

— REDOWAN DELOWAR

Context propagation in Go

Lessons from operating large-scale distributed systems
at DoorDash (Wolt)



Scan to get the slides

<https://rednafi.com/appearances>

```
$ whoami
```

Hi, I'm Redowan, [@rednafi](#) on the web.

I oscillate between product and platform work at DoorDash/Wolt.

Interested in databases, streaming, o11y, and failure modes of large-scale distsys.

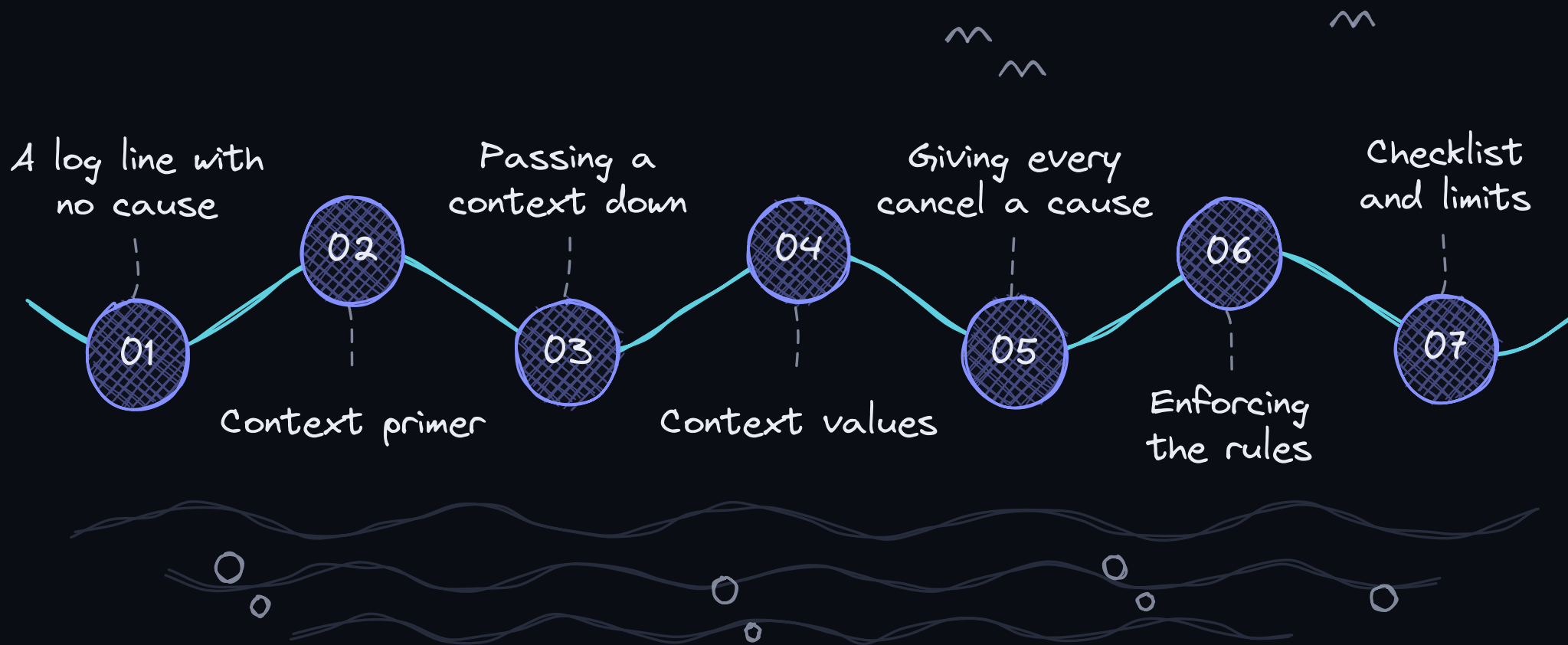
Writing at [rednafi.com](#).

[github.com/rednafi](#)

[linkedin.com/in/rednafi](#)



Overview



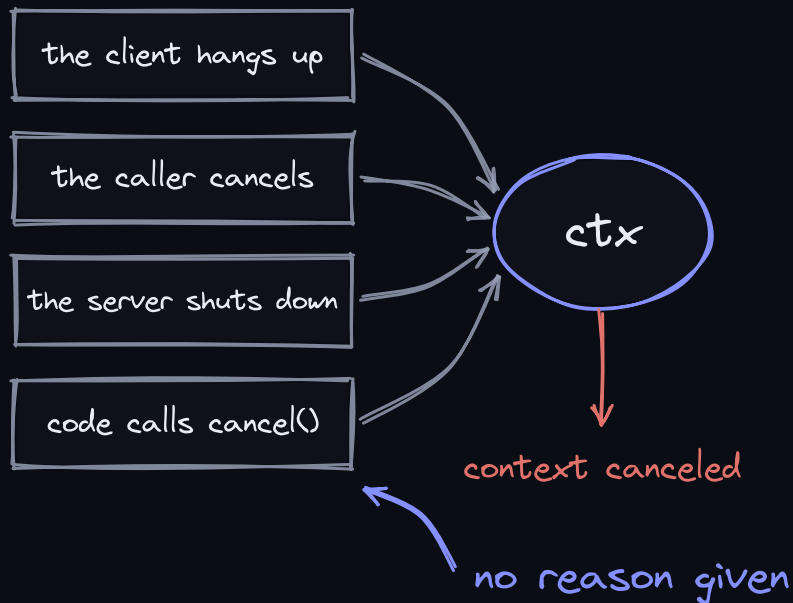
The log doesn't say why the context stopped

● ● ● order-service

```
ERROR place order: context canceled
ERROR fetch menu: context deadline exceeded
ERROR place order: context canceled
ERROR charge card: context canceled
ERROR fetch menu: context deadline exceeded
ERROR place order: context canceled
```



Any of these can stop it



The cancel often comes from another service or from a library.

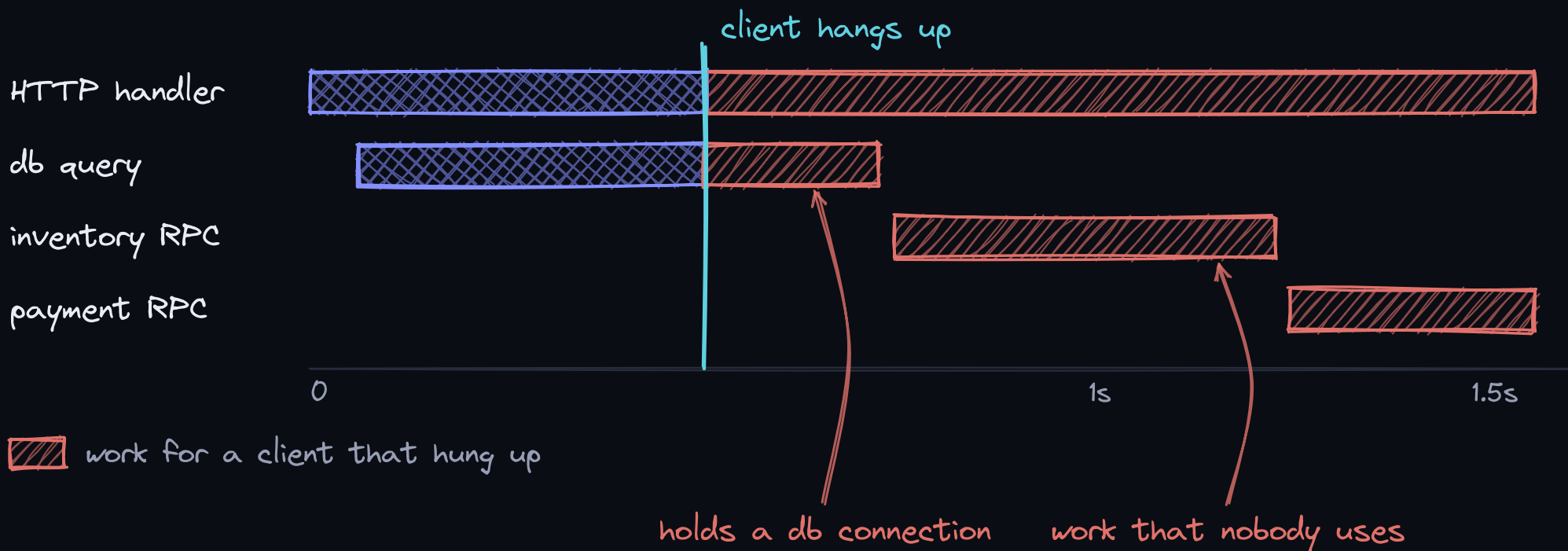
02

— PART 2

Context primer

02

The work continues after the client leaves



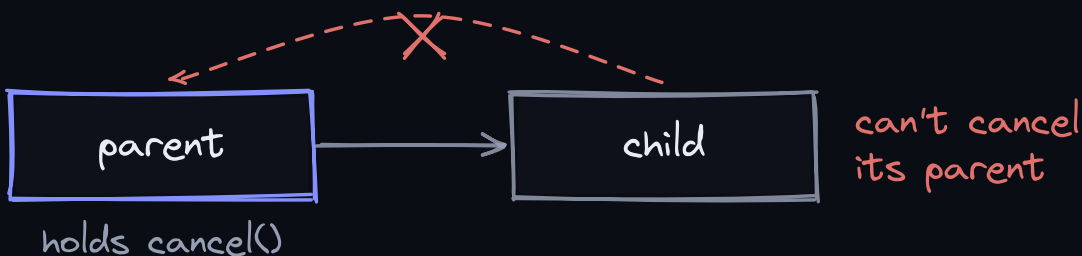
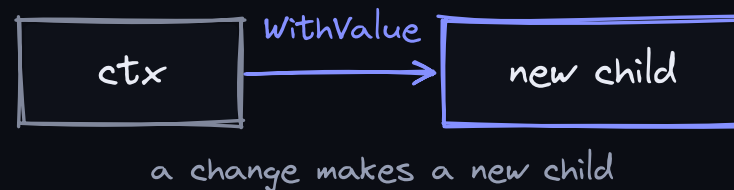
context.Context

“

Package context defines the Context type, which carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes.

[context package doc](#)

```
type Context interface {  
    Deadline() (deadline time.Time, ok bool) ← when to give up  
    Done() <-chan struct{} ← closes when the work should stop  
    Err() error ← Canceled or DeadlineExceeded  
    Value(key any) any ← data about the request  
}
```



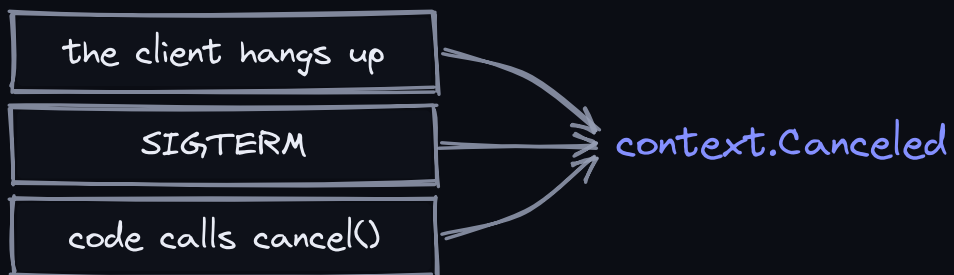
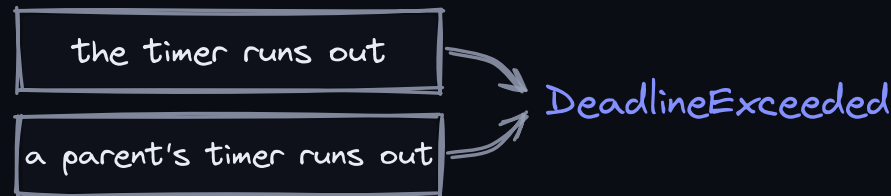
A context ends in one of two ways

```
ctx, cancel := context.WithCancel(parent)
cancel()
ctx.Err() // context.Canceled
```

← someone called cancel

```
ctx, cancel := context.WithTimeout(parent, 2*time.Second)
defer cancel()
<-ctx.Done()
ctx.Err() // context.DeadlineExceeded
```

← the deadline passed



Each With function makes a child, so contexts form a tree

`Background()`

The empty root context

`WithCancel(parent)`

Stop when the code calls cancel

`WithTimeout(parent, d)`

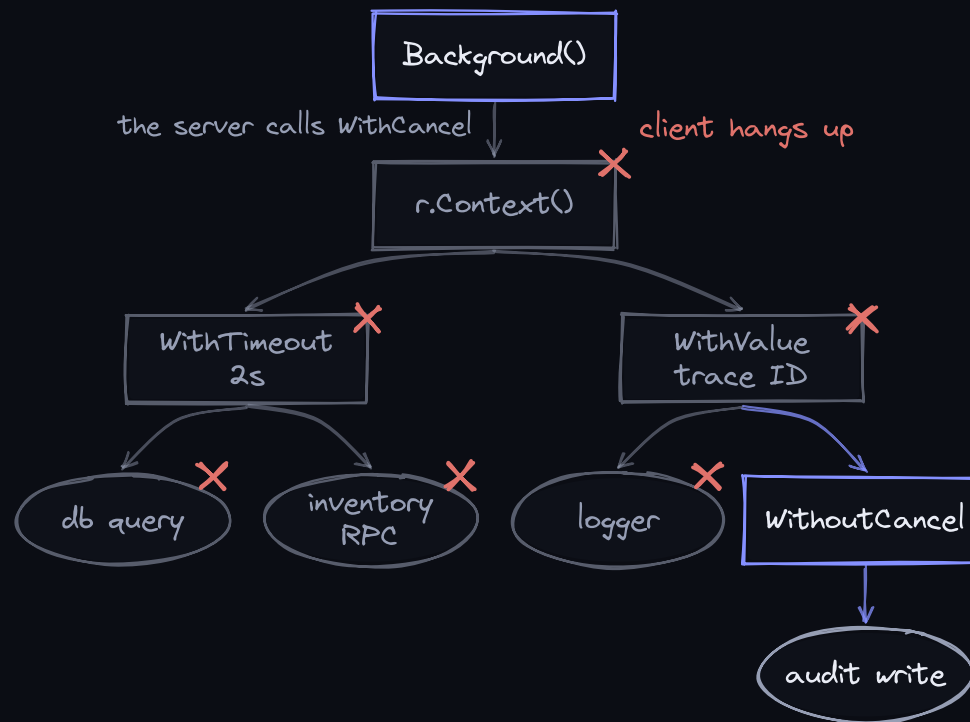
Stop after a duration. `WithDeadline` stops at a time

`WithValue(parent, k, v)`

Add one value

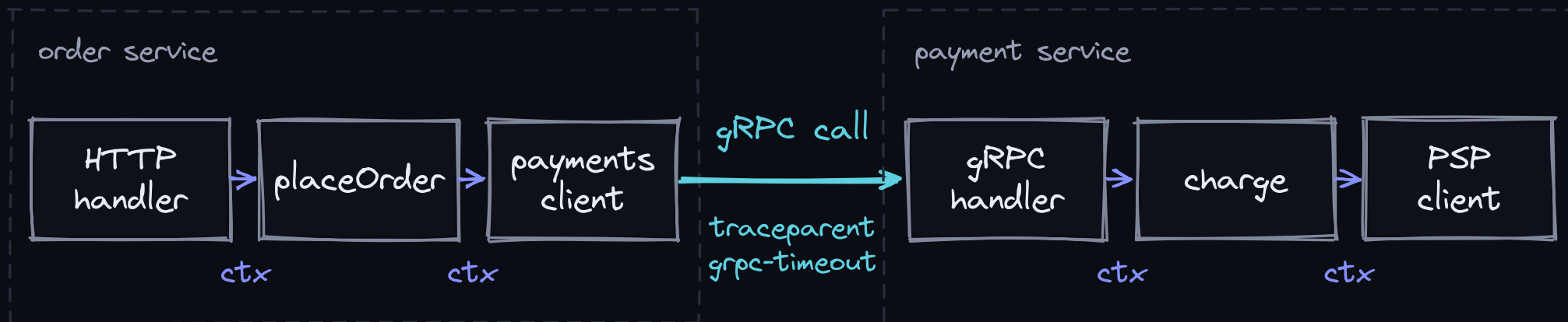
`WithoutCancel(parent)`

Keep the values and drop the stop signal (Go 1.21)



When the client hangs up, everything below `r.Context()` stops.
The `WithoutCancel` branch keeps going.

A context travels inside a process and between processes



In-process: your code passes `ctx` from function to function.

Distributed: gRPC and OpenTelemetry copy its deadline and trace into headers. They can copy only what your code passed down.

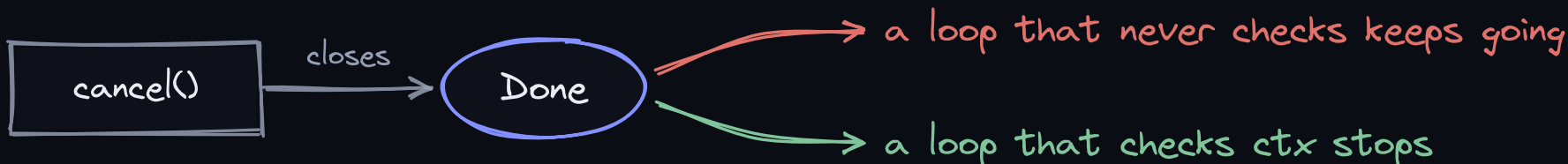
A cancel stops only the code that checks the context

```
// Don't: ignore ctx in the loop.
```

```
for _, img := range images {  
    resize(img) // 200ms each ← keeps going after a cancel  
}
```

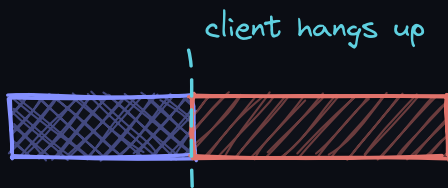
```
// Do: check ctx before each image.
```

```
for _, img := range images {  
    if err := ctx.Err(); err != nil { ← stops before the next image  
        return err  
    }  
    resize(img)  
}
```



Most context bugs I see are in these groups

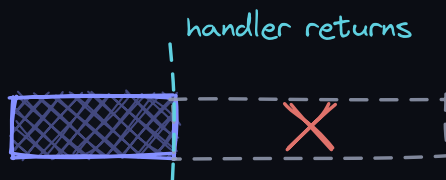
1



Work that runs too long

The client hangs up, but the query continues.

2



Work that stops too early

The audit write fails when the handler returns.

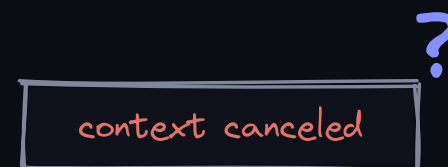
3



A value that is wrong or hidden

Another package's key hides the user.

4



A cancel with no cause

The log says only context canceled.

03

— PART 3

Passing a context down

Use the caller's context

// Don't: start a new context here.

`return s.store.Find(context.Background(), id)` ← loses the caller's deadline

// Do: pass the caller's context.

`return s.store.Find(ctx, id)`

in main and init
`Background()`

in tests (Go 1.24)
`t.Context()`

no ctx passed in yet
`TODO()`

never pass
`nil` ✗

Pass ctx to each call, and don't store it in a struct

```
// Don't: keep the context in a struct.
type Notifier struct {
    ctx    context.Context ← one deadline for every call
    client *sms.Client
}

// Do: pass the context to each call.
func (n *Notifier) Send(ctx context.Context, msg string) error {
    return n.client.Send(ctx, msg)
}
```

“

When you store the context in a struct, you obscure lifetime to the callers, or worse intermingle two scopes together in unpredictable ways.

Go blog, Contexts and structs, 2021

Wait on the timer and ctx.Done() together

// Don't: sleep without a check for cancel.

time.Sleep(backoff(attempt)) ← a cancel can't stop it

// Do: stop the wait when ctx is done.

select {

case <-time.After(backoff(attempt)):

case <-ctx.Done(): ← a cancel ends the wait

return ctx.Err() ← Part 5 returns the cause here

}

a timer

`<-time.After(d)`

+ `<-ctx.Done()`

a receive

`v := <-ch`

+ `<-ctx.Done()`

a send

`ch <- v`

+ `<-ctx.Done()`

Give background work its own context and timeout

// Don't: use the request context for background work.

go h.audit.Write(r.Context(), orderID) ← canceled when the handler returns

// Do: keep the values and set a new timeout.

detached := context.WithoutCancel(r.Context()) ← keeps the values (Go 1.21)

go func() {

 ctx, cancel := context.WithTimeout(detached, 5*time.Second) ← its own timeout

 defer cancel()

 h.audit.Write(ctx, orderID)

}()

Background()

✓ drops the cancel

✗ drops the trace ID and log fields

WithoutCancel(ctx)

✓ keeps the trace ID and log fields

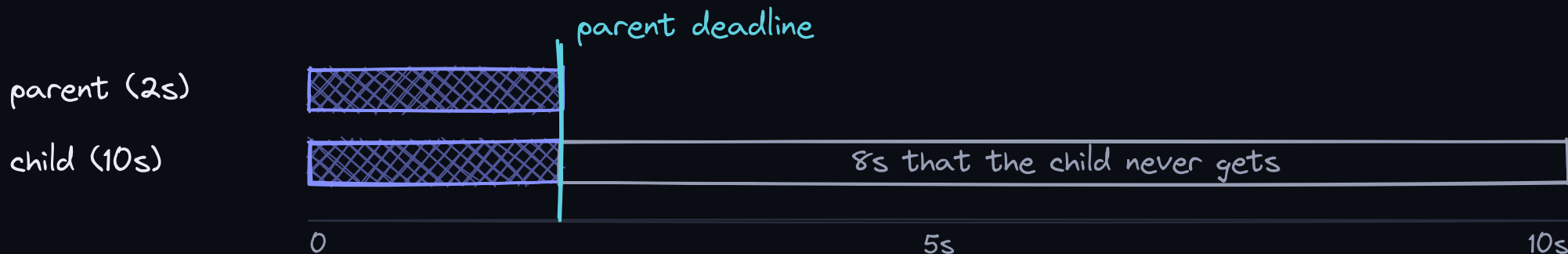
✓ drops the cancel and the deadline

A child context can't extend its parent's deadline

```
parent, cancelParent := context.WithTimeout(ctx, 2*time.Second) ← the parent gets 2s
defer cancelParent()
```

```
child, cancelChild := context.WithTimeout(parent, 10*time.Second) ← the child asks for 10s
defer cancelChild()
```

```
deadline, _ := child.Deadline() ← 2s from now
```



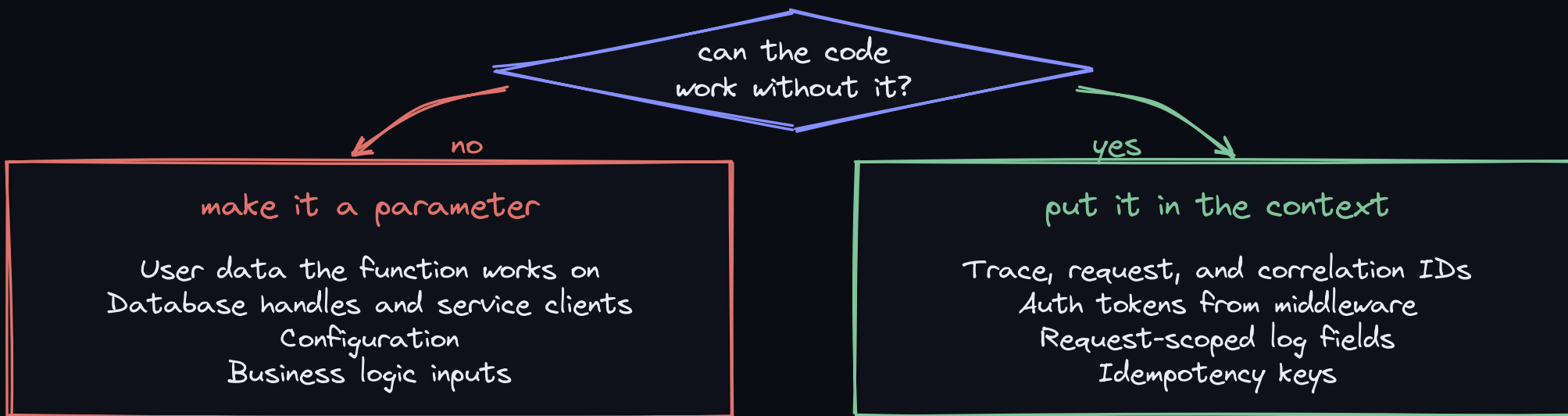
Set the deadline once, in the handler. Give a lower layer its own timeout only when it needs less time.

04

— PART 4

Context values

Put a value in the context only if the code works without it



“

Use context Values only for request-scoped data that transits processes and APIs, not for passing optional parameters to functions.

[context package doc](#)

An optional value also has to be request data.



Hide an unexported key behind a setter and a getter

```
// Don't: use a string as the key.
ctx = context.WithValue(ctx, "user", "alice") // package auth
ctx = context.WithValue(ctx, "user", "audit-bot") // package audit
ctx.Value("user") // "audit-bot" ← audit-bot hides alice

// Do: keep an unexported key behind a setter and a getter.
package auth

type ctxKey struct{} ← only auth can make one

func WithUser(ctx context.Context, u User) context.Context {
    return context.WithValue(ctx, ctxKey{}, u)
}

func UserFromContext(ctx context.Context) (User, bool) {
    u, ok := ctx.Value(ctxKey{}).(User) ← a typed value and a bool
    return u, ok
}
```

Don't hide a dependency in a context

// Don't: find the transaction in the context.

```
func (s *OrderStore) Create(ctx context.Context, o Order) (int64, error) {
```

var db DBTX = s.db ← then the store writes outside the transaction

if tx, ok := TxFromContext(ctx); ok { ← false if a caller forgets WithTx

db = tx

}

// ...

}

// Do: pass the transaction as an argument.

```
func (s *OrderStore) Create(ctx context.Context, tx DBTX, o Order) (int64, error) { ← callers pass it
```

// ...

}

05

— PART 5

Giving every cancel a cause

Cancel with a cause that says why

```
// Do: give each cancel a cause.
func placeOrder(ctx context.Context, id string) error {
    ctx, cancel := context.WithCancelCause(ctx) ← Go 1.20
    defer cancel(fmt.Errorf("placeOrder: completed: %w", context.Canceled)) ← normal return

    if err := checkInventory(ctx, id); err != nil {
        cancel(fmt.Errorf("inventory %s: %w: %w", id, err, context.Canceled)) ← error path
        return err
    }
    // ...
}
```

`context.Cause(ctx)` → `inventory ord-123: connection refused: context canceled`



Wrap DeadlineExceeded in the cause, so errors.Is still matches

```
// Don't: use a cause that doesn't wrap DeadlineExceeded.
```

```
ctx, cancel := context.WithTimeoutCause(ctx, 20*time.Millisecond,  
    errors.New("inventory: 20ms timeout")) ← errors.New doesn't wrap DeadlineExceeded  
defer cancel()
```

```
_, err := http.DefaultClient.Do(req.WithContext(ctx)) ← since Go 1.23, err wraps the cause  
errors.Is(err, context.DeadlineExceeded) ← false
```

```
// Do: wrap DeadlineExceeded with %w.
```

```
ctx, cancel := context.WithTimeoutCause(ctx, 20*time.Millisecond,  
    fmt.Errorf("inventory: 20ms timeout: %w", context.DeadlineExceeded)) ← errors.Is now returns true
```

Stack a cancel cause on the timeout

```
// Don't: stop at WithTimeoutCause.
```

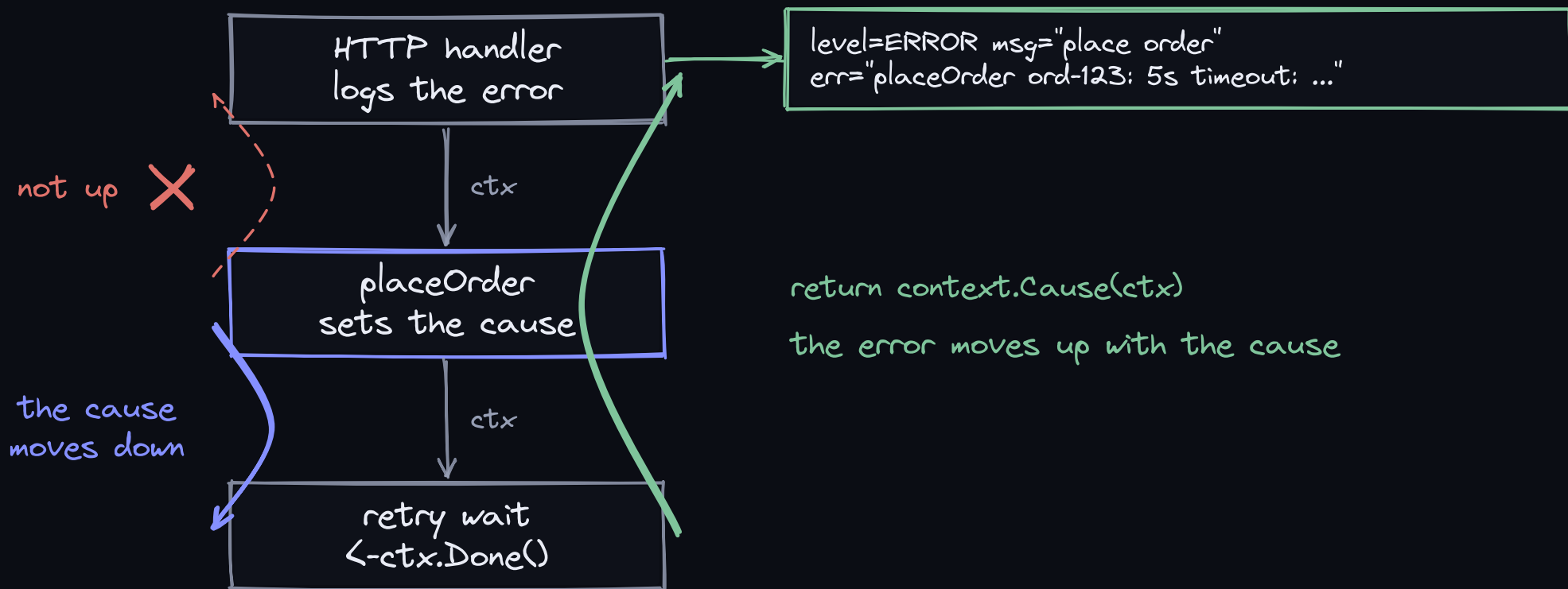
```
ctx, cancel := context.WithTimeoutCause(ctx, 5*time.Second,  
    fmt.Errorf("placeOrder %s: 5s timeout: %w", id, context.DeadlineExceeded))  
defer cancel()
```

← cancel() sets no cause

```
// Do: stack a cancel cause on the timeout.
```

```
ctx, cancelCause := context.WithCancelCause(ctx) ← a cause on every other path  
ctx, cancelTimeout := context.WithTimeoutCause(ctx, 5*time.Second, ← the timer and the deadline  
    fmt.Errorf("placeOrder %s: 5s timeout: %w", id, context.DeadlineExceeded))  
defer cancelTimeout()  
defer cancelCause(fmt.Errorf("placeOrder: completed: %w", context.Canceled)) ← this defer runs first
```

A cause moves down, so the error has to carry it up



Return `context.Cause(ctx)` where you return `ctx.Err()`

```
// Do: return the cause where you return ctx.Err().
select {
case <-time.After(backoff(attempt)):
case <-ctx.Done():
    return context.Cause(ctx) ← same errors.Is result, plus the reason
}
```

	<code>ctx.Err()</code>	<code>context.Cause(ctx)</code>
<code>errors.Is(err, DeadlineExceeded)</code>	✓ true	✓ true
the message	context deadline exceeded	placeOrder ord-123: 5s timeout: context deadline exceeded
no cause set	context deadline exceeded	context deadline exceeded same output

Log the error once, in the handler

```
// Don't: log the error in placeOrder, and also return it.
if err := retry(ctx, charge); err != nil {
    slog.ErrorContext(ctx, "retry failed", "err", err) ← the handler logs it again
    return err
}
```

```
level=ERROR msg="retry failed" err="placeOrder ord-123: 5s timeout: context deadline exceeded"
level=ERROR msg="place order"  err="placeOrder ord-123: 5s timeout: context deadline exceeded"
```

```
// Do: log once, in the handler. placeOrder only returns the error.
if err := placeOrder(ctx, id); err != nil {
    slog.ErrorContext(ctx, "place order", "err", err) ← the only log line
}
```

```
level=ERROR msg="place order" err="placeOrder ord-123: 5s timeout: context deadline exceeded"
```

06

— PART 6

Enforcing the rules

Linters can check some of these rules

go vet: lostcancel

a cancel func that
is never called

staticcheck: SA1012

a nil context passed
to a function

staticcheck: SA1029

a built-in type, like a
string, used as a key

containedctx

a context stored
in a struct field

contextcheck

a context that didn't come
from the caller

errorlint

err == context.Canceled,
which is false for a cause

Wrap the constructors so every cancel needs a cause

```
// Do: make the cancel func take a cause.
```

```
package xctx
```

```
func WithCancel(parent context.Context) (context.Context, context.CancelCauseFunc) {
```

```
    ctx, cancel := context.WithCancelCause(parent) //nolint:forbidigo
```

```
    return ctx, func(cause error) { ← the cancel func needs a cause
```

```
        if cause == nil { ← nil still gets a cause
```

```
            cause = errors.New("no cause given")
```

```
        }
```

```
        if !errors.Is(cause, context.Canceled) { ← wrap Canceled once, so errors.Is matches
```

```
            cause = fmt.Errorf("%w: %w", cause, context.Canceled)
```

```
        }
```

```
        cancel(cause)
```

```
    }
```

```
}
```

At DoorDash, Alex Hornbake built this as `xctx` for a Go monorepo that many teams share.

Block the plain constructors in CI

```
# Do: block the plain constructors in .golangci.yml.
```

```
version: "2"
```

```
linters:
```

```
  enable: [forbidigo]
```

```
  settings:
```

```
    forbidigo:
```

```
      analyze-types: true
```

```
      forbid:
```

```
        - pattern: ^context\.(With(Cancel|Timeout|Deadline)(Cause)?$
```

← a renamed import still matches

← blocks the Cause versions also

```
$ golangci-lint run
```

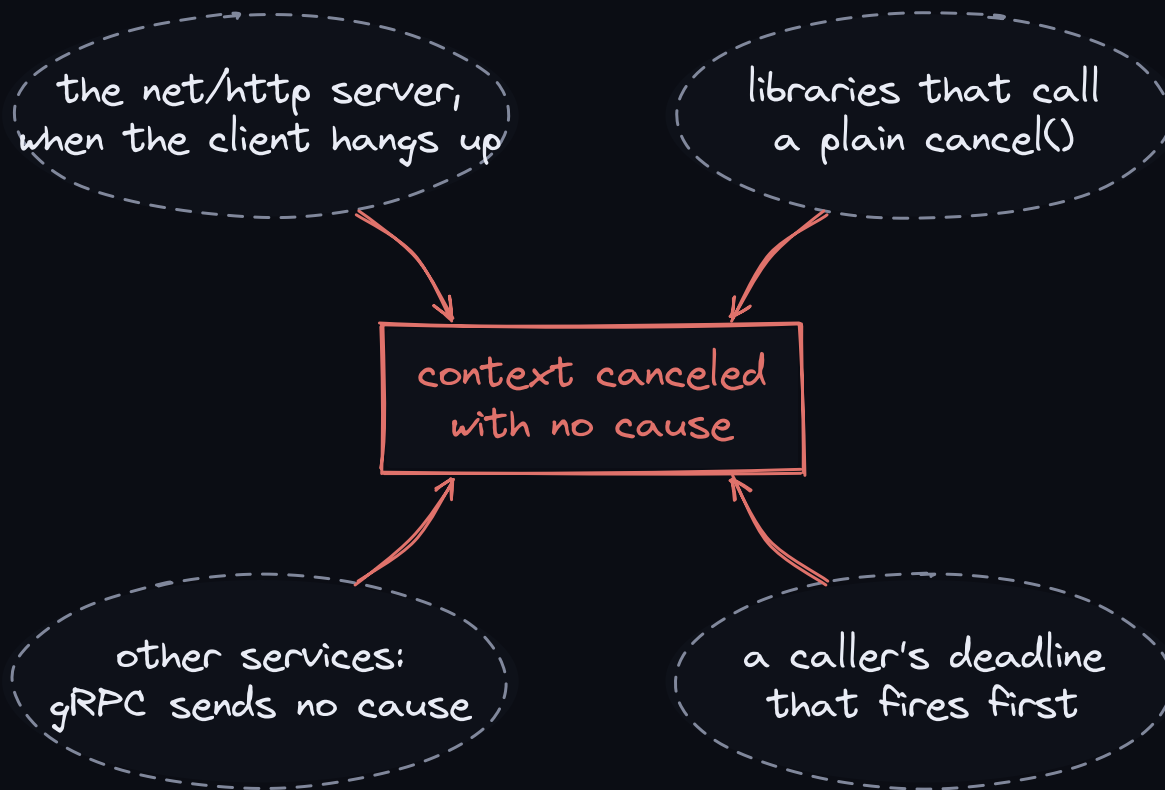
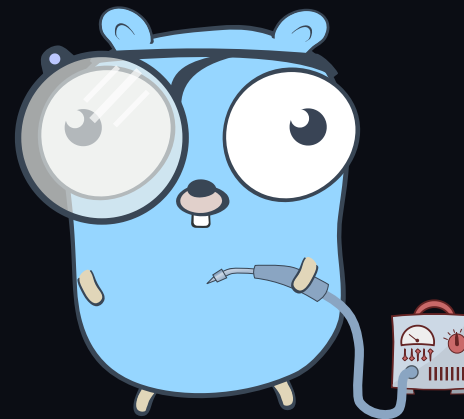
```
order/place.go:12:16: use of `context.WithTimeout` forbidden by pattern
```

```
  `^context\.(With(Cancel|Timeout|Deadline)(Cause)?$` (forbidigo)
```

The checklist I use in code review

- ☑ Pass `ctx` to each call, and don't store it in a struct
 - ☑ Check `ctx` in every loop and every blocking wait
 - ☑ Give background work `WithoutCancel` and a timeout
 - ☑ Put only optional request data in a context, and hide the key
 - ☑ Give every cancel a cause, including the normal return
 - ☑ Stack `WithCancelCause` on a timeout to cover every path
 - ☑ Wrap `Canceled` or `DeadlineExceeded` in every cause
 - ☑ Return `context.Cause(ctx)`, and log once in the handler
- how long work runs
- values
- causes

Some cancels still have no cause



Further reading

GO BLOG

- [Go Concurrency Patterns: Context](#), Sameer Ajmani, 2014
- [Go Concurrency Patterns: Pipelines and cancellation](#), Sameer Ajmani, 2014
- [Contexts and structs](#), Jean Barkhuysen and Matt T. Proud, 2021

MY POSTS

- [What canceled my Go context?](#)
- [Avoiding collisions in Go context keys](#)
- [What belongs in Go's context values?](#)
- [Is passing user ID through context an antipattern?](#)
- [Stacked log lines considered harmful](#)

GO ISSUES

- [#26356: context: ease debugging of where a context was canceled?](#)
- [#40221: context: add WithoutCancel](#)
- [#51365: context: add APIs for writing and reading cancelation cause](#)
- [#56661: context: add APIs for setting a cancelation cause when deadline or timer expires](#)

DOCS AND STYLE GUIDES

- [Package context](#)
- [Go 1.20 release notes: context](#)
- [Go 1.21 release notes: context](#)
- [Google Go style guide: Contexts](#)
- [Google Go style guide: Custom contexts](#)

LINTERS

- [go vet: lostcancel](#)
- [staticcheck: SA1012, SA1029](#)
- [golangci-lint: containedctx, contextcheck, errorlint, forbidigo](#)

Thanks

Questions?

REDNAFI.COM

