# Own Your Own Dependencies

Golang Reston

Michael Spiegel

January 16, 2019

# introduction

- a way to build a web service
- lessons learned along the way

# Outline

- Demo
- Assumptions
- Logging
- Contexts
- Errors
- Routing
- Integration
- Conclusions

# always be closing

- Always Be Closing is a GitHub service
- improves development workflow
- 5 min demo

# always be closing

- Always Be Closing is a GitHub service
- improves development workflow
- 5 min demo
- beta testers needed at thoughtdealership.com
- slides at github.com/thoughtdealership/howto
- howto is CC0 licensed

# tyranny of metrics

- lines of code metric used throughout the prsentation
- loc is generally a useless metric
- using it as a proxy for understanding the entire application
- we'll revisit this

# logging standards?

- lots of prior discussions about loggers
- Let's Talk About Logging
- The Hunt for a Logger Interface
- logging levels should be actionable
- level error goes to PagerDuty
- level warning goes to non-immediate reporting
- use structured logging to your advantage

# rs/zerolog to the rescue

- github.com/rs/zerolog
- leveled logger
- structured logger
- zero allocation (or low allocation) logger
- uses types not interfaces
- 5055 lines of code
- tradeoff: you can't inspect fields

# contexts

```
package context

// The provided key must be comparable and should not
// be of type string or any other built-in type to
// avoid collisions between packages using context.
// Users of WithValue should define their own types
// for keys. To avoid allocating when assigning to
// an interface{}, context keys often have concrete
// type struct{}.
func WithValue(parent Context, key,
                  val interface{}) Context {
  ...
  return &valueCtx{parent, key, val}
}
```

context → values → in → list

# contexts

```
package context

// The provided key must be comparable and should not
// be of type string or any other built-in type to
// avoid collisions between packages using context.
// Users of WithValue should define their own types
// for keys. To avoid allocating when assigning to
// an interface{}, context keys often have concrete
// type struct{}.
func WithValue(parent Context, key,
                    val interface{}) Context {
   ...
   return &valueCtx{parent, key, val}
}
```

context → values → in → list

Actually interface{} → interface{} → interface{} → interface{}

# type-safe contexts

```
package frame

type frameCtx struct{}

var frameKey frameCtx

type Frame struct {
    UUID   uuid.UUID
    Logger zerolog.Logger

    Foo string
    Bar bool
    Baz struct {
        A int
        B byte
        C string
    }
}
```

# type-safe contexts

```
package frame

func NewContext(ctx context.Context) context.Context {
    fr := new(Frame)
    fr.Logger = log.Logger
    return context.WithValue(ctx, frameKey, fr)
}

func FromContext(ctx context.Context) *Frame {
    fr := ctx.Value(frameKey)
    if fr == nil {
        return nil
    }
    return fr.(*Frame)
}
```

# error handling

- generate http response codes
- combine multiple errors
- generate http response codes for combined errors

# external errors

```
package exterror

type ExtError struct {
    Status int
    Err    error
}

func (e ExtError) Error() string {
    return e.Err.Error()
}

func Create(status int, err error) ExtError {
    return ExtError{Status: status, Err: err}
}
```

# combining errors

```
package multierr

type Error []error
```

- github.com/jonbodner/multierr
- 62 lines of code

# combining errors

```
func Append(e1 error, e2 error) error {
    if isNil(e1) && isNil(e2) { return nil }
    if isNil(e1) { return e2 }
    if isNil(e2) { return e1 }
    switch e1 := e1.(type) {
    case Error:
        switch e2 := e2.(type) {
            case Error: return append(e1, e2...)
            default: return append(e1, e2)
        }
    default:
        ...
    }
}
```

# combining external errors

```
// Convert generates an ExtError from
// a non-nil error input
func Convert(err error) ExtError {
    switch err := err.(type) {
    case ExtError:
        return err
    case multierr.Error:
        return ExtError{
            Status: convertMultiErr(err),
            Err: err
        }
    default:
        return ExtError{Status: 500, Err: err}
    }
}
```

# combining external errors

```
func convertMultiErr(errs multierr.Error) int {
    if !allExtError(errs) {
        return 500
    } else if allEqualStatus(errs) {
        return errs[0].(ExtError).Status
    } else if allRangeStatus(errs, 400, 500) {
        return 400
    }
    return 500
}
```

# so many routers...

- github.com/julienschmidt/httprouter
- limited scope
- explicit matches
- builds radix tree (trie) for routes
- zero allocation (or low allocation) router
- 1232 lines of code

# finally lets talk middleware

- github.com/justinas/alice
- syntactic sugar
- Transforms alice.New(Func1, Func2, Func3).Then(App)
- to Func1(Func2(Func3(App)))
- 112 lines of code
- bonus points for clever name

# create router

```
func Create() http.Handler {
    router := httprouter.New()

    router.Handle("GET", "/", Response(Hello))

    return alice.New(
        RecoveryHandler,
        FrameHandler,
        RequestIDHandler).
        Then(router)
}
```

# create frame

```
func FrameHandler(h http.Handler) http.Handler {
    return http.HandlerFunc(func(
            w http.ResponseWriter,
            r *http.Request) {
        ctx := r.Context()
        ctx = frame.NewContext(ctx)
        r = r.WithContext(ctx)
        h.ServeHTTP(w, r)
    })
}
```

# populate frame

```
func RequestIDHandler(h http.Handler) http.Handler {
    return http.HandlerFunc(func(
            w http.ResponseWriter,
            r *http.Request) {
        ctx := r.Context()
        fr := frame.FromContext(ctx)
        id := uuid.New()
        fr.UUID = id
        fr.Logger = fr.Logger.With().
            Str("uuid", id.String()).
            Logger()
        h.ServeHTTP(w, r)
    })
}
```

# what's a response?

```
type respHandle func(*http.Request,
    httprouter.Params) (string, error)

func Hello(r *http.Request,
    p httprouter.Params) (string, error) {
    return "world", nil
}

func UserError(r *http.Request,
    p httprouter.Params) (string, error) {
    err := exterror.Create(http.StatusBadRequest,
        errors.New("user error"))
    return "", err
}
```

# response handler

```
func Response(handle respHandle) httprouter.Handle {
    return func(w http.ResponseWriter,
        r *http.Request, p httprouter.Params) {
        msg, err := handle(r, p)
        if err != nil {
            HandleError(w, r, err)
        } else {
            HandleResult(w, r, msg)
        }
    }
}
```

# error response handler

```
func HandleError(w http.ResponseWriter,
    r *http.Request, err error) {

    ctx := r.Context()
    fr := frame.FromContext(ctx)
    exterr := exterror.Convert(ctx, err)
    if exterr.Status < 500 {
        fr.Logger.Warn().
            Err(err).
            Int("status", exterr.Status).
            Msg("user error reported")
    } else {
        fr.Logger.Error().
            Err(err).
            Int("status", exterr.Status).
            Msg("server error reported")
    }
    ...
```

# own your own dependencies

- github.com/google/uuid 868
- github.com/go-stack/stack 400
- github.com/ianschenck/envflag 192
- github.com/jonbodner/multierr 63
- github.com/julienschmidt/httprouter 1232
- github.com/justinas/alice 112
- github.com/rs/zerolog 5055

# revisiting assumptions

- we optimized for lines of code
- proxy for understanding the entire application
- what is highest priority for your application?
- is it understanding?

# revisiting assumptions

- we optimized for lines of code
- proxy for understanding the entire application
- what is highest priority for your application?
- ~~is it understanding?~~
- delivering features, security, performance
- whatever works for you

# thank you

- please - beta testers needed at thoughtdealership.com
- slides at github.com/thoughtdealership/howto
- howto is CC0 licensed