Logging in Go: A Comparison of the Top 8 Libraries | Better Stack Community

There's probably a 99% chance that if you're logging in Go, you're using a third-party logging framework since the built-in `log` package lacks even the most basic features required for production logging. This recently changed with the release of Go 1.21 where the new log/slog package for structured, leveled, and context-aware logging was one of the major highlights.

Since the Go ecosystem has already spawned many several comprehensive logging solutions, you might be wondering whether the Slog package is the harbinger of obsolescence for its predecessors, or merely another versatile tool in your Go observability arsenal.

This article aims to uncover some insights that should help you clarify that question by taking a look at eight logging solutions, discussing their pros and cons after considering the following factors: performance, flexibility, features, ease of use, and community support.

1. Zerolog

Zerolog currently holds the crown as the fastest structured logging framework for Go programs . It offers excellent performance for high-load systems with minimal to zero heap allocations.

Here's the simplest example of using Zerolog to log a message:

```
package main
```

```
import (
    "github.com/rs/zerolog/log"
)

func main() {
    log.Info().Msg("Hello from Zerolog logger")
}
```

The resulting log record will be in JSON format:

```
{"level":"info","time":"2023-07-12T11:57:28+02:00","message":"Hello from
Zerolog global logger"}
```

Zerolog can also be configured to produce binary logs in the CBOR format

(for reducing log storage requirements) but it does not support any other log format. In development environments where log readability is essential, you can use the `ConsoleWriter` type to log in a human-friendly, colorized output:

```
package main

import (
    "os"
    "time"

    "github.com/rs/zerolog"
)

func main() {
    logger := zerolog.New(
        zerolog.ConsoleWriter{Out: os.Stderr, TimeFormat: time.RFC3339},
    ).Level(zerolog.TraceLevel).With().Timestamp().Caller().Logger()

    logger.Info().Msg("Hello from Zerolog logger")
```

```
}
```

The ability to add contextual properties to your log records in the form of key/value pairs is fully supported. It also provides helpers for saving and retrieving a logger from a `context.Context` instance making it easy to contextualize your logging in the application.

Here's a basic example demonstrating this feature:

```
func SomeMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        ctx := r.Context()

        l := zerolog.New(os.Stdout)

        l.UpdateContext(func(c zerolog.Context) zerolog.Context {
            // add contextual properties to the logger's context like this
            return c.Str("user_id", "usr-1234")
        })

        // store the logger in the request's context
        r = r.WithContext(l.WithContext(ctx))

        next.ServeHTTP(w, r)
    })
}

// your HTTP handler
func handler(w http.ResponseWriter, r *http.Request) {
    // retrieve the logger from the request context
    l := zerolog.Ctx(r.Context())

    // all subsequent logs will contain the contextual properties
previously attached
    // to the logger along with any additional context added at log point
    l.Info().Str("doc_id", "doc-xyz").Msg("doc-xyz deleted")
}
```

Zerolog offers other notable features, including log volume reduction through sampling, the ability to execute code before logging for transformation, transportation or filtration purposes, logging to multiple destinations,

and automatic collection and formatting of stack traces when used with libraries like pkgerrors

or xerrors .

See our complete guide to Zerolog to learn more and view more complex usage examples.

2. Zap

Zap grew out of Uber's frustration with the slow logging times recorded on their high-performance servers. It pioneered the minimal allocations approach that was eventually adopted and improved on by Zerolog, although the latter gave up some flexibility to achieve the performance gains.

Zap closely follows Zerolog in terms of performance and memory allocations, but outshines it when it comes to customization capabilities, allowing you to closely tailor the framework to your specific needs.

This high level of customization is achieved through the Zapcore

package, which defines and implements the low-level interfaces on which Zap is built. By creating alternate implementations for these interfaces, you have the power to customize and extend Zap's behavior in any way you desire.

Getting started with logging in Zap is made easier with the availability of development and production presets. The former logs at the `DEBUG` level using a human-friendly format, while the production preset logs in JSON format at the `INFO` level. Unlike most logging frameworks, a pre-configured global logger is not automatically available. It needs to be explicitly initialized before use.

```
package main

import (
    "os"

    "go.uber.org/zap"
```

```
)

func init() {
    logger := zap.Must(zap.NewProduction())
    if os.Getenv("APP_ENV") == "development" {
        logger = zap.Must(zap.NewDevelopment())
    }

    zap.ReplaceGlobals(logger)
}

func main() {
    zap.L().Info("Hello from Zap!")
}
```

```
{"level":"info","ts":1689166454.9876506,"caller":"slog/main.go:12","msg":"H
from Zap!"}
```

Zap also has a unique approach to its logging APIs. It provides a base `Logger` type which provides type-safety and the best possible performance at the cost of verbosity, and the `SugaredLogger` type which wraps the `Logger` functionality in a slower but less verbose API. You can use both APIs in the same codebase and convert between them freely as you wish.

```
func main() {
    logger := zap.Must(zap.NewProduction())

    defer logger.Sync()

    // only strongly typed fields can be used with a `Logger`
    logger.Info("user signed in",
        zap.Int("userid", 123456),
        zap.String("provider", "facebook"),
    )

    sugar := logger.Sugar()

    // `SugaredLogger` supports weakly typed contextual fields
    sugar.Infow("user signed in", "userid", 123456, "provider",
"facebook")
```

```
}
```

Ensure to check out our comprehensive guide to Zap for more details.

3. Slog

Slog is the new built-in logging framework that just landed in Go 1.21

, residing at `log/slog`. It aims to address the need for a high-performance, structured, and leveled logging solution in the Go standard library. While it provides a pre-configured global logger like the `log` package, the default output format is not structured. However, creating a custom `Logger` instance allows you to change this behavior easily.

```
package main

import (
    "log/slog"
)

func main() {
    slog.Debug("Debug message")

    logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))
    logger.Debug("Debug message")
}
```

```
2023/07/12 15:29:22 INFO Info message
{"time":"2023-07-12T15:29:22.89164153+02:00","level":"INFO","msg":"Info
message"}
```

To understand how Slog works, there are three key aspects:

Slog offers a `Logger` frontend that provides level methods such as `Info()`, `Warn()`, and `Error()` for recording various events in the program.

Each log record created by the `Logger` is represented by the `Record` type.

The `Handler` type serves as the backend aspect of Slog. It determines how the log records are formatted and

processed.

Slog provides two built-in handlers by default: `TextHandler` and `JSONHandler`. The former produces Logfmt output, while the latter produces line-delimited JSON. You can also use any alternative backends produced by the community or create your own from scratch as you see fit.

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

    a.Info("Info message")
    b.Info("Info message")
}
```

```
{"time":"2023-07-12T15:33:28.941701701+02:00","level":"INFO","msg":"Info message"}
time=2023-07-12T15:33:28.941+02:00 level=INFO msg="Info message"
```

You even also use existing third-party logging frameworks as backends for Slog as long as they implement the `Handler` interface. Here's an example that uses [Zap's official slog.Handler implementation]

:

```
package main

import (
    "os"

    "go.uber.org/zap"
    "go.uber.org/zap/exp/zapslog"
    "golang.org/x/exp/slog"
)

func main() {
    zapLogger := zap.Must(zap.NewProduction())
```

```
    defer zapLogger.Sync()

    logger := slog.New(zapslog.NewHandler(zapLogger.Core()))

    logger.Info(
        "Using Slog frontend with Zap backend!",
        slog.Int("process_id", os.Getpid()),
    )
}
```

```
{"level":"info","ts":1689301952.7301471,"msg":"Using Slog frontend with Zap backend!","process_id":1662258}
```

When it comes to logging APIs, Slog draws some inspiration from Zap. First, it allows you to provide alternating key/value pairs for contextual logging. Alternatively, you can opt for strongly typed attributes, which minimize allocations and locking but increase verbosity. You can see a comparison of the two approaches below:

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

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),
)
```

See this article for a deep dive on Slog's features, capabilities, and how it enhances logging practices in Go applications.

4. Logrus

Logrus is one of the early pioneers of structured logging in Go, but it's now no longer being actively developed. It preserved compatibility with the built-in `log` package but added log levels, contextual logging, customizable output formats and integration with popular logging tools.

By default, Logrus utilizes the `TextFormatter` type to generate semi-structured and colorized output when logging to the terminal. However, it can be configured to produce fully structured output following the Logfmt standard. It also provides the `JSONFormatter` for logging in JSON.

```
package main

import (
    log "github.com/sirupsen/logrus"
)

func main() {
    log.SetFormatter(&log.JSONFormatter{})

    log.Info("Hello from Logrus!")
}
```

```
{"level":"info","msg":"Hello from
Logrus!","time":"2023-07-12T15:53:30+02:00"}
```

Context-aware logging is possible by using the `Fields` type which is an alias for `map[string]interface{}`. However, there is no type-safe option for contextual logging and its usage of maps is relatively inefficient .

```
func main() {
    log.SetFormatter(&log.JSONFormatter{})
```

```
    log.WithFields(log.Fields{
        "file":       "image.jpg",
        "size_bytes": 132932,
    }).Info("upload successful!")
}
```

```
{"file":"image.jpg","level":"info","msg":"upload
successful!","size_bytes":132932,"time":"2023-07-12T16:03:42+02:00"}
```

Logrus also supports hooks, allowing customization of the logging process. These hooks enable actions such as sending logs to a log management service or transforming and filtering logs before they reach their final destination. With over 80 hooks available , you have a wide range of options but if none of them suit your needs, you can always implement the `Hook` interface to address specific use cases.

5. Log15

Log15 is a logging framework that aims to produce logs that are both human-readable and machine-readable while providing a logging API that is easy to understand and use. Getting started with the framework is straightforward as it provides a pre-configured logger that is ready for use.

```
package main

import (
    log "github.com/inconshreveable/log15"
)

func main() {
```

```
    log.Info("Hello from Log15", "name", "John", "age", 20)
}
```

```
INFO[07-12|16:15:06] Hello from Log15                          name=John
age=20
```

The log message is the first argument to the level methods, followed by alternating key/value pairs for additional context. You can also create a new logger with some pre-defined fields by using the `New()` method. An alternative to variadic arguments for specifying contextual data is the `Ctx` type which, like Logrus' `Fields` type, is an alias for `map[string]inteface{}`.

```
func main() {
    logger := log.New("env", "prod", "go_version", "1.20")

    logger.Info("Hello from Log15", "name", "John", "age", 20)
    logger.Error("Something unexpected happened", log.Ctx{
        "error": errors.New("an error"),
    })
}
```

```
INFO[07-12|16:18:44] Hello from Log15                          env=prod
go_version=1.20 name=John age=20
EROR[07-12|16:18:44] Something unexpected happened             env=prod
go_version=1.20 error="an error"
```

When it comes to log formatting, the `Format` interface is provided along with a few implementations such as `JsonFormat`, `LogFmtFormat`, and `TerminalFormat`. The job of determining how and where log records are stored is left to the `Handler` interface, which provides a composable way to achieve a logging setup that suits your application.

```
func main() {
    handler := log.StreamHandler(os.Stdout, log.JsonFormat())
    logger := log.New()
    logger.SetHandler(handler)

    . . .
}
```

Some of the notable built-in `Handler` implementations include `StreamHandler` which writes to any `io.Writer`, `FilterHandler` for filtering records, `ChannelHandler` for writing logs to a channel, and `MultiHandler` for logging to multiple destinations simultaneously.

6. Logf

Logf is a logging framework that's dedicated to producing structured logs in the Logfmt format. It differentiates itself by offering a minimal API, making it a suitable choice for those seeking simplicity and a lightweight solution.

Logf provides five level methods: `Debug()`, `Info()`, `Warn()`, `Error()`, and `Fatal()`. Each one accepts the log message as the first argument, followed by alternating key/value pairs for contextual logging. It also supports adding default fields to all logs produced by a `Logger` instance.

```
package main

import (
    "time"

    "github.com/zerodha/logf"
)

func main() {
    logger := logf.New(logf.Opts{
        EnableColor:          true,
        Level:                logf.DebugLevel,
        EnableCaller:         true,
        TimestampFormat:      time.RFC3339Nano,
        DefaultFields:        []any{"go_version", "1.20"},
    })

    logger.Info("Hello from Logf!")
}
```

```
timestamp=2023-07-12T16:40:08.086175967+02:00 level=info message="Hello
from Logf!" caller=/home/ayo/dev/betterstack/demo/slog/main.go:19
go_version=1.20
```

The customization options are pretty minimal here. You can enable a colorized output, set any `io.Writer` to be the destination of the logs, change the timestamp format, set the default level, and enable the inclusion of the caller in the output. If your requirements extend beyond these options, you may need to explore other logging frameworks.

7. Apex/log

Apex is another well-regarded structured logging solution for Go. Although it has not seen recent updates, it remains a popular choice for logging in Go. Inspired by Logrus, Apex enhances the logging API in several ways:

It provides a `WithField()` method for attaching single key/value pairs to a record, while `WithFields()` for adding multiple contextual pairs.

Instead of Logrus' `Formatter` and `Hook` interfaces, Apex introduces a simpler `Handler` interface.

Several built-in handlers are available customize the log format and transport the generated logs to centralized log management systems.

Unique to Apex is its tracing support, which allows tracking operations, their durations, and any resulting errors with a single line of code.

Here's a basic example using Apex for logging:

```
package main

import (
    "github.com/apex/log"
)

func main() {
```

```
    logger := log.WithFields(log.Fields{
        "app": "myapp",
        "env": "prod",
    })

    logger.Info("Hello from Apex logger")
}
```

```
2023/07/12 11:46:34  info Hello from Apex logger    app=myapp env=prod
```

Without configuration, Apex produces a semi-structured log output where contextual fields are outputted in the Logfmt format, while the standard fields are presented as is without a corresponding key. However, it's easy to switch to fully structured format by importing either the `json` or `logfmt` handlers and using them as follows:

```
package main

import (
    "os"

    "github.com/apex/log"
    "github.com/apex/log/handlers/logfmt"
)

func main() {
    stdout := logfmt.New(os.Stdout)

    log.SetHandler(stdout)
    logger := log.WithFields(log.Fields{
        "app": "myapp",
        "env": "prod",
    })

    logger.Info("Hello from Apex logger")
}
```

```
timestamp=2023-07-12T11:47:18.796211807+02:00 level=info message="Hello
from Apex logger" app=myapp env=prod
```

The `multi` handler allows logging using multiple formats or sending logs to different destinations simultaneously:

```
stdout := logfmt.New(os.Stdout)
stderr := json.New(os.Stderr)

log.SetHandler(multi.New(stdout, stderr))
```

Apex places contextual fields in a fields object when serializing log entries to the JSON format. This approach ensures consistent schema, facilitating further processing of log entries.

```
timestamp=2023-07-12T11:50:22.470598627+02:00 level=info message="Hello
from Apex logger" app=myapp env=prod
{"fields":
{"app":"myapp","env":"prod"},"level":"info","timestamp":"2023-07-12T11:50:2
from Apex logger"}
```

You'll notice that Apex places contextual fields in a `fields` object when serializing log entries to the JSON format. This approach ensures consistent schema, facilitating further processing of log entries.

One notable feature of Apex is its tracing support, which allows for tracking operations, their durations, and resulting errors without writing multiple logging statements. By using the `Trace()` method with a deferred call, you can conveniently generate logs that indicate the start and duration of an operation:

```
package main

import (
    "fmt"
    "net/http"
    "os"

    "github.com/apex/log"
    "github.com/apex/log/handlers/logfmt"
)

func fetchRandomQuote(l log.Interface) (err error) {
    url := "https://api.quotable.io/random"

defer l.Trace("fetching random quote").Stop(&err)
```

```
    _, err = http.Get(url)

    return
}

func main() {
    stdout := logfmt.New(os.Stdout)

    log.SetHandler(stdout)
    logger := log.WithFields(log.Fields{
        "app": "myapp",
        "env": "prod",
    })

    fetchRandomQuote(logger)
}
```

The above is a minimal example that makes an HTTP GET request to an API endpoint. The deferred `Trace()` call in the example above generates two logs: one indicates the start of an HTTP request and the other displays the duration and any errors encountered.

Here's the output if the HTTP request succeeds:

```
timestamp=2023-07-12T11:34:53.007127029+02:00 level=info message="fetching
random quote" app=myapp env=prod
timestamp=2023-07-12T11:34:54.013085863+02:00 level=info message="fetching
random quote" app=myapp duration=1005 env=prod
```

If it fails, you will notice an `error` property in the second log entry:

```
timestamp=2023-07-12T11:40:40.283743165+02:00 level=info message="fetching
random quote" app=myapp env=prod
timestamp=2023-07-12T11:40:40.284696226+02:00 level=error
message="fetching random quote" app=myapp duration=0 env=prod error="Get
\"https://api.quotable.i/random\": dial tcp: lookup api.quotable.i: no
such host"
```

8 Logr

Logr is an abstraction for Go programs that offers a way to implement logging without being tied to a particular implementation. It offers a `Logger` type that provides a simple API for emitting logs, but the actual formatting and writing of the records is left to types that implement `LogSink` interface.

The general idea is that application and library authors use methods on the `Logger` type to write logging statements, but the actual logging functionality is provided by any logging framework that implements the `LogSink` interface. This makes the logging interface remains the same no matter what underlying framework you choose.

Here's a simple example using a Zerolog implementation via go-logr/zerologr

:

```
package main

import (
    "errors"
    "os"

    "github.com/go-logr/logr"
    "github.com/go-logr/zerologr"
    "github.com/rs/zerolog"
)

func main() {
    zl := zerolog.New(os.Stderr)
    zl = zl.With().Caller().Timestamp().Logger()
```

```
    var log logr.Logger = zerologr.New(&zl)

    log.Info("hello from Logr and Zerolog!")
    log.Error(
        errors.New("file not found"),
        "file open failed",
        "file_path",
        "/home/john/abc.txt",
    )
}
```

```
{"level":"info","v":0,"caller":"/home/ayo/dev/betterstack/demo/slog
/main.go:18","time":"2023-07-13T07:56:45+02:00","message":"hello from Logr
and Zerolog!"}
{"level":"error","error":"file not found","file_path":"/home
/john/abc.txt","caller":"/home/ayo/dev/betterstack/demo/slog
/main.go:19","time":"2023-07-13T07:56:45+02:00","message":"file open
failed"}
```

The `Logger` API provides only two methods for creating logs:

```
Info(Info(msg string, keysAndValues ...interface{})
Error(err error, msg string, keysAndValues ...interface{})
```

With the `zerologr` implementation, these methods map to Zerolog's `info` and `error` levels directly. To log at other levels, you need to set the verbosity level using the `V()` method and chain the `Info()` method to the resulting logger:

```
log.V(0).Info("level 0") // the default, mappped to zerolog.InfoLevel
log.V(1).Info("level 1") // mapped to zerolog.DebugLevel
log.V(2).Info("level 2") // mapped to zerolog.TraceLevel
```

The idea here is that the higher the number, the less severe the log entry. Since `zerolog.TraceLevel` is the lowest severity in Zerolog, using a number higher than 2 will not produce any output.

It may take a little experimenting to fully understand the ideas espoused by Logr, and with the release of Slog, it makes more sense to use its abstraction for new projects as it is sure to be more widely adopted.

Benchmarks

The following benchmark results were observed by running Zap's benchmarking suite

.

Package	Time	Objects Allocated
zerolog	81 ns/op	0 allocs/op
zap	193 ns/op	0 allocs/op
zap (sugared)	227 ns/op	1 allocs/op
slog	322 ns/op	0 allocs/op
apex/log	19518 ns/op	53 allocs/op
log15	19812 ns/op	70 allocs/op
logrus	21997 ns/op	68 allocs/op

Zerolog is clearly the fastest of the bunch with Zap and Slog not far behind so between the three it mostly comes down to which API you prefer and how much customization you need. The others are an order of magnitude slower due to the substantial heap allocations so you should probably avoid them for new projects.

Final thoughts

The introduction of the `log/slog` package changes the game when it comes to logging solutions in Go. By leaning on many of the prior innovations, it succeeds in bringing a high-performance logging framework to the standard library that should cover most needs. It is safe to say that it has now made using libraries using older techniques like Apex, Log15, Logurs, and Logr mostly obsolete.

However, libraries like Zerolog and Zap will definitely remain relevant for the foreseeable future due to their performance and unique features. As demonstrated above, you can even use them as alternate backends for Slog if its built-in handlers don't quite meet up to your expectations.

One knock against Slog is its overly convoluted logging API for guaranteeing strongly-typed fields. However, the ability to switch out the underlying framework without affecting the API is a massive plus that should make logging in the Go ecosystem more seamless and consistent.

Thanks for reading, and happy logging!

Further reading:

Redis caching in Node.js.

Redis caching in Golang.

Working with JSON in Go.