Frequently Asked Questions

Design

Why spend so much effort on logger performance?

Of course, most applications won't notice the impact of a slow logger: they already take tens or hundreds of milliseconds for each operation, so an extra millisecond doesn't matter.

On the other hand, why *not* make structured logging fast? The `SugaredLogger` isn't any harder to use than other logging packages, and the `Logger` makes structured logging possible in performance-sensitive contexts. Across a fleet of Go microservices, making each application even slightly more efficient adds up quickly.

Why aren't `Logger` and `SugaredLogger` interfaces?

Unlike the familiar `io.Writer` and `http.Handler`, `Logger` and `SugaredLogger` interfaces would include *many* methods. As Rob Pike points out, "The bigger the interface, the weaker the abstraction." Interfaces are also rigid — *any* change requires releasing a new major version, since it breaks all third-party implementations.

Making the `Logger` and `SugaredLogger` concrete types doesn't sacrifice much abstraction, and it lets us add methods without introducing breaking changes. Your applications should define and depend upon an interface that includes just the methods you use.

Why are some of my logs missing?

Logs are dropped intentionally by zap when sampling is enabled. The production configuration (as returned by `NewProductionConfig()` enables sampling which will cause repeated logs within a second to be sampled. See more details on why sampling is enabled in Why sample application logs.

Why sample application logs?

Applications often experience runs of errors, either because of a bug or because of a misbehaving user. Logging errors is usually a good idea, but it can easily make this bad situation worse: not only is your application coping with a flood of errors, it's also spending extra CPU cycles and I/O logging those errors. Since writes are typically serialized, logging limits throughput when you need it most.

Sampling fixes this problem by dropping repetitive log entries. Under normal conditions, your application writes out every entry. When similar entries are logged hundreds or thousands of times each second, though, zap begins dropping duplicates to preserve throughput.

Why do the structured logging APIs take a message in addition to fields?

Subjectively, we find it helpful to accompany structured context with a brief description. This isn't critical during development, but it makes debugging and operating unfamiliar systems much easier.

More concretely, zap's sampling algorithm uses the message to identify duplicate entries. In our experience, this is a practical middle ground between random sampling (which often drops the exact entry that you need while debugging) and hashing the complete entry (which is prohibitively expensive).

Why include package-global loggers?

Since so many other logging packages include a global logger, many applications aren't designed to accept loggers as explicit parameters. Changing function signatures is often a breaking change, so zap includes global loggers to simplify migration.

Avoid them where possible.

Why include dedicated Panic and Fatal log levels?

In general, application code should handle errors gracefully instead of using `panic` or `os.Exit`. However, every rule has exceptions, and it's common to crash when an error is truly unrecoverable. To avoid losing any information — especially the reason for the crash — the logger must flush any buffered entries before the process exits.

Zap makes this easy by offering `Panic` and `Fatal` logging methods that automatically flush before exiting. Of course, this doesn't guarantee that logs will never be lost, but it eliminates a common error.

See the discussion in uber-go/zap#207 for more details.

What's `DPanic`?

`DPanic` stands for "panic in development." In development, it logs at `PanicLevel`; otherwise, it logs at `ErrorLevel`. `DPanic` makes it easier to catch errors that are theoretically possible, but shouldn't actually happen, *without* crashing in production.

If you've ever written code like this, you need `DPanic`:

```
if err != nil {
  panic(fmt.Sprintf("shouldn't ever get here: %v", err))
}
```

Installation

What does the error `expects import "go.uber.org/zap"` mean?

Either zap was installed incorrectly or you're referencing the wrong package name in your code.

Zap's source code happens to be hosted on GitHub, but the import path is `go.uber.org/zap`. This gives us, the project maintainers, the freedom to move the source code if necessary. However, it means that you need to take a little care when installing and using the package.

If you follow two simple rules, everything should work: install zap with `go get -u go.uber.org/zap`, and always import it in your code with `import "go.uber.org/zap"`. Your code shouldn't contain *any* references to `github.com/uber-go/zap`.

Usage

Does zap support log rotation?

Zap doesn't natively support rotating log files, since we prefer to leave this to an external program like `logrotate`.

However, it's easy to integrate a log rotation package like `gopkg.in/natefinch/lumberjack.v2` as a `zapcore.WriteSyncer`.

```
// lumberjack.Logger is already safe for concurrent use, so we don't need to
// lock it.
w := zapcore.AddSync(&lumberjack.Logger{
  Filename:   "/var/log/myapp/foo.log",
  MaxSize:    500, // megabytes
  MaxBackups: 3,
  MaxAge:     28, // days
})
core := zapcore.NewCore(
  zapcore.NewJSONEncoder(zap.NewProductionEncoderConfig()),
  w,
  zap.InfoLevel,
)
logger := zap.New(core)
```

Extensions

We'd love to support every logging need within zap itself, but we're only familiar with a handful of log ingestion systems, flag-parsing packages, and the like. Rather than merging code that we can't effectively debug and support, we'd rather grow an ecosystem of zap extensions.

We're aware of the following extensions, but haven't used them ourselves:

Package	Integration
`github.com/tchap/zapext`	Sentry, syslog
`github.com/fgrosse/zaptest`	Ginkgo
`github.com/blendle/zapdriver`	Stackdriver
`github.com/moul/zapgorm`	Gorm
`github.com/moul/zapfilter`	Advanced filtering rules