Error Handling and Testing in Go

Anthony Lee

Errors

Typical

```
        tkn, err := s.Tokener.Token(ctx)
        if err != nil {
            http.Error(w, "could not retrieve token", http.StatusInternalServerError)
            return
        }
```

"Short"

```
        if err := t.UpdateToken(ctx); err != nil {
            return nil, errors.Wrap(err, "token update err")
        }
```

Long and deep history of discussions

Dave Cheney (2016)(https://dave.cheney.net/2016/04/27/dont-just-check-errors-handle-them-gracefully)

David Cheney (2019)(https://dave.cheney.net/2019/01/27/eliminate-error-handling-by-eliminating-errors)

Andrew Gerrand (2011)(https://blog.golang.org/error-handling-and-go)

Common errors that are easy to ignore

```
func Unmarshal(data []byte, v interface{}) error

// Do not:
json.Unmarshal(data, whiskey)
fmt.Println(whiskey) // Could panic!

// Do:
if err := json.Unmarshal(data, baloney); err != nil {
    // handle or punt
    return errors.Wrap(err, "json unmarshal failed")
}
fmt.Println(baloney)
```

Not checking errors leads to nil pointer dereferences or malformed data structures.

Errors and go routines

```
go func() {
    if err := doStuff(); err != nil {
        // ????
        // Panic?
        // Log?
        // Return to the ether?
    }
}
```

Go routines often run indefinitely or return afer the parent's scope has exited.

Handle it!

Generally, WaitGroups or channels are used to manage cooperative go routines.

ErrGroup (https://godoc.org/golang.org/x/sync/errgroup)

```
...
type Do func(interface{}) error

group, ctx := errgroup.WithContext(context.Background())
for _, task := tasks {
    // Note: unbounded concurrency for simplicity!
    task := task
    group.Go(func() error{
       return Do(task)
    })
}

if err := group.Wait() {
    return err
}
...
```

Using channels

ErrGroups often assume that each routine is critical for a cooperative effort, but there are cases where we don't really care if a single thread fails.

In situations where either:

- Individual failures should not halt concurrent executions.
- Only clients have the context required to effectively handle errors.

Use the concurrency primitive for thread safe communication: channels!

Using channels

```
...
func Do(tasks <-chan Task) (<-chan Product, <-chan error){
    products := make(chan Product)
    errors := make(chan error)
    go func() {
        defer close(products)
        defer close(Errors)
        // Note: unbounded concurrency for simplicity!
        for t := range tasks{
            go func(t Task) {
                p, err := Produce(t)
                if err != nil {
                    errors <- err
                    return
                }
                products <- product
            }(t)
        }
    }()
    return products, errors
}
...
```

Testing

Testing can be fun! ...ish

There's no need for testing frameworks in Go.

```
// via pkg/testing docs.
func TestAbs(t *testing.T) {
    got := Abs(-1)
    if got != 1 {
        t.Errorf("Abs(-1) = %d; want 1", got)
    }
}
```

Tests are like writing client code.

Table testing

Instead of writing out individual test funcs for each case, use tables.

Go Wiki (https://github.com/golang/go/wiki/TableDrivenTests)

```
func TestAbs(t *testing.T){
    // Create a slice of anonymous struct literals with testing components.
    var tests = []struct{
        arg int
        want int
    }{
        {
            arg: -1,
            want: 1,
        },
        {
            arg: 0,
            want: 0,
        },
        {
            arg: 1,
            want: 1,
        },
    }
    ...
```

Table testing (continued)

```
...
    // Iterate over test cases, performing any set up and tear down needed.
    for _, tt := range tests {
        t.Run(tt.in, func(t *testing.T) {
            got := Abs(t.arg)
            // Evaluate expectations and react appropriately.
            if got != tt.want {
                t.Errorf("Abs(%d) = %d; want %d", got, tt.want)
            }
        })
    }
}
```

Testing toast

```
// NewToaster initializes a Toaster.
func NewToaster(client *http.Client, cfg Config) *Toaster {
    if client == nil {
        client = http.DefaultClient
    }
    if cfg.RefreshIntervalMS < 1 {
        cfg.RefreshIntervalMS = int(time.Hour * 8 / time.Millisecond)
    }
    t := &Toaster{
        Client: client,
        Config: cfg,
        mu:     &sync.Mutex{},
    }
    return t
}
```

Initializers often set sensible defaults and perform validation checks on dependencies.
It'd be a good idea to make sure it's doing what we expect...

Testing toast (continued)

```
func TestNewToaster(t *testing.T) {
    type args struct {
        client *http.Client
        cfg    toast.Config
    }
    tests := []struct {
        name string
        args args
        want *toast.Toaster
    }{
        {
            name: "normal",
            args: args{
                client: http.DefaultClient,
                cfg: toast.Config{
                    RefreshIntervalMS: 3600,
                },
            },
            want: &toast.Toaster{
                Client: http.DefaultClient,
                Config: toast.Config{
                    RefreshIntervalMS: 3600,
                },
            },
        },
        {
            name: "missing config",
            want: &toast.Toaster{
                Client: http.DefaultClient,
                Config: toast.Config{
                    RefreshIntervalMS: int(time.Hour * 8 / time.Millisecond),
                },
            },
        },
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            opts := cmpopts.IgnoreUnexported(toast.Toaster{})
            if got := toast.NewToaster(tt.args.client, tt.args.cfg); !cmp.Equal(got, tt.want, opts) {
                t.Errorf(cmp.Diff(got, tt.want))
            }
        })
    }
}
```

Testing toast (continued)

```
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            opts := cmpopts.IgnoreUnexported(toast.Toaster{})
            if got := toast.NewToaster(tt.args.client, tt.args.cfg); !cmp.Equal(got, tt.want, opts) {
                t.Errorf(cmp.Diff(got, tt.want))
            }
        })
    }
}
```

Use cmp to compare more complex objects. (https://godoc.org/github.com/google/go-cmp/cmp)

Results

```
Running tool: /usr/local/opt/go/libexec/bin/go test -timeout 30s git.target.com/ae-authentication/toast/

=== RUN   TestNewToaster
=== RUN   TestNewToaster/normal
=== RUN   TestNewToaster/missing_config
--- PASS: TestNewToaster (0.00s)
    --- PASS: TestNewToaster/normal (0.00s)
    --- PASS: TestNewToaster/missing_config (0.00s)
PASS
coverage: 17.1% of statements
ok      git.target.com/ae-authentication/toast/pkg/toast    1.021s    coverage: 17.1% of statements
Success: Tests passed.
```

Run `go test` from your terminal.

Thank you

Anthony Lee