Operability in Go

Improving operations in Go programs

Who Am I

Ian N Schenck

SWE, Infrastructure, Oscar Health

@ianschenck (I don’t really tweet)

ian.schenck@gmail.com

Preface

I am: A SWE who keeps ending up in SRE.

I try to write *operable* code.

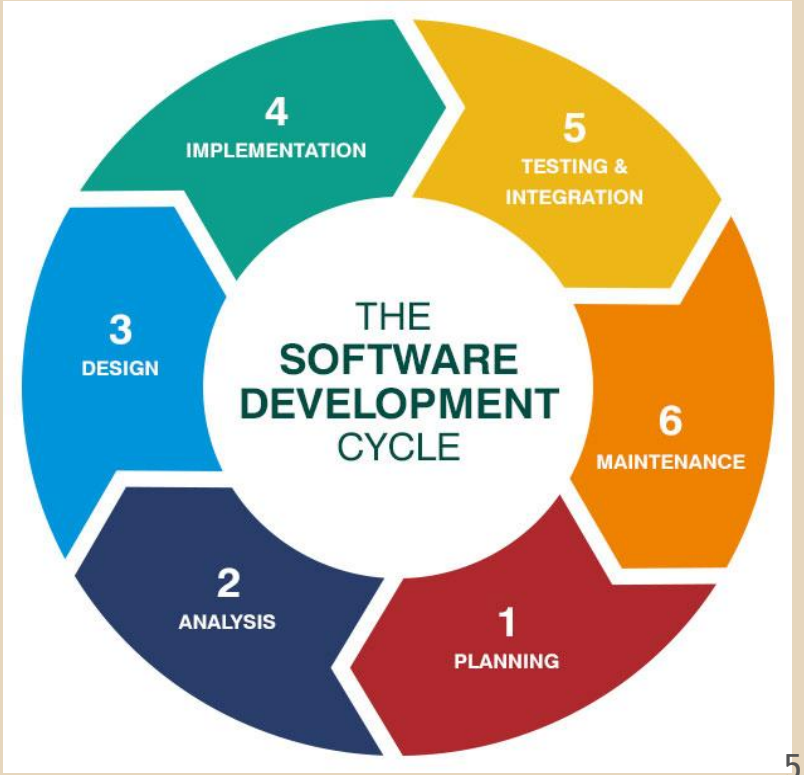
This is going to break, how do I make it *as easy as possible*?

What are Operations

Software Design Life Cycle (SDLC)

1. Plan
2. Analyze
3. Design
4. Implement
5. Document/Test
6. **Maintain**

*Not necessarily in that order



Maintenance

Even if your software is *perfect* and *entirely bug free*, it can still break.

- Environments are complex and changing
- Hardware can break
- Humans are buggy

Maintenance - Failure

When something fails, we have two *equal* objectives:

1. Fix it
2. Determine what went wrong

Maintenance - Failure

1. Fix it

Depends on the situation.

Let’s talk about failing well...

Failing Well

- Fail immediately when unrecoverable errors occur.
- Fail the smallest execution unit necessary.
- Err on the side of caution - fail as big as you need to (maybe the whole application).

Failing Well

In general, an unhandled/unrecoverable error should panic.

It should also give clear and concise information about what led to the panic.

Failing Well - Panic

Applications may panic, which will fail up to a deferred `recover()`

E.g. Panic in an HTTP handler will fail up to the serving goroutine.

Panic without `recover()` terminates.

Failing Well

Panic does give a stack trace, but could use more context.

Add context around application panics using logging.

Maintenance - Failure

2. Determine what went wrong.

If you're unable to determine what went wrong, you can't avoid repeating the failure.

Diagnosing Failure - 5 Whys

The vehicle will not start. (the problem)

1. Why? - The battery is dead.
2. Why? - The alternator is not functioning.
3. Why? - The alternator belt has broken.
4. Why? - The alternator belt wore out.
5. Why? - The vehicle was not maintained. (root cause)

Diagnosing Failure

We need (a lot of) information!

Killing a Stuck Process

`SIGQUIT` (kill -3) a process, get a stack trace:

```
goroutine 1 [IO wait, 5 minutes]:
...
net.(*TCPListener).AcceptTCP(0xc820124170, 0xc82005dbe0,
0x0, 0x0)
    /usr/local/go/src/net/tcpsock_posix.go:254 +0x4d
net.(*TCPListener).Accept(0xc820124170, 0x0, 0x0, 0x0,
0x0)
...
```

Sources of Information

Stack Trace

Logs

…

Logging

Provide context, don’t just:

```
log.Println(err)
```

E.g.

```
unexpected EOF
```

Unexpected EOF of ***what***!?

A Note on Errors

Some errors provide context:

```
listen tcp :33712: bind: address already in use
```

“Named” errors (`io.ErrUnexpectedEOF`) do not:

```
unexpected EOF
```

A Note on Errors

https://github.com/pkg/errors

You can add context to the error with `errors.Wrap`.

You can add context with the logger.

Logging Context

Structured logging adds key-value pairs to your log.

https://github.com/sirupsen/logrus

```
log.WithFields(logrus.Fields{
    "animal": "walrus",
    "number": 8,
}).Debug("Started observing beach")
```

Logging Context

Structured logging provides a way to add context in a machine and human consumable format.

```
INFO[0000] A group of walrus emerges from the ocean      animal=walrus size=10
WARN[0000] The group's number increased tremendously!    number=122 omg=true
INFO[0000] A giant walrus appears!                       animal=walrus size=10
INFO[0000] Tremendously sized cow enters the ocean.      animal=walrus size=9
FATA[0000] The ice breaks!                               number=100 omg=true
exit status 1
```

Logging Context

Structured loggers can output text or JSON format for easy consumption by logstash/ELK/Splunk.

Context can make all the difference...

Logging Anxiety

The anxiety over what to log, when. How much is too much, how much is enough?

Let’s set this aside for now.

Information

Other information?

Logs (action with context)
Environment
Flags
Stack Trace
More?

Information

Logging some of these may work, but perhaps there’s a better way.

Logging doesn’t work at all for some cases. E.g. what’s the current stack look like?

Information

What about exposing information outside of logging? Logging describes *action* with *context*.

`expvar` - in the standard library. Exposes *current state*.

expvar

Adds a route to the default ServerMux at `/debug/vars` as a side effect.

Also exposes a handler.

expvar

`expvar` provides an http handler/endpoint which exposes arbitrary data in JSON format.

What kind of data?

- cmdline
- memstats
- And more...

expvar

```
{
  "cmdline": [
    ".\/expvar_example"
  ],
  "memstats": {
    "Alloc": 136736,
    "TotalAlloc": 136736,
     ...
  }
}
```

MemStats

https://golang.org/pkg/runtime/#MemStats

Gives you various stats like:

- Allocated bytes (Heap/Sys/Total)
- GC statistics
- Allocations by size

expvar

Expose various `Var` types:

- Float*
- Int*
- Map*
- String

* Atomic operations

expvar

Can expose a variety of “Vars”, but notably there is `Publish(Func)`:

```
func init() {
    http.HandleFunc("/debug/vars", expvarHandler)
    Publish("cmdline", Func(cmdline))
    Publish("memstats", Func(memstats))
}
```

expvar

`expvar.Func`

Publish a function that returns `interface{}`.
The returned value is marshalled to JSON.

expvar

```
func memstats() interface{} {
    stats := new(runtime.MemStats)
    runtime.ReadMemStats(stats)
    return *stats
}
```

expvar

So what can we do with `expvar`?

Expose the environment:

```
expvar.Publish("env", expvar.Func(func () interface{}
{return os.Environ()}))
```

Exposing Environment

Want a `map` instead of a `slice`?

```
func publishEnv() interface{} {
    env := make(map[string]string)
    for _, line := range os.Environ() {
        parts := strings.SplitN(line, "=", 2)
        env[parts[0]] = parts[1]
    }
    return redactMap(env)
}
```

Exposing Secrets

You want to filter secret values somehow.

Replace the value with a hash so you can compare.

Exposing Flags

Flag values can be very useful for debugging

```
func publishFlags() interface{} {
    flagMap := make(map[string]interface{})
    flag.VisitAll(func(f *flag.Flag) {
        flagMap[f.Name] = f.Value
    })
    return redactMap(flagMap)
}
```

Exposing Flags

Flag magic values, for two reasons:

1. They can be changed.
2. They show up in `expvar`. (super useful)

Exposing a Stack Trace

Publish the stack trace:

```
func publishStack() interface{} {
    buf := make([]byte, 65535)
    n := runtime.Stack(buf, true)
    buf = buf[0:n]
    return string(buf)
}
```

expvar ALL the things!

```
expvar.Publish("env", expvar.Func(publishEnv))
expvar.Publish("flags", expvar.Func(publishFlags))
expvar.Publish("stack", expvar.Func(publishStack))
…
expvar.Publish("my-internal-value", ...)
```

Caution

Make sure you don’t publish very expensive functions!

Don’t make `expvar` too expensive or turn it into an accidental DoS vector.

Use Verbose Names

Exposing information is great. Make that information verbose/specific:

```
expvar.Publish("jobs", ...)
```

Better:

```
expvar.Publish("discovery-job-cache", ...)
```

Use Verbose Names

Flag Names:

- `addr`
- `init-timeout`
- `rpc-retries`

Better:

- `status-addr, status.addr`
- `discovery-init-timeout, discovery.init.timeout`
- `foo-rpc-retries, foo.rpc.retries`

Use Verbose Names

Environment Variables:

- `DB_PASSWORD`
- `INIT_TIMEOUT`

Better:

- `CLAIMS_DB_PASSWORD`
- `DISCOVERY_INIT_TIMEOUT`

expvar + structlog

These two work together.

Use expvar for *state*.

Use logging for *action*.

Less anxiety.

More with HTTP

`expvar` means we've already committed to having a port open responding to http requests.

What else can we do with this?

More with HTTP

- Health handler
- Specialized handlers for libraries (e.g. Vault integration)
- Shutdown handlers (quit and abort)
- Admin handlers

Specialized Endpoints

Probably only want to do modification/destruction on `POST`.

`GET` can return a form.

Specialized Endpoints - Monitoring

Prometheus is fantastic (you should use it)

Since you already have an internal/status http endpoint, dangle your prometheus metrics off of it.

Library Developers

Provide exported variables for application developers to expose in `expvar` or logs.

Or use `expvar` and prometheus directly? (But that side effect)

Recap

Think about failure at all times to guide:

- Panicking when necessary
- Exposing data via `expvar`
- Logging and context
- Naming (flags, environment variables, etc.)

Thanks

Come talk to me about operations and go!

Questions?