● JANUARY 2026 SERIES

FROM GO BUILD
TO GO RUN

GOLANG 2026 - NIV RAVE

#22

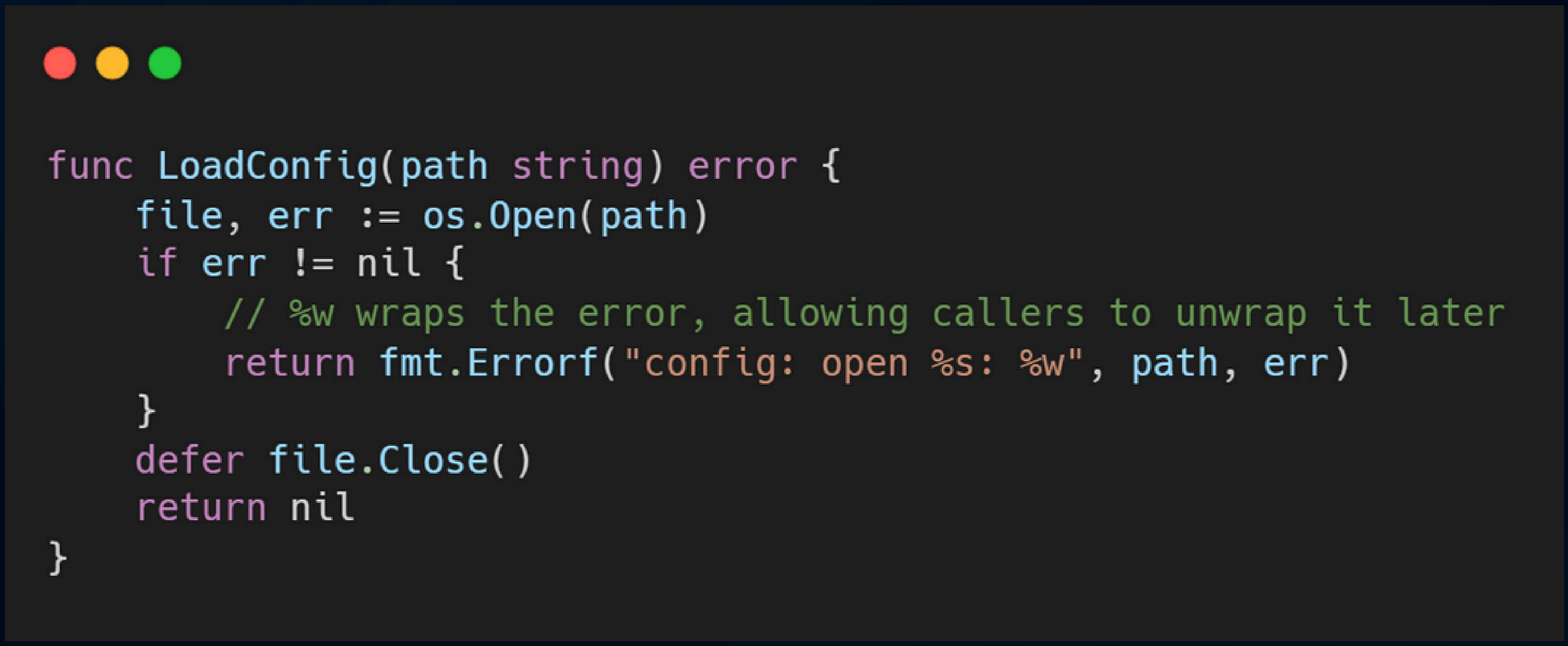
ERROR HANDLING - WRAPPING & CONTEXT

DON'T JUST RETURN THE ERROR; ADD VALUE TO IT

The *%w* Verb

Preserving the Chain

Since Go 1.13, we use *fmt.Errorf* with the *%w* verb to wrap an error. This adds context while preserving the original error for inspection.



```go
func LoadConfig(path string) error {
    file, err := os.Open(path)
    if err != nil {
        // %w wraps the error, allowing callers to unwrap it later
        return fmt.Errorf("config: open %s: %w", path, err)
    }
    defer file.Close()
    return nil
}
```

Wrapping creates a "Logical Stack Trace." It tells a story: "Config failed" because "Open failed" because "Permission denied."

Checking the Chain (*errors.Is*)

Detecting Specific Sentinels

Never use == to check a wrapped error. It will fail. Use *errors.Is*, which recursively unwraps the chain to find a match.

```go
err := LoadConfig("config.yaml")

// errors.Is looks deep into the wrapping layers
if errors.Is(err, os.ErrNotExist) {
    // We can handle 'missing file' differently than 'permission denied'
    return createDefaultConfig()
}
```

Retrieving Types (*errors.As*)

Extracting Detailed Context

If you need to access fields on a custom error struct hidden inside a chain, use *errors.As*.

```go
var perr *os.PathError
if errors.As(err, &perr) {
    // Now we have access to the Path, Op, and Err fields of PathError
    fmt.Printf("Operation %s failed on path %s", perr.Op, perr.Path)
}
```

Golden Rule: Use *As* when you need the "Why" (metadata), and *Is* when you only need the "What" (identity).

The "Handle or Return" Rule

Stop Double-Logging

A common junior mistake is logging an error and then returning it. This leads to 5 identical log entries for one single failure.

The Senior Strategy:

- **Option A**: Handle the error (log it and stop the process).
- **Option B**: Wrap it and return it (add context for the next guy).

One failure = One log entry

Multi-Errors (*errors.Join*)

Handling Multiple Failures (Go 1.20+)

Sometimes you run multiple operations (like closing 3 different resources) and want to collect all errors instead of just the first one.

```go
var errs error
errs = errors.Join(closeDB(), closeRedis(), closeFile())

if errs != nil {
    return errs // Contains all non-nil errors formatted together
}
```

Benefit: *errors.Is* and *errors.As* work on joined errors too! They will search through every error in the collection.

Concurrent Errors (*errgroup*)

Concurrency-Safe Error Handling

When running goroutines in parallel, *sync.WaitGroup* isn't enough because it doesn't handle errors. Use *golang.org/x/sync/errgroup*.

```go
g, ctx := errgroup.WithContext(mainCtx)

for _, url := range urls {
    url := url // avoid closure capture bug
    g.Go(func() error {
        return fetch(ctx, url)
    })
}

// Wait() returns the first error encountered by any goroutine
if err := g.Wait(); err != nil {
    return err
}
```

Privacy in Error Design

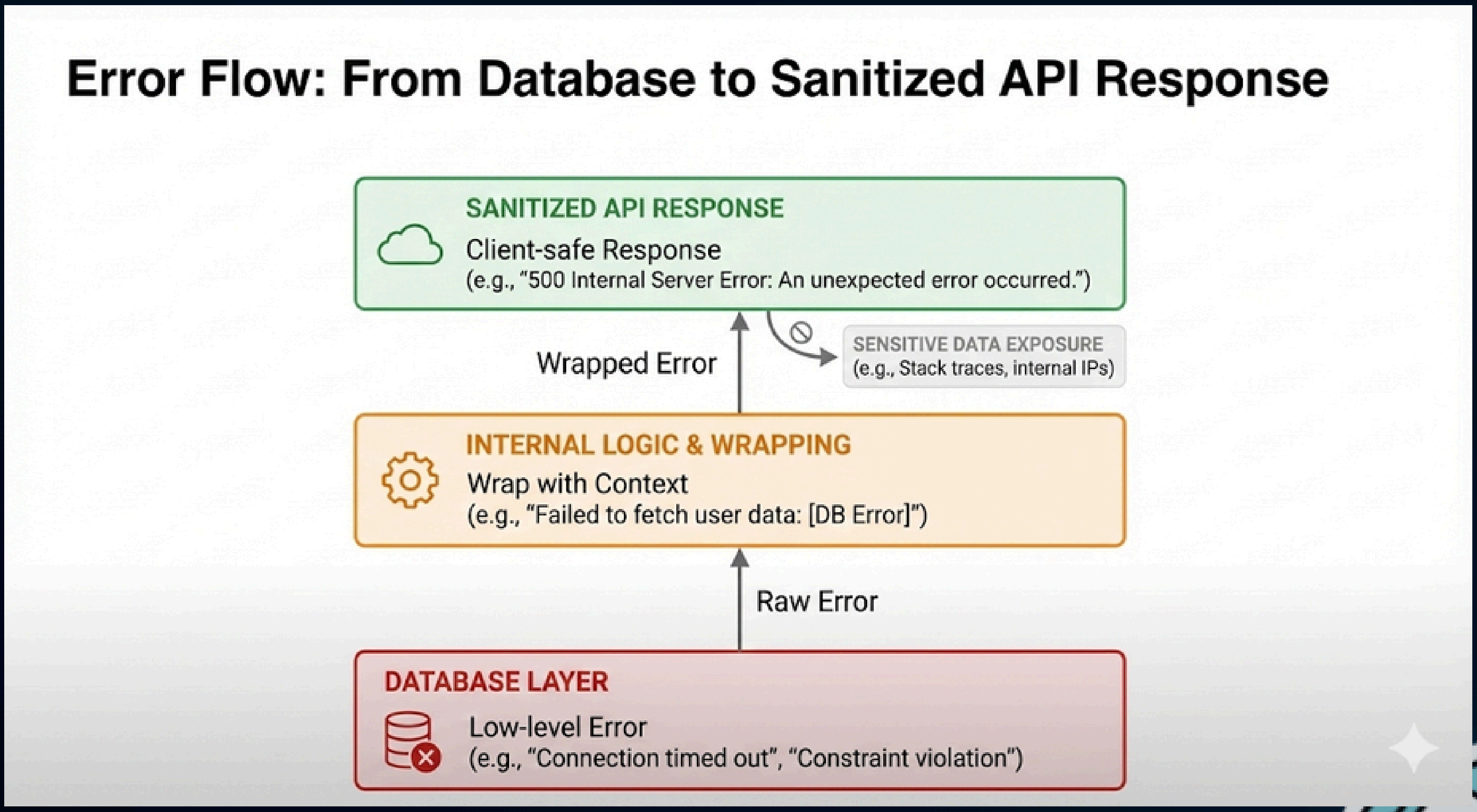
Error Sanitization (Internal vs External)

A database connection string or a SQL syntax error should never be sent to a client. It's a security risk.

The Pattern: Wrap internally for your logs, but use a "Sanitizer" at the API edge.

We would:

- Internal: *fmt.Errorf("db update failed for user %d: %w", id, queryErr)*
  External: *{"error": "Update failed. Please try again."}*


Error Flow: From Database to Sanitized API Response
SANITIZED API RESPONSE
Client-safe Response
(e.g., "500 Internal Server Error: An unexpected error occurred.")
Wrapped Error
SENSITIVE DATA EXPOSURE
(e.g., Stack traces, internal IPs)
INTERNAL LOGIC & WRAPPING
Wrap with Context
(e.g., "Failed to fetch user data: [DB Error]")
Raw Error
DATABASE LAYER
Low-level Error
(e.g., "Connection timed out", "Constraint violation")

Summary:

- Use %w for history, Is/As for inspection.
- Use errors.Join and errgroup for complex flows.
- Add context at every layer, log only once at the top.
-

Have an errorless day :)