Functional options

where user-friendly API begins

Mateusz Szostok
SAP Hybris

Agenda

- Define our problem
- Let's solve our problem!

Let's build an HTTP request wrapper to provide useful metrics

Let's build an HTTP request wrapper to provide useful metrics

You start with the simplest api

```
func NewMeteredHandler(u http.Handler, address string) (http.Handler, error) {
    svc := &MeteredHandler{
        underlying:    u,
        riemannClient: riemann.NewClient(address),
    }

    // register metrics
    err := reg.Register(MetricSuffixSucceeded, svc.succeededMetric)
    if err != nil {}
    // ....
    return svc, nil
}
```

Usage

```
rtr := mux.NewRouter()

meteredRtr, _ := riemann.NewMeteredHandler(rtr, "riemann:5555")

http.ListenAndServe(":8080", metredRtr)
```

Features, Features, Features

- "how to override default success or client failure status code?"
- "to determine if it's client failure I need to check response body, do you support that?"
- "can I change flush metrics frequency?"
- **.. and more, more, more...**

BREAKING CHANGE: Add possibility to change status codes

```
// NewMeteredHandler creates an http.Handler instance which produce metrics about status codes
// and execution time.
//
// u defines underlying http handler
// address defines address of riemann server
// successStatusCodes defines https success status codes
// clientFailureStatusCodes defines https client failure status codes
// serverFailureStatusCodes defines https server failure status codes
func NewMeteredHandler(u http.Handler, address string, successStatusCodes,
    clientFailureStatusCodes, serverFailureStatusCodes []int) (http.Handler, error) {
    // body
}
```

BREAKING CHANGE: Add possibility to change flush metrics frequency

```
// NewMeteredHandler creates an http.Handler instance which produce metrics about status codes
// and execution time.
//
// u defines underlying http handler
// address defines address of riemann server
// flushMetricFrequency defines how often metric will be flushed to Riemann server
// successStatusCodes defines https success status codes
// clientFailureStatusCodes defines https client failure status codes
// serverFailureStatusCodes defines https server failure status codes
func NewMeteredHandler(u http.Handler, address string, flushMetricFrequency time.Duration,
    succesStatusCodes, clientFailureStatusCodes,
    serverFailureStatusCodes []int) (http.Handler, error) {
    // body
}
```

That's the evilest thing I can imagine.

Solution 1: Use specialized functions, à la Server

```
// NewMeteredHandler creates an http.Handler instance which produce metrics about status codes
// and execution time.
func NewMeteredHandler(u http.Handler, address string) (http.Handler, error)

// NewMeteredHandlerWithFlushFrequency creates an http.Handler instance which
// produce metrics about status codes and execution time.
// Only given status code will be treated as success.
func NewMeteredHandlerWithSuccessStatusCodes(u http.Handler, address string,
    succesStatusCodes []int) (http.Handler, error)

// NewMeteredHandlerWithFlushFrequency creates an http.Handler instance which
// produce metrics about status codes and execution time
// which will be flushed to the server with given duration.
func NewMeteredHandlerWithFlushFrequency(u http.Handler, address string,
    flushMetricFrequency time.Duration) (http.Handler, error)
```

Props and cons of using specialized functions

props

- callers call dedicated function only with params which he is interested in

cons

- providing every possible permutation can quickly become overwhelming
- duplication of godoc

Solution 2: Use a configuration struct

```
// Config contains parameters to configure Riemann metered handler
type Config struct {
	// FlushMetricFrequency defines how often metric will be flushed to Riemann server
	FlushMetricFrequency time.Duration

	// SuccessStatusCodes defines https success status codes
	SuccessStatusCodes []int

	// ClientFailureStatusCodes defines https client failure status codes
	ClientFailureStatusCodes []int

	// ServerFailureStatusCodes defines https server failure status codes
	ServerFailureStatusCodes []int
}

func NewMeteredHandler(u http.Handler, address string, cfg Config) (http.Handler, error) {
    // body
}
```

Default values

```
func main() {
    metered, _ := riemann.NewMeteredHandler(rtr, "riemann:5555", Config{
        FlushMetricFrequency: 5*time.Second,
    })
}
```

Default values, problems

```
type Config struct {
    // Maximum allowed concurrent connections to the Riemann server
    // if unset defaults to 3
    MaxConcurrentConnection int
}
func main() {
    metered, _ := riemann.NewMeteredHandler(rtr, "riemann:5555", Config{
        MaxConcurrentConnection: 0, // oops
    })
}
```

"I just want a metered handler, I don't want to thing about it"

```
func main() {
    metered, _ := riemann.NewMeteredHandler(rtr, "riemann:5555", Config{})
}
```

Pass a pointer to the value instead

```
// Config contains parameters to configure Riemann metered handler
type Config struct {
    // Fields
}

func NewMeteredHandler(u http.Handler, address string, cfg *Config) (http.Handler, error) {
// body
}
```

Usage

```
func main() {
    metered, _ := riemann.NewMeteredHandler(rtr, "riemann:5555", nil)
}
```

Props and cons of using configuration struct

props

- new feature can be added over time, while the public API for creating a server itself remains unchanged.
- godoc on each variable instead of huge one comment
- potentially enables the callers to use the zero value to signify they they want the default

cons

- even if we do not want to change any of the configuration parameters, we still need to pass something
- allowed mutation after passing to constructor

You can't make everyone happy you're not a jar of Nutella.

Our Nutella: Functional options to the rescue!

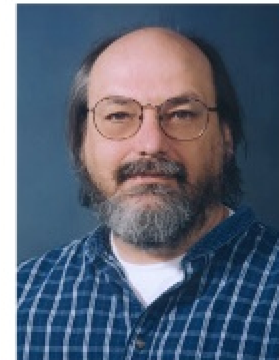
Designers

Robert Griesemer

V8 JavaScript engine, Java HotSpot VM

Rob Pike

UNIX, Plan 9, UTF-8



Ken Thompson

UNIX, Plan 9, B language, UTF-8

6

Functional option, how to

Define new type for functional options

```
// MeteredHandlerOption is a functional option type, which allows you to use optional configuration.
type MeteredHandlerOption func(*MeteredHandler) error
```

The function signature should include required fields like *address*, and everything else is an option

```
func NewMeteredHandler(u http.Handler, address string,
    options ...MeteredHandlerOption) (http.Handler, error)
```

New we can use it as before

```
meteredRtr, _ := riemann.NewMeteredHandler(rtr, "riemann:5555")
```

Functional option, how to

It's really easy to use defaults

```
func NewMeteredHandler(u http.Handler, address string,
    options ...MeteredHandlerOption) (http.Handler, error) {
    h := &MeteredHandler{
        underlying:             u,
        riemannClient:          riemann.NewClient(address),
        successPredicate:       defaultSuccessPredicate,
        clientFailurePredicate: defaultClientFailurePredicate,
        serverFailurePredicate: defaultServerFailurePredicate,
        flushMetricFrequency:   time.Second,
    }

    // apply each of the options individually in the order the user supplies them
    for _, option := range options {
        if err := option(h); err != nil {
            return nil, err
        }
    }
    return h, nil
}
```

Override default success status code (Non-breaking change)

```
// MeteredHandlerOption is a functional option type, which allows you to use optional configuration.
type MeteredHandlerOption func(*MeteredHandler) error

// WithSuccessStatusCodes is a functional option,
// which modifies success status code selection predicate.
func WithSuccessStatusCodes(statusCodes ...int) MeteredHandlerOption {
    return func(h *MeteredHandler) {
        h.successPredicate = statusCodes
    }
}
```

Usage

```
NewMeteredHandler(rtr, "riem:5555", WithSuccessStatusCodes(http.StatusOK, http.StatusNoContent))
```

Override flush metrics frequency (Non-breaking change)

```
// FlushMetricFrequency is a functional option, which modifies flush metric frequency.
func WithFlushMetricFrequency(d time.Duration) MeteredHandlerOption {
    return func(h *MeteredHandler) {
        h.flushMetricFrequency = d
    }
}
```

Usage

```
NewMeteredHandler(rtr, "riemann:5555",
    WithSuccessStatusCodes(http.StatusOK, http.StatusNoContent),
    WithFlushMetricFrequency(60*time.Second),
)
```

Unlimited combinations

```
NewMeteredHandler(rtr, "riemann:5555",
    WithClientFailureStatusCodes(http.StatusBadRequest),
    WithSuccessStatusCodes(http.StatusOK, http.StatusNoContent),
    WithFlushMetricFrequency(60*time.Second),
)
```

Who is using functional options?

gRPC

```
type DialOption func(*dialOptions)
  func WithAuthority(a string) DialOption
  func WithStatsHandler(h stats.Handler) DialOption

type ServerOption func(*options)
  func Creds(c credentials.TransportCredentials) ServerOption
  func StreamInterceptor(i StreamServerInterceptor) ServerOption
```

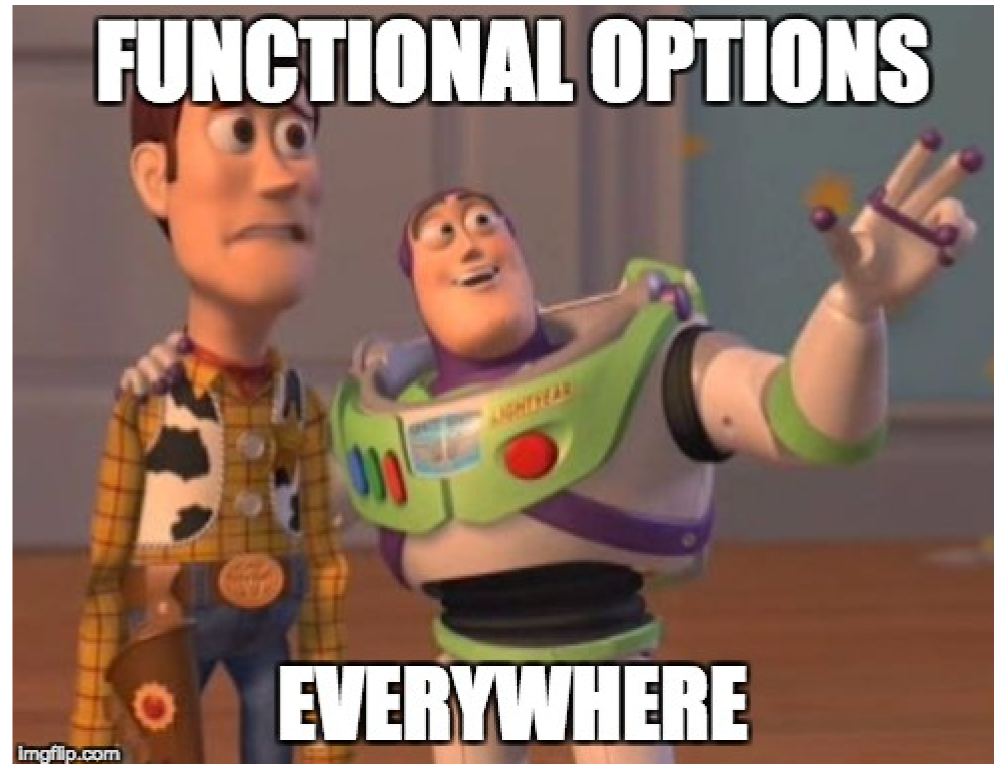
github.com/grpc/grpc-go (https://github.com/grpc/grpc-go)

nats-io

```
type Option func(*Options) error
  func Name(name string) Option
  func Secure(tls ...*tls.Config) Option
  func RootCAs(file ...string) Option
```

github.com/nats-io/go-nats (https://github.com/nats-io/go-nats)

Mistakes are a inevitable consequence of doing something new



Well know over engineering

```
// ServiceOption is a signature for function option
type ServiceOption func(*expiratorService)

// WithClock provides ability for setting clock function
func WithClock(clock clock.Clock) ServiceOption {
    return func(s *expiratorService) {
        s.clock = clock
    }
}

func NewService(cfg Config, log *logrus.Entry, expirator Expirator,
                options ...ServiceOption) (service.StartableService, error) {
    expSvc :=  &expiratorService{
        // init fields
    }

    for _, option := range options {
        if err := option(expSvc); err != nil {
            return nil, err
        }
    }
}
```

Used only in test

```
expSvc := expirator.NewService(expirator.Config{}, logger.New(&logger.Config{}), expiratorMock,
            expirator.WithClock(clockMock))
```

Better is to use export test.go pattern

```
// export_test.go

      package expirator

      func (e *expirator) SetClock(clock clock.Clock) {
          e.clock = clock
      }
```

Thank you

Mateusz Szostok
SAP Hybris
mateusz.szostok@sap.com (mailto:mateusz.szostok@sap.com)