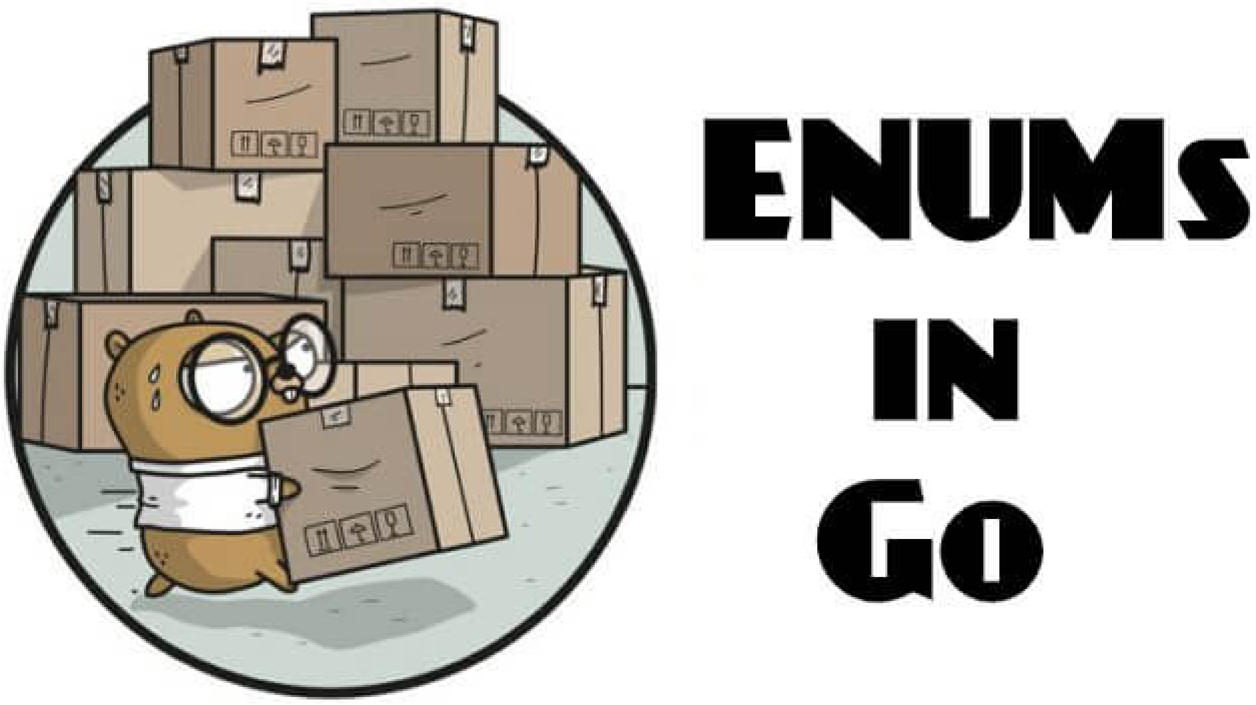
October 4, 2019 #programming | #golang

Working with enums in Go

Enums (short of enumerators), identifiers that behave like constants, are useful parts of many languages. Unfortunately, enums in Go aren't as useful due to Go's implementation. The biggest drawback is that they aren't strictly typed, thus you have to manually validate them.



Having a wide range of usages, ENUMs are a powerful feature of many languages. They let you define strict values of data you expect.

As an example, having an HTTP endpoint receiving employees leave type, the controller should allow only a few types, like ‘Annual Leave’, ‘Sick’, ‘Bank Holiday’, ‘Other’. In many languages, like Java, this would be defined as:

Copy

```
enum LeaveType {
  Annual Leave,
  Sick,
  Bank Holiday,
  Other
```

Now, if you expect the client to provide LeaveType, only these values will be accepted. Any other value will be returning `java.lang.IllegalArgumentException: No enum constant`.

With Go, it’s different. Most often you create a custom type (`LeaveType`) and define constants of that type. In the case of integer custom type, `iota` keyword can be used to simplify the definition.

Copy

```
type LeaveType string

const(
    AnnualLeave LeaveType = "AnnualLeave"
    Sick = "Sick"
    BankHoliday = "BankHoliday"
    Other = "Other"
)
```

If you’re expecting `LeaveType` in your controller, the client can send any valid string and there will be no errors. Why? Well, `LeaveType` is just a string, not an enum, thus allowing any valid string to be provided.

Copy

```
package main

import (
    "fmt"
)
```

```go
 6
 7    type LeaveType string
 8
 9    const(
10        AnnualLeave LeaveType = "AnnualLeave"
11        Sick = "Sick"
12        BankHoliday = "BankHoliday"
13        Other = "Other"
14    )
15
16    func main() {
17        var a LeaveType
18        a = "Hello"
19        fmt.Println(a)
20    }
```

The program above prints `"Hello"`. If it was a proper enum type, line 18 would fail as “Hello” is not a valid value for `LeaveType`.

There are a few ways to deal with this. When creating REST APIs I do one of the two:

If an empty value is acceptable for enum, I validate the values during JSON marshaling by writing a custom unmarshal function.

By using json.Unmarshal, we can provide nice errors if the wrong type is provided (for example int instead of string returns `json: cannot unmarshal number into Go value of type main.LT`):

Copy

```go
func (lt *LeaveType) UnmarshalJSON(b []byte) error {
    // Define a secondary type to avoid ending up with a recursive call t
    type LT LeaveType;
    var r *LT = (*LT)(lt);
    err := json.Unmarshal(b, &r)
    if err != nil{
        panic(err)
    }
    switch *lt {
    case AnnualLeave, Sick, BankHoliday, Other:
        return nil
    }
    return errors.New("Invalid leave type")
}
```

Keep in mind that above leaveType will hold the invalid value. If you don't return on errors, this may cause unexpected issues.

The other option is to unmarshal into a string and convert it to desired type:

Copy

```go
func (lt *LeaveType) UnmarshalJSON(b []byte) error {
    var s string
    json.Unmarshal(b, &s)
    leaveType := LeaveType(s)
    switch leaveType {
    case AnnualLeave, Sick, BankHoliday, Other:
        *lt = leaveType
        return nil
    }
    return errors.New("Invalid leave type")
}
```

The problem with the above implementation is that empty values are not being unmarshalled, thus you won't be getting an error if an empty value is provided and you don't want to support empty values.

Creating an `IsValid` function on LeaveType and call it after the values are unmarshalled:

Copy

```go
func (lt LeaveType) IsValid() error {
    switch lt {
    case AnnualLeave, Sick, BankHoliday, Other:
        return nil
    }
    return errors.New("Invalid leave type")
}
```

This prevents empty values, but involves additional check whenever you need to validate the value, which is more error-prone in case you forget it and ‘uglifies’ the code with repetition:

Copy

```
if err := r.LeaveType.IsValid(); err != nil{
    return nil, err
}
```

There are open proposals to implement proper enum support in Go 2, but for now, there are no better ways to handle enums.

Similar articles:

- dt - Go's missing datetime package
- Building a live chat with Go, NATS, Redis and Websockets
- Google Datastore with a relational data model
- Glice v2 - Adding support for go.mod
- HTTP logging in Go