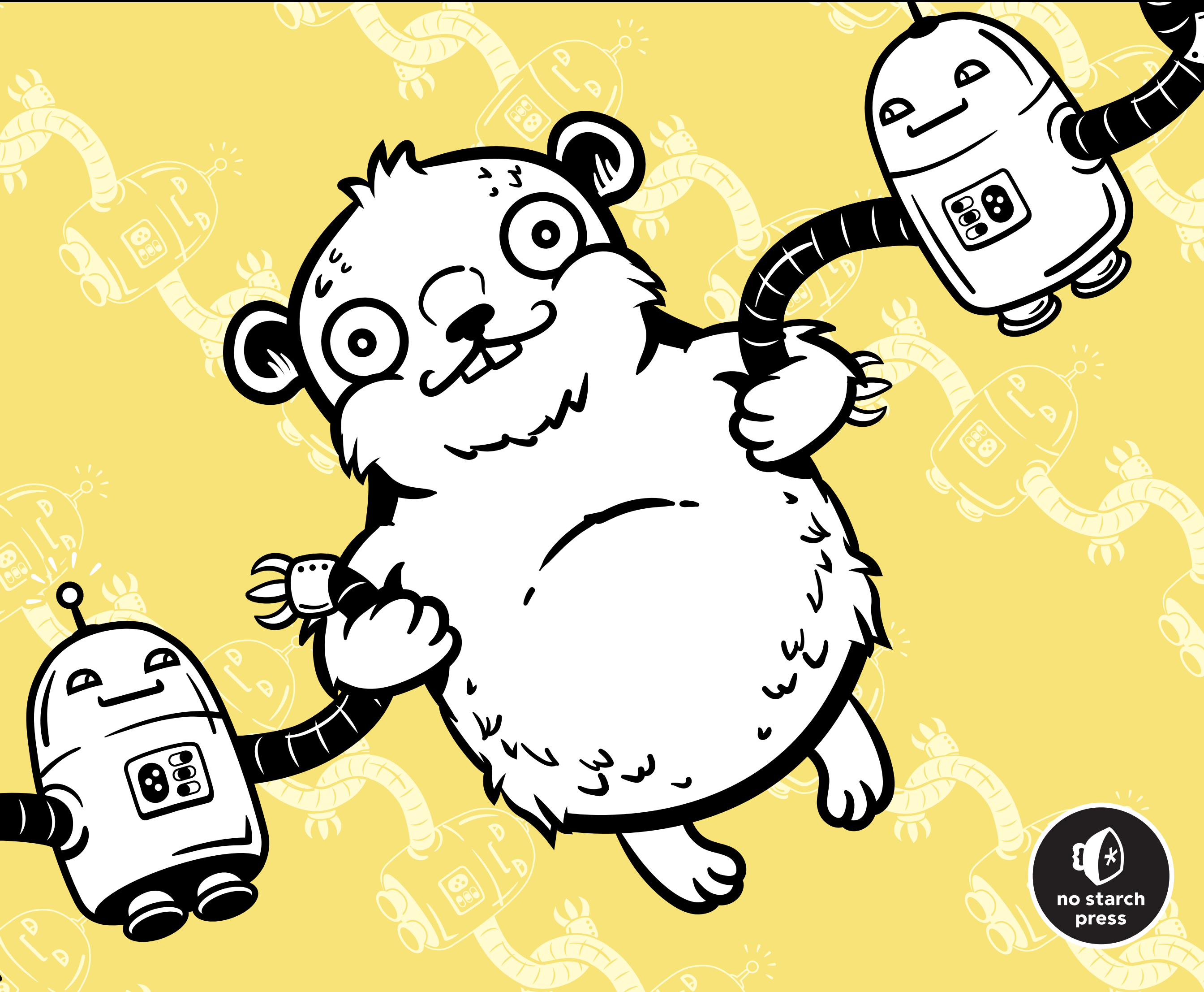
NETWORK PROGRAMMING WITH GO

CODE SECURE AND RELIABLE NETWORK SERVICES FROM SCRATCH

ADAM WOODBECK



no starch press

NETWORK PROGRAMMING WITH GO

Code Secure and Reliable Network Services from Scratch

Adam Woodbeck

no starch press
San Francisco

ISBN-13: 978-1-7185-0088-4 (print)
ISBN-13: 978-1-7185-0089-1 (ebook)

Publisher: William Pollock
Executive Editor: Barbara Yien
Production Editor: Kate Kaminski
Developmental Editor: Frances Saux
Cover Illustration: Gina Redman
Interior Design: Octopod Studios
Technical Reviewer: Jeremy Bowers
Copyeditor: Sharon Wilkey
Compositor: Jeff Lytle, Happenstance Type-O-Rama
Proofreader: Paula L. Fleming

For information on book distributors or translations, please contact No Starch Press, Inc. directly:
No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1-415-863-9900; info@nostarch.com
www.nostarch.com

`Library of Congress Control Number: 2020943331`

About the Author

Adam Woodbeck is a senior software engineer at Barracuda Networks, where he implemented a distributed cloud environment in Go to supplant the previous cloud infrastructure, profoundly increasing its scalability and performance. He's since served as the architect for many network-based services in Go.

About the Technical Reviewer

Jeremy Bowers is a distinguished software architect in the Office of CTO at Barracuda Networks. Equipped with many years of lead developer experience at Barracuda and security startups, especially in network engineering, Jeremy has successfully designed and implemented services that efficiently serve hundreds of thousands of customers worldwide. He holds a bachelor's and a master's degree in computer science from Michigan State University.

13

LOGGING AND METRICS

In an ideal world, our code would be free of bugs from the outset. Our network services would exceed our expectations for performance and capacity, and they would be robust enough to adapt to unexpected input without our intervention. But in the real world, we need to worry about unexpected and potentially malicious input, hardware degradation, network outages, and outright bugs in our code.

Monitoring our applications, no matter whether they are on premises or in the cloud, is vital to providing resilient, functional services to our users. Comprehensive logging allows us to receive timely details about errors, anomalies, or other actionable events, and metrics give us insight into the current state of our services, as well as help us identify bottlenecks. Taken together, logging and metrics allow us to manage service issues and focus our development efforts to avoid future failures.

You've used Go's `log` and `fmt` packages to give you feedback in previous chapters, but this chapter will take a deeper dive into logging and instrumenting your services. You will learn how to use log levels to increase or decrease the verbosity of your logs and when to use each log level. You'll learn how to add structure to your log entries so software can help you make better sense of log entries and zero in on relevant logs. I'll introduce you to the concept of wide event logging, which will help you maintain a handle on the amount of data you log as your services scale. You'll learn techniques for dynamically enabling debug logging and managing log file rotation from your code.

This chapter will also introduce you to Go kit's `metrics` package. Per Go kit's documentation, the `metrics` package "provides a set of uniform interfaces for service instrumentation." You'll learn how to instrument your services by using counters, gauges, and histograms.

By the end of this chapter, you should have a handle on how to approach logging, how to manage log files to prevent them from consuming too much hard drive space, and how to instrument your services to gain insight into their current state.

Event Logging

Logging is hard. Even experienced developers struggle to get it right. It's tough to anticipate what questions you'll need your logs to answer in the future, when your service fails—yet you should resist the urge to log everything just in case. You need to strike a balance in order to log the right information to answer those questions without overwhelming yourself with irrelevant log lines. Overzealous logging may suit you fine in development, where you control the scale of testing and overall entropy of your service, but it will quickly degrade your ability to find the needle in the haystack when you need to diagnose production failures.

In addition to figuring out what to log, you need to consider that logging isn't free. It consumes CPU and I/O time your application could otherwise use. A log entry added to a busy `for` loop while in development may help you understand what your service is doing. But it may become a bottleneck in production, insidiously adding latency to your service.

Instead, sampling these log entries, or logging on demand, may provide suitable compromises between log output and overhead. You might find it helpful to use *wide event* log entries, which summarize a transaction. For example, a service in development may log half a dozen entries about a request, any intermediate steps, and a response. In production, a single wide event log entry providing these details scales better. You'll learn more about wide event log entries in "Scaling Up with Wide Event Logging" on page 312.

Lastly, logging is subjective. An anomaly may be inconsequential in my application but indicative of a larger issue in your application. Whereas I could ignore the anomaly, you'd likely want to know about it. For this reason, it's best if we discuss logging in terms of best practices. These practices are a good baseline approach, but you should tailor them to each application.

The log Package

You have superficial experience using Go's `log` package, in earlier chapters, for basic logging needs, like timestamping log entries and optionally exiting your application with `log.Fatal`. But it has a few more features we have yet to explore. These require us to go beyond the package-level logger and instantiate our own `*log.Logger` instance. You can do this using the `log.New` function:

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

The `log.New` function accepts an `io.Writer`, a string prefix to use on each log line, and a set of flags that modify the logger's output. Accepting an `io.Writer` means the logger can write to anything that satisfies that interface, including an in-memory buffer or a network socket.

The default logger writes its output to `os.Stderr`, standard error. Let's look at an example logger in Listing 13-1 that writes to `os.Stdout`, standard output.

```
func Example_log() {
    l := log.New(❶os.Stdout, ❷"example: ", ❸log.Lshortfile)
    l.Print("logging to standard output")

    // Output:
    // example: ❹log_test.go:12: logging to standard output
}
```

*Listing 13-1: Writing a log entry to standard output (*log_test.go*)*

You create a new `*log.Logger` instance that writes to standard output ❶. The logger prefixes each line with the string *example:* ❷. The flags of the default logger are `log.Ldate` and `log.Ltime`, collectively `log.LstdFlags`, which print the timestamp of each log entry. Since you want to simplify the output for testing purposes when you run the example on the command line, you omit the timestamp and configure the logger to write the source code filename and line of each log entry ❸. The `l.Print` function on line 12 of the *log_test.go* file results in the output of those values ❹. This behavior can help with development and debugging, allowing you to zero in on the exact file and line of an interesting log entry.

Recognizing that the logger accepts an `io.Writer`, you may realize this allows you to use multiple writers, such as a log file and standard output or an in-memory ring buffer and a centralized logging server over a network. Unfortunately, the `io.MultiWriter` is not ideal for use in logging. An `io.MultiWriter` writes to each of its writers in sequence, aborting if it receives an error from any `Write` call. This means that if you configure your io.MultiWriter to write to a log file and standard output in that order, standard output will never receive the log entry if an error occurred when writing to the log file.

Fear not. It's an easy problem to solve. Let's create our own io.MultiWriter implementation, starting in Listing 13-2, that sustains writes across its writers and accumulates any errors it encounters.

```
package ch13

import (
    "io"

    "go.uber.org/multierr"
)

type sustainedMultiWriter struct {
    writers []io.Writer
}

func (s *sustainedMultiWriter) ❶Write(p []byte) (n int, err error) {
    for _, w := range s.writers {
        i, wErr := ❷w.Write(p)
        n += i
        err = ❸multierr.Append(err, wErr)
    }

    return n, err
}
```

*Listing 13-2: A multiwriter that sustains writing even after receiving an error (*writer.go*)*

As with io.MultiWriter, you'll use a struct that contains a slice of io.Writer instances for your sustained multiwriter. Your multiwriter implements the io.Writer interface ❶, so you can pass it into your logger. It calls each writer's Write method ❷, accumulating any errors with the help of Uber's multierr package ❸, before ultimately returning the total written bytes and cumulative error.

Listing 13-3 adds a function to initialize a new sustained multiwriter from one or more writers.

```
--snip--

func SustainedMultiWriter(writers ...io.Writer) io.Writer {
    mw := &sustainedMultiWriter{writers: ❶make([]io.Writer, 0, len(writers))}

    for _, w := range writers {
        if m, ok := ❷w.(*sustainedMultiWriter); ok {
            mw.writers = ❸append(mw.writers, m.writers...)
            continue
        }

        mw.writers = ❹append(mw.writers, w)
    }

    return mw
}
```

*Listing 13-3: Creating a sustained multiwriter (*writer.go*)*

First, you instantiate a new `*sustainedMultiWriter`, initialize its writers slice ❶, and cap it to the expected length of writers. You then loop through the given writers and append them to the slice ❹. If a given writer is itself a `*sustainedMultiWriter` ❷, you instead append its writers ❸. Finally, you return the pointer to the initialized `sustainedMultiWriter`.

You can now put your sustained multiwriter to good use in Listing 13-4.

```
package ch13

import (
    "bytes"
    "fmt"
    "log"
    "os"
)

func Example_logMultiWriter() {
    logFile := new(bytes.Buffer)
    w := ❶SustainedMultiWriter(os.Stdout, logFile)
    l := log.New(w, "example: ", ❷log.Lshortfile|log.Lmsgprefix)

    fmt.Println("standard output:")
    l.Print("Canada is south of Detroit")

    fmt.Print("\nlog file contents:\n", logFile.String())

    // Output:
    // standard output:
    // log_test.go:24: example: Canada is south of Detroit
    //
    // log file contents:
    // log_test.go:24: example: Canada is south of Detroit
}
```

*Listing 13-4: Logging simultaneously to a log file and standard output (*log_test.go*)*

You create a new sustained multiwriter ❶, writing to standard output, and a `bytes.Buffer` meant to act as a log file in this example. Next, you create a new logger using your sustained multiwriter, the prefix `example:`, and two flags ❷ to modify the logger's behavior. The addition of the `log.Lmsgprefix` flag (first available in Go 1.14) tells the logger to locate the prefix just before the log message. You can see the effect this has on the log entries in the example output. When you run this example, you see that the logger writes the log entry to the sustained multiwriter, which in turn writes the log entry to both standard output and the log file.

Leveled Log Entries

I wrote earlier in the chapter that verbose logging may be inefficient in production and can overwhelm you with the sheer number of log entries as your service scales up. One way to avoid this is by instituting *logging levels*, which assign a priority to each kind of event, enabling you to always log high-priority errors but conditionally log low-priority entries more suited for debugging and development purposes. For example, you'd always want to know if your service is unable to connect to its database, but you may care to log only details about individual connections while in development or when diagnosing a failure.

I recommend you create just a few log levels to begin with. In my experience, you can address most use cases with just an *error* level and a *debug* level, maybe even an *info* level on occasion. Error log entries should accompany some sort of alert, since these entries indicate a condition that needs your attention. Info log entries typically log non-error information. For example, it may be appropriate for your use case to log a successful database connection or to add a log entry when a listener is ready for incoming connections on a network socket. Debug log entries should be verbose and serve to help you diagnose failures, as well as aid development by helping you reason about the workflow.

Go's ecosystem offers several logging packages, most of which support numerous log levels. Although Go's `log` package does not have inherent support for leveled log entries, you can add similar functionality by creating separate loggers for each log level you need. Listing 13-5 does this: it writes log entries to a log file, but it also writes debug logs to standard output.

```
--snip--

func Example_logLevels() {
    lDebug := log.New(os.Stdout, ❶"DEBUG: ", log.Lshortfile)
    logFile := new(bytes.Buffer)
    w := SustainedMultiWriter(logFile, ❷lDebug.Writer())
    lError := log.New(w, ❸"ERROR: ", log.Lshortfile)

    fmt.Println("standard output:")
    lError.Print("cannot communicate with the database")
    lDebug.Print("you cannot hum while holding your nose")

    fmt.Print("\nlog file contents:\n", logFile.String())

    // Output:
    // standard output:
    // ERROR: log_test.go:43: cannot communicate with the database
    // DEBUG: log_test.go:44: you cannot hum while holding your nose
    //
    // log file contents:
    // ERROR: log_test.go:43: cannot communicate with the database
}
```

Listing 13-5: Writing debug entries to standard output and errors to both the log file and standard output (log_test.go)

First, you create a debug logger that writes to standard output and uses the `DEBUG:` prefix ❶. Next, you create a `*bytes.Buffer` to masquerade as a log file for this example and instantiate a sustained multiwriter. The sustained multiwriter writes to both the log file and the debug logger's `io.Writer` ❷. Then, you create an error logger that writes to the sustained multiwriter by using the prefix `ERROR:` ❸ to differentiate its log entries from the debug logger. Finally, you use each logger and verify that they output what you expect. Standard output should display log entries from both loggers, whereas the log file should contain only error log entries.

As an exercise, figure out how to make the debug logger conditional without wrapping its `Print` call in a conditional. If you need a hint, you'll find a suitable writer in the `io/ioutil` package that will let you discard its output.

This section is meant to demonstrate additional uses of the `log` package beyond what you've used so far in this book. Although it's possible to use this technique to log at different levels, you'd be better served by a logger with inherent support for log levels, like the Zap logger described in the next section.

Structured Logging

The log entries made by the code you've written so far are meant for human consumption. They are easy for you to read, since each log entry is little more than a message. This means that finding log lines relevant to an issue involves liberal use of the `grep` command or, at worst, manually skimming log entries. But this could become more challenging if the number of log entries increases. You may find yourself looking for a needle in a haystack. Remember, logging is useful only if you can quickly find the information you need.

A common approach to solving this problem is to add metadata to your log entries and then parse the metadata with software to help you organize them. This type of logging is called *structured logging*. Creating structured log entries involves adding key-value pairs to each log entry. In these, you may include the time at which you logged the entry, the part of your application that made the log entry, the log level, the hostname or IP address of the node that created the log entry, and other bits of metadata that you can use for indexing and filtering. Most structured loggers encode log entries as JSON before writing them to log files or shipping them to centralized logging servers. Structured logging makes the whole process of collecting logs in a centralized server easy, since the additional metadata associated with each log entry allows the server to organize and collate log entries across services. Once they're indexed, you can query the log server for specific log entries to better find timely answers to your questions.

Using the Zap Logger

Discussing specific centralized logging solutions is beyond the scope of this book. If you're interested in learning more, I suggest you initially investigate

Elasticsearch or Apache Solr. Instead, this section focuses on implementing the logger itself. You'll use the Zap logger from Uber, found at *https://pkg.go.dev/go.uber.org/zap/*, which allows you to integrate log file rotation.

Log file rotation is the process of closing the current log file, renaming it, and then opening a new log file after the current log file reaches a specific age or size threshold. Rotating log files is a good practice to prevent them from filling up your available hard drive space. Plus, searching through smaller, date-delimited log files is more efficient than searching through a single, monolithic log file. For example, you may want to rotate your log files every week and keep only eight weeks' worth of rotated log files. If you wanted to look at log entries for an event that occurred last week, you could limit your search to a single log file. Also, you can compress the rotated log files to further save hard drive space.

I've used other structured loggers on large projects, and in my experience, Zap causes the least overhead; I can use it in busy bits of code without a noticeable performance hit, unlike other heavyweight structured loggers. But your mileage may vary, so I encourage you to find what works best for you. You can apply the structured logging principles and log file management techniques described here to other structured loggers.

The Zap logger includes `zap.Core` and its options. The `zap.Core` has three components: a log-level threshold, an output, and an encoder. The *log-level threshold* sets the minimum log level that Zap will log; Zap will simply ignore any log entry below that level, allowing you to leave debug logging in your code and configure Zap to conditionally ignore it. Zap's *output* is a `zapcore.WriteSyncer`, which is an `io.Writer` with an additional `Sync` method. Zap can write log entries to any object that implements this interface. And the *encoder* can encode the log entry before writing it to the output.

Writing the Encoder

Although Zap provides a few helper functions, such as `zap.NewProduction` or `zap.NewDevelopment`, to quickly create production and development loggers, you'll create one from scratch, starting with the encoder configuration in Listing 13-6.

```
package ch13

import (
    "bytes"
    "fmt"
    "io/ioutil"
    "log"
    "os"
    "path/filepath"
    "runtime"
    "testing"
    "time"

    "go.uber.org/zap"
    "go.uber.org/zap/zapcore"
    "gopkg.in/fsnotify.v1"
    "gopkg.in/natefinch/lumberjack.v2"
)

var encoderCfg = zapcore.EncoderConfig{
    MessageKey: ❶"msg",
    NameKey:    ❷"name",

    LevelKey:    "level",
    EncodeLevel: ❸zapcore.LowercaseLevelEncoder,

    CallerKey:    "caller",
    EncodeCaller: ❹zapcore.ShortCallerEncoder,

 ❺ // TimeKey:   "time",
    // EncodeTime: zapcore.ISO8601TimeEncoder,
}
```

Listing 13-6: The encoder configuration for your Zap logger (zap_test.go)

The encoder configuration is independent of the encoder itself in that you can use the same encoder configuration no matter whether you're passing it to a JSON encoder or a console encoder. The encoder will use your configuration to dictate its output format. Here, your encoder configuration dictates that the encoder use the key `msg` ❶ for the log message and the key `name` ❷ for the logger's name in the log entry. Likewise, the encoder configuration tells the encoder to use the key `level` for the logging level and encode the level name using all lowercase characters ❸. If the logger is configured to add caller details, you want the encoder to associate these details with the `caller` key and encode the details in an abbreviated format ❹.

Since you need to keep the output of the following examples consistent, you'll omit the `time` key ❺ so it won't show up in the output. In practice, you'd want to uncomment these two fields.

Creating the Logger and Its Options

Now that you've defined the encoder configuration, let's use it in Listing 13-7 by instantiating a Zap logger.

```
--snip--

func Example_zapJSON() {
    zl := zap.New(
     ❶ zapcore.NewCore(
         ❷ zapcore.NewJSONEncoder(encoderCfg),
         ❸ zapcore.Lock(os.Stdout),
         ❹ zapcore.DebugLevel,
        ),
     ❺ zap.AddCaller(),
        zap.Fields(
         ❻ zap.String("version", runtime.Version()),
        ),
    )
    defer func() { _ = ❼zl.Sync() }()
    example := ❽zl.Named("example")
    example.Debug("test debug message")
    example.Info("test info message")

    // Output:
 ❾ // {"level":"debug","name":"example","caller":"ch13/zap_test.go:49",
"msg":"test debug message","version":"❿go1.15.5"}
    // {"level":"info","name":"example","caller":"ch13/zap_test.go:50",
"msg":"test info message","version":"go1.15.5"}
}
```

Listing 13-7: Instantiating a new logger from the encoder configuration and logging to JSON (zap_test.go)

The `zap.New` function accepts a `zap.Core` ❶ and zero or more `zap.Options`. In this example, you're passing the `zap.AddCaller` option ❺, which instructs the logger to include the caller information in each log entry, and a field ❻ named `version` that inserts the runtime version in each log entry.

The `zap.Core` consists of a JSON encoder using your encoder configuration ❷, a `zapcore.WriteSyncer` ❸, and the logging threshold ❹. If the `zapcore.WriteSyncer` isn't safe for concurrent use, you can use `zapcore.Lock` to make it concurrency safe, as in this example.

The Zap logger includes seven log levels, in increasing severity: `DebugLevel`, `InfoLevel`, `WarnLevel`, `ErrorLevel`, `DPanicLevel`, `PanicLevel`, and `FatalLevel`. The `InfoLevel` is the default. `DPanicLevel` and `PanicLevel` entries will cause Zap to log the entry and then panic. An entry logged at the `FatalLevel` will cause Zap to call `os.Exit(1)` after writing the log entry. Since your logger is using `DebugLevel`, it will log all entries.

I recommend you restrict the use of `DPanicLevel` and `PanicLevel` to development and `FatalLevel` to production, and only then for catastrophic startup errors, such as a failure to connect to the database. Otherwise, you're asking for trouble. As mentioned earlier, you can get a lot of mileage out of `DebugLevel`, `ErrorLevel`, and on occasion, `InfoLevel`.

Before you start using the logger, you want to make sure you defer a call to its `Sync` method ❼ to ensure all buffered data is written to the output.

You can also assign the logger a name by calling its `Named` method ❽ and using the returned logger. By default, a logger has no name. A named logger will include a name key in the log entry, provided you defined one in the encoder configuration.

The log entries ❾ now include metadata around the log message, so much so that the log line output exceeds the width of this book. It's also important to mention that the Go version ❿ in the example output is dependent on the version of Go you're using to test this example. Although you're encoding each log entry in JSON, you can still read the additional metadata you're including in the logs. You could ingest this JSON into something like Elasticsearch and run queries on it, letting Elasticsearch do the heavy lifting of returning only those log lines that are relevant to your query.

Using the Console Encoder

The preceding example included a bunch of functionality in relatively little code. Let's instead assume you want to log something a bit more human-readable, yet that has structure. Zap includes a console encoder that's essentially a drop-in replacement for its JSON encoder. Listing 13-8 uses the console encoder to write structured log entries to standard output.

```
--snip--

func Example_zapConsole() {
    zl := zap.New(
        zapcore.NewCore(
          ❶ zapcore.NewConsoleEncoder(encoderCfg),
            zapcore.Lock(os.Stdout),
          ❷ zapcore.InfoLevel,
        ),
    )
    defer func() { _ = zl.Sync() }()

    console := ❸zl.Named("[console]")
    console.Info("this is logged by the logger")
 ❹ console.Debug("this is below the logger's threshold and won't log")
    console.Error("this is also logged by the logger")

    // Output:
 ❺ // info   [console]   this is logged by the logger
    // error  [console]   this is also logged by the logger
}
```

*Listing 13-8: Writing structured logs using console encoding (*zap_test.go*)*

The console encoder ❶ uses tabs to separate fields. It takes instruction from your encoder configuration to determine which fields to include and how to format each.

Notice you don't pass the `zap.AddCaller` and `zap.Fields` options to the logger in this example. As a result, the log entries won't have `caller` and `version` fields. Log entries will include the `caller` field only if the logger has the `zap.AddCaller` option and the encoder configuration defines its `CallerKey`, as in Listing 13-6.

You name the logger ❸ and write three log entries, each with a different log level. Since the logger's threshold is the `info` level ❷, the debug log entry ❹ does not appear in the output because `debug` is below the `info` threshold.

The output ❺ lacks key names but includes the field values delimited by a tab character. Although not obvious in print, there's a tab character between the log level, the log name, and the log message. If you type this into your editor, be mindful to add tab characters between those elements lest the example fail when you run it.

Logging with Different Outputs and Encodings

Zap includes useful functions that allow you to concurrently log to different outputs, using different encodings, at different log levels. Listing 13-9 creates a logger that writes JSON to a log file and console encoding to standard output. The logger writes only the debug log entries to the console.

```
--snip--

func Example_zapInfoFileDebugConsole() {
    logFile := ❶new(bytes.Buffer)
    zl := zap.New(
        zapcore.NewCore(
            zapcore.NewJSONEncoder(encoderCfg),
            zapcore.Lock(❷zapcore.AddSync(logFile)),
            zapcore.InfoLevel,
        ),
    )
    defer func() { _ = zl.Sync() }()

 ❸ zl.Debug("this is below the logger's threshold and won't log")
    zl.Error("this is logged by the logger")
```

*Listing 13-9: Using *bytes.Buffer as the log output and logging JSON to it (zap_test.go)*

You're using `*bytes.Buffer` ❶ to act as a mock log file. The only problem with this is that `*bytes.Buffer` does not have a `Sync` method and does not implement the `zapcore.WriteSyncer` interface. Thankfully, Zap includes a helper function named `zapcore.AddSync` ❷ that intelligently adds a no-op `Sync` method to an `io.Writer`. Aside from the use of this function, the rest of the logger implementation should be familiar to you. It's logging JSON to the log file and excluding any log entries below the `info` level. As a result, the first log entry ❸ should not appear in the log file at all.

Now that you have a logger writing JSON to a log file, let's experiment with Zap and create a new logger in Listing 13-10 that can simultaneously write JSON log entries to a log file and console log entries to standard output.

```
--snip--

    zl = ❶zl.WithOptions(
      ❷ zap.WrapCore(
            func(c zapcore.Core) zapcore.Core {
                ucEncoderCfg := encoderCfg
              ❸ ucEncoderCfg.EncodeLevel = zapcore.CapitalLevelEncoder
                return ❹zapcore.NewTee(
                    c,
                  ❺ zapcore.NewCore(
                        zapcore.NewConsoleEncoder(ucEncoderCfg),
                        zapcore.Lock(os.Stdout),
                        zapcore.DebugLevel,
                    ),
                )
            },
        ),
    )

    fmt.Println("standard output:")
  ❻ zl.Debug("this is only logged as console encoding")
    zl.Info("this is logged as console encoding and JSON")

    fmt.Print("\nlog file contents:\n", logFile.String())

    // Output:
    // standard output:
    // DEBUG   this is only logged as console encoding
    // INFO    this is logged as console encoding and JSON
    //
    // log file contents:
    // {"level":"error","msg":"this is logged by the logger"}
    // {"level":"info","msg":"this is logged as console encoding and JSON"}
}
```

*Listing 13-10: Extending the logger to log to multiple outputs (*zap_test.go*)*

Zap's `WithOptions` method ❶ clones the existing logger and configures the clone with the given options. You can use the `zap.WrapCore` function ❷ to modify the underlying `zap.Core` of the cloned logger. To mix things up, you make a copy of the encoder configuration and tweak it to instruct the encoder to output the level using all capital letters ❸. Lastly, you use the `zapcore.NewTee` function, which is like the `io.MultiWriter` function, to return a `zap.Core` that writes to multiple cores ❹. In this example, you're passing in the existing core and a new core ❺ that writes `debug`-level log entries to standard output.

When you use the cloned logger, both the log file and standard output receive any log entry at the `info` level or above, whereas only standard output receives debugging log entries ❻.

Sampling Log Entries

One of my warnings to you with regard to logging is to consider how it impacts your application from a CPU and I/O perspective. You don't want logging to become your application's bottleneck. This normally means taking special care when logging in the busy parts of your application.

One method to mitigate the logging overhead in critical code paths, such as a loop, is to sample log entries. It may not be necessary to log each entry, especially if your logger is outputting many duplicate log entries. Instead, try logging every *n*th occurrence of a duplicate entry.

Conveniently, Zap has a logger that does just that. Listing 13-11 creates a logger that will constrain its CPU and I/O overhead by logging a subset of log entries.

```
--snip--

func Example_zapSampling() {
    zl := zap.New(
      ❶ zapcore.NewSamplerWithOptions(
            zapcore.NewCore(
                zapcore.NewJSONEncoder(encoderCfg),
                zapcore.Lock(os.Stdout),
                zapcore.DebugLevel,
            ),
          ❷time.Second, ❸1, ❹3,
        ),
    )
    defer func() { _ = zl.Sync() }()

    for i := 0; i < 10; i++ {
        if i == 5 {
          ❺ time.Sleep(time.Second)
        }
      ❻ zl.Debug(fmt.Sprintf("%d", i))
      ❼ zl.Debug("debug message")
    }

    // ❽Output:
    // {"level":"debug","msg":"0"}
    // {"level":"debug","msg":"debug message"}
    // {"level":"debug","msg":"1"}
    // {"level":"debug","msg":"2"}
    // {"level":"debug","msg":"3"}
    // {"level":"debug","msg":"debug message"}
    // {"level":"debug","msg":"4"}
    // {"level":"debug","msg":"5"}
    // {"level":"debug","msg":"debug message"}
    // {"level":"debug","msg":"6"}
    // {"level":"debug","msg":"7"}
    // {"level":"debug","msg":"8"}
    // {"level":"debug","msg":"debug message"}
    // {"level":"debug","msg":"9"}
}
```

*Listing 13-11: Logging a subset of log entries to limit CPU and I/O overhead (*zap_test.go*)*

The `NewSamplerWithOptions` function ❶ wraps `zap.Core` with sampling functionality. It requires three additional arguments: a sampling interval ❷, the number of initial duplicate log entries to record ❸, and an integer ❹ representing the *n*th duplicate log entry to record after that point. In this example, you are logging the first log entry, and then every third duplicate log entry that the logger receives in a one-second interval. Once the interval elapses, the logger starts over and logs the first entry, then every third duplicate for the remainder of the one-second interval.

Let's look at this in action. You make 10 iterations around a loop. Each iteration logs both the counter ❻ and a generic debug message ❼, which stays the same for each iteration. On the sixth iteration, the example sleeps for one second ❺ to ensure that the sample logger starts logging anew during the next one-second interval.

Examining the output ❽, you see that the debug message prints during the first iteration and not again until the logger encounters the third duplicate debug message during the fourth loop iteration. But on the sixth iteration, the example sleeps, and the sample logger ticks over to the next one-second interval, starting the logging over. It logs the first debug message of the interval in the sixth loop iteration and the third duplicate debug message in the ninth iteration of the loop.

Granted, this is a contrived example, but one that illustrates how to use this log-sampling technique as a compromise in CPU- and I/O-sensitive portions of your code. One place this technique may be applicable is when sending work to worker goroutines. Although you may send work as fast as the workers can handle it, you might want periodic updates on each worker's progress without having to incur too much logging overhead. The sample logger allows you to temper the log output and strike a balance between timely updates and minimal overhead.

Performing On-Demand Debug Logging

If debug logging introduces an unacceptable burden on your application under normal operation, or if the sheer amount of debug log data overwhelms your available storage space, you might want the ability to enable debug logging on demand. One technique is to use a semaphore file to enable debug logging. A *semaphore file* is an empty file whose existence is meant as a signal to the logger to change its behavior. If the semaphore file is present, the logger outputs `debug`-level logs. Once you remove the semaphore file, the logger reverts to its previous log level.

Let's use the `fsnotify` package to allow your application to watch for filesystem notifications. In addition to the standard library, the `fsnotify` package uses the `x/sys` package. Before you start writing code, let's make sure our `x/sys` package is current:

```
$ go get -u golang.org/x/sys/...
```

Not all logging packages provide safe methods to asynchronously modify log levels. Be aware that you may introduce a race condition if you attempt to modify a logger's level at the same time that the logger is reading the log level. The Zap logger allows you to retrieve a `sync/atomic`-based leveler to dynamically modify a logger's level while avoiding race conditions. You'll pass the atomic leveler to the `zapcore.NewCore` function in place of a log level, as you've previously done.

The `zap.AtomicLevel` struct implements the `http.Handler` interface. You can integrate it into an API and dynamically change the log level over HTTP instead of using a semaphore.

Listing 13-12 begins an example of dynamic logging using a semaphore file. You'll implement this example over the next few listings.

```
--snip--

func Example_zapDynamicDebugging() {
    tempDir, err := ioutil.TempDir("", "")
    if err != nil {
        log.Fatal(err)
    }
    defer func() { _ = os.RemoveAll(tempDir) }()

    debugLevelFile := ❶filepath.Join(tempDir, "level.debug")
    atomicLevel := ❷zap.NewAtomicLevel()

    zl := zap.New(
        zapcore.NewCore(
            zapcore.NewJSONEncoder(encoderCfg),
            zapcore.Lock(os.Stdout),
          ❸ atomicLevel,
        ),
    )
    defer func() { _ = zl.Sync() }()
```

Listing 13-12: Creating a new logger using an atomic leveler (zap_test.go)

Your code will watch for the *level.debug* file ❶ in the temporary directory. When the file is present, you'll dynamically change the logger's level to `debug`. To do that, you need a new atomic leveler ❷. By default, the atomic leveler uses the `info` level, which suits this example just fine. You pass in the atomic leveler ❸ when creating the core as opposed to specifying a log level itself.

Now that you have an atomic leveler and a place to store your semaphore file, let's write the code that will watch for semaphore file changes in Listing 13-13.

```
--snip--

    watcher, err := ❶fsnotify.NewWatcher()
    if err != nil {
        log.Fatal(err)
    }
    defer func() { _ = watcher.Close() }()

    err = ❷watcher.Add(tempDir)
    if err != nil {
        log.Fatal(err)
    }

    ready := make(chan struct{})
    go func() {
        defer close(ready)

        originalLevel := ❸atomicLevel.Level()

        for {
            select {
            case event, ok := ❹<-watcher.Events:
                if !ok {
                    return
                }
                if event.Name == ❺debugLevelFile {
                    switch {
                    case event.Op&fsnotify.Create == ❻fsnotify.Create:
                        atomicLevel.SetLevel(zapcore.DebugLevel)
                        ready <- struct{}{}
                    case event.Op&fsnotify.Remove == ❼fsnotify.Remove:
                        atomicLevel.SetLevel(originalLevel)
                        ready <- struct{}{}
                    }
                }
            case err, ok := ❽<-watcher.Errors:
                if !ok {
                    return
                }
                zl.Error(err.Error())
            }
        }
    }()
```

*Listing 13-13: Watching for any changes to the semaphore file (*zap_test.go*)*

First, you create a filesystem watcher ❶, which you'll use to watch the temporary directory ❷. The watcher will notify you of any changes to or within that directory. You also want to capture the current log level ❸ so that you can revert to it when you remove the semaphore file.

Next, you listen for events from the watcher ❹. Since you're watching a directory, you filter out any event unrelated to the semaphore file ❺ itself. Even then, you're interested in only the creation of the semaphore file or its removal. If the event indicates the creation of the semaphore file ❻, you change the atomic leveler's level to `debug`. If you receive a semaphore file removal event ❼, you set the atomic leveler's level back to its original level.

If you receive an error from the watcher ❽ at any point, you log it at the `error` level.

Let's see how this works in practice. Listing 13-14 tests the logger with and without the semaphore file present.

```
--snip--

❶ zl.Debug("this is below the logger's threshold")

  df, err := ❷os.Create(debugLevelFile)
  if err != nil {
      log.Fatal(err)
  }
  err = df.Close()
  if err != nil {
      log.Fatal(err)
  }
  <-ready

❸ zl.Debug("this is now at the logger's threshold")

  err = ❹os.Remove(debugLevelFile)
  if err != nil {
      log.Fatal(err)
  }
  <-ready
```

```
❺ zl.Debug("this is below the logger's threshold again")
❻ zl.Info("this is at the logger's current threshold")

    // Output:
    // {"level":"debug","msg":"this is now at the logger's threshold"}
    // {"level":"info","msg":"this is at the logger's current threshold"}
}
```

Listing 13-14: Testing the logger's use of the semaphore file (zap_test.go)

The logger's current log level via the atomic leveler is `info`. Therefore, the logger does not write the initial debug log entry ❶ to standard output. But if you create the semaphore file ❷, the code in Listing 13-13 should dynamically change the logger's level to `debug`. If you add another debug log entry ❸, the logger should write it to standard output. You then remove the semaphore file ❹ and write both a debug log entry ❺ and an info log entry ❻. Since the semaphore file no longer exists, the logger should write only the info log entry to standard output.

Scaling Up with Wide Event Logging

Wide event logging is a technique that creates a single, structured log entry per event to summarize the transaction, instead of logging numerous entries as the transaction progresses. This technique is most applicable to request-response loops, such as API calls, but it can be adapted to other use cases. When you summarize transactions in a structured log entry, you reduce the logging overhead while conserving the ability to index and search for transaction details.

One approach to wide event logging is to wrap an API handler in middleware. But first, since the `http.ResponseWriter` is a bit stingy with its output, you need to create your own response writer type (Listing 13-15) to collect and log the response code and length.

```
package ch13

import (
    "io"
    "io/ioutil"
    "net"
    "net/http"
    "net/http/httptest"
    "os"

    "go.uber.org/zap"
    "go.uber.org/zap/zapcore"
)

type wideResponseWriter struct {
❶ http.ResponseWriter
    length, status int
}
```

```
func (w *wideResponseWriter) ❷WriteHeader(status int) {
    w.ResponseWriter.WriteHeader(status)
    w.status = status
}

func (w *wideResponseWriter) ❸Write(b []byte) (int, error) {
    n, err := w.ResponseWriter.Write(b)
    w.length += n

    if w.status == 0 {
        w.status = ❹http.StatusOK
    }

    return n, err
}
```

*Listing 13-15: Creating a `ResponseWriter` to capture the response status code and length (*wide_test.go*)*

The new type embeds an object that implements the `http.ResponseWriter` interface ❶. In addition, you add `length` and `status` fields, since those values are ultimately what you want to log from the response. You override the `WriteHeader` method ❷ to easily capture the status code. Likewise, you override the `Write` method ❸ to keep an accurate accounting of the number of written bytes and optionally set the status code ❹ should the caller execute `Write` before `WriteHeader`.

Listing 13-16 uses your new type in wide event logging middleware.

```
--snip--

func WideEventLog(logger *zap.Logger, next http.Handler) http.Handler {
    return http.HandlerFunc(
        func(w http.ResponseWriter, r *http.Request) {
            wideWriter := ❶&wideResponseWriter{ResponseWriter: w}

         ❷ next.ServeHTTP(wideWriter, r)

            addr, _, _ := net.SplitHostPort(r.RemoteAddr)
         ❸ logger.Info("example wide event",
                zap.Int("status_code", wideWriter.status),
                zap.Int("response_length", wideWriter.length),
                zap.Int64("content_length", r.ContentLength),
                zap.String("method", r.Method),
                zap.String("proto", r.Proto),
                zap.String("remote_addr", addr),
                zap.String("uri", r.RequestURI),
                zap.String("user_agent", r.UserAgent()),
            )
        },
    )
}
```

*Listing 13-16: Implementing wide event logging middleware (*wide_test.go*)*

The wide event logging middleware accepts both a `*zap.Logger` and an `http.Handler` and returns an `http.Handler`. If this pattern is unfamiliar to you, please read "Handlers" on page 193.

First, you embed the `http.ResponseWriter` in a new instance of your wide event logging–aware response writer ❶. Then, you call the `ServeHTTP` method of the next `http.Handler` ❷, passing in your response writer. Finally, you make a single log entry ❸ with various bits of data about the request and response.

Keep in mind that I'm taking care here to omit values that would change with each execution and break the example output, like call duration. You would likely have to write code to deal with these in a real implementation.

Listing 13-17 puts the middleware into action and demonstrates the expected output.

```
--snip--

func Example_wideLogEntry() {
    zl := zap.New(
        zapcore.NewCore(
            zapcore.NewJSONEncoder(encoderCfg),
            zapcore.Lock(os.Stdout),
            zapcore.DebugLevel,
        ),
    )
    defer func() { _ = zl.Sync() }()

    ts := httptest.NewServer(
     ❶ WideEventLog(zl, http.HandlerFunc(
            func(w http.ResponseWriter, r *http.Request) {
                defer func(r io.ReadCloser) {
                    _, _ = io.Copy(ioutil.Discard, r)
                    _ = r.Close()
                }(r.Body)
                _, _ = ❷w.Write([]byte("Hello!"))
            },
        )),
    )
    defer ts.Close()

    resp, err := ❸http.Get(ts.URL + "/test")
    if err != nil {
     ❹ zl.Fatal(err.Error())
    }
    _ = resp.Body.Close()

    // ❺Output:
    // {"level":"info","msg":"example wide event","status_code":200,
"response_length":6,"content_length":0,"method":"GET","proto":"HTTP/1.1",
"remote_addr":"127.0.0.1","uri":"/test","user_agent":"Go-http-client/1.1"}
}
```

Listing 13-17: Using the wide event logging middleware to log the details of a GET call (wide_test.go)

As in Chapter 9, you use the `httptest` server with your `WideEventLog` middleware ❶. You pass `*zap.Logger` into the middleware as the first argument and `http.Handler` as the second argument. The handler writes a simple *Hello!* to the response ❷ so the response length is nonzero. That way, you can prove that your response writer works. The logger writes the log entry immediately before you receive the response to your GET request ❸. As before, I must wrap the JSON output ❺ for printing in this book, but it consumes a single line otherwise.

Since this is just an example, I elected to use the logger's `Fatal` method ❹, which writes the error message to the log file and calls `os.Exit(1)` to terminate the application. You shouldn't use this in code that is supposed to keep running in the event of an error.

Log Rotation with Lumberjack

If you elect to output log entries to a file, you could leverage an application like *logrotate* to keep them from consuming all available hard drive space. The downside to using a third-party application to manage log files is that the third-party application will need to signal to your application to reopen its log file handle lest your application keep writing to the rotated log file.

A less invasive and more reliable option is to add log file management directly to your logger by using a library like *Lumberjack*. Lumberjack handles log rotation in a way that is transparent to the logger, because your logger treats Lumberjack as any other `io.Writer`. Meanwhile, Lumberjack keeps track of the log entry accounting and file rotation for you.

Listing 13-18 adds log rotation to a typical Zap logger implementation.

```
--snip--

func TestZapLogRotation(t *testing.T) {
    tempDir, err := ioutil.TempDir("", "")
    if err != nil {
        t.Fatal(err)
    }
    defer func() { _ = os.RemoveAll(tempDir) }()

    zl := zap.New(
        zapcore.NewCore(
            zapcore.NewJSONEncoder(encoderCfg),
          ❶ zapcore.AddSync(
              ❷ &lumberjack.Logger{
                    Filename:   ❸filepath.Join(tempDir, "debug.log"),
                    Compress:   ❹true,
                    LocalTime:  ❺true,
                    MaxAge:     ❻7,
                    MaxBackups: ❼5,
                    MaxSize:    ❽100,
                },
            ),
            zapcore.DebugLevel,
        ),
    )
    defer func() { _ = zl.Sync() }()

    zl.Debug("debug message written to the log file")
}
```

Listing 13-18: Adding log rotation to the Zap logger using Lumberjack (zap_test.go)

Like the `*bytes.Buffer` in Listing 13-9, `*lumberjack.Logger` ❷ does not implement the `zapcore.WriteSyncer`. It, too, lacks a `Sync` method. Therefore, you need to wrap it in a call to `zapcore.AddSync` ❶.

Lumberjack includes several fields to configure its behavior, though its defaults are sensible. It uses a log filename in the format *<processname>-lumberjack.log*, saved in the temporary directory, unless you explicitly give it a log filename ❸. You can also elect to save hard drive space and have Lumberjack compress ❹ rotated log files. Each rotated log file is timestamped using UTC by default, but you can instruct Lumberjack to use local time ❺ instead. Finally, you can configure the maximum log file age before it should be rotated ❻, the maximum number of rotated log files to keep ❼, and the maximum size in megabytes ❽ of a log file before it should be rotated.

You can continue using the logger as if it were writing directly to standard output or `*os.File`. The difference is that Lumberjack will intelligently handle the log file management for you.

Instrumenting Your Code

Instrumenting your code is the process of collecting metrics for the purpose of making inferences about the current state of your service—such as the duration of each request-response loop, the size of each response, the number of connected clients, the latency between your service and a third-party API, and so on. Whereas logs provide a record of how your service got into a certain state, metrics give you insight into that state itself.

Instrumentation is easy, so much so that I'm going to give you the opposite advice I did for logging: instrument everything (initially). Fine-grained instrumentation involves hardly any overhead, it's efficient to ship, and it's inexpensive to store. Plus, instrumentation can solve one of the challenges of logging I mentioned earlier: that you won't initially know all the questions you'll want to ask, particularly for complex systems. An insidious problem may be ready to ruin your weekend because you lack critical metrics to give you an early warning that something is wrong.

This section will introduce you to metric types and show you the basics for using those types in your services. You will learn about Go kit's `metrics` package, which is an abstraction layer that provides useful interfaces for popular metrics platforms. You'll round out the instrumentation by using Prometheus as your target metrics platform and set up an endpoint for Prometheus to scrape. If you elect to use a different platform in the future, you will need to swap out only the Prometheus bits of this code; you could leave the Go kit code as is. If you're just getting started with instrumentation, one option is to use Grafana Cloud at *https://grafana.com/products/cloud/* to scrape and visualize your metrics. Its free tier is adequate for experimenting with instrumentation.

Setup

To abstract the implementation of your metrics and the packages they depend on, let's begin by putting them in their own package (Listing 13-19).

```
package metrics

import (
    "flag"

 ❶ "github.com/go-kit/kit/metrics"
 ❷ "github.com/go-kit/kit/metrics/prometheus"
 ❸ prom "github.com/prometheus/client_golang/prometheus"
)

var (
    Namespace = ❹flag.String("namespace", "web", "metrics namespace")
    Subsystem = ❺flag.String("subsystem", "server1", "metrics subsystem")
```

*Listing 13-19: Imports and command line flags for the metrics example (*instrumentation/metrics/metrics.go*)*

You import Go kit's `metrics` package ❶, which provides the interfaces your code will use, its `prometheus` adapter ❷ so you can use Prometheus as your metrics platform, and Go's Prometheus client package ❸ itself. All Prometheus-related imports reside in this package. The rest of your code will use Go kit's interfaces. This allows you to swap out the underlying metrics platform without the need to change your code's instrumentation.

Prometheus prefixes its metrics with a namespace and a subsystem. You could use the service name for the namespace and the node or hostname for the subsystem, for example. In this example, you'll use `web` for the namespace ❹ and `server1` for the subsystem ❺ by default. As a result, your metrics will use the `web_server1_` prefix. You'll see this prefix in Listing 13-30's command line output.

Now let's explore the various metric types, starting with counters.

Counters

Counters are used for tracking values that only increase, such as request counts, error counts, or completed task counts. You can use a counter to calculate the rate of increase for a given interval, such as the number of connections per minute.

Listing 13-20 defines two counters: one to track the number of requests and another to account for the number of write errors.

```
--snip--

    Requests ❶metrics.Counter = ❷prometheus.NewCounterFrom(
     ❸ prom.CounterOpts{
            Namespace: *Namespace,
            Subsystem: *Subsystem,
            Name:    ❹"request_count",
            Help:    ❺"Total requests",
        },
        []string{},
    )

    WriteErrors metrics.Counter = prometheus.NewCounterFrom(
        prom.CounterOpts{
            Namespace: *Namespace,
            Subsystem: *Subsystem,
            Name:      "write_errors_count",
            Help:      "Total write errors",
        },
        []string{},
    )
```

*Listing 13-20: Creating counters as Go kit interfaces (*instrumentation/metrics/metrics.go*)*

Each counter implements Go kit's `metrics.Counter` interface ❶. The concrete type for each counter comes from Go kit's `prometheus` adapter ❷ and relies on a `CounterOpts` struct ❸ from the Prometheus client package for configuration. Aside from the namespace and subsystem values we covered, the other important values you set are the metric name ❹ and its help string ❺, which describes the metric.

Gauges

Gauges allow you to track values that increase or decrease, such as the current memory usage, in-flight requests, queue sizes, fan speed, or the number of ThinkPads on my desk. Gauges do not support rate calculations, such as the number of connections per minute or megabits transferred per second, while counters do.

Listing 13-21 creates a gauge to track open connections.

```
--snip--

    OpenConnections ❶metrics.Gauge = ❷prometheus.NewGaugeFrom(
     ❸ prom.GaugeOpts{
            Namespace: *Namespace,
            Subsystem: *Subsystem,
            Name:      "open_connections",
            Help:      "Current open connections",
        },
        []string{},
    )
```

*Listing 13-21: Creating a gauge as a Go kit interface (*instrumentation/metrics/metrics.go*)*

Creating a gauge is much like creating a counter. You create a new variable of Go kit's `metrics.Gauge` interface ❶ and use the `NewGaugeFrom` function ❷ from Go kit's `prometheus` adapter to create the underlying type. The Prometheus client's `GaugeOpts` struct ❸ provides the settings for your new gauge.

Histograms and Summaries

A *histogram* places values into predefined buckets. Each bucket is associated with a range of values and named after its maximum one. When a value is observed, the histogram increments the maximum value of the smallest bucket into which the value fits. In this way, a histogram tracks the frequency of observed values for each bucket.

Let's look at a quick example. Assuming you have three buckets valued at 0.5, 1.0, and 1.5, if a histogram observes the value 0.42, it will increment the counter associated with bucket 0.5, because 0.5 is the smallest bucket that can contain 0.42. It covers the range of 0.5 and smaller values. If the histogram observes the value 1.23, it will increment the counter associated with the bucket 1.5, which covers values in the range of above 1.0 through 1.5. Naturally, the 1.0 bucket covers the range of above 0.5 through 1.0.

You can use a histogram's distribution of observed values to estimate a percentage or an average of all values. For example, you could use a histogram to calculate the average request sizes or response sizes observed by your service.

A *summary* is a histogram with a few differences. First, a histogram requires predefined buckets, whereas a summary calculates its own buckets. Second, the metrics server calculates averages or percentages from histograms, whereas your service calculates the averages or percentages from summaries. As a result, you can aggregate histograms across services on the metrics server, but you cannot do the same for summaries.

The general advice is to use summaries when you don't know the range of expected values, but I'd advise you to use histograms whenever possible so that you can aggregate histograms on the metrics server. Let's use a histogram to observe request duration (see Listing 13-22).

```
--snip--

    RequestDuration ❶metrics.Histogram = ❷prometheus.NewHistogramFrom(
     ❸ prom.HistogramOpts{
            Namespace: *Namespace,
            Subsystem: *Subsystem,
            Buckets: ❹[]float64{
                0.0000001, 0.0000002, 0.0000003, 0.0000004, 0.0000005,
                0.000001, 0.0000025, 0.000005, 0.0000075, 0.00001,
                0.0001, 0.001, 0.01,
            },
            Name: "request_duration_histogram_seconds",
            Help: "Total duration of all requests",
        },
        []string{},
    )
)
```

*Listing 13-22: Creating a histogram metric (*instrumentation/metrics/metrics.go*)*

Both the summary and histogram metric types implement Go kit's `metrics.Histogram` interface ❶ from its `prometheus` adapter. Here, you're using a histogram metric type ❷, using the Prometheus client's `HistogramOpts` struct ❸ for configuration. Since Prometheus's default bucket sizes are too large for the expected request duration range when communicating over localhost, you define custom bucket sizes ❹. I encourage you to experiment with the number of buckets and bucket sizes.

If you'd rather implement `RequestDuration` as a summary metric, you can substitute the code in Listing 13-22 for the code in Listing 13-23.

```
--snip--

    RequestDuration ❶metrics.Histogram = prometheus.NewSummaryFrom(
        prom.SummaryOpts{
            Namespace: *Namespace,
            Subsystem: *Subsystem,
            Name: "request_duration_summary_seconds",
            Help: "Total duration of all requests",
        },
        []string{},
    )
)
```

Listing 13-23: Optionally creating a summary metric

As you can see, this looks a lot like a histogram, minus the `Bucket` method. Notice that you still use the `metrics.Histogram` interface ❶ with a Prometheus summary metric. This is because Go kit does not distinguish between histograms and summaries; only your implementation of the interface does.

Instrumenting a Basic HTTP Server

Let's combine these metric types in a practical example: instrumenting a Go HTTP server. The biggest challenges here are determining what you want to instrument, where best to instrument it, and what metric type is most appropriate for each value you want to track. If you use Prometheus for your metrics platform, as you'll do here, you'll also need to add an HTTP endpoint for the Prometheus server to scrape.

Listing 13-24 details the initial code needed for an application that comprises an HTTP server to serve the metrics endpoint and another HTTP server to pass all requests to an instrumented handler.

```
package main

import (
    "bytes"
    "flag"
    "fmt"
    "io"
    "io/ioutil"
    "log"
    "math/rand"
    "net"
    "net/http"
    "sync"
    "time"

 ❶ "github.com/prometheus/client_golang/prometheus/promhttp"

 ❷ "github.com/awoodbeck/gnp/ch13/instrumentation/metrics"
)

var (
    metricsAddr = ❸flag.String("metrics", "127.0.0.1:8081",
        "metrics listen address")
    webAddr = ❹flag.String("web", "127.0.0.1:8082", "web listen address")
)
```

*Listing 13-24: Imports and command line flags for the metrics example (*instrumentation/main.go*)*

The only imports your code needs are the `promhttp` package for the metrics endpoint and your `metrics` package to instrument your code. The `promhttp` package ❶ includes an `http.Handler` that a Prometheus server can use to scrape metrics from your application. This handler serves not only your metrics but also metrics related to the runtime, such as the Go version, number of cores, and so on. At a minimum, you can use the metrics provided by the Prometheus handler to gain insight into your service's memory utilization, open file descriptors, heap and stack details, and more.

All variables exported by your `metrics` package ❷ are Go kit interfaces. Your code doesn't need to concern itself with the underlying metrics platform or its implementation, only how these metrics are made available to the metrics server. In a real-world application, you could further abstract the Prometheus handler to fully remove any dependency other than your metrics package from the rest of your code. But in the interest of keeping this example succinct, I've included the Prometheus handler in the `main` package.

Now, onto the code you want to instrument. Listing 13-25 adds the function your web server will use to handle all incoming requests.

```
--snip--

func helloHandler(w http.ResponseWriter, _ *http.Request) {
 ❶ metrics.Requests.Add(1)
    defer func(start time.Time) {
     ❷ metrics.RequestDuration.Observe(time.Since(start).Seconds())
    }(time.Now())
    _, err := w.Write([]byte("Hello!"))
    if err != nil {
     ❸ metrics.WriteErrors.Add(1)
    }
}
```

Listing 13-25: An instrumented handler that responds with random latency (instrumentation/main.go)

Even in such a simple handler, you're able to make three meaningful measurements. You increment the requests counter upon entering the handler ❶ since it's the most logical place to account for it. You also immediately defer a function that calculates the request duration and uses the request duration summary metric to observe it ❷. Lastly, you account for any errors writing the response ❸.

Now, you need to put the handler to use. But first, you need a helper function that will allow you to spin up a couple of HTTP servers: one to serve the metrics endpoint and one to serve this handler. Listing 13-26 details such a function.

```
--snip--

func newHTTPServer(addr string, mux http.Handler,
    stateFunc ❶func(net.Conn, http.ConnState)) error {
    l, err := net.Listen("tcp", addr)
    if err != nil {
        return err
    }

    srv := &http.Server{
        Addr:              addr,
        Handler:           mux,
        IdleTimeout:       time.Minute,
        ReadHeaderTimeout: 30 * time.Second,
        ConnState:         stateFunc,
    }

    go func() { log.Fatal(srv.Serve(l)) }()

    return nil
}

func ❷connStateMetrics(_ net.Conn, state http.ConnState) {
    switch state {
    case http.StateNew:
     ❸ metrics.OpenConnections.Add(1)
    case http.StateClosed:
     ❹ metrics.OpenConnections.Add(-1)
    }
}
```

Listing 13-26: Functions to create an HTTP server and instrument connection states (instrumentation/main.go)

This HTTP server code resembles that of Chapter 9. The exception here is you're defining the server's `ConnState` field, accepting it as an argument ❶ to the `newHTTPServer` function.

The HTTP server calls its `ConnState` field anytime a network connection changes. You can leverage this functionality to instrument the number of open connections the server has at any one time. You can pass the `connStateMetrics` function ❷ to the `newHTTPServer` function anytime you want to initialize a new HTTP server and track its open connections. If the server establishes a new connection, you increment the open connections gauge ❸ by 1. If a connection closes, you decrement the gauge ❹ by 1. Go kit's gauge interface provides an `Add` method, so decrementing a value involves adding a negative number.

Let's create an example that puts all these pieces together. Listing 13-27 creates an HTTP server to serve up the Prometheus endpoint and another HTTP server to serve your instrumented handler.

```
--snip--

func main() {
    flag.Parse()
    rand.Seed(time.Now().UnixNano())

    mux := http.NewServeMux()
 ❶ mux.Handle("/metrics/", promhttp.Handler())
    if err := newHTTPServer(*metricsAddr, mux, ❷nil); err != nil {
        log.Fatal(err)
    }
    fmt.Printf("Metrics listening on %q ...\n", *metricsAddr)

    if err := newHTTPServer(*webAddr, ❸http.HandlerFunc(helloHandler),
      ❹connStateMetrics); err != nil {
        log.Fatal(err)
    }
    fmt.Printf("Web listening on %q ...\n\n", *webAddr)
```

Listing 13-27: Starting two HTTP servers to serve `metrics` and the `helloHandler` (instrumentation/main.go)

First, you spawn an HTTP server with the sole purpose of serving the Prometheus handler ❶ at the `/metrics/` endpoint where Prometheus scrapes metrics from by default. Since you do not pass in a function for the third argument ❷, this HTTP server won't have a function assigned to its `ConnState` field to call on each connection state change. Then, you spin up another HTTP server to handle each request with the `helloHandler` ❸. But this time, you pass in the `connStateMetrics` function ❹. As a result, this HTTP server will gauge open connections.

Now, you can spin up many HTTP clients to make a bunch of requests to affect your metrics (see Listing 13-28).

```
--snip--

    clients := ❶500
    gets := ❷100
    wg := new(sync.WaitGroup)

    fmt.Printf("Spawning %d connections to make %d requests each ...",
        clients, gets)
    for i := 0; i < clients; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()

            c := &http.Client{
                Transport: ❸http.DefaultTransport.(*http.Transport).Clone(),
            }

            for j := 0; j < gets; j++ {
                resp, err := ❹c.Get(fmt.Sprintf("http://%s/", *webAddr))
                if err != nil {
                    log.Fatal(err)
                }
                _, _ = ❺io.Copy(ioutil.Discard, resp.Body)
                _ = ❻resp.Body.Close()
            }
        }()
    }
❼   wg.Wait()
    fmt.Print(" done.\n\n")
```

*Listing 13-28: Instructing 500 HTTP clients to each make 100 GET calls (*instrumentation/main.go*)*

You start by spawning 500 HTTP clients ❶ to each make 100 GET calls ❷. But first, you need to address a problem. The `http.Client` uses the `http.DefaultTransport` if its `Transport` method is nil. The `http.DefaultTransport` does an outstanding job of caching TCP connections. If all 500 HTTP clients use the same transport, they'll all make calls over about two TCP sockets. Our open connections gauge would reflect the two idle connections when you're done with this example, which isn't really the goal.

Instead, you must make sure to give each HTTP client its own transport. Cloning the default transport ❸ is good enough for our purposes.

Now that each client has its own transport and you're assured each client will make its own TCP connection, you iterate through a GET call ❹ 100 times with each client. You must also be diligent about draining ❺ and closing ❻ the response body so each client can reuse its TCP connection.

Once all 500 HTTP clients complete their 100 calls ❼, you can move on to Listing 13-29 and check the current state of the metrics.

```
--snip--

    resp, err := ❶http.Get(fmt.Sprintf("http://%s/metrics", *metricsAddr))
    if err != nil {
        log.Fatal(err)
    }

    b, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        log.Fatal(err)
    }
    _ = resp.Body.Close()

    metricsPrefix := ❷fmt.Sprintf("%s_%s", *metrics.Namespace,
        *metrics.Subsystem)
    fmt.Println("Current Metrics:")
    for _, line := range bytes.Split(b, []byte("\n")) {
        if ❸bytes.HasPrefix(line, []byte(metricsPrefix)) {
            fmt.Printf("%s\n", line)
        }
    }
}
```

*Listing 13-29: Displaying the current metrics matching your namespace and subsystem (*instrumentation/main.go*)*

You retrieve all the metrics from the metrics endpoint ❶. This will cause the metrics web server to return all metrics stored by the Prometheus client, in addition to details about each metric it tracks, which includes the metrics you added. Since you're interested in only your metrics, you can check each line starting with your namespace, an underscore, and your subsystem ❷. If the line matches this prefix ❸, you print it to standard output. Otherwise, you ignore the line and move on.

Let's run this example on the command line and examine the resulting metrics in Listing 13-30.

```
$ go run instrumentation/main.go
Metrics listening on "127.0.0.1:8081" ...
Web listening on "127.0.0.1:8082" ...

Spawning 500 connections to make 100 requests each ... done.

Current Metrics:
web_server1_open_connections ❶500
web_server1_request_count ❷50000
web_server1_request_duration_histogram_seconds_bucket{le="1e-07"} ❸0
web_server1_request_duration_histogram_seconds_bucket{le="2e-07"} 1
web_server1_request_duration_histogram_seconds_bucket{le="3e-07"} 613
web_server1_request_duration_histogram_seconds_bucket{le="4e-07"} 13591
web_server1_request_duration_histogram_seconds_bucket{le="5e-07"} 33216
web_server1_request_duration_histogram_seconds_bucket{le="1e-06"} 40183
```

```
web_server1_request_duration_histogram_seconds_bucket{le="2.5e-06"} 49876
web_server1_request_duration_histogram_seconds_bucket{le="5e-06"} 49963
web_server1_request_duration_histogram_seconds_bucket{le="7.5e-06"} 49973
web_server1_request_duration_histogram_seconds_bucket{le="1e-05"} 49979
web_server1_request_duration_histogram_seconds_bucket{le="0.0001"} 49994
web_server1_request_duration_histogram_seconds_bucket{le="0.001"} 49997
web_server1_request_duration_histogram_seconds_bucket{le="0.01"} ❹50000
web_server1_request_duration_histogram_seconds_bucket{le="+Inf"} 50000
web_server1_request_duration_histogram_seconds_sum ❺0.04102166899999979
web_server1_request_duration_histogram_seconds_count ❻50000
```

Listing 13-30: Web server output and resulting metrics

As expected, 500 connections were open ❶ at the time you queried the metrics. These connections are idle. You can experiment with the HTTP client by invoking its `CloseIdleConnections` method after it's done making 100 GET calls; see how that change affects the open connections gauge. Likewise, see what happens to the open connections when you don't define their `Transport` field.

The request count is 50,000 ❷, so all requests succeeded.

Do you notice what's missing? The write errors counter. Since no write errors occur, the write errors counter never increments. As a result, it doesn't show up in the metrics output. You could make a call to `metrics.WriteErrors.Add(0)` to make the metric show up without changing its value, but its absence probably bothers you more than it bothers Prometheus. Just be aware that the metrics output may not include all instrumented metrics, just the ones that have changed since initialization.

The underlying Prometheus histogram is a *cumulative* histogram: any value that increments a bucket's counter also increments the counters for all buckets less than the value. Therefore, you see increasing values in each bucket until you reach the 0.01 bucket ❹. Even though you define a range of buckets, Prometheus adds an infinite bucket for you. In this example, you defined a bucket smaller than all observed values ❸, so its counter is still zero.

A histogram and a summary maintain two additional counters: the sum of all observed values ❺ and the total number of observed values ❻. If you use a summary, the Prometheus endpoint will present only these two counters. It will not detail the summary's buckets as it does with a histogram. Therefore, the Prometheus server can aggregate histogram buckets but cannot do the same for summaries.

What You've Learned

Logging is hard. Instrumentation, not so much. Be frugal with your logging and generous with your instrumentation. Logging isn't free and can quickly add latency if you aren't mindful of where and how much you log. You cannot go wrong by logging actionable items, particularly ones that should trigger an alert. On the other hand, instrumentation is very efficient. You should instrument everything, at least initially. Metrics detail the current state of your service and provide insight into potential problems, whereas logs provide an immutable audit trail of sorts that explains the current state of your service and helps you diagnose failures.

Go's `log` package provides enough functionality to satisfy basic log requirements. But it becomes cumbersome when you need to log to more than one output or at varying levels of verbosity. At that point, you're better off with a comprehensive solution such as Uber's Zap logger. No matter what logger you use, consider adding structure to your log entries by including additional metadata. Structured logging allows you to leverage software to quickly filter and search log entries, particularly if you centralize logs across your infrastructure.

On-demand debug logging and wide event logging are two methods you can use to collect important information while minimizing logging's impact on the performance of your service. You can use the creation of a semaphore file to signal your logger to enable debug logging. When you remove the semaphore file, the logger immediately disables debug logging. Wide event logs summarize events in a request-response loop. You can replace numerous log entries with a single wide event log without hindering your ability to diagnose failures.

One approach to instrumentation is to use Go kit's `metrics` package, which provides interfaces for common metric types and adapters for popular metrics platforms. It allows you to abstract the details of each metrics platform away from your instrumented code.

The `metrics` package supports counters, gauges, histograms, and summaries. Counters monotonically increase and can be used to calculate rates of change. Use counters to track values like request counts, error counts, or completed tasks. Gauges track values that can increase and decrease, such as current memory usage, in-flight requests, and queue sizes. Histograms and summaries place observed values in buckets and allow you to estimate averages or percentages of all values. You could use a histogram or summary to approximate the average request duration or response size.

Taken together, logging and metrics give you necessary insight into your service, allowing you to proactively address potential problems and recover from failures.