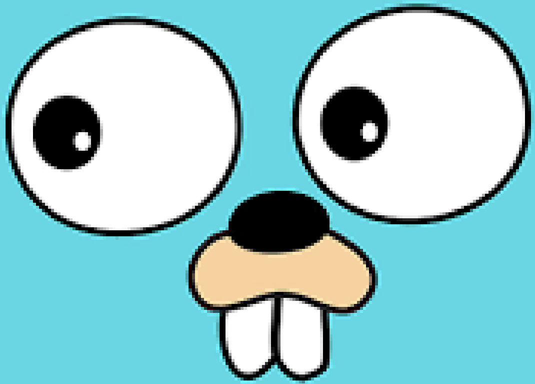


Advanced Error Handling

Discussion Points

- Custom errors
- Error wrapping
- Best practices in error handling

Custom errors

Error Philosophy in Go?

Never ignore errors because ignoring them can lead to program crashes.

The Go way to handle errors is for functions and methods to return an error object as their only or last return value—or nil if no error occurred—and for the code calling functions to always check the error they receive.

Go makes a distinction between **critical** and **non-critical** errors:

- non-critical errors are returned as normal return values
- critical errors, the panic-recover mechanism is used.

Custom errors

Error Philosophy in Go?

Errors are not exceptions: When an error occurs, it is propagated up the call stack until it is handled. There's no try...catch or try...except mechanism to handle errors.

Two key points that are kept in mind while Golang error handling is:

- Keep it simple.
- Plan where it can go wrong.

Custom errors

What says convention about Error?

If the function needs to return more than one value, then the error is the last return value in the list. This is all by convention; there is nothing in the compiler that enforces this.

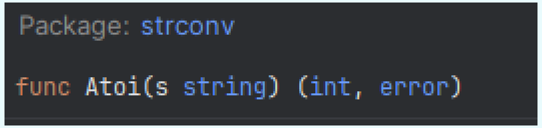
Custom errors

How to handle an Error?

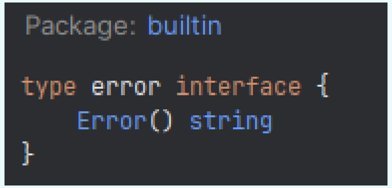
There are a few important points to note here:

- Errors can be returned as nil, and in fact, it’s the default, or “zero”, value of on error in Go. This is important since checking if err != nil is the idiomatic way to determine if an error was encountered
- Errors are typically returned as the last argument in a function
- Error messages should start with a lower-case letter
- Error messages should not have line breaks or periods

```go
num, err := strconv.Atoi("Twenty One")

if err != nil {
   // Handle error
}

fmt.Println(num)
```



```
Package: strconv

func Atoi(s string) (int, error)
```



```
Package: builtin

type error interface {
    Error() string
}
```

Custom errors

Constructing Error?

We can create variables of type error in two ways:

- By using **New()** function provided by built-in **errors** package
- By using **Errorf()** function provided by **fmt** package

```
err := errors.New("math - square root of negative number")
```

```
err := fmt.Errorf("math - square root of negative number %d", num)
```

Custom errors

Defining Custom Error Types?

While the strategy covers most error-handling scenarios, there may be situations where you need additional functionality. For instance, you might want an error to include extra data fields, or you may need the error message to dynamically populate with specific values when it's printed.

```go
func Sqrt(f float64) (float64, error) {
    if f < 0 {
      return 0, errors.New("math - square root of negative number")
    }
    // implementation of Sqrt - Square root
}

if f, err := Sqrt(-1); err != nil {
    fmt.Printf("Error: %s\n", err)
}
```

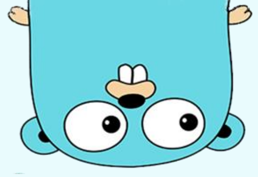
Custom errors

Handling Multiple Errors?

When a function can produce several different errors, it's crucial to handle each error case individually to ensure clear and helpful feedback is given to the user or calling function.

To achieve this, consider defining custom error types or specific error variables for each unique error case. Doing so allows you to address each error in a precise and focused way.

```go
var ErrDivideByZero = errors.New("cannot divide by zero")
var ErrInvalidInput = errors.New("invalid input")

func Divide(a, b int) (int, error) {
   if b == 0 {
      return 0, ErrDivideByZero
   }
   if a < 0 || b < 0 {
      return 0, ErrInvalidInput
   }
   return a / b, nil
}
```



Custom errors

Type Assertion of Errors?

Type assertions are extremely useful when working with errors. They help extract information from an interface value, and since error handling often involves custom implementations of the error interface, using assertions on errors becomes a powerful tool.

When managing errors, consider using type assertions or comparing error variables to handle each error scenario distinctly.

```go
result, err := Divide(-10, 0)

if errors.Is(err, ErrDivideByZero) {
   fmt.Println("Error: cannot divide by zero")
} else if errors.Is(err, ErrInvalidInput) {
   fmt.Println("Error: invalid input")
} else if err != nil {
   fmt.Println("Error:", err)
}
```

Error wrapping

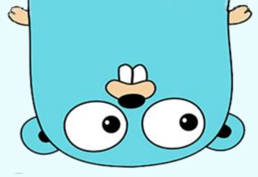
Wrapping Errors?

At times, you might need to add **extra context** to an error before passing it back to the caller. You can achieve this by using the **fmt.Errorf()** function, which lets you create a formatted error message while wrapping the original error.

```
// Capture connection properties.
cfg := mysql.Config{...}
// Get a database handle.
db, err := sql.Open("mysql", cfg.FormatDSN())

return fmt.Errorf("Failed to connect to database: %w", err)
```

Error wrapping

Wrapping Errors?



pkg/errors package also introduces the errors.

- Wrap and errors.
- Wrapf functions.

These functions add context to an error with a message and stack trace at the point they were called. This way, instead of simply returning an error, you can wrap it with its context and important debug data.

```go
if err != nil {
   return errors.Wrap(err, "Failed to connect to database")
}
```

Error wrapping

Wrapping Errors?

Unfortunately, the standard built-in errors do not include stack traces, which makes debugging challenging when you don't know where the error originated. The error might pass through several function calls before reaching the code that prints it out.

To address this issue, you can use the **pkg/errors** package. This package offers useful features like error wrapping, unwrapping, formatting, and stack trace recording. You can install it by running:

```
go get github.com/pkg/errors
```

Wrapping Errors?

```go
import "github.com/pkg/errors"

func BaseError() error {
   return errors.New("Cannot convert string to int:")
}

func check() error {
   _, err := strconv.Atoi("Twenty One")
   if err != nil {
     return BaseError()
   }
   return nil
}

func main() {
   fmt.Printf("Error: %+v", check())
}
```

```
// Error: Cannot convert string to int:
// main.BaseError
//        ./test.go:11
// main.check
//        ./test.go:20
// main.main
//        ./test.go:28
// runtime.main
//        ../Go/src/runtime/proc.go:272
// runtime.goexit
//        ../Go/src/runtime/asm_amd64.s:1700
// Process finished with the exit code 0
```

Error wrapping

Wrap or not to Wrap?

To address this question, need to examining potential scenarios where error wrapping is essential and analyzing their implications.

Error from a Third-Party Package Call:
When an error originates from a third-party package, wrapping it provides additional context about where the error occurred in your code. This makes debugging much easier by clarifying the source of the failure.
Error from a Dependency Call:
Similarly, errors from dependency calls benefit from wrapping. Without wrapping, it can be difficult to trace the error back through the dependency chain and understand the intent behind the failed function calls.

In both cases, without error wrapping, identifying the complete trace of the error and understanding why the function call failed would be considerably more challenging.

Best practices in error handling

Return Errors Early (Fail Fast)

Don’t Panic, Return Errors

Log Errors at the Right Level

Add Context to Errors

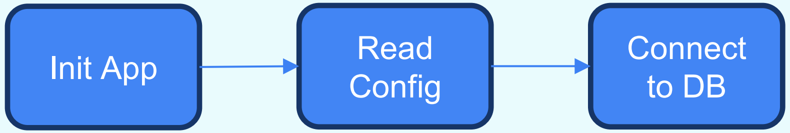
Use errors.Is and errors.As for Error Comparison

Use struct based errors types returning Error() method

Best practices in error handling

Error Handling in a Complex System

Consider a scenario where multiple functions are
involved in reading a configuration file and connecting to a database.
Error handling should be consistent and add relevant context at each level.



Thank you.

Questions?

- Custom errors
- Error wrapping
- Best practices in error handling

“Learn from yesterday, live for today, hope for tomorrow. The important thing is not to stop questioning.”

- Albert Einstein