Error types

You can find all the code here

Creating your own types for errors can be an elegant way of tidying up your code, making your code easier to use and test.

Pedro on the Gopher Slack asks

> If I'm creating an error like `fmt.Errorf("%s must be foo, got %s", bar, baz)`, is there a way to test equality without comparing the string value?

Let's make up a function to help explore this idea.

```
// DumbGetter will get the string body of url if it gets a 200
func DumbGetter(url string) (string, error) {
        res, err := http.Get(url)

        if err != nil {
                return "", fmt.Errorf("problem fetching from %s, %v", url, err)
        }

        if res.StatusCode != http.StatusOK {
                return "", fmt.Errorf("did not get 200 from %s, got %d", url, res.Sta
        }

        defer res.Body.Close()
        body, _ := ioutil.ReadAll(res.Body) // ignoring err for brevity

        return string(body), nil
}
```

It's not uncommon to write a function that might fail for different reasons and we want to make sure we handle each scenario correctly.

As Pedro says, we *could* write a test for the status error like so.

```
t.Run("when you don't get a 200 you get a status error", func(t *testing.T) {
```

```
2
3         svr := httptest.NewServer(http.HandlerFunc(func(res http.ResponseWriter, req
4                 res.WriteHeader(http.StatusTeapot)
5         }))
6         defer svr.Close()
7
8         _, err := DumbGetter(svr.URL)
9
10        if err == nil {
11                t.Fatal("expected an error")
12        }
13
14        want := fmt.Sprintf("did not get 200 from %s, got %d", svr.URL, http.StatusTe
15        got := err.Error()
16
17        if got != want {
18                t.Errorf(`got "%v", want "%v"`, got, want)
19        }
20 })
```

This test creates a server which always returns `StatusTeapot` and then we use its URL as the argument to `DumbGetter` so we can see it handles non `200` responses correctly.

Problems with this way of testing

This book tries to emphasise *listen to your tests* and this test doesn't *feel* good:

- We're constructing the same string as production code does to test it
- It's annoying to read and write
- Is the exact error message string what we're *actually concerned with* ?

What does this tell us? The ergonomics of our test would be reflected on another bit of code trying to use our code.

How does a user of our code react to the specific kind of errors we return? The best they can do is look at the error string which is extremely error prone and horrible to write.

What we should do

With TDD we have the benefit of getting into the mindset of:

> How would *I* want to use this code?

What we could do for `DumbGetter` is provide a way for users to use the type system to understand what kind of error has happened.

What if `DumbGetter` could return us something like

```
type BadStatusError struct {
        URL    string
        Status int
}
```

Rather than a magical string, we have actual *data* to work with.

Let's change our existing test to reflect this need

```
t.Run("when you don't get a 200 you get a status error", func(t *testing.T) {

        svr := httptest.NewServer(http.HandlerFunc(func(res http.ResponseWriter, req
                res.WriteHeader(http.StatusTeapot)
        }))
        defer svr.Close()

        _, err := DumbGetter(svr.URL)

        if err == nil {
                t.Fatal("expected an error")
        }

        got, isStatusErr := err.(BadStatusError)

        if !isStatusErr {
                t.Fatalf("was not a BadStatusError, got %T", err)
        }

        want := BadStatusError{URL: svr.URL, Status: http.StatusTeapot}

```

```
        if got != want {
                t.Errorf("got %v, want %v", got, want)
        }
})
```

We'll have to make `BadStatusError` implement the error interface.

```
func (b BadStatusError) Error() string {
        return fmt.Sprintf("did not get 200 from %s, got %d", b.URL, b.Status)
}
```

What does the test do?

Instead of checking the exact string of the error, we are doing a type assertion on the error to see if it is a `BadStatusError`. This reflects our desire for the *kind* of error clearer. Assuming the assertion passes we can then check the properties of the error are correct.

When we run the test, it tells us we didn't return the right kind of error

```
--- FAIL: TestDumbGetter (0.00s)
    --- FAIL: TestDumbGetter/when_you_dont_get_a_200_you_get_a_status_error (0.00s)
        error-types_test.go:56: was not a BadStatusError, got *errors.errorString
```

Let's fix `DumbGetter` by updating our error handling code to use our type

```
if res.StatusCode != http.StatusOK {
        return "", BadStatusError{URL: url, Status: res.StatusCode}
}
```

This change has had some *real positive effects*

- Our `DumbGetter` function has become simpler, it's no longer concerned with the intricacies of an error string, it just creates a `BadStatusError`.
- Our tests now reflect (and document) what a user of our code *could* do if they decided they wanted to do some more sophisticated error handling than just logging. Just do a type assertion and then you get easy access to the properties of the error.

- It is still "just" an `error`, so if they choose to they can pass it up the call stack or log it like any other `error`.

Wrapping up

If you find yourself testing for multiple error conditions don't fall in to the trap of comparing the error messages.

This leads to flaky and difficult to read/write tests and it reflects the difficulties the users of your code will have if they also need to start doing things differently depending on the kind of errors that have occurred.

Always make sure your tests reflect how *you'd* like to use your code, so in this respect consider creating error types to encapsulate your kinds of errors. This makes handling different kinds of errors easier for users of your code and also makes writing your error handling code simpler and easier to read.

Addendum

As of Go 1.13 there are new ways to work with errors in the standard library which is covered in the Go Blog

```
t.Run("when you don't get a 200 you get a status error", func(t *testing.T) {

        svr := httptest.NewServer(http.HandlerFunc(func(res http.ResponseWriter, req
                res.WriteHeader(http.StatusTeapot)
        }))
        defer svr.Close()

        _, err := DumbGetter(svr.URL)

        if err == nil {
                t.Fatal("expected an error")
        }

        var got BadStatusError
        isBadStatusError := errors.As(err, &got)
        want := BadStatusError{URL: svr.URL, Status: http.StatusTeapot}

        if !isBadStatusError {
```

```
20          }       t.Fatalf("was not a BadStatusError, got %T", err)

21
22          if got != want {
23                  t.Errorf("got %v, want %v", got, want)
24          }
25  })
```

In this case we are using `errors.As` to try and extract our error into our custom type. It returns a `bool` to denote success and extracts it into `got` for us.