Logging in Go with Slog: The Ultimate Guide | Better Stack Community

Structured logging involves recording notable application events in a well-defined format (usually JSON), adding a level of organization and consistency to the generated logs, and thus making them easier to process.

Such log records are composed of key/value pairs that capture relevant contextual information about the event being logged, such as its severity, timestamp, source code location, correlation ID, or any other relevant metadata.

This article will dive deep into the world of structured logging in Go with a specific focus on the recently introduced Slog package which aims to bring high-performance, structured, and leveled logging to the Go standard library.

We'll begin by examining the its predecessor, the `log` package and its limitations, before introducing Slog and discussing its most important concepts. We'll also explore Slog's impact on the wider Go logging ecosystem and provide some best practices to improve the usefulness of your application logs.

The log package: a brief recap

Before we discuss the new structured logging package, let's briefly examine the `log` package which provides a simple way to write log messages to the console, a file, or any type that implements the `io.Writer` interface. Here's the most basic way to write log messages in Go:

```
package main

import "log"

func main() {
    log.Println("Hello from Go application!")
```

```
}
```

```
2023/03/08 11:43:09 Hello from Go application!
```

The output contains the log message and a timestamp in the local time zone that indicates when the entry was generated. The `Println()` method is one of the methods accessible on the preconfigured global `Logger`, and it prints to the standard error. The following other methods are available:

```
log.Print()
log.Printf()
log.Fatal()
log.Fatalf()
log.Fatalln()
log.Panic()
log.Panicf()
log.Panicln()
```

The difference between the `Fatal` and `Panic` family of methods above is that the former calls `os.Exit(1)` after logging a message, while the latter calls `panic()`.

You can customize the default `Logger` by retrieving it through the `log.Default()` method. Afterward, call the relevant method on the returned `Logger`. The example below configures the default logger to write to the standard output instead of the standard error:

```
func main() {
    defaultLogger := log.Default()


    defaultLogger.SetOutput(os.Stdout)


    log.Println("Hello from Go application!")
}
```

You can also create a completely custom logger through the `log.New()` method which has the following signature:

```
func New(out io.Writer, prefix string, flag int) *Logger
```

This way you can customize its output more by using a set of constants

to add details to each log message.

```
logger := log.New(
  os.Stderr,
  "MyApplication: ",
  log.Ldate|log.Ltime|log.Lmicroseconds|log.LUTC|log.Lshortfile,
)
```

```
MyApplication: 2023/03/08 10:47:12.348478 main.go:14: Hello from Go
application!
```

The `MyApplication:` prefix appears at the beginning of each log entry, and the UTC timestamp now includes microseconds. The file name and line number are also included in the output to help you locate the source of each entry in the codebase.

Limitations of the log package

Although the `log` package in Go provides a convenient way to initiate logging, it is not ideal for production use due to several limitations, such as the following:

Lack of log levels: log levels are one of the staple features in most logging packages, but they are missing from the `log` package in Go. All log messages are treated the same way, making it difficult to filter or separate log messages based on their importance or severity.

No support for structured logging: the `log` package in Go only outputs plain text messages. It does not support structured logging, where the events being recorded are represented in a structured format (usually JSON), which can be subsequently parsed and analyzed programmatically for monitoring, alerting, auditing, creating dashboards, and other forms of analysis.

No context-aware logging: support for context-aware logging is missing from the `log` package, making it difficult to attach contextual information (such as request IDs, User IDs, and other variables) to log messages automatically.

Limited configuration options: the `log` package only supports basic configuration options, such as setting the log output destination and prefix are supported. Advanced logging libraries offer way more configuration opportunities, such as custom log formats, filtering, automatically adding contextual data, enabling asynchronous logging, error handling behavior, and more!

In light of the aforementioned limitations, a new logging package called `slog` has been introduced to fill the existing gap in Go's standard library. In short, this package aims to enhance logging capabilities in the language by introducing structured logging with levels, and create a standard interface for logging that other packages can extend freely.

Structured logging in Go with Slog

The `slog` package has its origins in [this GitHub discussion opened by Jonathan Amsterdam](), which later led to the [proposal]() describing the exact design of the package. Once finalized, it was released in [Go v1.21]() and now resides at `log/slog`.

Let's begin our exploration of the `slog` package by walking through its design and architecture. It three main types that you need to be familiar with:

`Logger`: the "frontend" for logging with Slog. It provides level methods such as (`Info()` and `Error()`) for recording events of interest.

`Record`: represents each self-contained log record object created by a `Logger`.

`Handler`: the "backend" of the Slog package. It is an interface that, once implemented, determines the formatting and destination of each `Record`. Two handlers are included with the `slog` package by default: `TextHandler` and `JSONHandler`.

In the following sections, we will present a comprehensive examination of each of these types, accompanied

with detailed examples.

Getting started with Slog

Like most Go logging libraries, the `slog` package exposes a default `Logger` accessible through top-level functions on the package. This logger defaults to the `INFO` level, and it logs a plaintext output to the standard output (similar to the `log` package):

```
package main

import (
    "log/slog"
)

func main() {
    slog.Debug("Debug message")
    slog.Info("Info message")
    slog.Warn("Warning message")
    slog.Error("Error message")
}
```

```
2023/03/15 12:55:56 INFO Info message
2023/03/15 12:55:56 WARN Warning message
2023/03/15 12:55:56 ERROR Error message
```

You can also create a custom `Logger` instance through the `slog.New()` method. It accepts a non-nil `Handler` interface, which determines how the logs are formatted and where they are written to. Here's an example that uses the built-in `JSONHandler` type to log to the standard output in JSON format:

```
func main() {
    logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))


    logger.Debug("Debug message")
    logger.Info("Info message")
    logger.Warn("Warning message")
    logger.Error("Error message")
}
```

```
{"time":"2023-03-15T12:59:22.227408691+01:00","level":"INFO","msg":"Info
message"}
```

```
{"time":"2023-03-15T12:59:22.227468972+01:00","level":"WARN","msg":"Warning message"}
{"time":"2023-03-15T12:59:22.227472149+01:00","level":"ERROR","msg":"Error message"}
```

Notice that the custom logger also defaults to `INFO`, which causes the suppression of the `DEBUG` entry. If you use the `TextHandler` type instead, each log record will be formatted according to the [logfmt standard]

:

```
logger := slog.New(slog.NewTextHandler(os.Stdout, nil))
```

```
time=2023-03-15T13:00:11.333+01:00 level=INFO msg="Info message"
time=2023-03-15T13:00:11.333+01:00 level=WARN msg="Warning message"
time=2023-03-15T13:00:11.333+01:00 level=ERROR msg="Error message"
```

Customizing the default logger

To configure the default `Logger`, the most straightforward approach is to utilize the `slog.SetDefault()` method, which allows you to substitute the default logger with a custom one.

```
func main() {
    logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))

    slog.SetDefault(logger)



    slog.Info("Info message")
}
```

```
{"time":"2023-03-15T13:07:39.105777557+01:00","level":"INFO","msg":"Info message"}
```

You'll observe that the package's top-level logging methods now produce JSON logs as seen above. Using the `SetDefault()` method also alters the default `log.Logger` employed by the `log` package. This behavior allows existing applications that utilize the older `log` package to seamlessly transition to [structured logging].

```
func main() {
    logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))

    slog.SetDefault(logger)

    log.Println("Hello from old logger")
}
```

```
{"time":"2023-03-16T15:20:33.783681176+01:00","level":"INFO","msg":"Hello from old logger"}
```

The `slog.NewLogLogger()` method is also available for converting an `slog.Logger` to a `log.Logger` when you need to utilize APIs that require the latter (such as `http.Server.ErrorLog`):

```
func main() {
  handler := slog.NewJSONHandler(os.Stdout, nil)

  logger := slog.NewLogLogger(handler, slog.LevelError)

  server := http.Server{
    ErrorLog: logger,
  }
}
```

Logging contextual attributes

Logging in a structured format offers a significant advantage over traditional plaintext formats by allowing the inclusion of arbitrary attributes as key/value pairs in log records. These attributes provide additional context about the logged event, which can be valuable for tasks such as troubleshooting, generating metrics, auditing, and various other purposes. Here is an example illustrating how it works in Slog:

```
logger.Info(
  "incoming request",
  "method", "GET",
  "time_taken_ms", 158,
  "path", "/hello/world?q=search",
  "status", 200,
  "user_agent", "Googlebot/2.1 (+http://www.google.com/bot.html)",
)
```

```
{
  "time":"2023-02-24T11:52:49.554074496+01:00",
  "level":"INFO",
  "msg":"incoming request",
  "method":"GET",
  "time_taken_ms":158,
  "path":"/hello/world?q=search",
  "status":200,
  "user_agent":"Googlebot/2.1 (+http://www.google.com/bot.html)"
}
```

All the level methods (`Info()`, `Debug()`, etc.) accept a log message as their first argument, and an unlimited number of loosely-typed key/value pairs thereafter. This API is similar to the SugaredLogger API in Zap (specifically its level methods ending in `w`) as it prioritizes brevity at the cost of additional memory allocations. However, note that it can also lead to strange problems if you're not careful. Most notably, unbalanced key/value pairs will yield a problematic output:

```
logger.Info(
  "incoming request",
  "method", "GET",
  "time_taken_ms", // the value for this key is missing
)
```

Since the `time_taken_ms` key does not have a corresponding value, it will be treated as a value with key `!BADKEY`:

```
{
  "time": "2023-03-15T13:15:29.956566795+01:00",
  "level": "INFO",
  "msg": "incoming request",
  "method": "GET",
"!BADKEY": "time_taken_ms"


}
```

This isn't great because a property misalignment could lead to bad entries being created, and you may not know about it until you need to use the logs. While the proposal suggests a vet check

to catch missing key/value problems in methods where they can occur, extra care also needs to be taken during the review process to ensure that each key/value pair in the entry are balanced, and the types are correct.

To prevent such mistakes, it's best only to use [strongly-typed contextual attributes]

as shown below:

```
logger.Info(
  "incoming request",
  slog.String("method", "GET"),
  slog.Int("time_taken_ms", 158),
  slog.String("path", "/hello/world?q=search"),
  slog.Int("status", 200),
  slog.String(
    "user_agent",
    "Googlebot/2.1 (+http://www.google.com/bot.html)",
  ),
)
```

While this is a much better approach to contextual logging, it's not fool-proof as nothing is stopping you from mixing strongly-typed and loosely-typed key/value pairs like this:

```
logger.Info(
  "incoming request",
  "method", "GET",
  slog.Int("time_taken_ms", 158),
  slog.String("path", "/hello/world?q=search"),
  "status", 200,
  slog.String(
```

```
    "user_agent",
    "Googlebot/2.1 (+http://www.google.com/bot.html)",
  ),
)
```

To guarantee type safety when adding contextual attributes to your records, you must use the `LogAttrs()` method like this:

```
logger.LogAttrs(
  context.Background(),
  slog.LevelInfo,
  "incoming request",
  slog.String("method", "GET"),
  slog.Int("time_taken_ms", 158),
  slog.String("path", "/hello/world?q=search"),
  slog.Int("status", 200),
  slog.String(
    "user_agent",
    "Googlebot/2.1 (+http://www.google.com/bot.html)",
  ),
)
```

This method only accepts the `slog.Attr` type for custom attributes, so it's not possible to have an unbalanced key/value pair. However, its API is more convoluted as you always need to pass a context (or `nil`) and the log level to the method in addition to the log message and custom attributes.

Grouping contextual attributes

Slog also provides the ability to group multiple attributes under a single name name. The way it is displayed depends on the `Handler` in use. For example, with `JSONHandler`, the group is treated as a separate JSON object:

```
logger.LogAttrs(
  context.Background(),
  slog.LevelInfo,
  "image uploaded",
  slog.Int("id", 23123),
  slog.Group("properties",
    slog.Int("width", 4000),
```

```
    slog.Int("height", 3000),

    slog.String("format", "jpeg"),

),

)
```

```
{
  "time":"2023-02-24T12:03:12.175582603+01:00",
  "level":"INFO",
  "msg":"image uploaded",
  "id":23123,
  "properties":{
    "width":4000,
    "height":3000,
    "format":"jpeg"
  }
}
```

When using the `TextHandler`, each key in the group will be prefixed by the group name like this:

```
time=2023-02-24T12:06:20.249+01:00 level=INFO msg="image uploaded"
id=23123
  properties.width=4000 properties.height=3000 properties.format=jpeg
```

Creating and using child loggers

Including the same attributes in all records within a specific program scope can be beneficial to ensure their presence without repetitive logging statements. This is where child loggers prove helpful, as they create a new logging context inheriting from their parent logger while allowing the addition of additional fields.

In Slog, creating child loggers is accomplished using the `Logger.With()` method. It accepts one or more key/value pairs, and returns a new `Logger` that includes the specified attributes. Consider the following code snippet that adds the program's process ID and the Go version used for compilation to each log record, storing them in a `program_info` property:

```
func main() {
    handler := slog.NewJSONHandler(os.Stdout, nil)
```

```
    buildInfo, _ := debug.ReadBuildInfo()

    logger := slog.New(handler)

    child := logger.With(

        slog.Group("program_info",

            slog.Int("pid", os.Getpid()),

            slog.String("go_version", buildInfo.GoVersion),

),

)



    . . .
}
```

With this configuration in place, all records created by the `child` logger will contain the specified attributes under the `program_info` property as long as it is not overridden at log point:

```
func main() {
    . . .

    child.Info("image upload successful", slog.String("image_id",
"39ud88"))
    child.Warn(
        "storage is 90% full",
        slog.String("available_space", "900.1 mb"),
    )
}
```

```
{
  "time": "2023-02-26T19:26:46.046793623+01:00",
```

```
  "level": "INFO",
  "msg": "image upload successful",
  "program_info": {
    "pid": 229108,
    "go_version": "go1.20"
  },
  "image_id": "39ud88"
}
{
  "time": "2023-02-26T19:26:46.046847902+01:00",
  "level": "WARN",
  "msg": "storage is 90% full",
  "program_info": {
    "pid": 229108,
    "go_version": "go1.20"
  },
  "available_space": "900.1 MB"
}
```

You can also use the `WithGroup()` method to create a child logger that starts a group such that all attributes added to the logger (including those added at log point) are nested under the group name:

```
handler := slog.NewJSONHandler(os.Stdout, nil)
buildInfo, _ := debug.ReadBuildInfo()

logger := slog.New(handler).WithGroup("program_info")



child := logger.With(
  slog.Int("pid", os.Getpid()),
  slog.String("go_version", buildInfo.GoVersion),
)

child.Info("image upload successful", slog.String("image_id", "39ud88"))
child.Warn(
  "storage is 90% full",
  slog.String("available_space", "900.1 MB"),
```

```
)
```

```
{
  "time": "2023-05-24T19:00:18.384085509+01:00",
  "level": "INFO",
  "msg": "image upload successful",
  "program_info": {
    "pid": 1971993,
    "go_version": "go1.20.2",
    "image_id": "39ud88"
  }
}
{
  "time": "2023-05-24T19:00:18.384136084+01:00",
  "level": "WARN",
  "msg": "storage is 90% full",
  "program_info": {
    "pid": 1971993,
    "go_version": "go1.20.2",
    "available_space": "900.1 mb"
  }
}
```

Customizing Slog log levels

The `slog` package provides four log levels by default, and each one is associated with an integer value: `DEBUG` (-4), `INFO` (0), `WARN` (4), and `ERROR` (8). The gap of 4 between each level is a deliberate design decision made to accommodate logging schemes with custom levels between the default ones. For example, you can create a custom `NOTICE` level between `INFO` and `WARN` with a value of 1, 2, or 3.

You've probably noticed that all loggers are configured to log at the `INFO` level by default, which causes events logged at a lower severity (such as `DEBUG`) to be suppressed. You can customize this behavior through the

HandlerOptions type as shown below:

```
func main() {
    opts := &slog.HandlerOptions{


        Level: slog.LevelDebug,


}



    handler := slog.NewJSONHandler(os.Stdout, opts)



    logger := slog.New(handler)
    logger.Debug("Debug message")
    logger.Info("Info message")
    logger.Warn("Warning message")
    logger.Error("Error message")
}
```

```
{"time":"2023-05-24T19:03:10.70311982+01:00","level":"DEBUG","msg":"Debug
message"}
{"time":"2023-05-24T19:03:10.703187713+01:00","level":"INFO","msg":"Info
message"}
{"time":"2023-05-24T19:03:10.703190419+01:00","level":"WARN","msg":"Warning
message"}
{"time":"2023-05-24T19:03:10.703192892+01:00","level":"ERROR","msg":"Error
message"}
```

Note that this approach fixes the minimum level of the `handler` throughout its lifetime. If you need the minimum level to be dynamically varied, you must use the `LevelVar` type as illustrated below:

```
logLevel := &slog.LevelVar{} // INFO

opts := slog.HandlerOptions{
    Level: logLevel,
}

// you can change the level anytime like this
```

```
logLevel.Set(slog.LevelDebug)
```

Creating custom log levels

If you need custom levels beyond what Slog provides by default, you can create them by implementing the Leveler interface which is defined by a single method:

```
type Leveler interface {
    Level() Level
}
```

It's also easy to implement the `Leveler` interface through the `Level` type as shown below (since `Level` itself implements `Leveler`):

```
const (
    LevelTrace  = slog.Level(-8)
    LevelNotice = slog.Level(2)
    LevelFatal  = slog.Level(12)
)
```

Once you've defined custom levels as above, you can use them as follows:

```
opts := &slog.HandlerOptions{
    Level: LevelTrace,
}

logger := slog.New(slog.NewJSONHandler(os.Stdout, opts))

ctx := context.Background()
logger.Log(ctx, LevelTrace, "Trace message")
logger.Log(ctx, LevelNotice, "Notice message")
logger.Log(ctx, LevelFatal, "Fatal level")
```

```
{"time":"2023-02-24T09:26:41.666493901+01:00","level":"DEBUG-
4","msg":"Trace level"}
{"time":"2023-02-24T09:26:41.66659754+01:00","level":"INFO+2","msg":"Notice
```

```
level"}
{"time":"2023-02-24T09:26:41.666602404+01:00","level":"ERROR+4","msg":"Fata
level"}
```

Notice how the custom levels are labelled in terms of the defaults. This probably isn't what you want, so you should customize the level names through the `HandlerOptions` type:

```
. . .

var LevelNames = map[slog.Leveler]string{
    LevelTrace:      "TRACE",
    LevelNotice:     "NOTICE",
    LevelFatal:      "FATAL",
}

func main() {
    opts := slog.HandlerOptions{
        Level: LevelTrace,
        ReplaceAttr: func(groups []string, a slog.Attr) slog.Attr {

if a.Key == slog.LevelKey {

                level := a.Value.Any().(slog.Level)

                levelLabel, exists := LevelNames[level]

if !exists {

                    levelLabel = level.String()

}



                a.Value = slog.StringValue(levelLabel)
```

```
}



return a


},


    }


    . . .
}
```

The `ReplaceAttr()` function is used to customize how each key/value pair in a `Record` is handled by a `Handler`. It can be used to customize the name of the key, or transform the value in some way. In the above example, it maps the custom log levels to their respective labels producing `TRACE`, `NOTICE`, and `FATAL` respectively.

```
{"time":"2023-02-24T09:27:51.747625912+01:00","level":"TRACE","msg":"Trace
level"}
{"time":"2023-02-24T09:27:51.747732118+01:00","level":"NOTICE","msg":"Notic
level"}
{"time":"2023-02-24T09:27:51.747737319+01:00","level":"FATAL","msg":"Fatal
level"}
```

Customizing Slog Handlers

As mentioned earlier, both `TextHandler` and `JSONHandler` can be customized using the `HandlerOptions` type. You've already learned how to adjust the minimum level and modify attributes before they are logged. Another customization that can be accomplished through `HandlerOptions` is adding the source of the log message, if required:

```
opts := slog.HandlerOptions{
  AddSource: true,


  Level: slog.LevelDebug,
}
```

```
{"time":"2023-05-24T19:39:27.005871442+01:00","level":"DEBUG","source":
```

```
{"function":"main.main","file":"/home/ayo/dev/demo/slog
/main.go","line":30},"msg":"Debug message"}
```

It's also easy to switch handlers based on the application environment. For example, you might prefer to use the `TextHandler` for your development logs since its a little easier to read, then switch to `JSONHandler` in production for greater compatibility with various logging tools. You can easily enable such behavior through an environmental variable:

```
var appEnv = os.Getenv("APP_ENV")

func main() {
    opts := &slog.HandlerOptions{
        Level: slog.LevelDebug,
    }

    var handler slog.Handler = slog.NewTextHandler(os.Stdout, opts)
    if appEnv == "production" {
        handler = slog.NewJSONHandler(os.Stdout, opts)
    }

    logger := slog.New(handler)

    logger.Info("Info message")
}
```

```
time=2023-02-24T10:36:39.697+01:00 level=INFO msg="Info message"
```

```
APP_ENV=production go run main.go
```

```
{"time":"2023-02-24T10:35:16.964821548+01:00","level":"INFO","msg":"Info
message"}
```

Creating custom Handlers

Since `Handler` is an interface, its easy to create custom handlers for formatting the logs differently, or writing them to some other destination. Its signature is as follows:

```
type Handler interface {
    Enabled(context.Context, Level) bool
    Handle(context.Context, r Record) error
    WithAttrs(attrs []Attr) Handler
```

```
    WithGroup(name string) Handler
}
```

Here's what each of the methods do:

`Enabled()` determines if a log record should be handled or discarded based on its level. The `context` can also used to make a decision.

`Handle()` processes each log record sent to the handler. It is called only if `Enabled()` returns `true`.

`WithAttrs()` creates a new handler from an existing one and adds the specified attributes it.

`WithGroup()` creates a new handler from an existing one and adds the specified group name to it such that subsequent attributes are qualified by the name.

Here's an example that uses the `log`, `json`, and color packages to implement a prettified development output for log records:

```
// NOTE: Not well tested, just an illustration of what's possible
package main

import (
    "context"
    "encoding/json"
    "io"
    "log"
    "log/slog"

    "github.com/fatih/color"
)

type PrettyHandlerOptions struct {
    SlogOpts slog.HandlerOptions
}

type PrettyHandler struct {
```

```
    slog.Handler
    l *log.Logger
}

func (h *PrettyHandler) Handle(ctx context.Context, r slog.Record) error {
    level := r.Level.String() + ":"

    switch r.Level {
    case slog.LevelDebug:
        level = color.MagentaString(level)
    case slog.LevelInfo:
        level = color.BlueString(level)
    case slog.LevelWarn:
        level = color.YellowString(level)
    case slog.LevelError:
        level = color.RedString(level)
    }

    fields := make(map[string]interface{}, r.NumAttrs())
    r.Attrs(func(a slog.Attr) bool {
        fields[a.Key] = a.Value.Any()

        return true
    })

    b, err := json.MarshalIndent(fields, "", "  ")
    if err != nil {
        return err
    }

    timeStr := r.Time.Format("[15:05:05.000]")
    msg := color.CyanString(r.Message)

    h.l.Println(timeStr, level, msg, color.WhiteString(string(b)))

    return nil
}
```

```
func NewPrettyHandler(
    out io.Writer,
    opts PrettyHandlerOptions,
) *PrettyHandler {
    h := &PrettyHandler{
        Handler: slog.NewJSONHandler(out, &opts.SlogOpts),
        l:       log.New(out, "", 0),
    }

    return h
}
```

When you use the `PrettyHandler` in your code like this:

```
func main() {
    opts := PrettyHandlerOptions{
        SlogOpts: slog.HandlerOptions{
            Level: slog.LevelDebug,
        },
    }
    handler := NewPrettyHandler(os.Stdout, opts)
    logger := slog.New(handler)
    logger.Debug(
        "executing database query",
        slog.String("query", "SELECT * FROM users"),
    )
    logger.Info("image upload successful", slog.String("image_id",
"39ud88"))
    logger.Warn(
        "storage is 90% full",
        slog.String("available_space", "900.1 MB"),
    )
    logger.Error(
        "An error occurred while processing the request",
        slog.String("url", "https://example.com"),
    )
}
```

You will observe the following colorized output when you execute the program:

You can find several custom handlers created by the community on GitHub

. Some notable examples include:

tint - writes tinted (colorized) logs.

slog-sampling - improves logging throughput by dropping repetitive log records.

slog-multi - implements workflows such as middleware, fanout, routing, failover, load balancing.

slog-formatter - provides more flexible attribute formatting.

Hiding sensitive fields with the LogValuer interface

The `LogValuer` interface allows you to standardize your logging output by specifying how your custom types should be logged. Here's its signature:

```
type LogValuer interface {
    LogValue() Value
}
```

A prime use case for implementing this interface is for hiding sensitive fields in your custom types. For example, here's a `User` type that does not implement the `LogValuer` interface. Notice how sensitive details are exposed when type is logged:

```
// User does not implement `LogValuer` here
type User struct {
    ID        string `json:"id"`
    FirstName string `json:"first_name"`
    LastName  string `json:"last_name"`
    Email     string `json:"email"`
    Password  string `json:"password"`
}

func main() {
    handler := slog.NewJSONHandler(os.Stdout, nil)
    logger := slog.New(handler)

    u := &User{
        ID:        "user-12234",
        FirstName: "Jan",
        LastName:  "Doe",
        Email:     "jan@example.com",
        Password:  "pass-12334",
    }

    logger.Info("info", "user", u)
}
```

```
{
  "time": "2023-02-26T22:11:30.080656774+01:00",
  "level": "INFO",
```

```
  "msg": "info",
  "user": {
    "id": "user-12234",
    "first_name": "Jan",
    "last_name": "Doe",
    "email": "jan@example.com",
    "password": "pass-12334"
  }
}
```

Without implementing the `LogValuer` interface, the entire `User` type will be logged as shown above. This is problematic since the type contains secret fields that should not be present in the logs (such as emails and passwords), and it can also make your logs unnecessarily verbose.

You can solve this issue by specifying how you'd like the type to be handled in the logs. For example, you may specify that only the `ID` field should be logged as follows:

```
// implement the `LogValuer` interface
func (u *User) LogValue() slog.Value {
    return slog.StringValue(u.ID)
}
```

You will now observe the following output:

```
{
  "time": "2023-02-26T22:43:28.184363059+01:00",
  "level": "INFO",
  "msg": "info",
  "user": "user-12234"
}
```

You can also group multiple attributes like this:

```
func (u *User) LogValue() slog.Value {
    return slog.GroupValue(
        slog.String("id", u.ID),
        slog.String("name", u.FirstName+" "+u.LastName),
    )
}
```

```
{
  "time": "2023-03-15T14:44:24.223381036+01:00",
```

```
  "level": "INFO",
  "msg": "info",
  "user": {
    "id": "user-12234",
    "name": "Jan Doe"
  }
}
```

Using third-party libraries with Slog

One of Slog's major design goals is to provide a unified logging frontend (`slog.Logger`) for Go applications while the backend (`slog.Handler`) remains customizable from program to program.

This way, the logging API is consistent across all dependencies even if the backends differ. It also avoids coupling the logging implementation to a specific package by making it trivial to switch a different backend if requirements change in your project.

Here's an example that uses the Slog frontend with a Zap backend, providing the best of both worlds:

```
go get go.uber.org/zap/exp/zapslog
```

```
package main

import (
    "log/slog"

    "go.uber.org/zap"
    "go.uber.org/zap/exp/zapslog"
)

func main() {
    zapL := zap.Must(zap.NewProduction())

    defer zapL.Sync()

    logger := slog.New(zapslog.NewHandler(zapL.Core(), nil))

    logger.Info(
        "incoming request",
        slog.String("method", "GET"),
```

```
        slog.String("path", "/api/user"),
        slog.Int("status", 200),
    )
}
```

This snippet creates a new Zap production logger that is subsequently used as a handler for the Slog package through `zapslog.NewHandler()`. With this in place, you only need to write your logs using methods provided on `slog.Logger` but the resulting records will be processed according to the provided `zapL` configuration.

```
{"level":"info","ts":1697453912.4535635,"msg":"incoming
request","method":"GET","path":"/api/user","status":200}
```

Switching to a different logging is really straightforward since logging is done in terms of `slog.Logger`. For example, you can switch from Zap to Zerolog like this:

```
go get github.com/rs/zerolog
```

```
go get github.com/samber/slog-zerolog
```

```
package main

import (
    "log/slog"
    "os"

"github.com/rs/zerolog"


    slogzerolog "github.com/samber/slog-zerolog"


)

func main() {
    zerologL := zerolog.New(os.Stdout).Level(zerolog.InfoLevel)



    logger := slog.New(

        slogzerolog.Option{Logger: &zerologL}.NewZerologHandler(),


)



    logger.Info(
        "incoming request",
        slog.String("method", "GET"),
        slog.String("path", "/api/user"),
        slog.Int("status", 200),
    )
}
```

```
{"level":"info","time":"2023-10-16T13:22:33+02:00","method":"GET","path":"/
/user","status":200,"message":"incoming request"}
```

In the above snippet, the Zap handler has been replaced with a [custom Zerolog one](#)

. Since logging isn't done using either library's custom APIs, the migration process only takes a couple of minutes compared to a situation where you have to switch out one logging API for another across your entire application.

Best practices for writing and storing Go logs

Once you've configured Slog or your preferred third-party Go logging framework, it necessary to adopt the following best practices to ensure that you get the most out of your application logs:

1. Standardize your logging interfaces

Implementing the `LogValuer` interface allows you to standardize how the various types in your application are logged to ensure that their representation in the logs is consistent throughout your application. It's also an effective strategy for ensuring that sensitive fields are omitted from your application logs as we explored earlier in this article.

2. Log unexpected errors with a stack trace

To improve your ability to debug unexpected issues in production, adding a stack trace to your error logs is crucial. This way, it'll be easy to pinpoint where the error originated within the codebase and the program flow that led to the problem. Slog does not currently provide a way to easily add stack traces to errors, but you can implement a custom function using the `ReplaceAttr` function and a library like pkgerrors

or go-xerrors .

See an example on the Go playground here .

3. Centralize your logs, but persist them in local files first

It's generally better to decouple the task of writing logs from shipping them to a centralized log management system. Writing logs to local files first ensures there's a backup in case the log management system or network faces issues, preventing potential loss of crucial data. Additionally, storing logs locally before sending them off helps buffer the logs, allowing for batch transmissions which can optimize network bandwidth usage and minimize impact on application performance.

Local log storage also affords greater flexibility so that if there's a need to transition to a different log management system, modifications are required only in the shipping method rather than the entire application logging mechanism. See our articles on using specialized log shippers like Vector or Fluentd for more details.

Logging to files does not necessarily require you to configure your chosen framework to write directly to a file as Systemd can easily redirect the application's standard output and error streams to a file. Docker also defaults to collecting all data sent to both streams and routing them to local files on the host machine.

4. Sample your logs

Log sampling is the practice of recording only a representative subset of log entries instead every single log event. This technique is especially useful in high-traffic environments where systems generate vast amounts of log data, and processing every entry could be quite costly since centralized logging solutions often charge based on data ingestion rates or storage.

```
package main

import (
    "fmt"
    "log/slog"
    "os"

    slogmulti "github.com/samber/slog-multi"
    slogsampling "github.com/samber/slog-sampling"
)
```

```
func main() {
    // Will print 20% of entries.
    option := slogsampling.UniformSamplingOption{
        Rate: 0.2,
    }

    logger := slog.New(
        slogmulti.
            Pipe(option.NewMiddleware()).
            Handler(slog.NewJSONHandler(os.Stdout, nil)),
    )

    for i := 1; i <= 10; i++ {
        logger.Info(fmt.Sprintf("a message from the gods: %d", i))
    }
}
```

```
{"time":"2023-10-18T19:14:09.820090798+02:00","level":"INFO","msg":"a
message from the gods: 4"}
{"time":"2023-10-18T19:14:09.820117844+02:00","level":"INFO","msg":"a
message from the gods: 5"}
```

Third-party frameworks such as Zerolog and Zap provide built-in log sampling features. With Slog, you'd have to integrate a third-party handler such as slog-sampling or develop a custom solution. You can also choose to sample logs through a dedicated log shipper such as Vector.

5. Use a log management system

Centralizing your logs in a log management system makes it easy to search, analyze, and monitor your application's behavior across multiple servers and environments. With all the logs in one place, your ability to identify and diagnose issues is sped up significantly as you'd no longer need to jump between different servers to gather information about your service.

While here are several log management solutions out there, Better Stack provides an easy way to set up centralized log management in a few minutes, with live tailing, alerting, dashboards, and uptime monitoring, and incident management features built-into a modern and intuitive interface. Give it try with a completely free plan by clicking the link below:

Final thoughts

I hope this post has provided you with an understanding of the new structured logging package in Go, and how you can start using it in your projects. If you want to explore this topic further, I recommend checking out the

full proposal and package documentation

.

Thanks for reading, and happy logging!

Further reading:

The Top 8 Go logging libraries.

Best Practices for Formatting Your Logs in Production.

Redis caching in Go: A Beginner's Guide.