— title: “Why I like Go’s error handling” date: 2023-03-27T08:30:43-03:00 draft: false —

One of the most controversial issues in the Go language is how we handle errors. I remember when I started working with the language, in mid-2015, after having used PHP for a few years, and I was surprised that it didn’t use the famous **try/catch**.

After the first impact passed, I tried to understand why and the official answer was, “in Go, errors are first-class citizens.” Therefore, you are responsible for handling them explicitly.

I will show an example to illustrate this. I recently came across the following code in Java:

```
private JsonNode get(String query) {
    try {
        SimpleHttp.Response response = SimpleHttp.doGet(query, this.session)
ectionRequestTimeoutMillis(Integer.parseInt(model.get(ConsumerProviderConstants.TIMEOUT_CONSUMER)))
.connectTimeoutMillis(Integer.parseInt(model.get(ConsumerProviderConstants.CONN_TIMEOUT_CONSUMER)))
socketTimeOutMillis(Integer.parseInt(model.get(ConsumerProviderConstants.SOCKET_TIMEOUT_CONSUMER)))
                .header(HttpHeaders.CONTENT_TYPE, "application/json; charset=utf-8")
                .asResponse();
        switch (response.getStatus()) {
            case HttpStatus.SC_OK:
                return response.asJson();
            default:
                logger.errorf("private get(%s) - ResponseBody: {%s} -
StatusCode(%d)", query, response.asString(),
                        response.getStatus());
                return null;
        }
    } catch (Exception e) {
        throw new ServiceException("Error on retrieve data");
    }
}
```

I got to this code after finding some **“Error on retrieve data”** occurrences in the application logs.

What’s the problem with the code above? Finding which part of the code is causing the error is tricky. It can be in **SimpleHttp.doGet**, **model.get**, **Integer.parseInt**, etc.

It took a refactoring so that we could make the code more robust. As a result, it looks like this (we could do more refactorings, but for this example, it’s enough):

```
public int tryParseInt(String stringToParse, int defaultValue, String varName) {
    try {
        return Integer.parseInt(stringToParse);
    } catch (NumberFormatException ex) {
        logger.errorf("tryParseInt(String %s, varName %s)", stringToParse, varName);
        return defaultValue;
    }
}

private JsonNode get(String query) {
    try {
        SimpleHttp.Response response = SimpleHttp.doGet(query, this.session)
                .connectionRequestTimeoutMillis(

this.tryParseInt(model.get(ConsumerProviderConstants.TIMEOUT_CONSUMER), 1000,
                                    ConsumerProviderConstants.TIMEOUT_CONSUMER))
 .connectTimeoutMillis(this.tryParseInt(model.get(ConsumerProviderConstants.CONN_TIMEOUT_CONSUMER),
                                1000, ConsumerProviderConstants.CONN_TIMEOUT_CONSUMER))
```

```
.socketTimeOutMillis(this.tryParseInt(model.get(ConsumerProviderConstants.SOCKET_TIMEOUT_CONSUMER),
                                5000, ConsumerProviderConstants.SOCKET_TIMEOUT_CONSUMER))
                        .header(HttpHeaders.CONTENT_TYPE, "application/json; charset=utf-8")
                        .asResponse();

                if (response == null) {
                    logger.errorf("Response was return null");
                    throw new ServiceException("Error response null");
                }

                switch (response.getStatus()) {
                    case HttpStatus.SC_OK:
                        return response.asJson();
                    default:
                        logger.errorf("private get(%s) - ResponseBody: {%s} -
            StatusCode(%d)", query, response.asString(),
                                response.getStatus());
                        return null;
                }
            } catch (Exception e) {
                logger.errorf("ConsumerService Response error (%s)", e);
                throw new ServiceException("Error on retrieve data");
            }
        }
```

Much better because now we have identified that the error was happening when converting the **TIMEOUT_CONSUMER** environment variable to an integer since it was empty. A point of attention concerning this new code is that now the **tryParseInt** function knows and handles the **NumberFormatException**. The new code generated a coupling between them. If, at some point, the name of this exception changes or if the class starts to throw new exceptions, we may need to change our code to adapt to this change.

But what does this have to do with Go? The problem presented here doesn't happen only with Java, or the culprit is the concept of **try/catch**, much less from the person who wrote the code or reviewed it (by the way, I was one of the people who reviewed this code). My point is that Java (and other languages) allows this construct.

Meanwhile, being a strongly opinionated language, Go makes this type of situation much more difficult. So this is my interpretation of the code above in Go:

```
package main

import (
"encoding/json"
"fmt"
"io/ioutil"
"net/http"
"strconv"
"time"
)

const (
timeoutConsumer = "timeout.consumer"
)

type model struct{}
```

```
// just to emulate the model
func (m model) get(name string) string {
return "0"
}

func main() {
json, err := getJSON("http://url_to_be_consumed")
if err != nil {
panic(err) // handle error
}
fmt.Println(json)
}

func getJSON(query string) (string, error) {
m := model{}
tc, err := strconv.Atoi(m.get(timeoutConsumer))
if err != nil {
return "", fmt.Errorf("error converting timeout: %w", err)
}
req, err := http.NewRequest(http.MethodGet, query, nil)
if err != nil {
return "", fmt.Errorf("error creating request: %w", err)
}
req.Header.Set("Content-Type", "application/json; charset=utf-8")

client := http.Client{
Timeout: time.Duration(tc) * time.Second,
}
res, err := client.Do(req)
if err != nil {
return "", fmt.Errorf("error executing request: %w", err)
}
if res == nil {
return "", fmt.Errorf("empty response")
}
if res.StatusCode != http.StatusOK {
return "", fmt.Errorf("expected %d received %d", http.StatusOK, res.StatusCode)
}

resBody, err := ioutil.ReadAll(res.Body)
if err != nil {
return "", fmt.Errorf("error reading body: %w", err)
}
var result map[string]any
err = json.Unmarshal(resBody, &result)
if err != nil {
return "", fmt.Errorf("error parsing json: %w", err)
}
jsonStr, err := json.Marshal(result)
if err != nil {
return "", fmt.Errorf("error converting json: %w", err)
}
return string(jsonStr), nil
}
```

The code has gotten a lot more verbose. More lines to write the same functionality, and “OMG, that bunch of if/else!!!”. I agree with that argument. Indeed, the code became longer, but its reading

became much more apparent. The fact that each function always returns an error forces whoever is writing the code to handle or ignore it, which would be very visible. For example, it would be possible to change the snippet:

```
tc, err := strconv.Atoi(m.get(timeoutConsumer))
if err != nil {
  return "", fmt.Errorf("error converting timeout: %w", err)
}
```

By:

```
tc, _ := strconv.Atoi(m.get(timeoutConsumer))
```

But in that case, anyone could, in the code review, make the observation:

> Hey, are you ignoring the conversion error? Are you sure it's okay to do that?

And as we saw in this example, conversion errors can happen :)

Here is my account of why I like how Go handles errors. Of course, it could be better as other languages do it differently. It's verbose, but I'd prefer to write more lines and have an easier-to-maintain code.

Originally published at https://eltonminetto.dev on January 25, 2023.