshal(d, &raw); err != nil {
        return err
    }
    var success Success
    var fail Error
    result := make(Responses, len(raw))
    for i, msg := range raw {
        _ = json.Unmarshal(msg, &success)
        if success.Status == "ok" {
            result[i] = success
            continue
        }
        _ = json.Unmarshal(msg, &fail)
        result[i] = fail
    }
    *r = result
    return nil
}
```

Using a container, #2

```
func main() {
    data := []byte(`[
        {"status":"ok","results": ["one", "two"]},
        {"status":"not found", "reason": "The requested object does not exist"}
    ]`)
    var x Responses
    if err := json.Unmarshal(data, &x); err != nil {
        panic(err)
    }
    spew.Dump(x)
}
```

Run

Unknown types, #3



Unknown types, #3

Let's suppose we may receive any of three types:

```
{"type": "person", "name": "Bob", "age": 32}

{"type": "animal", "species": "dog", "name": "Spot"}

{"type": "address", "city": "Rotterdam", "street": "Goudsesingel"}
```

Unkwnown tyes, #3

```
type Person struct {
    Name string `json:"name"`
    Age  int    `json:"age"`
}

type Animal struct {
    Species string `json:"species"`
    Name    string `json:"name"`
}

type Address struct {
    City   string `json:"city"`
    Street string `json:"street"`
}

type Responses []interface{}
```

Unknown types, #3

```
func (r *Responses) UnmarshalJSON(d []byte) error {
    var tmp []json.RawMessage
    if err := json.Unmarshal(d, &tmp); err != nil {
        return err
    }
    var header struct {
        Type string `json:"type"`
    }
    result := make(Responses, len(tmp))
```

continued...

Unknown types, #3

```
    for i, raw := range tmp {
        _ = json.Unmarshal(raw, &header)
        switch header.Type {
        case "person":
            tgt := Person{}
            _ = json.Unmarshal(raw, &tgt)
            result[i] = tgt
        case "animal":
            tgt := Animal{}
            _ = json.Unmarshal(raw, &tgt)
            result[i] = tgt
        case "address":
            tgt := Address{}
            _ = json.Unmarshal(raw, &tgt)
            result[i] = tgt
        }
    }
    *r = result
    return nil
}
```

Unknown types, #3

```
func main() {
    data := []byte(`[
        {"type": "person", "name": "Bob", "age": 32},
        {"type": "animal", "species": "dog", "name": "Spot"},
        {"type": "address", "city": "Rotterdam", "street": "Goudsesingel"}
    ]`)
    var x Responses
    if err := json.Unmarshal(data, &x); err != nil {
        panic(err)
    }
    spew.Dump(x)
}
```

Run

Unknown types, #4

Unknown types, #4

The last example works well, but is not efficient. We can do better.

Unknown types, #4

```
func (r *Responses) UnmarshalJSON(d []byte) error {
    dec := json.NewDecoder(bytes.NewReader(d))
    _, _ = dec.Token() // Consume opening "["
    result := make(Responses, 0)
    var header struct {
        Type string `json:"type"`
    }
```

continued...

Unknown types, #4

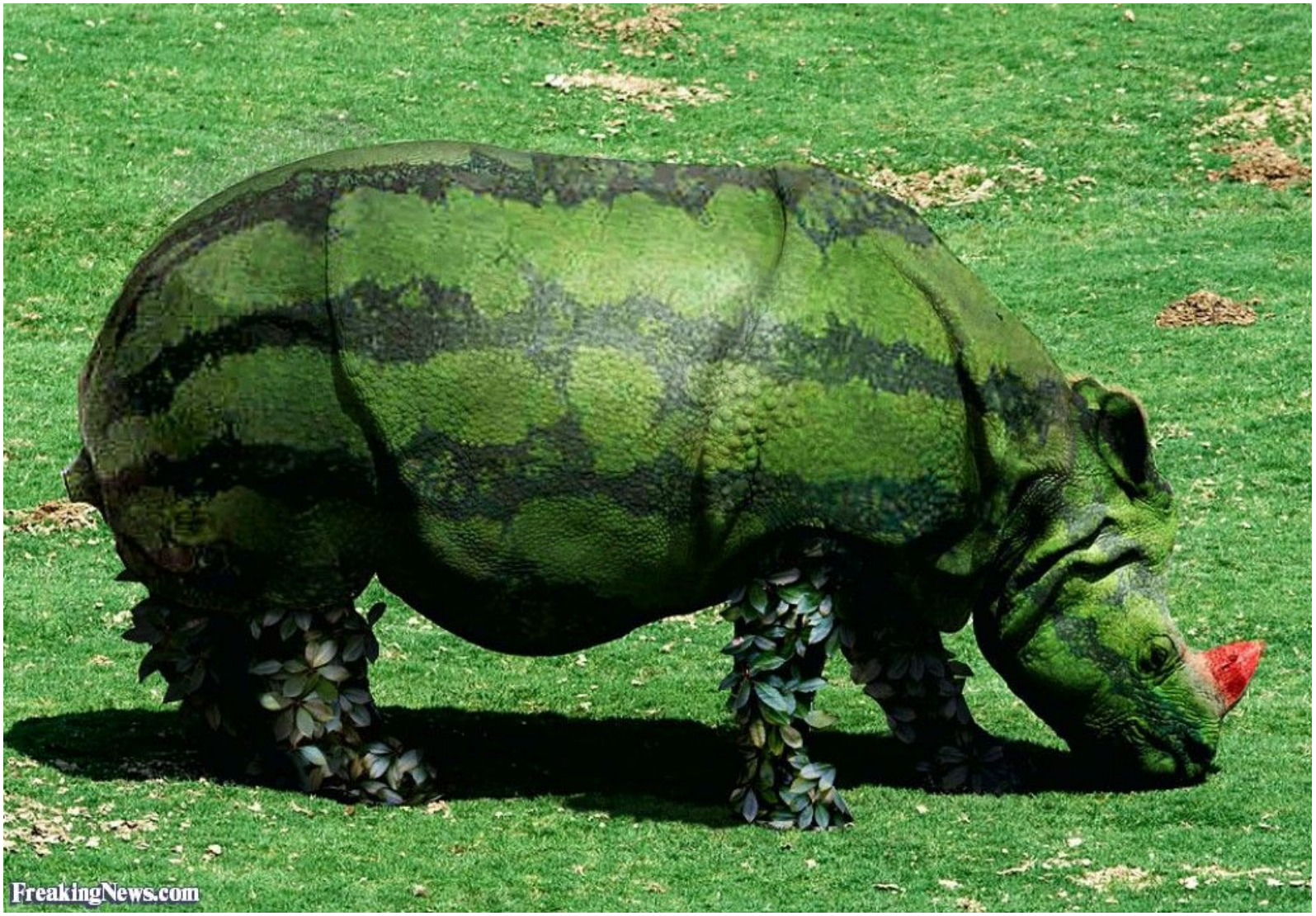
```
    var raw json.RawMessage
    for dec.More() {
        _ = dec.Decode(&raw)
        _ = json.Unmarshal(raw, &header)
        switch header.Type {
        case "person":
            tgt := Person{}
            _ = json.Unmarshal(raw, &tgt)
            result = append(result, tgt)
        case "animal":
            tgt := Animal{}
            _ = json.Unmarshal(raw, &tgt)
            result = append(result, tgt)
        case "address":
            tgt := Address{}
            _ = json.Unmarshal(raw, &tgt)
            result = append(result, tgt)
        }
    }
    *r = result
    return nil
}
```

Unknown types, #4

```
func main() {
    data := []byte(`[
        {"type": "person", "name": "Bob", "age": 32},
        {"type": "animal", "species": "dog", "name": "Spot"},
        {"type": "address", "city": "Rotterdam", "street": "Goudsesingel"}
    ]`)
    var x Responses
    if err := json.Unmarshal(data, &x); err != nil {
        panic(err)
    }
    spew.Dump(x)
}
```

Run

Hybrid struct/map



Hybrid struct/map

Maybe your API sends a response with some known, and some unknown fields.

```
{"_id":"bob","type":"user","name":"Bob"}

{"_id":"meetup","type":"website","url":"https://meetup.com/"}

{"_id":"soup","type":"recipe","ingredients":["broth","chicken","noodles"]}
```

Hybrid struct/map

```
type Item struct {
    ID   string                 `json:"_id"`
    Type string                 `json:"type"`
    Data map[string]interface{} `json:"-"`
}
```

Hybrid struct/map

```
func (i *Item) UnmarshalJSON(d []byte) error {
    var x struct {
        Item
        UnmarshalJSON struct{}
    }
    if err := json.Unmarshal(d, &x); err != nil {
        return err
    }

    var y map[string]interface{}
    _ = json.Unmarshal(d, &y)
    delete(y, "_id")
    delete(y, "type")
    *i = x.Item
    i.Data = y
    return nil
}
```

Hybrid struct/map

```
func main() {
    data := []byte(`[
        {"_id":"bob","type":"user","name":"Bob"},
        {"_id":"meetup","type":"website","url":"https://meetup.com/"},
        {"_id":"soup","type":"recipe","ingredients":["broth","chicken","noodles"]}
    ]`)
    x := []Item{}
    if err := json.Unmarshal(data, &x); err != nil {
        panic(err)
    }
    spew.Dump(x)
}
```

Run

Encoding a hybrid struct/map

Encoding a hybrid struct/map

```
func (i *Item) MarshalJSON() ([]byte, error) {
    data, _ := json.Marshal(i.Data)
    tmp := struct{
        *Item
        MarshalJSON struct{} `json:"-"`
    }{Item: i}
    obj, _ := json.Marshal(tmp)
    obj[len(obj)-1] = ','
    return append(obj, data[1:]...), nil
}
```

Encoding a hybrid struct/map

```
func main() {
    x := []Item{
        {ID: "bob", Type:"user",Data:map[string]interface{}{"name":"Bob"}},
        {ID:"meetup",Type:"website",Data:map[string]interface{}{"url":"https://meetup.com/"}},
        {ID:"soup",Type:"recipe",Data:map[string]interface{}{"ingredients":[]string{"broth","chicken","n
    }
    data, err := json.MarshalIndent(x, "", "  ")
    if err != nil {
        panic(err)
    }
    fmt.Println(string(data))
}
```

Run

Questions?

Notes, links, slides online:

jhall.io/posts/go-json-advanced (https://jhall.io/posts/go-json-advanced)

Thank you

19:40 05 Mar 2020

Jonathan Hall
DevOps Evangelist / Go Developer / Clean Coder / Salsa Dancer
jonathan@jhall.io (mailto:jonathan@jhall.io)
https://jhall.io/ (https://jhall.io/)
@DevOpsHabits (http://twitter.com/DevOpsHabits)