Go kit

Peter Bourgon
Gopher

The story

The modern enterprise

A company that is

- Tech-oriented
- Consumer-focused
- Successful — exponential growth
- 50–1000 engineers
- Microservice architecture

Modern enterprises

- Google — too big
- Amazon — too big
- Twitter
- Netflix
- Spotify
- SoundCloud
- Etsy — monolith

Maturity

We're getting there.

Library support

A lack of higher-order library support held Go back at SoundCloud.

Modern enterprises **need** a set of code, tools, idioms, and best practices on which to build their business logic.

Enter Go kit

- Higher-order abstractions for microservices
- Like Go, strong idioms and opinions
- But not *too* opinionated — you don't need to reinvent your infrastructure

Similar to...

- Twitter's **Finagle**
- Netflix's **Ribbon** (and friends)

Goals

Goal: make Go a first-class citizen at the application layer

- Border — HAProxy, nginx...
- Application — **here**
- Data — Cassandra, PostgreSQL, Redis, Elasticsearch, Riak...

Where your business logic lives.

Goal: (micro)service architecture

Go kit is focused on things that are important to microservices.

Our target market is overwhelmingly adopting this architecture.

Goal: RPC messaging pattern

Just to start — we're not fundamentally opposed to other patterns.

Focus on the 80% case, to get to a useful state.

Goal: operate in a mixed environment

We don't expect to have complete buy-in at your organization.

Go kit services must play nice with existing services.

Goal: pluggable transports

Your business logic should be decoupled from how you access it.

One service should be available over many transports — even in the same process.

- HTTP/JSON
- Thrift
- Protobuf
- net/rpc
- gRPC
- ...

(Remember, we don't want you to reinvent your infrastructure.)

Goal: strongly encourage best practices

Go kit contributors have experience in large-scale infrastructures.

We've made mistakes that you don't have to repeat.

This can be a big selling point to organizations on the fence.

Non-goals

Non-goal: require any specific infrastructure

Go kit won't have any infrastructure dependencies.

We'll work well in any scale environment, and grow with your organization.

Non-goal: opinions about orchestration

Go kit services can be deployed, run, and supervised however your organization does it.

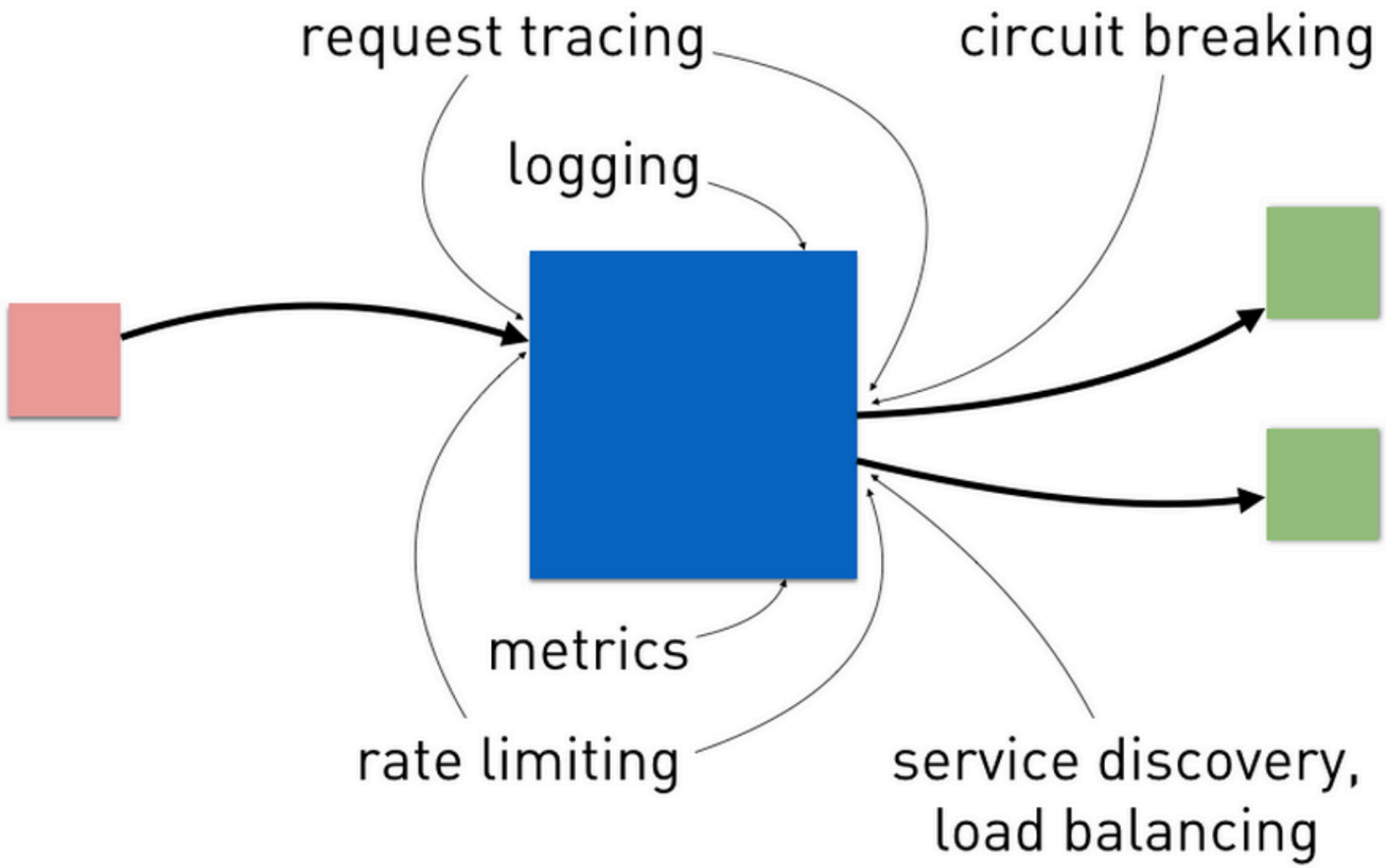
Non-goal: opinions about configuration

Go kit won't mandate a specific mechanism of build or runtime configuration.

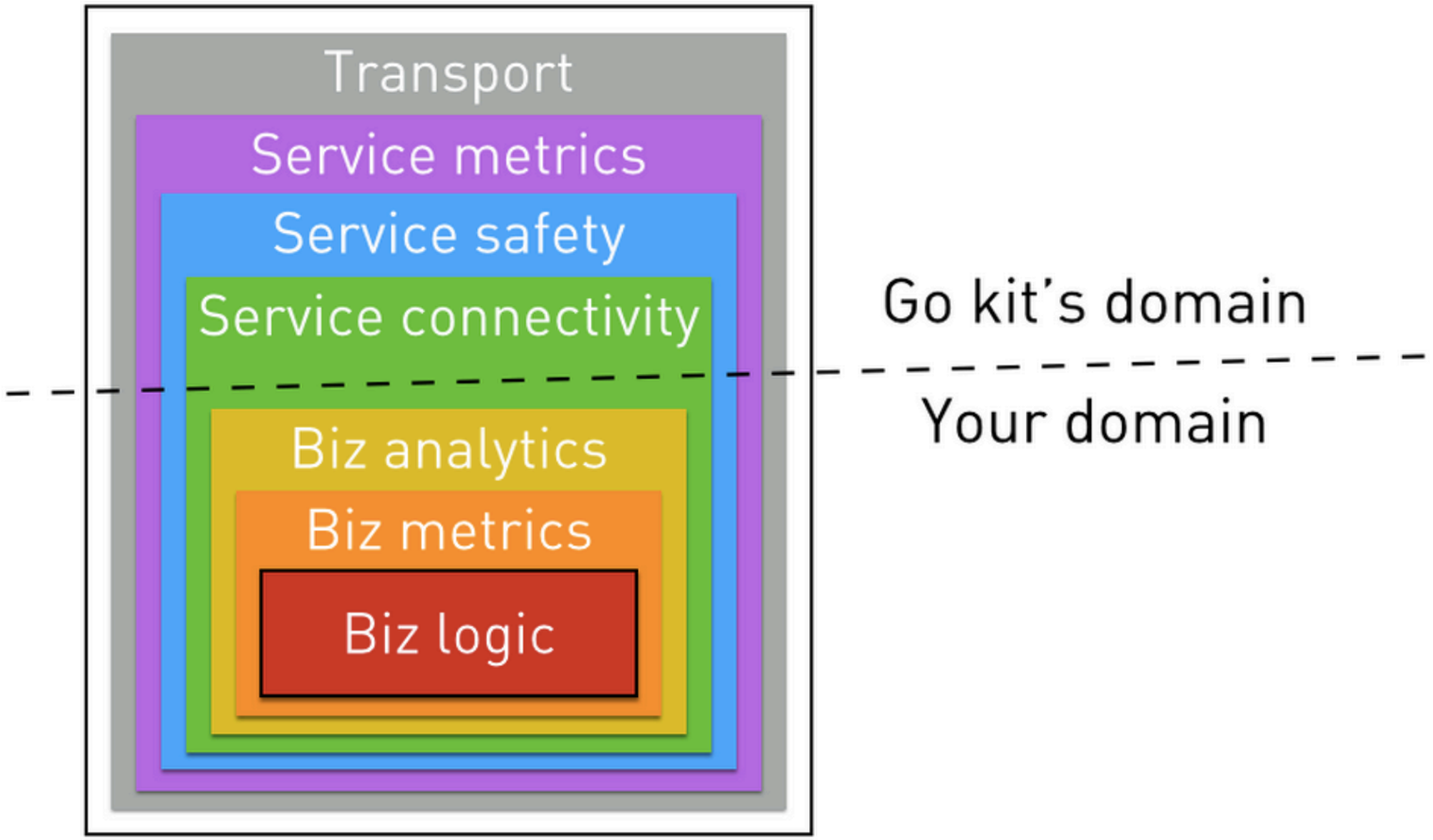
We'll work with whatever your organization prefers: flags, env vars, conf files — all OK.

Components

A service


request tracing
circuit breaking
logging
metrics
rate limiting
service discovery,
load balancing

A service


Transport
Service metrics
Service safety
Service connectivity
Biz analytics
Biz metrics
Biz logic
Go kit's domain
Your domain

package endpoint

```
// Endpoint is the fundamental building block of servers and clients.
// It represents a single RPC method.
type Endpoint func(ctx context.Context, request interface{}) (response interface{}, err error)

// Middleware is a chainable behavior modifier for endpoints.
type Middleware func(Endpoint) Endpoint
```

Implement your service as a collection of endpoints.

Transport

Most bindings can be made directly from your service to the transport library.

```
type grpcBinding struct{ endpoint.Endpoint }

func (b grpcBinding) MyMethod(ctx context.Context, req *pb.MyRequest) (*pb.MyReply, error) {
    reply, err := b.Endpoint(ctx, convertRequest(req))
    return convertReply(reply), err
}
```

package log

```
type Logger interface {
    Log(keyvals ...interface{}) error
}
```

A hard-won lesson: structured logging is mandatory.

But, equally useful for both application logging and log-structured data pipelines.

Still iterating on the best API — big thanks to Chris Hines.

package metrics

```
type Counter interface {
    With(Field) Counter
    Add(delta uint64)
}

type Gauge interface {
    With(Field) Gauge
    Set(value float64)
    Add(delta float64)
}

type Histogram interface {
    With(Field) Histogram
    Observe(value int64)
}
```

Thanks to Prometheus & Coda Hale.

package ratelimit

```
func NewTokenBucketLimiter(tb *ratelimit.Bucket) endpoint.Middleware {
    return func(next endpoint.Endpoint) endpoint.Endpoint {
        return func(ctx context.Context, request interface{}) (interface{}, error) {
            if tb.TakeAvailable(1) == 0 {
                return nil, ErrLimited
            }
            return next(ctx, request)
        }
    }
}
```

Thanks to Roger Peppe and Tomás Senart.

package circuitbreaker

```
func HandyBreaker(cb breaker.Breaker) endpoint.Middleware {
    return func(next endpoint.Endpoint) endpoint.Endpoint {
        return func(ctx context.Context, request interface{}) (response interface{}, err error) {
            if !cb.Allow() {
                return nil, breaker.ErrCircuitOpen
            }

            defer func(begin time.Time) {
                if err == nil {
                    cb.Success(time.Since(begin))
                } else {
                    cb.Failure(time.Since(begin))
                }
            }(time.Now())

            response, err = next(ctx, request)
            return
        }
    }
}
```

An important safety feature for all microservices.

package loadbalancer

```
// Publisher is backed by a data source: DNS SRV, Consul, etcd...
type Publisher interface {
    Subscribe(chan<- []endpoint.Endpoint)
    Unsubscribe(chan<- []endpoint.Endpoint)
    Stop()
}

// LoadBalancer is implemented by strategies: random, round-robin...
type LoadBalancer interface {
    Get() (endpoint.Endpoint, error)
}

func Retry(max int, timeout time.Duration, lb LoadBalancer) endpoint.Endpoint {
    return func(ctx context.Context, request interface{}) (interface{}, error) {
        // Get and try endpoints from the load balancer until you succeed,
        // reach max attempts, or exceed the timeout -- whichever comes first.
    }
}
```

Publisher implementation for DNS SRV. Consul and etcd support is planned. (Others?)

package tracing

```
func AnnotateServer(newSpan NewSpanFunc, c Collector) endpoint.Middleware {
    return func(next endpoint.Endpoint) endpoint.Endpoint {
        return func(ctx context.Context, request interface{}) (interface{}, error) {
            span, ok := fromContext(ctx)
            if !ok {
                span = newSpan(newID(), newID(), 0)
                ctx = context.WithValue(ctx, SpanContextKey, span)
            }
            span.Annotate(ServerReceive)
            defer func() { span.Annotate(ServerSend); c.Collect(span) }()
            return next(ctx, request)
        }
    }
}

func AnnotateClient(newSpan NewSpanFunc, c Collector) endpoint.Middleware {
    // ...
}
```

Currently just Zipkin. Appdash support is planned. (HTrace, Salp, others?)

An example service

addsvc

Let's define a one-method service.

```
type addService interface {
    Add(int, int) int
}
```

It can be simplified to just a function type.

```
type addFunc func(int, int) int
```

And a simple implementation.

```
func pureAdd(a, b int) int { return a + b }
```

addRequest, addResponse

To fit genearlized RPC semantics, we'll define request and response types.

```
type addRequest struct{ A, B int }
```

```
type addResponse struct{ V int }
```

makeEndpoint

The signature of the Add function is in our business domain. Let's write an adapter to lift it into the endpoint domain.

```
func makeEndpoint(f addFunc) endpoint.Endpoint {
    return func(ctx context.Context, request interface{}) (interface{}, error) {
        responses := make(chan interface{}, 1)
        go func() {
            req := request.(addRequest)
            responses <- addResponse{V: f(req.A, req.B)}
        }()
        select {
        case <-ctx.Done():
            return nil, context.DeadlineExceeded
        case response := <-responses:
            return response, nil
        }
    }
}
```

Binding

We'll need to bind our endpoint to one or more transports. Let's pick gRPC.

```
type grpcBinding struct{ endpoint.Endpoint }

func (b grpcBinding) Add(ctx context.Context, req *pb.AddRequest) (*pb.AddReply, error) {
    convertedRequest := addRequest{
        A: int(req.A),
        B: int(req.B),
    }

    resp, err := b.Endpoint(ctx, convertedRequest)
    if err != nil {
        return nil, err
    }

    v := int64(resp.(addResponse).V)
    return &pb.AddReply{V: v}, nil
}
```

Binding

Note: this requires a Protobuf definition.

```
syntax = "proto3";

package pb;

service Add {
  rpc Add (AddRequest) returns (AddReply) {}
}

message AddRequest {
  int64 a = 1;
  int64 b = 2;
}

message AddReply {
  int64 v = 1;
}
```

Wire it up

Wrap an addFunc with an endpoint, and an endpoint with a binding, and then expose the binding according to the rules of your transport.

```
var (
    function = pureAdd
    endpoint = makeEndpoint(function)
    binding  = grpcBinding{endpoint}
)

s := grpc.NewServer()
pb.RegisterAddServer(s, binding)
ln, _ := net.Listen("tcp", ":8001") // I'm so sorry for ignoring the error :(
s.Serve(ln)
```

(This will be slightly different for every transport.)

Logging

We want to log our requests from the perspective of our business domain.

We could implement it as an **endpoint** middleware...

```
type Middleware func(Endpoint) Endpoint
```

...but then we'd need type assertions to get our parameters and return values.

Alternately, we can define a **business** middleware.

```
type addMiddleware func(addFunc) addFunc
```

Logging

Now we can implement logging as a composable middleware.

```
func logging(logger log.Logger) addMiddleware {
    return func(next addFunc) addFunc {
        return func(a, b int) (v int) {

            defer func(begin time.Time) {
                logger.Log("a", a, "b", b, "v", v, "took", time.Since(begin))
            }(time.Now())

            v = next(a, b)
            return

        }
    }
}
```

Declarative composition

Compose business middlewares around the addFunc by using well-understood chaining techniques.

```
var a addFunc = pureAdd
a = logging(logger)(a)
a = instrument(requests, durations)(a)
```

Compose endpoint middlewares the same way.

```
var e endpoint.Endpoint = makeEndpoint(a)
e = ratelimit.NewTokenBucketLimiter(rl.NewBucketWithRate(100, 100))(e)
```

This **declarative composition** is easy to read and reason about.

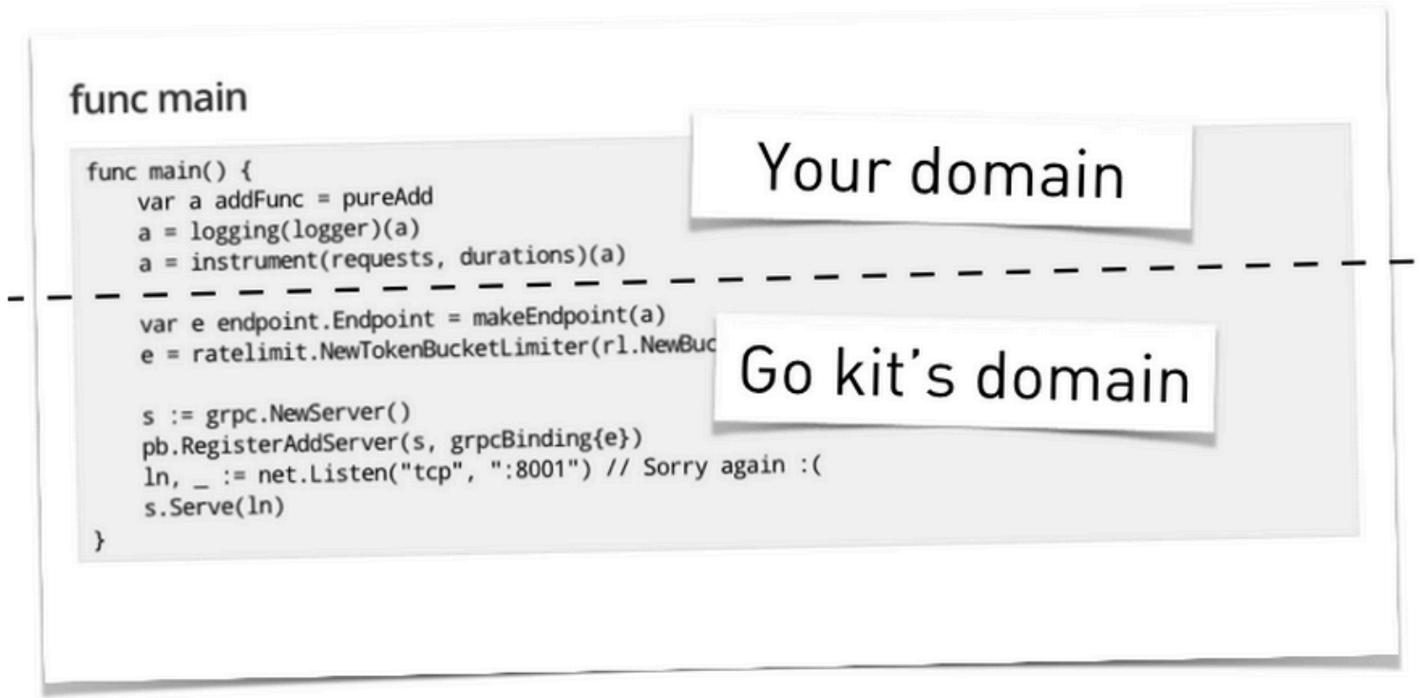
func main

```
func main() {
    var a addFunc = pureAdd
    a = logging(logger)(a)
    a = instrument(requests, durations)(a)

    var e endpoint.Endpoint = makeEndpoint(a)
    e = ratelimit.NewTokenBucketLimiter(rl.NewBucketWithRate(100, 100))(e)

    s := grpc.NewServer()
    pb.RegisterAddServer(s, grpcBinding{e})
    ln, _ := net.Listen("tcp", ":8001") // Sorry again :(
    s.Serve(ln)
}
```

Slide-ception



Clients

An endpoint can be used in both servers and clients.

```
func makeClient(cc *grpc.ClientConn) endpoint.Endpoint {
    client := pb.NewAddClient(cc)
    return func(ctx context.Context, request interface{}) (interface{}, error) {
        // Convert our request to a gRPC request
        // Invoke the gRPC client
        // Convert the gRPC reply to our response
    }
}
```

We can use the same set of endpoint middlewares.

```
    var e endpoint.Endpoint = makeClient(grpcDial("addsvc.datacenter.local"))
    e = circuitbreaker.Gobreaker(gobreaker.NewCircuitBreaker(gobreaker.Settings{}))(e)
    e = ratelimit.NewTokenBucketLimiter(rl.NewBucketWithRate(100, 100))(e)
```

Is this a good idea?

I dunno.

But it seems like a good place to start.

What's next

TODO

Near-term:

- package loadbalancer: Consul, etcd support
- package log: continue to refine the API
- package tracing: Appdash support
- package transport: Avro support
- package transport: comprehensive client support

Long-term:

- Tools to generate adapters and bindings
- Expand example coverage
- Support for messaging patterns beyond RPC

Your use cases

Go kit is community-driven, and benefits from everyone's experience.

If you've done this at your organization, I want to hear from you.

If you need something before you can start using Go kit, I want to hear from you!

The next level

Go is already becoming the language of the server.

I think Go can become the language of the modern enterprise.

Let's Go to the next level.

Thank you

Peter Bourgon
Gopher
@peterbourgon
http://gokit.io