Some things I keep repeating about Go

GopherConAU 2024

https://dave.cheney.net/

Hello!

Some things I keep repeating about Go

A talk in ~~five~~ four unequal parts

Names; variables, types, functions and packages

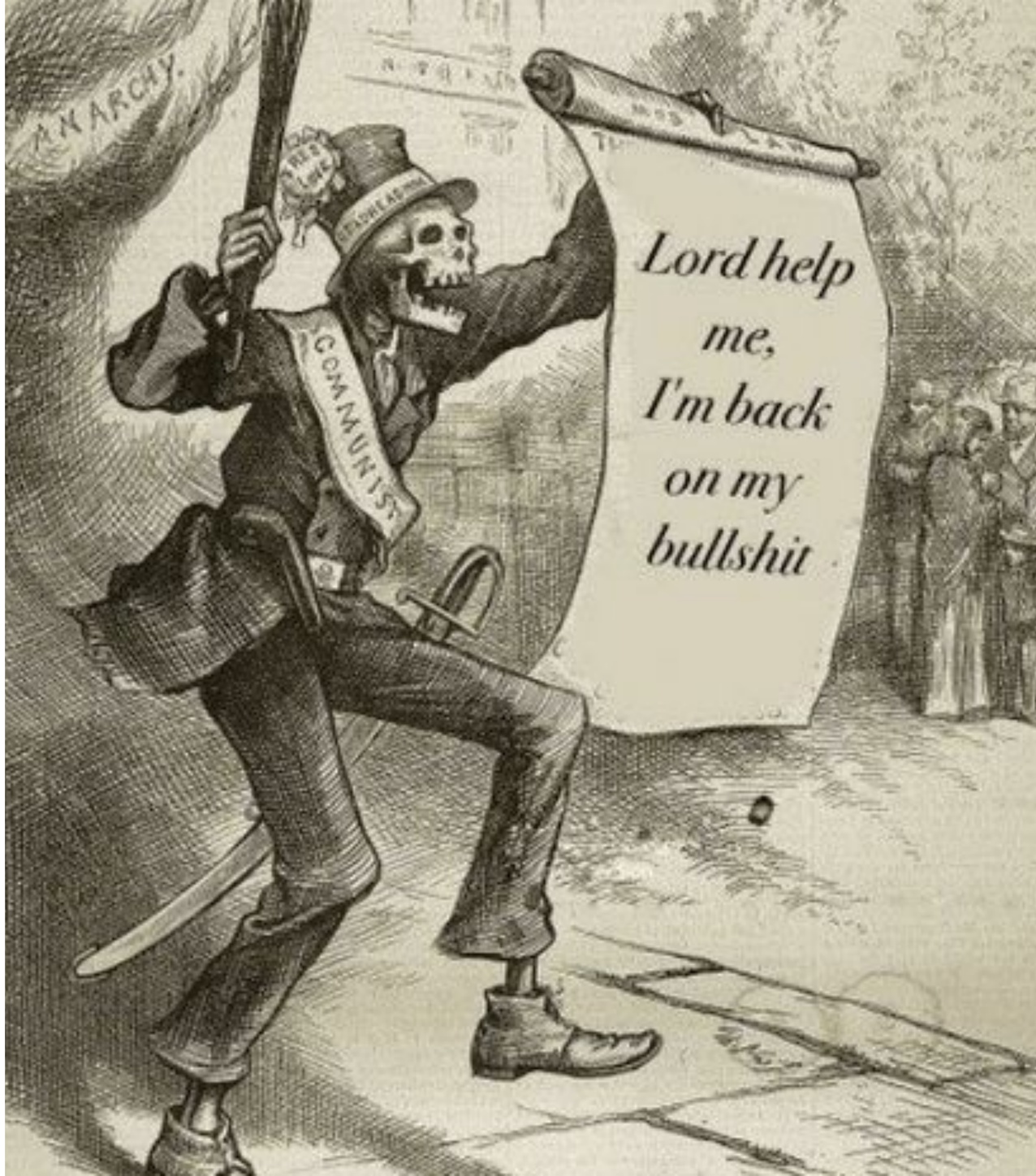
Initialisation; just once, thank you

Helpers; extending your vocabulary and being clearer about what your code is trying to say

Goroutine management; no goroutine left behind

Top tip: buy me a beer and ask me about the fifth topic

Names, yes we're talking about names



Long vs short?

What's the ideal length for a variable?

The greater the distance between a name's declaration and its uses, the longer the name should be.

Andrew Gerrand — https://go.dev/talks/2014/names.slide#4

```
func isSet(boolPtr *bool) bool {
    if boolPtr != nil {
        return *boolPtr
    }
    return false
}
```

```
func isSet(p *bool) bool {
    if p != nil {
        return *p
    }
    return false
}
```

```
func isSet(boolPtr *bool) bool {
    return boolPtr != nil && *boolPtr
}
```

Package names

A good name should describe what the package provides, not what it contains.

Short name is a good name?

```
io

strings

net/http

github.com/mycompany/whimisalmicroservice/api
```

Be consistent

```
package completion

type Request struct { … }

func (r *Request) String() string { … }

func (h *Handler) ServeHTTP(rw http.ResponseWriter, req http.Request) {
    creq, err := completion.NewRequest(req)
    if err != nil { … }
    …
}
```

```
package completion

type Handler struct { … }

func (h *Handler) Serve(req *Request) error { … }
```

```
package completion

type Request struct { … }

func (creq *Request) String() string { … }

func (h *Handler) ServeHTTP(rw http.ResponseWriter, req http.Request) {
    creq, err := completion.NewRequest(req)
    if err != nil { … }
    …
}

func (h *Handler) Serve(creq *Request) error { … }
```

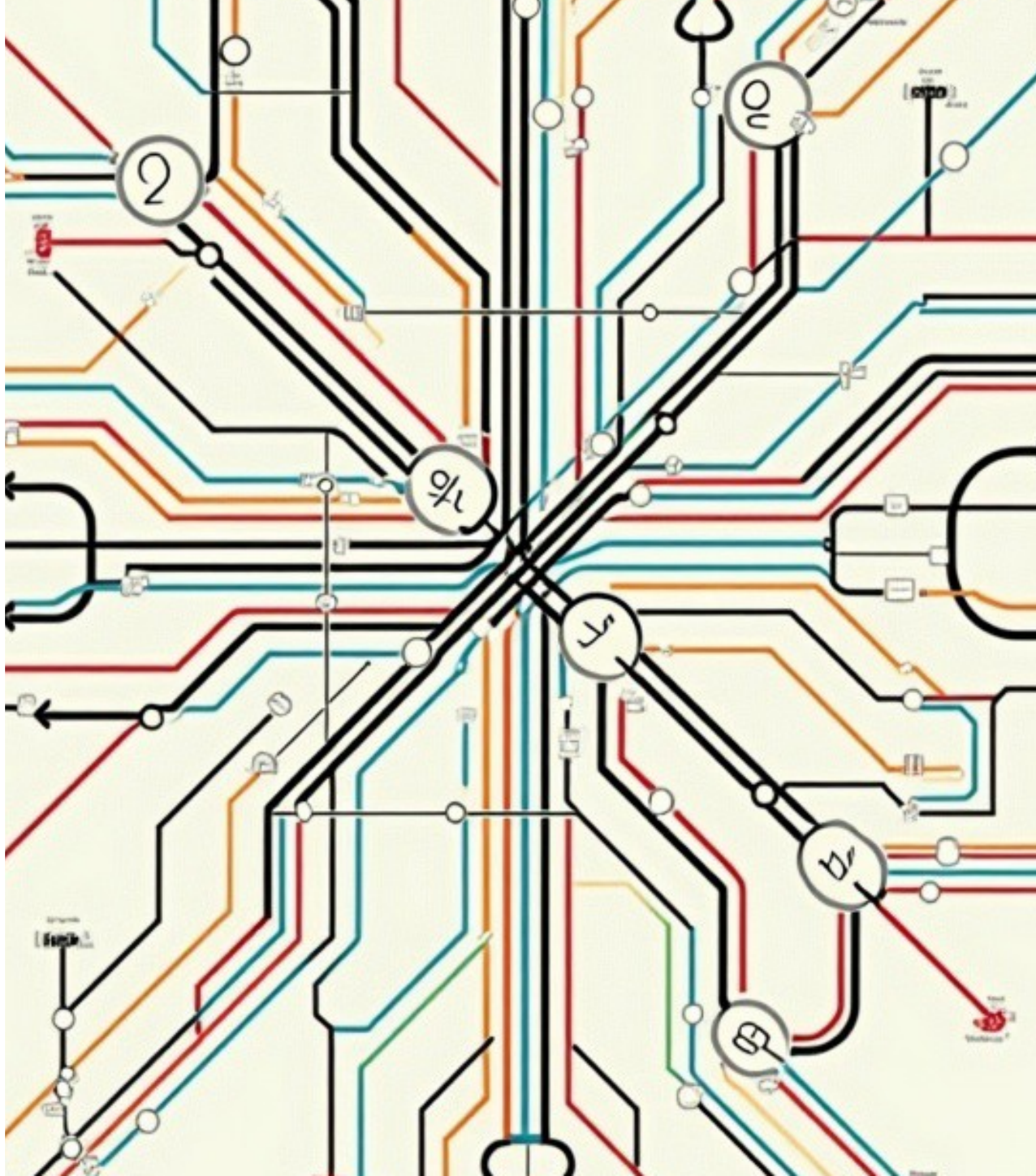
Function names are more important than comments

```
// ValidateChildren validates the children of *parent.
func ValidateChildren(parent *Node) []*Node { … }
```

Not everything needs a name

```
package completion

func NewRequest(r io.Reader) (*Request, error) {
    var v struct {
        Prompt string `json:"prompt"`
        Stop Stop `json:"stop"`
        …
    }
    err := json.NewDecoder(r).Decode(&v)
    if err != nil { … }
    var creq Request
    // validate v and populate creq
    return &creq, nil
}
```

Writing fewer conditionals for more readable code



if is bad, else is worse

Once, and only once

```
var thing Thing
if ctx.Environment == "production" {
    thing = &RealThing{ … }
} else {
    thing = &MockThing{ … }
}
```

```
var thing Thing = &MockThing{ … }
if etc.Environment == "production" {
    thing = &RealThing{ … }
}
```

```
func newThing(env string) Thing {
        switch env {
        case "production":
                return &RealThing{ … }
        default:
                return &MockThing{ … }
        }
}

thing := newThing(ctx.Environment)
```

```
thing := newThing(ctx.Environment)
```

thing is always initalised when it's declared.

newThing is now a free function. You can wrap a unit test around it easily.

newThing is now clearer about what it is doing; selecting an implementation based on the environment.

Keep to the left

```
func newThing(env string) Thing {
    if env != "production" {
        return &MockThing{ … }
    }
    return &RealThing{ … }
}
```


```
func newThing(env string) Thing {
    switch env {
    case "production":
        return &RealThing{ … }
    default:
        return &MockThing{ … }
    }
}
```

main.main calls main.run

```
package main

import (
        "fmt"
        "io"
        "os"
)

func main() {
    err := run(os.Args, os.Stdin, os.Stdout)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}

func run(args []string, stdin io.Reader, stdout io.Writer) error { … }
```

```
func run(args []string, stdin io.Reader, stdout io.Writer) error {
    log, err := setupLogger(stdout)
    if err != nil {
        return fmt.Errorf("failed to initalise logger: %w", err)
    }
    db, err := setupDB(stdin)
    if err != nil {
        return fmt.Errorf("failed to initalise db pool: %w", err)
    }
    svc, err := setupService(log, db, args)
    if err != nil {
        return fmt.Errorf("failed to initalise service: %w", err)
    }
    return svc.Serve()
}
```

Conditionals should naturally be true

```
if !creq.Metadata.Token.RestrictedTelemetry {
    // send headers to ask not to record
}
```

```
if creq.DenyTelemetry() {
    // send headers to ask not to record
}

func (creq *Request) DenyTelemetry() bool {
    return !creq.PermitTelemetry()
}

func (creq *Request) PermitTelemetry() bool {
    return creq.Metadata.Token.RestrictedTelemetry
}
```

```
if !creq.Stream {
    return fmt.Errorf("non-streaming mode not supported”)
}
```


```
if creq.Stream == false {
    return fmt.Errorf("non-streaming mode not supported”)
}
```

```
if !creq.StreamingMode() {
    return fmt.Errorf("non-streaming mode not supported”)
}
```


```
if creq.StreamingDisabled() {
    return fmt.Errorf("non-streaming mode not supported”)
}
```

Prefer > to >=
and friends

```
failure := resp.StatusCode >= 400
```


```
failure := resp.StatusCode > 399
```

Why eliminate such small syntactic details?

Seven, plus or minus two

Judge Dread holding a sign "7 plus or minus 2"

Put your logic into a helper

Helpers, helpers, helpers

Helpers give anonymous operations names

Helpers push conditionals down the stack, making them opaque to the caller and more testable for the author

ptr.To

```go
package ptr

// To returns a pointer to the given value.
func To[T any](v T) *T {
	return &v
}
```

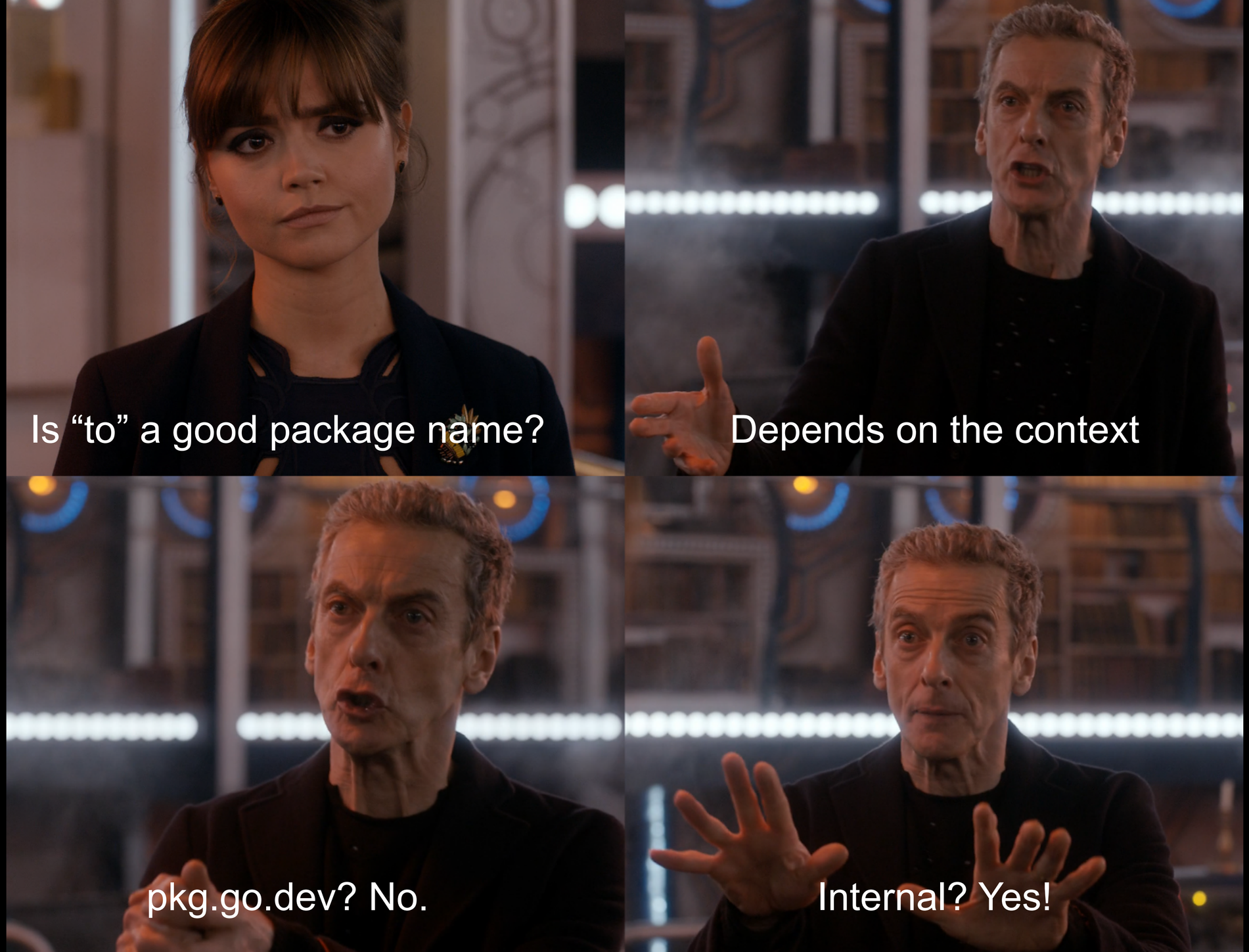
```
id := 9001
message := &SomeStruct{
    Id: &id,
    …
}
```


```
message := &SomeStruct{
    Id: ptr.To(9001),
    …
}
```

```
// package to contains functions for converting between types.
package to

import (
	"io"
	"net/http"
	"github.com/go-json-experiment/json"
	"github.com/go-json-experiment/json/jsontext"
)

// JSON writes the given object to the response body as JSON.
func JSON(rw http.ResponseWriter, obj any, opts ...func(w io.Writer) io.Writer) error {
	rw.Header().Set("Content-Type", "application/json; charset=utf-8")
	w := io.Writer(rw)
	for _, opt := range opts {
		w = opt(w)
	}
	return json.MarshalWrite(w, obj, jsontext.WithIndent("  "))
}
```

```
func NodeInfoIndex(rw http.ResponseWriter, req *http.Request) error {
    rw.Header().Set("cache-control", "max-age=259200, public")
    return to.JSON(rw, map[string]any{
        "links": []any{
            map[string]any{
                "rel":  "http://nodeinfo.diaspora.software/ns/schema/2.0",
                "href": fmt.Sprintf("https://%s/nodeinfo/2.0", r.Host),
            },
            map[string]any{
                "rel":  "http://nodeinfo.diaspora.software/ns/schema/2.1",
                "href": fmt.Sprintf("https://%s/nodeinfo/2.1", r.Host),
            },
        },
    })
}
```

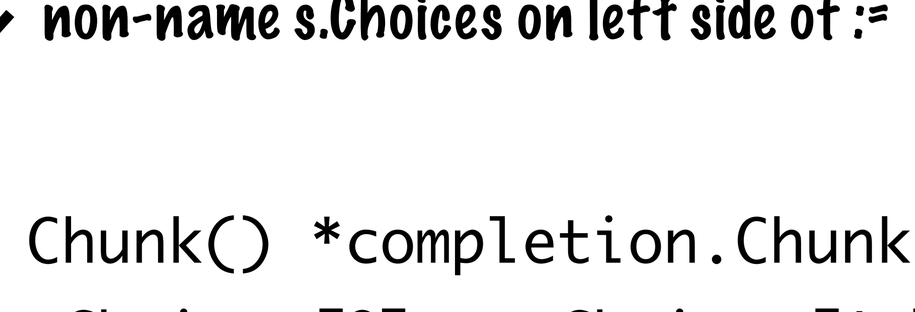
```
func NodeInfoIndex(rw http.ResponseWriter, req *http.Request) error {
    rw.Header().Set("cache-control", "max-age=259200, public")
    return json.NewEncoder(rw).Encode(map[string]any{
        "links": []any{
            map[string]any{
                "rel":  "http://nodeinfo.diaspora.software/ns/schema/2.0",
                "href": fmt.Sprintf("https://%s/nodeinfo/2.0", r.Host),
            },
            map[string]any{
                "rel":  "http://nodeinfo.diaspora.software/ns/schema/2.1",
                "href": fmt.Sprintf("https://%s/nodeinfo/2.1", r.Host),
            },
        },
    })
}
```


Is “to” a good package name?
Depends on the context
pkg.go.dev? No.
Internal? Yes!

Pushing and popping

```
func (s *streamIterator) Chunk() *completion.Chunk {
    choice := s.Choices[0]
    s.Choices = s.Choices[1:]
    return &completion.Chunk{
        ChunkMetadata: s.ChunkMetadata,
        ChoicePart:    choice,
    }
}
```

```
func (s *streamIterator) Chunk() *completion.Chunk {
    first, rest := s.Choices[0], s.Choices[1:]
    s.Choices = rest
    return &completion.Chunk{
        ChunkMetadata: s.ChunkMetadata,
        ChoicePart:    first,
    }
}
```




```
func (s *streamIterator) Chunk() *completion.Chunk {
    first, s.Choices := s.Choices[0], s.Choices[1:]
    return &completion.Chunk{
        ChunkMetadata: s.ChunkMetadata,
        ChoicePart:    first,
    }
}
```

```
// popFirst removes the first element from the slice
// and returns it.
func popFirst[T any](s *[]T) T {
    v := (*s)[0]
    *s = (*s)[1:]
    return v
}
```

```
func (s *streamIterator) Chunk() *completion.Chunk {
    return &completion.Chunk{
        ChunkMetadata: s.ChunkMetadata,
        ChoicePart:    popFirst(&s.Choices),
    }
}
```

```
type List[T any] []T

func (l *List[T]) PopFront() T {
    v := (*l)[0]
    *l = (*l)[1:]
    return v
}
```

```
func (s *streamIterator) Chunk() *completion.Chunk {
    return &completion.Chunk{
        ChunkMetadata: s.ChunkMetadata,
        ChoicePart:    s.Choices.PopFront(),
    }
}
```

```
s.Choices = append(s.Choices, choice)
```


```
func (l *List[T]) Push(v ...T) {
        *l = append(*l, v...)
}
```

```
s.Choices.Push(choice)
```

context.Canceled is viral

```
err := svc.Do(req.Context())
if errors.Is(err, context.Canceled) {
    return nil
}
return err
```

```
func ignoreContextCanceled(err error) error {
    if errors.Is(err, context.Canceled) {
        return nil
    }
    return err
}

err := svc.Do(req.Context())
return ignoreContextCanceled(err)
```

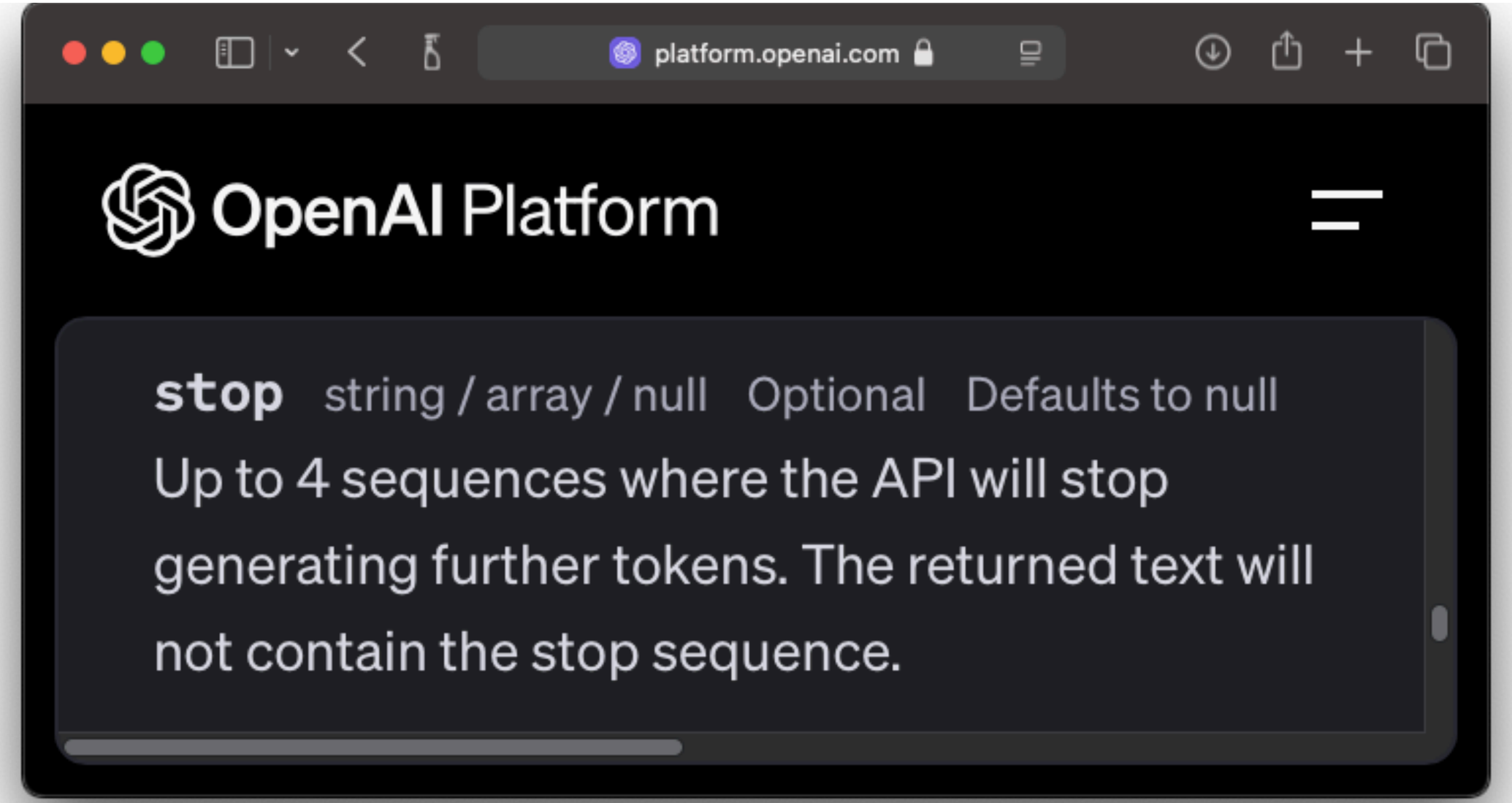
Keep debugging cruft out of your tests

```
require := require.New(t)

req := …
resp, err := http.DefaultClient.Do(req)
require.NoError(err)
defer resp.Body.Close()

require.Equal(http.StatusOK, resp.StatusCode,
     "unexpected status code: %v", resp.status)
// more tests
```

```
func dumpBody(r io.Reader) string {
    buf, _ := io.ReadAll(d.r)
    return string(buf)
}

resp, err := http.DefaultClient.Do(req)
require.NoError(err)
defer resp.Body.Close()

require.Equal(http.StatusOK, resp.StatusCode,
 "unexpected status code: %v %q", resp.Status, dumpBody(resp.Body))
```

```
type dumpBody struct {
    resp *http.Response
}

func (d dumpBody) String() string {
    buf, _ := io.ReadAll(d.resp.Body)
    return string(buf)
}

require.Equal(http.StatusOK, resp.StatusCode,
  "unexpected status code: %v %v", resp.Status, dumpBody{resp})
```

Dealing with unhelpful JSON


platform.openai.com
OpenAI Platform
stop string / array / null Optional Defaults to null
Up to 4 sequences where the API will stop generating further tokens. The returned text will not contain the stop sequence.

```
package openai

type Request struct {
    Stop any `json:"stop"`
    …
}
```

```
package openai

type Stop []string

func (s *Stop) UnmarshalJSON(b []byte) error { … }

type Request struct {
    Stop Stop `json:"stop"`
    …
}
```

```
// Stop is a []string that represents an OpenAI stop sequence.
// A stop sequence is a single string, or an array of up to 4 strings.
type Stop []string

func (s *Stop) UnmarshalJSON(b []byte) error {
    var stop any
    if err := json.Unmarshal(b, &stop); err != nil {
        return err
    }
    switch stop := stop.(type) {
    case string:
        *s = []string{stop}
        return nil
        case []interface{}:
           for i, v := range stop {
               if i > 3 {
                   return fmt.Errorf("stop must be an array of four or less strings but contained %d", len(stop))
               }
               st, ok := v.(string)
               if !ok {
                   return fmt.Errorf("stop array must contain only strings but contained %T", v)
               }
               *s = append(*s, st)
           }
        default:
            return fmt.Errorf("stop must be a string or an array of strings but was %T", stop)
    }
    if len(*s) == 0 {
         return errors.New("stop array must contain at least one string")
    }
    return nil
}
```

No goroutine left behind

```
// A group manages the lifetime of a set of goroutines from a common context.
// The first goroutine in the group to return will cause the context to be canceled,
// terminating the remaining goroutines.
type group struct {
    // ctx is the context passed to all goroutines in the group.
    ctx     context.Context
    cancel  context.CancelFunc
    done    sync.WaitGroup
    errOnce sync.Once
    err     error
}

// newGroup returns a new group using the given context.
func newGroup(ctx context.Context) *group {
    ctx, cancel := context.WithCancel(ctx)
    return &group{
        ctx:    ctx,
        cancel: cancel,
    }
}
```

```
package main

import (
    "context"
    "fmt"
    "os"
)

func main() {
    if err := run(); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}

func run() error {
    ctx := context.Background()
    g := newGroup(ctx)
    g.add(newHTTPService().Run)
    g.add(newRPCService().Run)
    g.add(newMetricsService().Run)
    return g.wait()
}
```

```
type Service struct { … }

func newService() *Service { return &Service{…} }

func (svc *Service) Run(ctx context.Context) error {
    if err := svc.start(ctx); err != nil {
        return fmt.Errorf("fail to start: %w", err)
    }
    // run returns when context is done or on error
    return svc.run(ctx)
}
```

```
func run() error {
    ctx := context.Background()
    // hook SIGTERM, SIGINT to gracefully shutdown the service.
    ctx, _ = signal.NotifyContext(ctx, syscall.SIGTERM, os.Interrupt)

    g := newGroup(ctx)
    g.add(newHTTPService().Run)
    g.add(newRPCService().Run)
    g.add(newMetricsService().Run)
    return g.wait()
}
```

Bringing it home

I do maintain that if your naming is wrong, your entire design is wrong.

Morrissey

Naming is everything

All things being equal, shorter is better than longer

Unique parts at the front, not the back

Name the receiver and the local var the same

Name packages after what they can do for you, not what they contain.

If is bad, else is worse

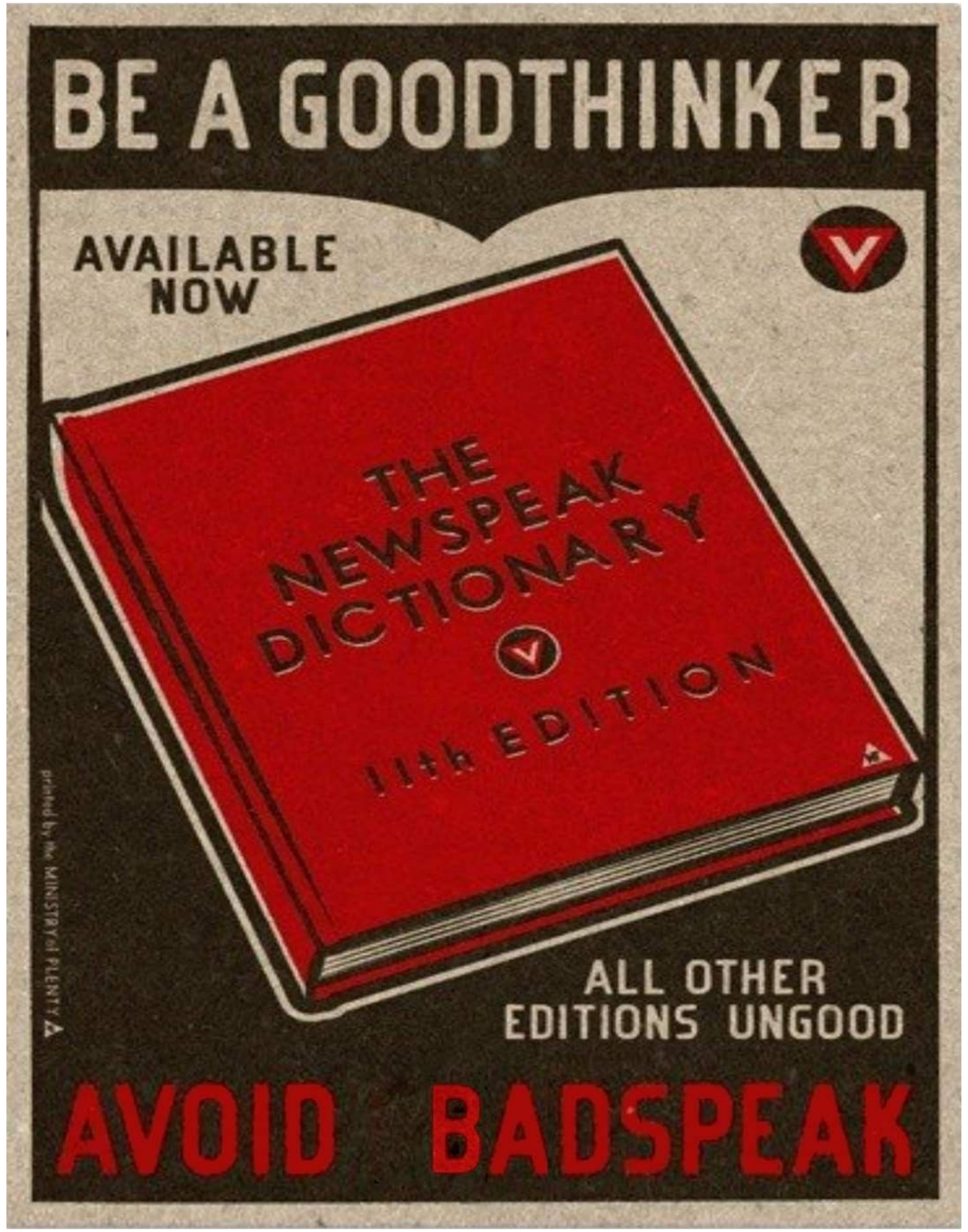
Initialise once and only once

Return early rather than else

Prefer switch for selection

Don’t repeat (your logic), instead write helpers

The language you use to describe a problem directly influences how you think about the problem



That's it!

Go write some code that makes you proud!

Slides