JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#49

SLOG: STRUCTURED LOGGING

MOVING FROM TEXT FOR HUMANS TO DATA FOR MACHINES

Why "Structured" Matters

Beyond the String

Traditional logging is just a pile of strings: *log.Printf("User %d logged in", id)*. While easy for a human to read, it’s a nightmare for an automated system to index or search.

The Shift:
Structured logging treats every log entry as a Key-Value pair.
Instead of searching for the string "logged in," you can query your logs for *event="login"* and *user_id=123*.
This turns your logs into a searchable database of your system's behavior.

Getting Started with slog

The New Standard

The *log/slog* package brings high-performance structured logging to the standard library. It's designed to be fast, flexible, and compatible with the logging levels we all know (Info, Debug, Warn, Error).

By default, *slog* can output in a developer-friendly text format or a production-ready JSON format. You choose the "Handler," and the logger does the rest.

```go
// Initialize a JSON logger for production
logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))

// Logging with attributes
logger.Info("incoming_request",
    "method", "GET",
    "path",   "/users",
    "status", 200,
)
```

Type-Safe Attributes

Performance vs. Convenience

slog offers two ways to log: loosely-typed "Key, Value" pairs or strongly-typed attributes. While the loose version is faster to write, the typed version avoids expensive allocation and is significantly faster in high-throughput paths.
Use the loose syntax for everyday debugging. Switch to *slog.Int*, *slog.String*, or *slog.Attr* when you're logging inside a tight loop or a high-traffic middleware where every nanosecond counts.

```
// Loosely typed (easy but slower)
logger.Info("upload", "size", 1024)

// Strongly typed (best performance)
logger.LogAttrs(ctx, slog.LevelInfo, "upload",
    slog.Int("size", 1024),
    slog.String("user", "niv"),
)
```

Contextual Logging

Carrying the Trace

A log entry is much more valuable if it includes a Trace ID or Request ID. This allows you to correlate logs from five different microservices into a single cohesive story.

The Implementation:
By wrapping the *slog.Handler*, you can create a logger that automatically pulls these IDs from the *context.Context* and injects them into every log line without you having to pass them manually every time.

Grouping and Scoping

Organizing Complex Data

When logging complex objects (like a User or a DB Config), you don't want a flat list of 50 keys. *slog* allows you to Group related attributes, creating a nested structure in your JSON output.

The Logic:

Grouping keeps your log schema organized. It prevents "Key Collisions" – where two different parts of your code try to use the key *id* by nesting them under namespaces like *user.id* and *request.id*.

```
logger.Info("user_update",
    slog.Group("user",
        slog.Int("id", 42),
        slog.String("email", "nivikr@gmail.com"),
    ),
)
// Output: {"level":"INFO","msg":"user_update","user":{"id":42,"email":"nivikr@gmail.com"}}
```

Handlers and Global State

Customizing the Output

One of the most powerful features of *slog* is the ability to swap the *Handler*. You can write a handler that sends logs to a file, to a Slack channel, or to an OpenTelemetry collector.

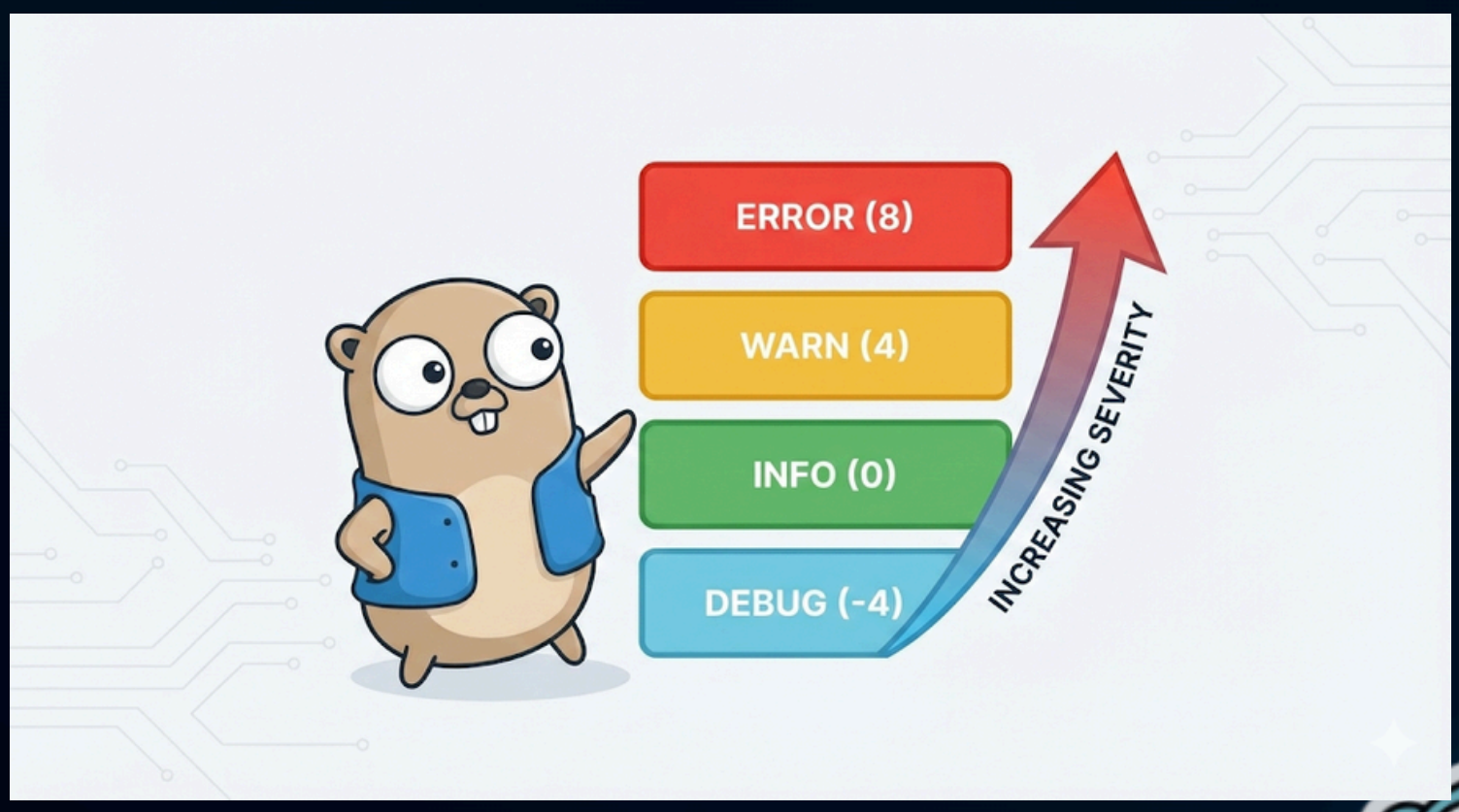
Set your global logger once in *main.go*. Because *slog* is part of the standard library, third-party packages can use the same global logger, ensuring your entire application - including its dependencies - logs in the same consistent format.

Levels and Filtering

Plugging in the Infrastructure

Logging too much is as bad as logging too little. In production, you might only want *Info* and above, while in development, you want *Debug*.

The Dynamic Switch:
With *slog*, you can change the log level at runtime without restarting your service. This is a lifesaver when you need to "turn up the lights" to investigate a live issue without drowning your ELK stack in debug data indefinitely.



GOLANG /// BA

Summary:

- **Structured**: Log keys and values, not just formatted strings.
- **slog**: Use the standard library for future-proof, high-performance logging.
- **Attributes**: Use typed attributes in performance-critical paths.
- **Context**: Inject Trace IDs to correlate logs across services.

Question: Do you prefer JSON logs for local development, or do you find the "pretty-printed" text format easier to scan during a coding session? 👇