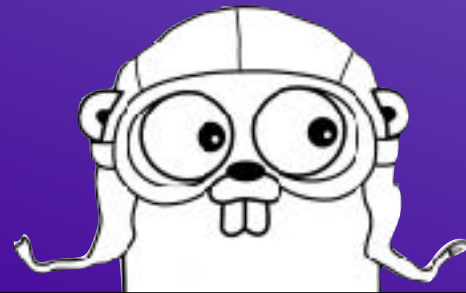


Go Anti-patterns



The name of the talk on the schedule is “Go Anti-Patterns”.

Idiomatic Go



But while writing this talk I realized that what I was after is a way to explore what Idiomatic Go is.

So I'll cover a few of the anti-patterns I've come across as a way to do that.

But first I should introduce myself.

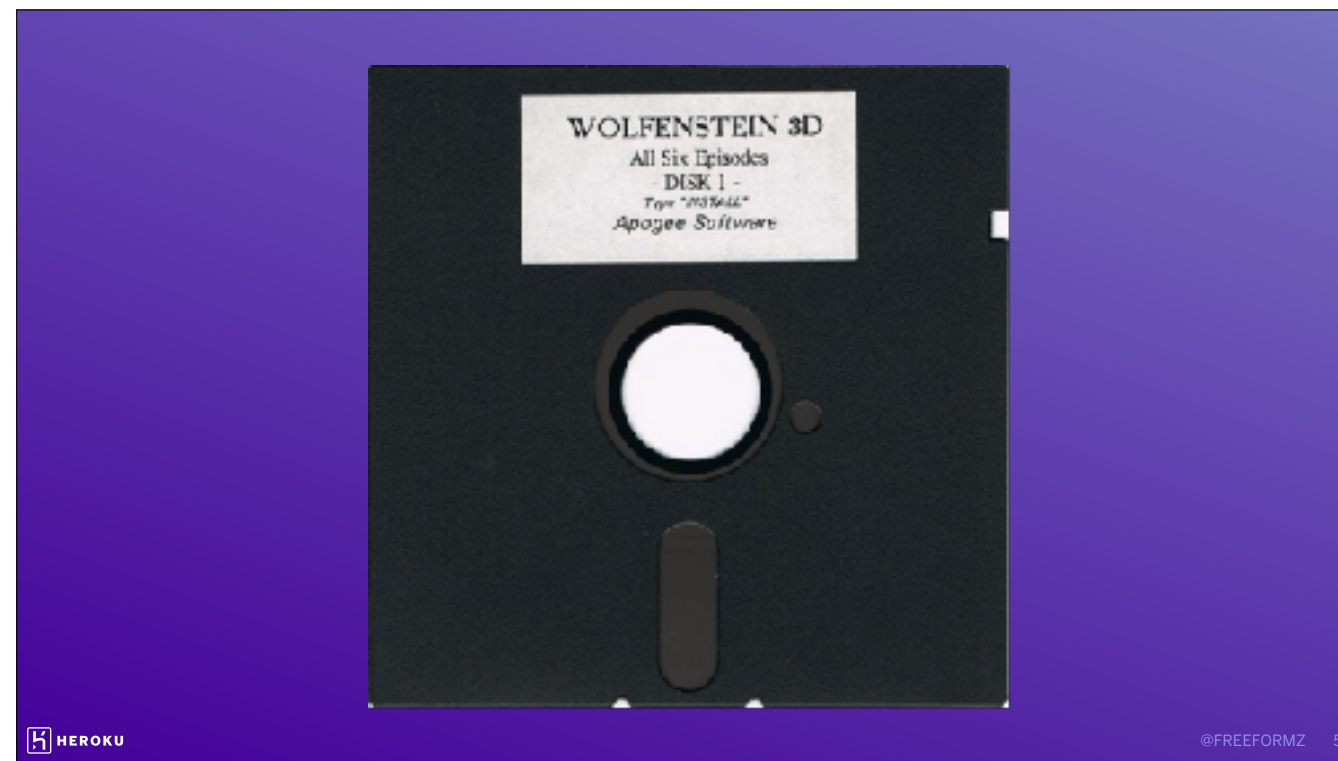


My name is Edward Muller.

Most places online I'm known as "freeformz".



I've been an engineer @ Heroku for the last 6 years.



But I've been writing software since shareware was distributed on 5.25" floppies.



In 2012 I wrote my first line of Go and pretty much fell in love.

If you Google these words: “Why I like Go”, a GitHub gist I wrote a few years back is the top result. I wrote that gist because, at the time, I was being asked this question a lot and decided to summarize my thoughts in a post I could share.

The gist lists some language features, but overall runtime, tooling and standard library features are more prominent:

Concurrency with goroutines, garbage collection, static binaries, memory safety, lack of a package system, built in profiling, fast compile times and others are called out.

These features drew me to Go and are often cited as why someone likes or should use Go for a project.



My advocacy for Go and the rise of Go internally, eventually led me to the role of “Go Language Owner” at Heroku.



What does a “language owner” do? We’re the engineers responsible for supporting customers using a given language.

To the best of our abilities we internally represent the community behind the language we support.

If you’ve deployed Go applications to Heroku, had a support ticket escalated to the “Go Team”, read a Heroku Devcenter article on some aspect of Go, or started with one of our Go sample programs, then you’ve interacted with some of my work and I hope you’ve had a good experience.

If you didn’t, please let me know after the talk.



So why do I care about idiomatic Go?

Language owners provide end user support. As a result I've had the opportunity to read, comment on, and help organize a lot of Go code, both internal and external.

When a customer has an issue and opens a ticket, I am the one reading their code to try to make sense of what is going on and what they are trying to do.

Go is also a primary language inside of Heroku. I am often asked to provide guidance and/or review code when a team adopts Go, or otherwise has questions.

That process is always easier, always faster and less painful for both parties when that code is idiomatic.

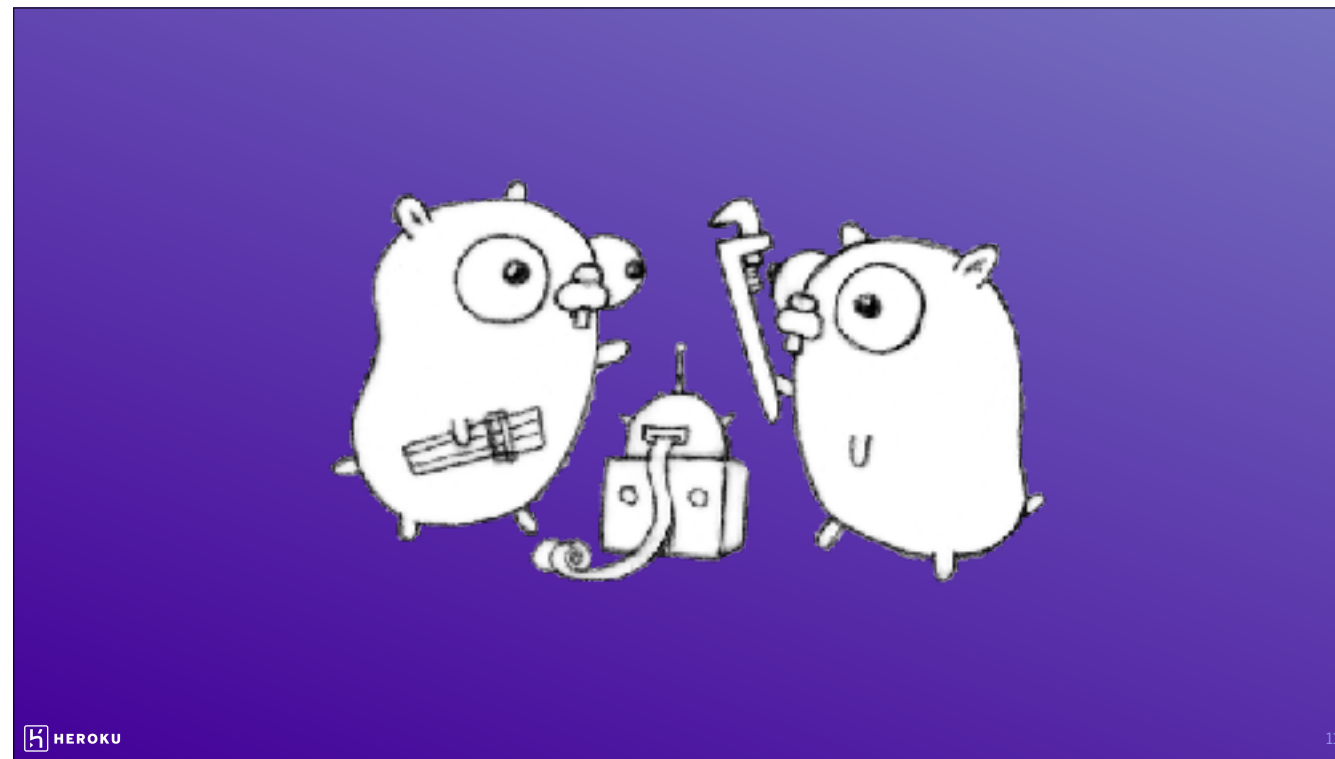


But a lot of that Go, just like the Go I started writing 5 years ago, and sometimes still write, isn't all that idiomatic.

I don't say this to chide anyone.

It's natural to bring the idioms you've learned in other languages to your new language. And Go is a newer language. There are fewer native Go speakers than there are for many older languages.

A lot of my initial Go looked more like Ruby got drunk and picked a fight with some channels and goroutines only to be locked in jail for the night.



Reading the Go code from this diverse group of software developers helped me appreciate the more subtle aspects of Go.

These aspects weren't listed in the gist I wrote over 4 years ago.

These aspects are reflected in what I am calling "Idiomatic Go".

Go's Design Principles



Orthogonality

Simplicity

Readability

<http://bit.ly/GoIntroVideo>

<http://bit.ly/GoLessIsMore>

<http://bit.ly/GoProverbs>



12

But what is idiomatic Go?

As best as I can tell, Go code that is idiomatic adheres to the design principles of the Go language:

orthogonality

simplicity

and readability

The Google Tech talk introducing Go in 2009, Rob Pike's blog post "less is exponentially more" and the Go proverbs all either directly talk about these design principles or reinforce them.

Let's cover these traits quickly.


#1 Orthogonality



#1 Orthogonality.

In programming terms Orthogonality means that pieces are independent from each other. Changes to one part, a type, package, program, etc, has minimal to no effect on other parts.

(the Toblerone is a “real life” example of Orthogonal Vectors. “math joke”)




```
package io

type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}

# How many types on my system implement io.Reader ?
$ guru implements /usr/local/go/src/io/io.go:#3309 | wc -l
    762

# io.Writer ?
$ guru implements /usr/local/go/src/io/io.go:#3800 | wc -l
    901
```

14

The go standard library contains many orthogonal pieces.

My favorite examples are the interfaces in the io package, especially io.Reader and io.Writer.

Each of these interfaces are implemented over 700 times on my system.

These implementations vary greatly from reading or writing local files, network connections, images, streaming logs and others.

A change to one has no effect on the others or the interface.

Said another way: The interfaces and their implementations are orthogonal.



#2 Simplicity, often referred to as reduced complexity

Go does this by removing “features” found in other programming languages

The Go language has a regular syntax and doesn’t need a symbol table to parse it.

Go has no classes;

methods can be added to any type.

It has no inheritance.

Interfaces are implicitly satisfied.

Types stand alone by themselves, they just are and have no hierarchy.

Methods aren’t special, they’re just functions.

Assignment is not an expression.

Go has no pointer arithmetic and every type has a zero value.

And you know what?



Go doesn't have Generics.



In essence, Go embraces less is more and promotes simplicity.

Go does several things to promote readability:

- There are no header files or forward declarations.
- Everything is declared exactly once, it is an error to redeclare a variable in the same block;
- It has few keywords, leading to a simple, unsurprising syntax;
- Type derivation and the declare and initialize construct reduce typing and help avoid stuttering.

Go Anti-Patterns

So, given these traits how do we know idiomatic Go when we see it?

For me, at least, it's always been easier to identify code smells, things that somehow aren't quite right.

I've identified a set of Go anti-patterns from smelling a lot of code.

As a way to spot design issues and help guide you to more idiomatic Go, let's cover some of these anti-patterns.

Anti-pattern: Tiny Package Syndrome

The tiny package syndrome anti-pattern usually appears with a directory structure that looks similar to this....

```
context/  
  cqlsession/  
    cqlsession.go  
  dao/  
    dao.go  
  api/  
    api.go  
  logtoken/  
    logtoken.go  
  metricsapi/  
    metricsapi.go  
  outlet/  
    outlet.go  
  producers/  
    producers.go  
  
context (cont...)/  
  requestid/  
    requestid.go  
  sinkctx/  
    sink.go  
  starttime/  
    starttime.go  
  time/  
    time.go  
  tx/  
    tx.go  
  user/  
    user.go  
  version/  
    version.go
```

Lots of packages, each usually containing a single file, especially missing tests.

In this particular example all of these files store and retrieve things from context.

They are all grouped by kind, under the notion of being related to context and are therefore located in a “context” directory.

```
package bar

import (
    "context"

    "github.com/foo/bar"
)

type key int

const barKey key = iota

func WithBar(ctx context.Context, t bar.Thing) context.Context {
    return context.WithValue(ctx, barKey, t)
}

func FromContext(ctx context.Context) (bar.Thing, ok bool) {
    b, ok := ctx.Value(barKey).(bar.Thing)
    return b, ok
}
```

This example is an archetype, so while I show context here, this isn't context specific. We'll cover context later.

The content of most of those files look like what is on this slide.

A key type and constants are defined along with one or more With<SomeType> and FromContext functions.

This context related package, and the original package often have the same name.

If not "C T X" or "context" are appended to the package name in order to avoid import conflicts.

```
context/  
  cqlsession.go  
  dao.go  
  api.go  
  logtoken.go  
  metricsapi.go  
  outlet.go  
  producers.go  
  requestid.go  
  sink.go  
  starttime.go  
  time.go  
  tx.go  
  user.go  
  version.go
```

A better way to do this would be to collapse the directory structure like this.

Making this a larger package named “context” containing a bunch of With<SomeType> and <SomeType>From functions.

```
package bar

type ctxKey int

const key ctxKey = iota

func WithThing(ctx context.Context, t Thing) context.Context {
    return context.WithValue(ctx, key, t)
}

func ThingFromContext(ctx context.Context) (Thing, ok bool) {
    t, ok := ctx.Value(key).(bar.Thing)
    return t, ok
}

// rest of the bar package
```

An even better way, and what I generally recommend, is to make the `With<SomeType>` and `<SomeType>FromContext` functions part of the original package, as seen here.

This keeps the complete API of `bar` located in a single package, not spread out across multiple.

This isn't the only way tiny package syndrome shows up.

So look for...


```
context/  
  cqlsession/  
    cqlsession.go  
  dao/  
    dao.go  
  api/  
    api.go  
  logtoken/  
    logtoken.go  
  metricsapi/  
    metricsapi.go  
  outlet/  
    outlet.go  
  producers/  
    producers.go
```

```
context (cont...)/  
  requestid/  
    requestid.go  
  sinkctx/  
    sink.go  
  starttime/  
    starttime.go  
  time/  
    time.go  
  tx/  
    tx.go  
  user/  
    user.go  
  version/  
    version.go
```

Lots of small packages.

```
context/  
  cqlsession/  
    cqlsession.go  
  dao/  
    dao.go  
  api/  
    api.go  
  logtoken/  
    logtoken.go  
  metricsapi/  
    metricsapi.go  
  outlet/  
    outlet.go  
  producers/  
    producers.go
```

```
context (cont...)/  
  requestid/  
    requestid.go  
  sinkctx/  
    sink.go  
  starttime/  
    starttime.go  
  time/  
    time.go  
  tx/  
    tx.go  
  user/  
    user.go  
  version/  
    version.go
```

Grouped by kind or otherwise organized in a by kind hierarchy.

```
context/  
  cqlsession/  
    cqlsession.go  
  dao/  
    dao.go  
  api/  
    api.go  
  logtoken/  
    logtoken.go  
  metricsapi/  
    metricsapi.go  
  outlet/  
    outlet.go  
  producers/  
    producers.go
```

```
context (cont...)/  
  requestid/  
    requestid.go  
  sinkctx/  
    sink.go  
  starttime/  
    starttime.go  
  time/  
    time.go  
  tx/  
    tx.go  
  user/  
    user.go  
  version/  
    version.go
```

With a small number of files per package. Especially if they don't have tests.

Anti-pattern: Premature Exportation

Premature exportation anti-pattern is closely related to the last one.

```
package foo

type Bit struct{}

func Bar(input string) string {
    var b Bit
    // ... do stuff with b
    return fmt.Sprintf("bits: %s", b)
}
```

Developers have a drive to taxnomize things and split them into the smallest possible pieces.

This segregation, this excessive over taxonification, in order to derive the smallest, DRYest piece of code can result in many small packages. This has a follow on effect: In order to make the package usable, most or all of the package's content needs to be exported.

This ends up becoming a pattern and suddenly every type, function, variable and constant in every package is being exported.

For instance, in this code, why is Bit exported if it's just used internally?

When I see this I often have to remind others that Go has no type or object hierarchy.

```
project/  
  cmd/  
    webserver/  
      main.go // imports pkg/foo and pkg/bar  
  pkg/  
    foo/  
      foo.go  
      ...  
    bar/  
      bar.go  
      ...  
  client/  
    client.go // doesn't import pkg/foo or pkg/bar  
  
otherproject/  
  main.go // imports project/pkg/foo
```

Another way this anti-pattern emerges is when packages that are internal to a project are exported wholesale.

In the project presented in this slide anyone who can access the project's source code can use `pkg/foo` and `pkg/bar`, but the developers of the project only intend for those packages to be used internally. They aren't expecting feature requests from outside, they don't discuss changes with others, etc.

But along comes `otherproject`, written by another team. And they see something in project's code that they think may be useful and start using it.

```
project/  
  cmd/  
    webserver/  
      main.go // imports internal/foo and internal/bar  
  internal/  
    foo/  
      foo.go  
      ...  
    bar/  
      bar.go  
      ...  
  client/  
    client.go // doesn't import internal/foo or internal/bar, but could  
  
otherproject/  
  main.go // can't import internal/foo or internal/bar
```

The developers of project can instead signal their intent that foo and bar are for not for external consumption and place the foo and bar packages in an internal directory.

Packages located in an internal directory can only be imported by internal's direct neighbors and their decedents.

Here the developers of outsideproject can't import foo or bar. They can either copy the code into otherproject or start a dialog with the maintainers of project about their needs with the intent of making some are all of the features they need publicly available.

Anti-pattern: Premature exportation

- Don't export types, variables, function and constants until there is a need to do so.
- The DRYest, smallest, most segmented packages lead to the need to export everything.
- Keep packages that are not meant for external use in an internal folder.

In summary...

Don't export types, variables, function and constants until there is a need to do so.

The DRYest, smallest, most segmented packages lead to the need to export everything.

Keep packages that are not meant for external consumption in an internal folder.

This relays the expectations the authors of those packages have, around who they expect to use the package to those who may have access to it.

Anti-pattern: package util

The package util anti-pattern.

```
package util

func GenerateRandomBytes(n int) ([]byte, error)
util.GenerateRandomBytes(10)

func GenerateRandomString(n int) (string, error)
util.GenerateRandomString(10)

func Cert(hostname string) (string, string, error)
util.Cert("foozle.com")
```

Here is an extraction from a util package I've recently run across.

The name util says nothing about the purpose of this package.

In Go package names have semantic meaning. The name provides the context the users of the package need to understand what the package provides. The name of a package should describe its purpose, not its contents. The only part of the package import path that matters is the right most name. That name is what is used when referencing the package's contents.

I see util packages often form when a project is doing a lot of group by kind and has some pieces left over. Where do they end up going? Into a util package of course.

Do you notice anything particular about each of these functions?

The all generating things.

```
package generate

func RandomBytes(n int) ([]byte, error) { ... }
generate.RandomBytes(10)

func RandomString(n int) (string, error) { ... }
generate.RandomString(10)

func Cert(hostname string) (string, string, error) { ... }
generate.Cert("foozle.com")
```

Why not make a generate package instead?

Now there is `generate.RandomBytes`, `generate.RandomString` and `generate.Cert`

The name `generate` conveys the purpose of the package.

The function names specify the what.

Anti-pattern: `package util`

- Package names have semantic meaning
- Package names should describe the purpose of the package, not its contents.
- The only part of the package import path that matters is the right most name.
- `util` says nothing.

To summarize.

Package names have semantic meaning.

Package names should describe the purpose of the package, not it's contents.

The only part of the package import path that matters is the right most name.

`util` says nothing about the purpose of a package beyond a grouping of bits.

Anti-pattern: Config Structs

The config or options struct anti-pattern.

```
package main

import (
    "log"
    "os"

    "github.com/foo/bar/internal/cmd"
    "github.com/foo/bar/internal/db"
    "github.com/foo/bar/internal/doer"
)

func main() {
    f, err := os.Open(configFile)
    if err != nil {
        log.Fatalf("Error opening %q: %s", configFile, err)
    }
    c := cmd.NewConfig(f)
    db := db.NewConnection(c)
    d := doer.NewFoo(c, db)
    d.Do()
}
```

I have seen, and even written, code that essentially looks like this.

```
package main

import (
    "log"
    "os"

    "github.com/foo/bar/internal/cmd"
    "github.com/foo/bar/internal/db"
    "github.com/foo/bar/internal/doer"
)

func main() {
    f, err := os.Open(configFile)
    if err != nil {
        log.Fatalf("Error opening %q: %s", configFile, err)
    }
    c := cmd.NewConfig(f)
    db := db.NewConnection(c)
    d := doer.NewFoo(c, db)
    d.Do()
}
```

There is a cmd package that contains one or more configuration types and functions dedicated to parsing the environment or configuration files into values of those types.

```
c := cmd.NewConfig(f)

func NewConfig(rdr io.Reader) Config

type Config struct {
    MaxLineLength      int
    BackBuff           int
    BatchSize          int
    NumOutlets         int
    InputFormat        int
    MaxAttempts        int
    Prival             string
    Procid             string
    Hostname           string
    Appname            string
    Msgid              string
    // a total of 28 members
}
```

If we look at the value that the `cmd.NewConfig` function returns we'll see that it has a total of 28 members.


```
package main

import (
    "log"
    "os"

    "github.com/foo/bar/internal/config"
    "github.com/foo/bar/internal/db"
    "github.com/foo/bar/internal/doer"
)

func main() {
    f, err := os.Open(configFile)
    if err != nil {
        log.Fatalf("Error opening %q: %s", configFile, err)
    }
    c := config.Parse(f)
    db := db.NewConnection(c)
    d := doer.NewFoo(c, db)
    d.Do()
}
```

Back in our program we see that the Config struct value is blindly passed through to the other functions NewConnection and NewFoo.

Does db.NewConnection have any need for the InputFormat member of the config struct?

Does doer.NewFoo have any need for the BatchSize member of the config struct?

```
db := db.NewConnection(c)

func NewConnection(cfg cmd.Config) *sql.DB

d := doer.NewFoo(c, db)

func NewFoo(cfg cmd.Config, db *sql.DB) *Foo
```

Can we even determine, from the function signatures, what members of the Config struct are used by NewConnection and NewFoo?

Nope.

One way this anti-pattern develops is when a developer cargo cults the mantra that “package globals are bad”.

They decide to stick those globals into a config struct and pass values of that struct along, in it’s entirety everywhere those globals were used before.

```
type Options struct {  
    Time time.Duration  
    ID    string  
    Type  string  
    OtherID string  
    AnotherID string  
    Active bool  
}  
  
func (d *Doer) DoThing(opts Options) error {  
    r := d.thingRequest(http.DeleteMethod, urlForThing(opts))  
    return d.doRequest(r)  
}
```

Similarly this example shows an Options struct.

The code base this was extracted from contains several methods like DoThing, some simpler and some more complex.

All use a subset of the members of the Options struct. Additionally, there are multiple Option structs in this code base, each different, just in different packages.

Just like the Config example, it's hard to tell which members are used w/o pulling apart the functions that opts is passed to.

This usually develops as the complexity of a type or set of types increases and they require additional parameters. Over time the parameter list gets large.

Eventually too large and then, like in this slide, the parameters are condensed down into a single “options” parameter.

So what members of options are being used here?

```
type Options struct {  
    Time time.Duration  
    ID    string  
    Type  string  
    OtherID string  
    AnotherID string  
    Active bool  
}  
  
func (d *Doer) DoThing(opts Options) error {  
    r := d.thingRequest(http.DeleteMethod, urlForThing(opts))  
    return d.doRequest(r)  
}  
  
func urlForThings(opts Options) string {  
    return &url.URL{  
        Path: fmt.Sprintf("/foo/%s/bar/%s/bits/%s", opts.ID, opts.Type, opts.OtherID),  
    }.String()  
}
```

Let's look.

Here we see that ID, Type and OtherID are the 3 fields that are actually required. We had to dig down at least 2 levels to figure that out.

Picture the Option struct being imported and shared with other packages, especially as Doer grows more methods that require different options.

Imagine this pattern repeated over and over again in significantly more complex code.

What is the impact to readability?

To simplicity?

```
func (d *Doer) DoThing(ID, typ, otherID string) error {  
    r := d.thingRequest(http.MethodDelete, urlForThing(ID, typ, otherID))  
    return d.doRequest(r)  
}  
  
func urlForThing(foo, bar, bits string) string {  
    u := url.URL{  
        Path: fmt.Sprintf("/foo/%s/bar/%s/bits/%s", foo, bar, bits),  
    }  
    return u.String()  
}
```

Here is a better way to write this code.

DoThing now takes a discreet set of inputs, the ID, type and otherID of something and calculates the url of that thing using those inputs.

In urlForThings these are mapped to variables, whose names make more sense in the context of generating the url.

When reading the signature of DoThing, you know what it needs.

And, because we're not copying the whole Options struct, the amount of data being copied is reduced.

Anti-pattern: config structs

- Increases coupling
- Obfuscate the API of the functions or types that accept them.
- Hides complexity

To review...

Config structs couple the package containing it to every other package that needs the Config type. Coupling makes it easier to create non-orthogonal code.

They obfuscate the API of the functions or types that accept the config struct. Un-used config members create clutter.

It papers over the fact that the functions or types are complex. Possibly too complex, hiding the need for a redesign.

Anti-pattern: config structs

- Use only the values you need.

```
doer.NewThing(c.Foo, c.Bar, c.Baz)
```

- Functional options.

```
type func(t *Thing) Opt
func WithBar(b string) Opt {
    return func(t *Thing) {
        t.Bar = b
    }
}
```

<http://bit.ly/GoFunctionalConfig>

- Redesign.

So what what are the options?

If the function or type only needs a few values, pass only the values you need.

If you have too many values, try a functional configuration approach. This style is my favorite and is especially useful when the default or zero values need to be tweaked. It also provides a mechanism for users to write their own functional options. Dave Cheney's "Functional options for friendly APIs" talk and follow up blog post, are a great guide.

If that is still not enough, your functions and/or types may be doing too much and you should consider redesigning them to be more focused.



Pointer all the things.

No, please don't. I want you to consider what a pointer means.

A pointer doesn't automatically mean "it will be faster".

If you believe it will be faster, benchmark it to prove it before committing to it.

A pointer doesn't mean that less garbage will be created, although it can mean that less data is copied when a function is called.

Taking a pointer allocates a second value, the pointer value, in addition to the value being pointed to. If the pointer value "escapes" the function that created it, it may be moved to the heap, along with the value it points to.



Pointers are about ownership. When you pass a pointer to a function you are delegating ownership of the pointee to that function.

So I'm not saying don't use pointers.

I am saying that you should be aware of the fact that you are sharing and decide when it's the correct action or not.

Anti-pattern: Pointer All The Things

- Pointers are about ownership of data.
- Not necessarily faster.
- Additional overhead.
- Go proverb: Don't communicate by sharing memory, share memory by communicating.

In summary...

Pointers are about the ownership of data.

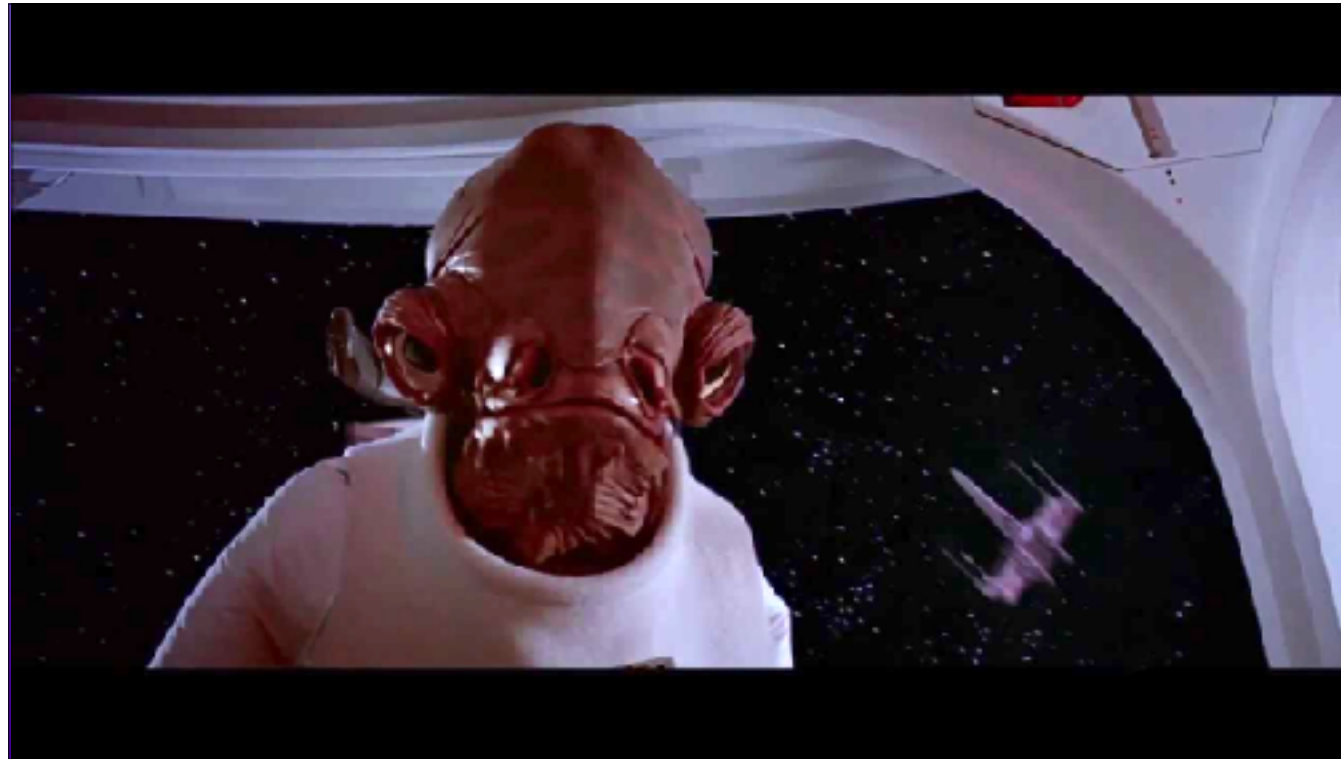
They aren't necessarily faster. Use benchmarks to prove that the additional overhead is worth it.

Remember one to the Go proverbs: Don't communicate by sharing memory, share memory by communicating.

Anti-pattern: `context.Value`

The `context.Value` anti-pattern.

Let me start by saying that `context.Value()`...



It's a trap. There are others ways to pass values along the request chain.

I won't cover those here, that's another talk for a different time.

```
package main

func handleSearch(w http.ResponseWriter, req *http.Request) {
    // ...
    userIP, _ := userip.FromRequest(req)
    ctx = userip.NewContext(ctx, userIP)
    results, err := google.Search(ctx, query)
    // ...
}

package google
func Search(ctx context.Context, query string) (Results, error) {
    // ...
    // If ctx is carrying the user IP address, forward it to the server.
    // Google APIs use the user IP to distinguish server-initiated requests
    // from end-user requests.
    if userIP, ok := userip.FromContext(ctx); ok {
        q.Set("userip", userIP.String())
    }
    // ...
}
```

This example code is pulled from the context blog post on [golang.org](https://blog.golang.org/context).

What is the problem here?

```
package main

func handleSearch(w http.ResponseWriter, req *http.Request) {
    // ...
    userIP, _ := userip.FromRequest(req)
    ctx = userip.NewContext(ctx, userIP)
    results, err := google.Search(ctx, query)
    // ...
}

package google
func Search(ctx context.Context, query string) (Results, error) {
    // ...
    // If ctx is carrying the user IP address, forward it to the server.
    // Google APIs use the user IP to distinguish server-initiated requests
    // from end-user requests.
    if userIP, ok := userip.FromContext(ctx); ok {
        q.Set("userip", userIP.String())
    }
    // ...
}
```

We've now created a side channel, undocumented api for `google.Search()`.

Even worse, this undocumented API avoids the type system by wrapping everything in the empty interface. We'll discuss the empty interface later.

This makes the code less clear, less understandable and the API less discoverable.

```
package main

func handleSearch(w http.ResponseWriter, req *http.Request) {
    // ...
    results, err := google.Search(ctx, query, userip.FromRequest(req))
    // ...
}

package google
func Search(ctx context.Context, query string, userIP net.IP) (Results, error) {...}
```

The client's IP address is needed for backend requests

Here is another take on that code. It is more in line with the explanation given in the context blog post that states... (click)

“The client's IP address is needed for backend requests”.

We see that google.Search now requires a net.IP, just like the blog said it needs to.

```
func handleSearch(w http.ResponseWriter, req *http.Request) {
    results, err := google.Search(ctx, query, userip.FromRequest(req))
}

func Search(ctx context.Context, query string, userIP net.IP) (Results, error) {
    err := httpDo(ctx, req, func(resp *http.Response, err error) error { ... })
}

func httpDo(ctx context.Context, req *http.Request, f func(*http.Response, error) error) error {
    c := make(chan error, 1)
    go func() { c <- f(client.Do(req)) }()
    select {
    case <-ctx.Done():
        <-c // Wait for f to return.
        return ctx.Err()
    case err := <-c:
        return err
    }
}
```

This is an expanded version of what was on the last slide.

Note that a context.Context is still a part of the Search function.

That's because the other properties of context are a very useful abstraction around signaling cancelation, as seen here in the httpDo function.

Anti-pattern: `context.Value`

- Use `Value()` / `WithValue()` sparingly, if at all.
- Use `WithTimeout()`, `WithDeadline()` & `WithCancel()`

In summary....

Use `Value()` and `WithValue()` sparingly. They create undocumented, side channel APIs.

If you still feel the need to use them, document the values that may be extracted from the context and the purpose for those values.

Use `WithTimeout()`, `WithDeadline()` and `WithCancel()` as they are a great abstraction around cancelation.

Anti-pattern: Asynchronous APIs

The Asynchronous APIs anti-pattern.

```
func Logs() <-chan logs {  
    c := make(chan logs)  
    go func() { // Receive Logs }  
    Return c  
}  
  
func main() {  
    for l := range Logs() {  
        // Do stuff with each l  
    }  
}
```

This is a simplified version of an API I wrote when I was starting to write Go.

There are a bunch of problems with it though:

How is the goroutine going to be shutdown?;

How is the size of the channel controlled?;

Closing the channel received from Logs() is likely to result in a panic;

How do errors from the goroutine get communicated?;

and what if a synchronous API is needed?

Most of these issues can be handled by making the Logs() function take additional parameters.

Or by making it a type and adding additional methods to that type.

But it would still be an asynchronous API and it's complexity will have grown.

```
type Reader struct {  
    Err      error  
    Current log  
}  
  
func (r *Reader) Next(ctx context.Context) bool {  
    if r.Err != nil { return false }  
    r.Current, r.Err = r.readNext(ctx)  
    return true  
}  
  
func main() {  
    r := &log.Reader{}  
    ctx := context.Background()  
    for r.Next(ctx) {  
        fmt.Println("I got a log": r.Current)  
    }  
    if r.Err != nil { fmt.Println("Error:", err) }  
}
```

Here is a synchronous version that answers all of those questions.

We still have a type, but it has a much more accurate name: Reader.

The user of a Reader value is in control of the reading loop and error handling.

Next is called until it returns false, indicating no more logs and provides the Current item unless there is an Error.

Reader itself likely uses channels in an async fashion underneath in the readNext function, especially since it can be canceled via the context. But those channels are not exposed.

This synchronous API can be made asynchronous if and when it's needed by it's consumers.

Anti-pattern: Asynchronous APIs

- Synchronous APIs are easier to make asynchronous than vice versa.
- Use channels internally, don't expose them.
- For inspiration, see the `stdlib http` package and `buffio.Scanner` type.

In summary...

Provide synchronous APIs.

It is possible to make a synchronous API async, but much harder, if not impossible, to do the inverse. Leave the concurrency to someone else.

As a general rule, it should be uncommon to expose channels in an API. Use them internally instead.

For inspiration see the `stdlib http` package. This package has a fairly large API surface and does quite a lot. It uses channels internally, but only exposes one in the `CloseNotifier` type.

Anti-pattern: If - Then - Else

The If Then Else Anti-pattern

```
def things(x)
  if x > 2
    foo
  else
    bar
  end
end
```

This is fairly idiomatic Ruby.

It relies on the fact that functions in Ruby return the last evaluated statement.

```
def things(x)
  if x > 2
    foo
  else
    bar
  end
end
```

foo is returned when $x > 2$ as that was the last statement evaluated in the function before returning.


```
def things(x)
  if x > 2
    foo
  else
    bar
  end
end
```

Otherwise bar is returned for the same reason.

```
type someType int

var (
    foo someType = 100
    Bar someType = 200
)

func things(x int) someType {
    if x > 2 {
        return foo
    } else {
        return bar
    }
}
```

if block ends with a return statement, so drop this else and outdent its block

Directly transliterated to Go, we would have a function that looks like this.

And running it through the go linter will produce the following warning:

(Click)

“if block ends with a return statement, so drop this else and outdent its block”

The go tools are there to help you.

```
type someType int

var (
    foo someType = 100
    Bar someType = 200
)

func things(x int) someType {
    if x > 2 {
        return foo
    }
    return bar
}
```

With that change implemented we get this.

The path with a conditional, $x > 2$, is handled above the non conditional path and returns early.

(Click)

The non conditional path, also known as the “happy path”, is de-dented to the left and is a simple return.

```
func NestedThings() {  
    err := Thing1()  
    if err == nil {  
        err2 := Thing2()  
        if err2 == nil {  
            // etc  
        }  
    }  
}
```

```
func EarlyReturnThings() {  
    if err := Thing1(); err != nil {  
        return  
    }  
    if err := Thing2(); err != nil {  
        return  
    }  
    // etc  
}
```

Here is another example.

It is harder to visually trace what is going on in the function on the left.

But the function on the right handles the error conditions immediately.

It returns on error and keeps the flow of the function aligned to the left, making visual parsing easier.

```
func newEvent(startRaw, endRaw string) (event, error) {  
    var e event  
    if startRaw != "" {  
        start, err := time.Parse(time.RFC3339, startRaw)  
        if err != nil {  
            return e, err  
        }  
        e.startTime = start  
    } else {  
        now1h := time.Now().Add(-2 * time.Hour)  
        e.startTime = now1h  
    }  
    // Do this again with endRaw for e.endTime,  
    // with a different default.  
    return e, nil  
}
```

Sometimes it's not as easy though.

The code here is parsing some raw start and end times for an event.

For brevity the end time logic, which is almost identical to the start time logic, is elided.

```
func newEvent(startRaw, endRaw string) (event, error) {  
    var e event  
    if startRaw != "" {  
        start, err := time.Parse(time.RFC3339, startRaw)  
        if err != nil {  
            return e, err  
        }  
        e.startTime = start  
    } else {  
        now1h := time.Now().Add(-2 * time.Hour)  
        e.startTime = now1h  
    }  
    // Do this again with endRaw for e.endTime,  
    // with a different default.  
    return e, nil  
}
```

If the raw start time is empty, a default value is calculated and assigned to the event.

```
func newEvent(startRaw, endRaw string) (event, error) {  
    var e event  
    if startRaw != "" {  
        start, err := time.Parse(time.RFC3339, startRaw)  
        if err != nil {  
            return e, err  
        }  
        e.startTime = start  
    } else {  
        now1h := time.Now().Add(-2 * time.Hour)  
        e.startTime = now1h  
    }  
    // Do this again with endRaw for e.endTime,  
    // with a different default.  
    return e, nil  
}
```

If the raw start time is not empty, the time is parsed.

If there is an error in parsing, that error is returned and the function ends.

If not the parsed start time is assigned to the event.

```
func newEvent(startRaw, endRaw string) (event, error) {  
    var e event  
    if startRaw != "" {  
        start, err := time.Parse(time.RFC3339, startRaw)  
        if err != nil {  
            return e, err  
        }  
        e.startTime = start  
    } else {  
        now1h := time.Now().Add(-2 * time.Hour)  
        e.startTime = now1h  
    }  
    // Do this again with endRaw for e.endTime,  
    // with a different default.  
    return e, nil  
}
```

But it's visually harder to parse the flow of the function than it needs to be.

And as mentioned earlier, the end time logic isn't shown here.

Let's do better.


```
func parseTimeOrDefault(raw string, def time.Time) (time.Time, error) {  
    if raw == "" {  
        return def, nil  
    }  
    return time.Parse(time.RFC3339, raw)  
}  
  
func newEvent(startRaw, endRaw string) (event, error) {  
    var e event  
    t, err := parseTimeOrDefault(startRaw, time.Now().Add(-2 * time.Hour))  
    if err != nil {  
        return e, err  
    }  
    e.startTime = t  
    t, err = parseTimeOrDefault(endRaw, time.Now())  
    if err != nil {  
        return e, err  
    }  
    e.endTime = t  
    return e, nil  
}
```

In this new version, the function `parseTimeOrDefault` contains the app specific logic for parsing and returning a time.

```
func parseTimeOrDefault(raw string, def time.Time) (time.Time, error) {  
    if raw == "" {  
        return def, nil  
    }  
    return time.Parse(time.RFC3339, raw)  
}  
  
func newEvent(startRaw string) (event, error) {  
    var e event  
    t, err := parseTimeOrDefault(startRaw, time.Now().Add(-2 * time.Hour))  
    if err != nil {  
        return e, err  
    }  
    t.startTime = t  
    t, err = parseTimeOrDefault(endRaw, time.Now())  
    if err != nil {  
        return e, err  
    }  
    e.endTime = t  
    return e, nil  
}
```

The parseTimeOrDefault and newThing functions handle error logic before the happy path.

(Click)

There is a clean flow with the happy paths flush to the left in both functions.

And there is also enough space to include the end time logic.

Anti-pattern: If – Then – Else

- Handle errors early and return often.
- Keep common or happy paths de-dented.
- When it's not possible refactor and/or redesign.

To summarize:

Handle errors early and return often.

Keep the common or happy paths de-dented and to the left.

When that's not possible refactor / redesign.

Anti-pattern: Panic In a Lib

The panic in a lib anti-pattern

```
func doHTTP(req http.Request) {  
    err := http.DefaultClient.Do(req)  
    if err != nil {  
        panic(err)  
    }  
}  
  
package main  
  
func stuff() {  
    defer func() {  
        if r := recover(); r != nil {  
            fmt.Println("recovered in stuff", r)  
        }  
    }()  
    doHTTP(http.Request{})  
}
```

Panic is as close to exceptions as Go gets.

In languages that utilize exceptions for error handling, raising an error is a natural thing.

But in Go, errors are values that need to be handled.

If the doHTTP function above was part of a 3rd party dependency, any app that uses it would need to add recover checks to every function that calls doHTTP or risk having the panic bubble up and terminate the program.

This forces the user of the dependency to litter defer functions around their code base.

To further complicate matters, it may be a transitive dependency, so the panic may bubble up into the application, even though it's not being used directly.

You usually find out where recovery checks need to be added after the fact.

After your program is crashing due to an unhandled panic at runtime

in production.

```
func doHTTP(req http.Request) error {  
    err := http.DefaultClient.Do(req)  
    return err  
}  
  
package main  
  
func main() {  
    if err := doHTTP(http.Request{}); err != nil {  
        fmt.Println("error doing http request: ",err)  
    }  
}
```

In Go, treating the error as a value and handling it is idiomatic.

The error goes from being a nebulous part of the contract around using doHTTP to a part of the function's signature, making it discoverable via documentation and other tooling.

Anti-pattern: Panic in a Lib

- Return errors, don't panic.
- Only panic when an error can't be handled directly or the handling can't be delegated to the upstream caller.
- Mark up errors with github.com/pkg/errors to add context before returning them.

To summarize:

Return errors, don't panic.

Only panic when an error can't be handled directly or the handling can't be delegated to the upstream caller.

This should only happen when the program cannot make any forward progress.

We didn't cover it, but I encourage everyone to add context to errors with this errors package before returning them.

Anti-pattern: Interface all the things

The interface all the things anti-pattern.


```
type Batch struct {
    logLines [][]byte
}

// ---

var buf bytes.Buffer
for i := range batch.logLines {
    buf.Write(batch.logLines[i])
}
http.Post("/logs", &buf)

// VS

func (b *Batch) Read(p []byte) (int, error) { ... }
http.Post("/logs", &b)
```

While I immediately grasped Go interfaces, it took me a while to really appreciate the power of small, simple interfaces.

This appreciation came when I realized it would be possible to re-implement a batch of logs being sent to a http service as an `io.Reader`.

This change made the batch easier to send because Go's HTTP Client works with `io.Reader`s.

This also improved performance by eliminating a copy into a buffer, before making the HTTP call.

Additionally this same Batch could be passed to anything taking an `io.Reader`. Or even an `io.Writer`, when adapted by an `io.Pipe`.

That is when I internalized the potential of small, focused interfaces.

```
type Thinger interface {  
    CreateThingConfig(...  
    DestroyThingConfig(...  
    CreateThing(...  
    DeleteThing(...  
    FetchThing(...  
    UpdateThing(...  
    SuspendThing(...  
    TerminateThing(...  
    DescribeThing(...  
    DescribeThingConfig(...  
    CreateThingHook(...  
    DetachThing(...  
    ...  
}
```

But I run into a lot of large interfaces with 6 to 12 or even more methods defined.

These large interfaces tend to also have only 2 implementations.

The first being the only concrete implementation of the interface.

The second being a mock for the purposes of testing.

These large interfaces tend to be defined up-front. They aren't discovered across implementations at a later date.

They tend to not be an abstraction, but the encoding of already declared interactions. There is little to no abstraction in these interfaces. And they remind me of the go proverb: The bigger the interface, the weaker the abstraction.

Interfaces should be discovered from existing types and extracted out of them.

The `io.Reader` and `io.Writer` interfaces weren't designed up front, they were discovered later. The network, file and other byte handling types shared a similar implementation. Out of those similarities the `io.Reader` and `io.Writer` interfaces were born.

The interface example in the slide isn't real, but extracted out of real code. The real one has more methods, 17 to be exact.

The single concrete implementation of this interface talks to a service via http.

The only other implementation is the mock implementation.

```
http.Client.Do(*http.Request) (*http.Response, error)

func (t *Thing) CreateRequest(Opts) (*http.Request, error)
func (t *Thing) DeleteRequest(Opts) (*http.Request, error)
func (t *Thing) FetchRequest(Opts) (*http.Request, error)
func NewThingFromResponse(r *http.Response) (Thing, error)

req, _ := t.CreateRequest(Opts)
resp, _ := http.DefaultClient.Do(req)
t, _ := NewThingFromResponse(resp)

s := httptest.NewServer(MockCreationHandler)
defer s.Close()
req, err := thing.CreateRequest(s.URL)
resp, err := http.DefaultClient.Do(req)
if err != nil {
    t.Fatal("Error creating thing": err)
}
```

A different approach to the same problem domain would be to use the http Client in the standard lib instead.

After all, the concrete implementation is a wrapper around HTTP requests.

```
http.Client.Do(*http.Request) (*http.Response, error)

func (t *Thing) CreateRequest(Opts) (*http.Request, error)
func (t *Thing) DeleteRequest(Opts) (*http.Request, error)
func (t *Thing) FetchRequest(Opts) (*http.Request, error)
func NewThingFromResponse(r *http.Response) (Thing, error)

req, _ := t.CreateRequest(Opts)
resp, _ := http.DefaultClient.Do(req)
t, _ := NewThingFromResponse(resp)

s := httptest.NewServer(MockCreationHandler)
defer s.Close()
req, err := thing.CreateRequest(s.URL)
resp, err := http.DefaultClient.Do(req)
if err != nil {
    t.Fatal("Error creating thing": err)
}
```

Create types for each of the domain concerns: Thing, ThingHook & ThingConfig.

Each of those types has methods for each of the different interactions, each of which create and return a http.Request for that interaction.

```
http.Client.Do(*http.Request) (*http.Response, error)

func (t *Thing) CreateRequest(Opts) (*http.Request, error)
func (t *Thing) DeleteRequest(Opts) (*http.Request, error)
func (t *Thing) FetchRequest(Opts) (*http.Request, error)
func NewThingFromResponse(r *http.Response) (Thing, error)

req, _ := t.CreateRequest(Opts)
resp, _ := http.DefaultClient.Do(req)
t, _ := NewThingFromResponse(resp)

s := httptest.NewServer(MockCreationHandler)
defer s.Close()
req, err := thing.CreateRequest(s.URL)
resp, err := http.DefaultClient.Do(req)
if err != nil {
    t.Fatal("Error creating thing": err)
}
```

This allows users of the API to use the standard http Client's Do method.

```
http.Client.Do(*http.Request) (*http.Response, error)

func (t *Thing) CreateRequest(Opts) (*http.Request, error)
func (t *Thing) DeleteRequest(Opts) (*http.Request, error)
func (t *Thing) FetchRequest(Opts) (*http.Request, error)
func NewThingFromResponse(r *http.Response) (Thing, error)

req, _ := t.CreateRequest(Opts)
resp, _ := http.DefaultClient.Do(req)
t, _ := NewThingFromResponse(resp)

s := httptest.NewServer(MockCreationHandler)
defer s.Close()
req, err := thing.CreateRequest(s.URL)
resp, err := http.DefaultClient.Do(req)
if err != nil {
    t.Fatal("Error creating thing": err)
}
```

A nice side effect of this is that the `httptest.Server` can be used to test more of your code, instead of testing a mock.

This solution eliminates the giant interface, uses more of the power of the `stdlib` and improves test coverage.

```
http.Client.Do(*http.Request) (*http.Response, error)

func (t *Thing) CreateRequest(Opts) (*http.Request, error)
func (t *Thing) DeleteRequest(Opts) (*http.Request, error)
func (t *Thing) FetchRequest(Opts) (*http.Request, error)
func NewThingFromResponse(r *http.Response) (Thing, error)

req, _ := t.CreateRequest(Opts)
resp, _ := http.DefaultClient.Do(req)
t, _ := NewThingFromResponse(resp)

s := httptest.NewServer(MockCreationHandler)
defer s.Close()
req, err := thing.CreateRequest(s.URL)
resp, err := http.DefaultClient.Do(req)
if err != nil {
    t.Fatal("Error creating thing": err)
}
```

A nice side effect of this is that the `httptest.Server` can be used to test more of your code, instead of testing a mock.

Anti-pattern: Interface all the things

- Go proverb: The bigger the interface, the weaker the abstraction.
- Pivot away from the what is being done (CreateThing) and focus on the how it is being done.
- There are other ways to test things then mocking an interface.

In summary:

The bigger the interface, the weaker the abstraction.

Rethinking the abstraction and pivoting away from what is being done to how it's being done can help. Though sometimes the inverse is true.

There are other ways to test things then mocking an interface. The `httptest.Server` type's handlers and network servers are pretty easy to write in Go.

PS: I'm not against mocks but they are over used in testing to the point where I've seen tests that really end up testing a set of mocks instead of the actual implementations.

Anti-pattern: Naked Return Values

The naked return values anti-pattern.

```
func SmallThing(b int) (n int, err error) {  
    n = rand.Intn(b)  
    err = fmt.Errorf("This is an error: %d",b)  
    return  
}
```

Naked returns are often used with named return values. The above returns the random integer and the error.

This is because `n` and `err` are named in the return parameters.

```
func something() error {  
    return fmt.Errorf("something error")  
}  
func SmallThing(b int) (n int, err error) {  
    n = rand.Intn(b)  
    if err := something(); err != nil {  
        fmt.Println("something() Err:", err)  
    }  
    return  
}  
  
func main() {  
    _, err := SmallThing(10)  
    fmt.Println("main err:", err)  
}
```

But as this function grows it's easy for naked returns to lead to subtle bugs.

Here SmallThing has been expanded to call something() and “handle” the error.

```
func something() error {  
    return fmt.Errorf("something error")  
}  
func SmallThing(b int) (n int, err error) {  
    n = rand.Intn(b)  
    if err := something(); err != nil {  
        fmt.Println("something() Err:", err)  
    }  
    return  
}  
  
func main() {  
    _, err := SmallThing(10)  
    fmt.Println("main err:", err)  
}  
something() Err: something error  
main err: <nil>
```

But, If you expect to get the error from something returned to main, you would be mistaken.

This is because err is shadowed in the block.

Once naked returns show up in code, they tend to stick around.

They get cargo culted forward to new functions.

They're not factored out as those functions expand.

They inevitably cause subtle bugs.

Naked returns are okay if the function is a handful of lines. Once it's a medium sized function, be explicit with your return values. Corollary: **it's not worth it to name result parameters just because it enables you to use naked returns**. Clarity of docs is always more important than saving a line or two in your function.

[HTTP://BIT.LY/GOCODEREVIEWCOMMENTS](http://bit.ly/gocodereviewcomments)

The guidance of the CodeReviewComments section of the golang wiki states:

Naked returns are okay if the function is a handful of lines. Once it's a medium sized function, be explicit with your return values.

Corollary: it's not worth it to name result parameters just because it enables you to use naked returns.

Clarity of docs is always more important than saving a line or two in your function.

Anti-pattern: Naked Returns

- Don't use naked return values, even if you use named result parameters.

I go one step further and say:

You should not use naked return values at all,
even when you use named result parameters.

Anti-pattern: interface{}

The empty interface anti-pattern.

```
func Voila(i interface{}) { ... }
```

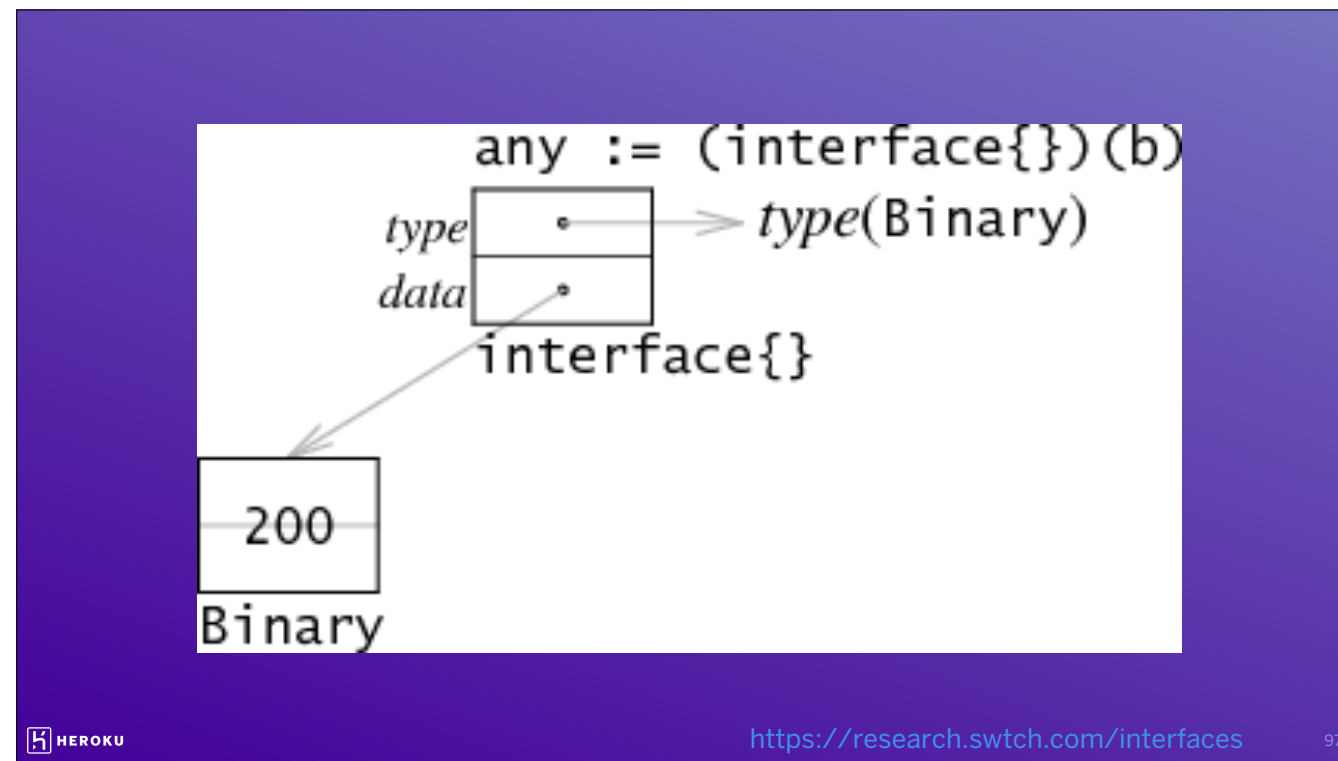
interface{} says nothing.

What does the function Voila know about value “i”?

To quote the Go proverbs:

(Click)

The empty interface says nothing.



The Go runtime still knows what type the empty interface is wrapping.

The above image shows the two pointers the runtime holds in an empty interface.

The first is a pointer to the type that the interface is wrapping so the runtime knows what it is.

The second is a pointer to the data itself.

```
func Voila(i interface{}) {  
    switch tv := i.(type) {  
        case int: fmt.Println("%d", tv)  
        case float64: fmt.Println("%f", tv)  
        case string: fmt.Println("%s", tv)  
    }  
}  
  
Voila(SomeType{})
```

But Voila will need to resort to type assertions or switch statements to determine how “i” should be handled.

The empty interface sidesteps static type checking and Voila can’t force any guarantees on the caller.

What happens above if something other than an int, float64 or string is passed to Voila?

Nothing?

A case wasn’t written to handle it.

```
func Voila(i interface{}) {  
    switch tv := i.(type) {  
        case int: fmt.Println("%d", tv)  
        case float64: fmt.Println("%f", tv)  
        case string: fmt.Println("%s", tv)  
        default:  
            fmt.Printf("Unknown type: %T\n", tv)  
        }  
    }  
}
```

We can handle that case, but that's only useful as a generic fall through.

```
func Voila(s fmt.Stringer) {  
    fmt.Println(s.String())  
}
```

Instead of using the empty interface try to create an interface with a method that defines the behavior you need.

One may already exist.

```
b, _ := json.Marshal(SomeType{})  
  
v, _ := json.Unmarshal([]byte("hello"))
```

But sometimes we do need to use the empty interface, especially when dealing with unknown data.

This is commonly needed when encoding/decoding data. It's needed at times when the type isn't known.

This still requires a lot of reflection.

If you are interested in how much I suggest reading through the json package in the stdlib.

Anti-patterns: empty interface

- Use the empty interface when dealing with unknown data.
- Otherwise try to tease out an interface that declares the behavior you need.

In Summary...

Use the empty interface when dealing with unknown data.

Otherwise try to tease out an interface that declares the behavior you need.



So, in conclusion, let's recap.

Orthogonality

With regard to orthogonality, idiomatic Go is made up of small pieces, that do one thing well.

This principle can be observed in small, focused interfaces that generalize a behavior, not an implementation.

In go contains packages that are loosely coupled with each.

Readability

With regard to readability, idiomatic Go has clear lines of sight.

Functions declare their dependencies via their parameter lists making it easy to understand what type a value is and where that type came from.

A function's happy path is unindented, sitting all the way to the left.

And the strategy of return early and return often is used before the happy path to handle errors.

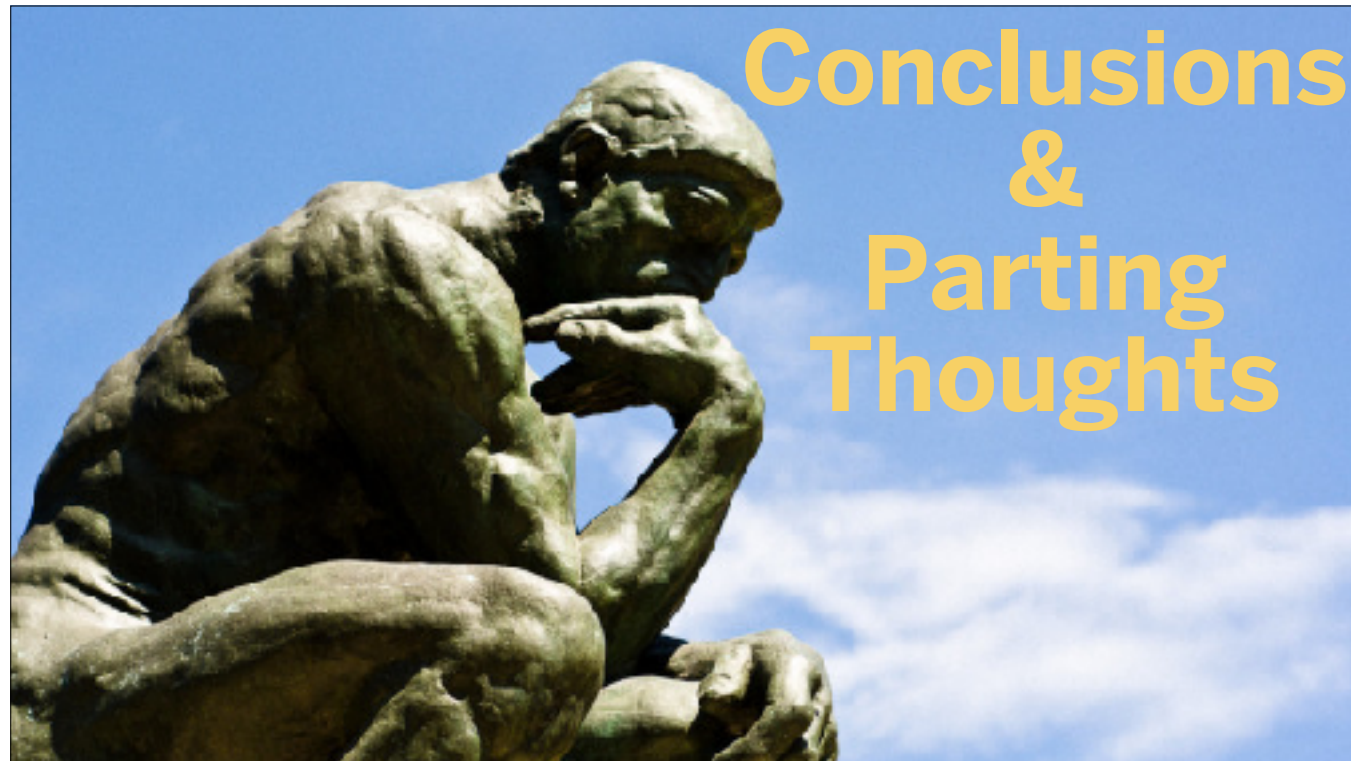
Idiomatic Go avoids stuttering, reducing clutter and noise.

Idiomatic Go is well documented, focusing on the why of its packages, types and functions.

Simplicity

With regard to simplicity, idiomatic Go uses `interface{}` sparingly and with care, both in test and non test code.

Packages and projects export a minimal surface area and has names that clearly indicate their purpose.



I hope that through these anti-patterns I've shown you some things to look for when writing or reviewing Go code.

I hope I have shown how the design principles of the Go language help identify non idiomatic Go code.

I'm sure these anti-patterns aren't complete and I look forward to future discussions.

Thank You.

Special Thanks To

- Renee French
- Olexandr Shalakhin
- Sea of Clouds
- My family
- Dave Cheney
- Ines Sombra
- Go Team
- Gophercon Team



108

Before questions, I'd like to extend a special thanks to....

Renee French for the Go Gopher

Olexandr Shalakhin for even more gophers. Sorry, I know that's probably not the correct way to pronounce your name.

Sea of Clouds for the Go Haiku

My family for putting up with me while I focused on this talk instead of going to the pool, on a scavenger hunt and otherwise being absent at times.

Dave Cheney, Ines Sombra & Sam Boyer for help whipping this talk into shape

The entire Go Team for creating a language that values the things I value

The Gophercon team for putting this conference on every year.

And last, but certainly not least, everyone who attended today.

Questions ?

Questions?