Organizing Our Go Code

lessons from the chat backend

Jesse Allen
@jessecarl
Software Engineer
ASAPP (www.asapp.com)

Make the business logic clear.

What makes business logic unclear?

- Impedance mismatch – dependencies
- Noise – boilerplate
- A little bit of both – instrumentation

How do we get here?

- DRY applied to keystrokes instead of data ownership
  - Ex. Constructors that call constructors
- Exploration without synthesis
  - Ex. sql.NullBool with no DB in sight

Dependencies cause impedance mismatch

- Names
- Interfaces
- Mental models

Use vocabulary specific to the business logic

- `pkg/bikeshed` package acts as glossary for project
- `pkg/redis`, `pkg/rpc`, etc. packages match `bikeshed` vocabulary to dependency vocabulary
- `cmd/bikeshed` command puts it all together

What goes in the `pkg/bikeshed` package?

- Data types 😂
- Interfaces 😂
- Errors 😂
- Dependency 😱
- Core business logic 😅

Adapt the rest

- Use data types from `bikeshed`
- Implement and consume interfaces from `bikeshed`
- Implement and consume interfaces that match your dependencies

Sometimes there are more layers

- `pkg/redis` becomes like `pkg/bikeshed` to `pkg/redis/redigo`
- `pkg/redis/redigo` depends on `pkg/redis` and maybe `pkg/bikeshed`
- `pkg/redis` depends on `pkg/bikeshed`
- `pkg/bikeshed` depends on **nothing**

Instrumentation makes this difficult

- Tightly coupled with implementation
- Can add a lot of long lines to the code
- Metrics, logging, and tracing are dependencies too

Hooks in our own code

```
type Service struct {
    Hooks *ServiceHooks

    // …Other dependencies
}

func (s *Service) DoAThing(ctx context.Context,
req *DoAThingRequest) (resp *DoAThingResponse, err
error) {
    ctx = s.Hooks.beforeDoAThing(ctx)
    defer func() { // may not need to be in a
defer if there are few enough exit points
        s.Hooks.afterDoAThing(ctx, err)
    }()

    // …work

    return resp, nil
}
```

```
type ServiceHooks struct {
    BeforeDoAThing func(context.Context)
context.Context
    AfterDoAThing  func(context.Context, error)
}

func (h *ServiceHooks) beforeDoAThing(ctx
context.Context) context.Context {
    if h == nil || h.BeforeDoAThing == nil {
        return ctx
    }
    return h.BeforeDoAThing(ctx)
}

func (h *FooServiceHooks) afterDoAThing(ctx
context.Context, err error) {
    if h == nil || h.AfterDoAThing == nil {
        return
    }
    h.AfterDoAThing(ctx, err)
}
```

Hooks in our own code

- No dependencies!
- Very little overhead at instrumentation site
- With a little (generatable) boilerplate, safe defaults

Instrumentation gets a package

```
type Foo struct {
    Logger          log.Logger
    DoAThingRequest struct {
        Total     metrics.Counter
        Errors    metrics.Counter
        DurationS metrics.Histogram `labels:"failed"`
    } `labels:"client_type"`
}

func (f *Foo) ServiceHooks() *foo.ServiceHooks {
    return &foo.ServiceHooks{
        BeforeDoAThing: func(ctx context.Context) context.Context {
            s.DoAThingRequest.Total.With(fooMetricsKVs(ctx)...).Add(1)
            level.Debug(fooLogCtx(ctx, s.Logger)).Log("msg", "Started doing a thing")
            return context.WithValue(ctx, fooKeyStart, time.Now())
        },
        AfterDoAThing: func(ctx context.Context, err error) {
            logger := fooLogCtx(ctx, s.Logger)
            if err != nil {
                s.DoAThingRequest.Errors.With(fooMetricsKVs(ctx)...).Add(1)
                level.Error(logger).Log("msg", "Error while doing a thing", "error", err)
            }
            level.Debug(logger).Log("msg", "Finished doing a thing")
            s.DoAThingRequest.DurationS.With(fooMetricsKVs(ctx)...).With("failed", strconv.FormatBool(err != nil)).
                Observe(time.Since(ctx.Value(fooKeyStart).(time.Time)).Seconds())
        },
    }
}
```

Instrumentation gets a package

- Been using `cmd/bikeshed/instrumentation`, but `pkg/instrumentation` could work
- Despite being almost pure boilerplate, it’s somewhat clear
- Dependencies are still explicit

Explicit dependency == a lot of boilerplate

- More boilerplate in `main`
- Less boilerplate in business logic
- Find patterns to exploit

Configuration

```
type Log struct {
    Level  LogLevel
    Path   string
    Format LogFormat
}

func (c *Log) AddFlags(fs *flag.FlagSet) {
    fs.Var(&c.Level, "log-level", "Threshold for writing leveled logs. Unleveled logs are always
written.")
    fs.StringVar(&c.Path, "log-path", c.Path,
        "Path to allow expose runtime log level changes (empty means no log level change endpoint).")
    fs.Var(&c.Format, "log-format", "Format for log output (nop|json|logfmt).")
}

func NewLog() Log {
    return Log{
        Level:  InfoLogLevel,
        Path:   "/logz",
        Format: JSONLogFormat,
    }
}
```

Base library for common setup

- Loggers
- HTTP Servers
- Metrics
- Signal Capture

Starter template with common conventions

- Uses base library
- `DefaultServeMux` on separate port (with pprof, metrics, log level change, etc.)
- oklog `run.Group` to manage operation lifetimes

Conventions + tools = reduced boilerplate

```
type ConnMetrics struct {
    AcquiredS  metrics.Histogram `labels:"reused,was_idle" help:"time taken to acquire a connection"`
    IdleS      metrics.Histogram `buckets:".001,.0025,.005,.01,.025,.05,.1,.25,.5,1,2.5,5" help:"time
connection spent idle before being acquired"`
    IdledTotal metrics.Counter   `labels:"failed" help"connections put to idle"`
}
```

- Metrics are our most common dependency
- Reflect-based tool recursively populates metrics with help of tags
- Keeps declaration closer to use

Make the business logic clear.

- Separate dependency vocabulary from domain vocabulary
- Separate instrumentation details from instrumented code
- Concentrate boilerplate to reveal patterns for abstraction

Thank You