Handling Go Errors

Marwan Sulaiman
The New York Times
GopherCon, July 2019

github.com/marwan-at-work
@MarwanSulaiman

Background

- Backend Engineer at The New York Times for 2 years
- ~2 YR as a Front End Engineer at Work & Co
- Ruby On Rails Bootcamp at App Academy
- No CS Background 🙃

I'd like to make an assumption

Go was not your first programming language

Why is this important?

Learning concepts

vs

Learning syntax

Learning Syntax

How do I parse a JSON string in Go?

Learning Concepts

What is data serialization?

Learning Syntax

How do I import a library in Go?

Learning Concepts

What are dependencies?

Learning Syntax

How do I catch an error in Go?

Learning Concepts

What is error handling?

Conclusion

Learn like a beginner

The Concept Of Errors

Errors are values

PRO

you get to define the importance of errors

CON

you get to define the importance of errors

1. Errors are i/o

- Sometimes you need to read an error
- Sometimes you need to write an error

Context matters

- Is your program a CLI tool?
- Is your program a library?
- Is your program a long running system?
- Who is consuming your program? And How?

Goal: make a sandwich


TRADER
JOE'S


TRADER
JOE'S


TRADER JOE'S
WHOLE FOODS MARKET


TRADER JOE'S
WHOLE FOODS MARKET


TRADER JOE'S
WHOLE FOODS MARKET

```
package main

import (
    "markets/wholefoods"
    "markets/traderjoes"
    "markets/shoppers"
)
```

```
func BuyAvocados() (Ingredient, error)

func BuyEggs() (Ingredient, error)

func BuyBread() (Ingredient, error)
```

```
func getIngredients() ([]Ingredient, error) {
    avocados, err := wholefoods.BuyAvocados()

    boiledEggs, err := traderjoes.BuyEggs()

    bread, err := shoppers.BuyBread()

    return []Ingredient{avocados, boiledEggs, bread}, nil
}
```

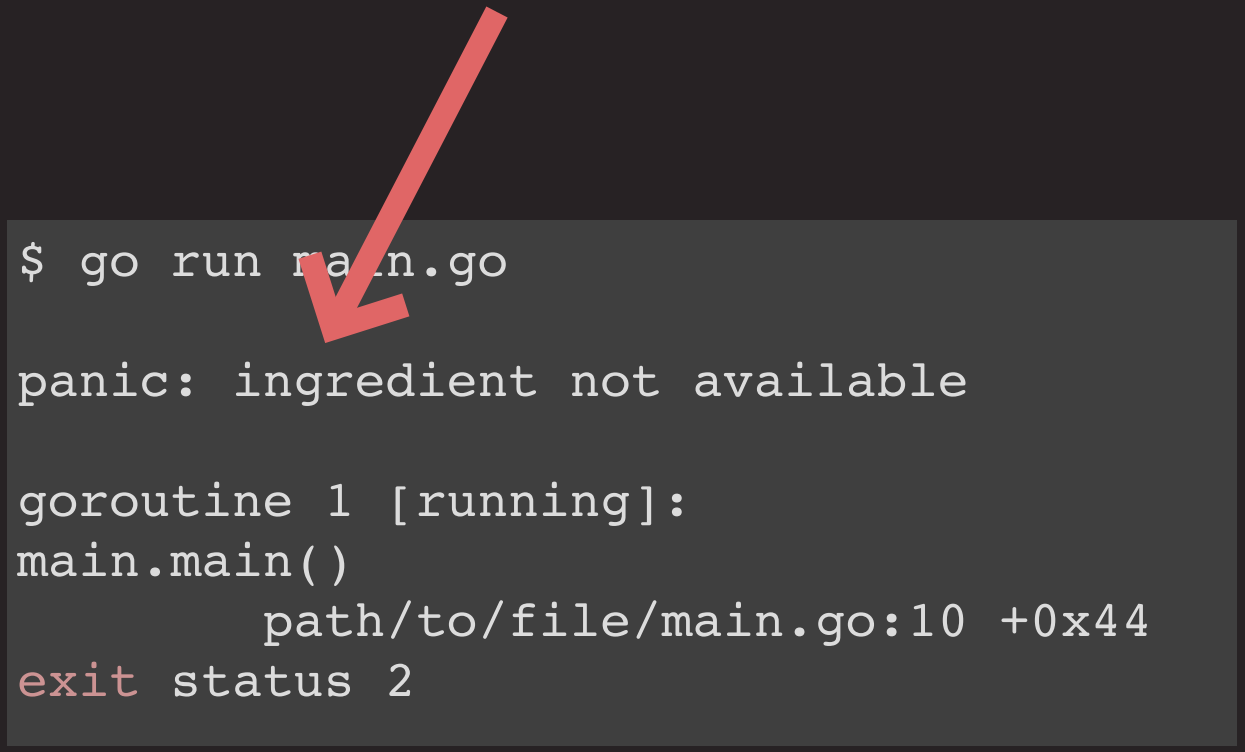
How do I handle errors in Go?

```
type error interface {
    Error() string
}
```

```
func getIngredients() ([]Ingredient, error) {
    avocados, err := wholefoods.BuyAvocados()
    if err != nil {
        return nil, err
    }

    boiledEggs, err := traderjoes.BuyEggs()
    if err != nil {
        return nil, err
    }

    bread, err := shoppers.BuyBread()
    if err != nil {
        return nil, err
    }

    return []Ingredient{avocados, boiledEggs, bread}, nil
}
```

```
func main() {
    ingredients, err := getIngredients()
    if err != nil {
        panic(err)
    }

    makeSandwich(ingredients)
}
```

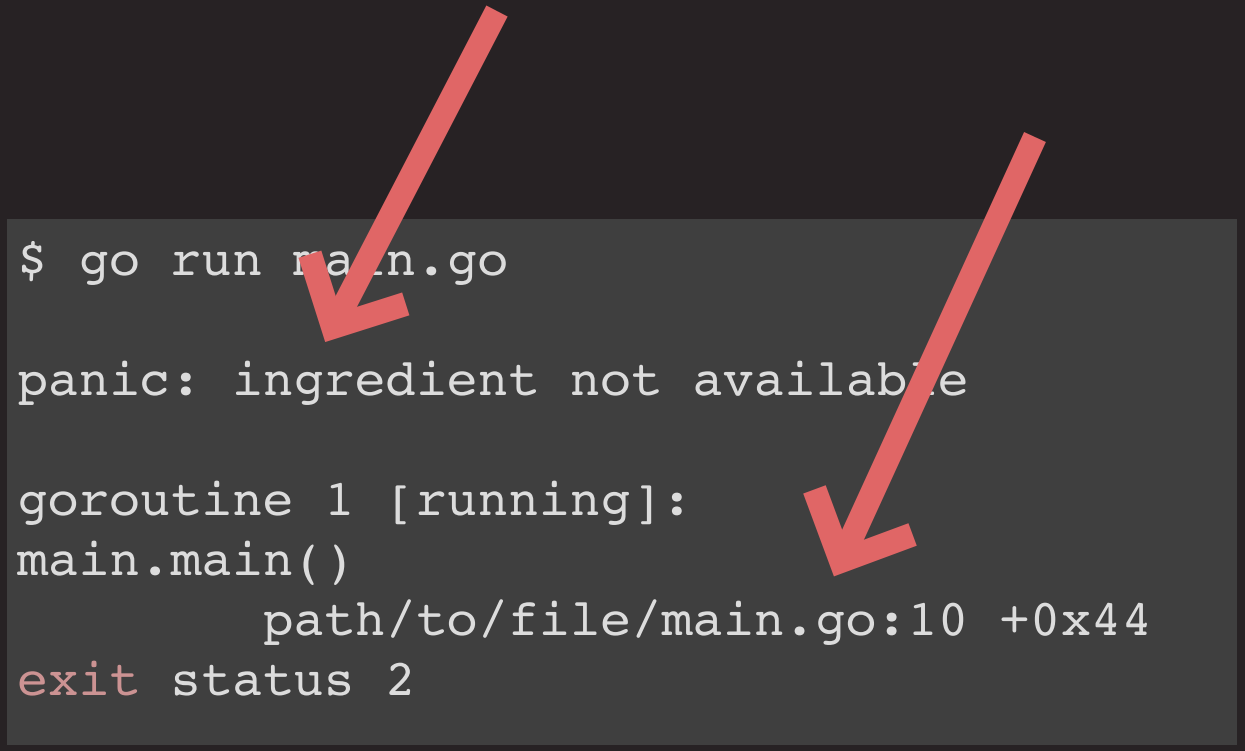
```
$ go run main.go

panic: ingredient not available

goroutine 1 [running]:
main.main()
        path/to/file/main.go:10 +0x44
exit status 2
```


$ go run main.go
panic: ingredient not available
goroutine 1 [running]:
main.main()
path/to/file/main.go:10 +0x44
exit status 2

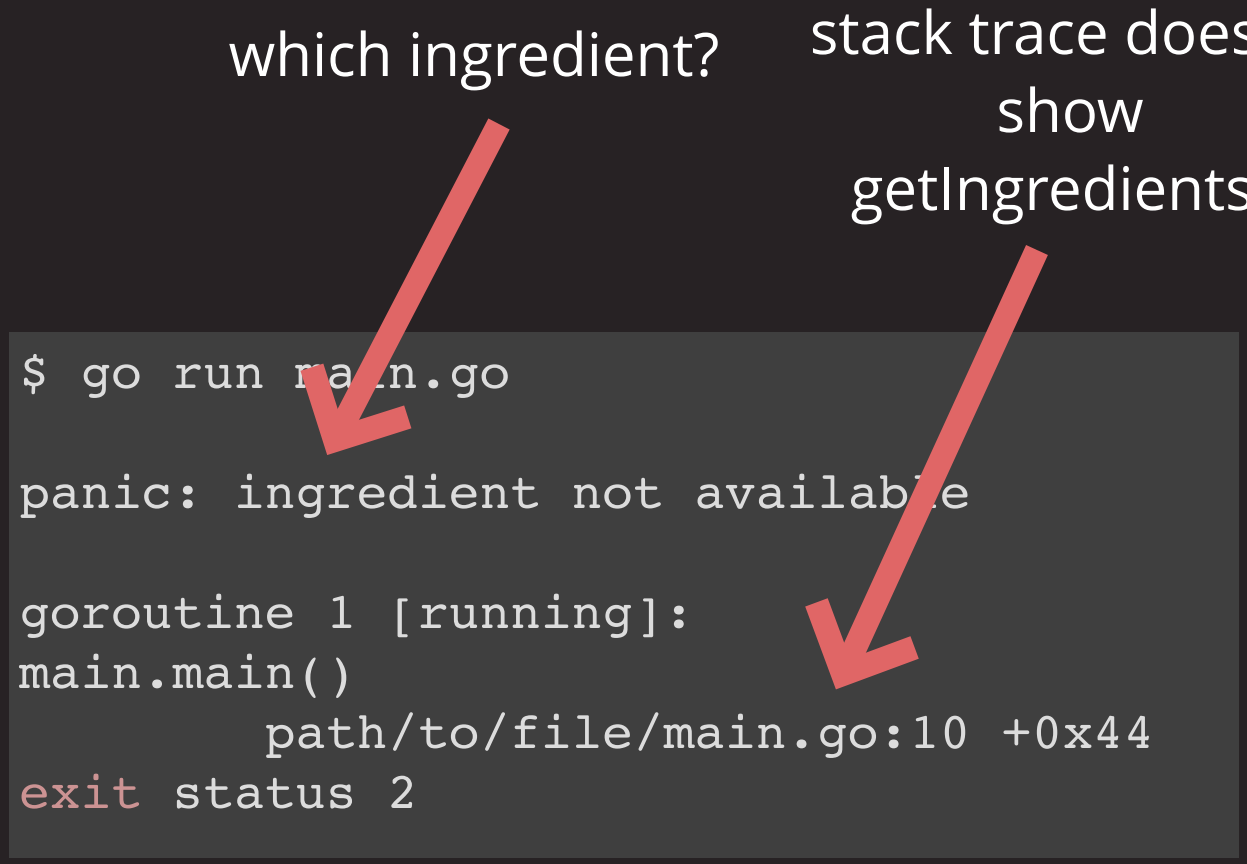
which ingredient?

```
$ go run main.go

panic: ingredient not available

goroutine 1 [running]:
main.main()
        path/to/file/main.go:10 +0x44
exit status 2
```

which ingredient?



```
$ go run main.go

panic: ingredient not available

goroutine 1 [running]:
main.main()
        path/to/file/main.go:10 +0x44
exit status 2
```


which ingredient?
stack trace doesn't show getIngredients()
$ go run main.go
panic: ingredient not available
goroutine 1 [running]:
main.main()
path/to/file/main.go:10 +0x44
exit status 2

2. Don't just check errors

- Handle them gracefully

How do I decorate errors in Go?

1

```
return fmt.Errorf("unique error message: %w", err)
```

2

```
import "github.com/pkg/errors"
```

```
return errors.Wrap(err, "unique error message")
```

```
import "github.com/pkg/errors"

func getIngredients() ([]Ingredient, error) {
    avocados, err := wholefoods.BuyAvocados()
    if err != nil {
        return nil, errors.Wrap(err, "could not buy avocados")
    }

    boiledEggs, err := traderjoes.BuyEggs()
    if err != nil {
        return nil, errors.Wrap(err, "could not buy eggs")
    }

    bread, err := shoppers.BuyBread()
    if err != nil {
        return nil, errors.Wrap(err, "could not buy bread")
    }

    return []Ingredient{avocados, boiledEggs, bread}, nil
}
```

```
$ go run main.go

panic: could not buy eggs: ingredient not available

goroutine 1 [running]:
main.main()
        path/to/file/main.go:10 +0x44
exit status 2
```

```
$ go run main.go

panic: could not buy eggs: ingredient not available

goroutine 1 [running]:
main.main()
        path/to/file/main.go:10 +0x44
exit status 2
```

```
$ go run main.go

panic: could not buy eggs: ingredient not available

goroutine 1 [running]:
main.main()
        path/to/file/main.go:10 +0x44
exit status 2
```

```
$ go run main.go

panic: could not buy eggs: ingredient not available

goroutine 1 [running]:
main.main()
        path/to/file/main.go:10 +0x44
exit status 2
```

```
$ go run main.go

panic: could not buy eggs: ingredient not available

goroutine 1 [running]:
main.main()
        path/to/file/main.go:10 +0x44
exit status 2
```

3. Stack traces are for disasters

- They're hard to read
- They're hard to parse
- At best, they say *where* an error went wrong, and not *why*

What if we want to act on an error?

```go
import "github.com/pkg/errors"

func getIngredients() ([]Ingredient, error) {
    avocados, err := wholefoods.BuyAvocados()
    if err != nil {
        return nil, errors.Wrap(err, "could not buy avocados")
    }

    boiledEggs, err := traderjoes.BuyEggs()
    if err == tradejoes.ErrNotAvailable {
        boiledEggs, err = wholefoods.BuyEggs()
    }

    if err != nil {
        return nil, errors.Wrap(err, "could not buy eggs")
    }

    bread, err := shoppers.BuyBread()
    if err != nil {
        return nil, errors.Wrap(err, "could not buy bread")
    }

    return []Ingredient{avocados, boiledEggs, bread}, nil
}
```

Takeaways:

- We can handle errors gracefully
- We can trace the error back to the code
- We can act upon an error

Is this enough?

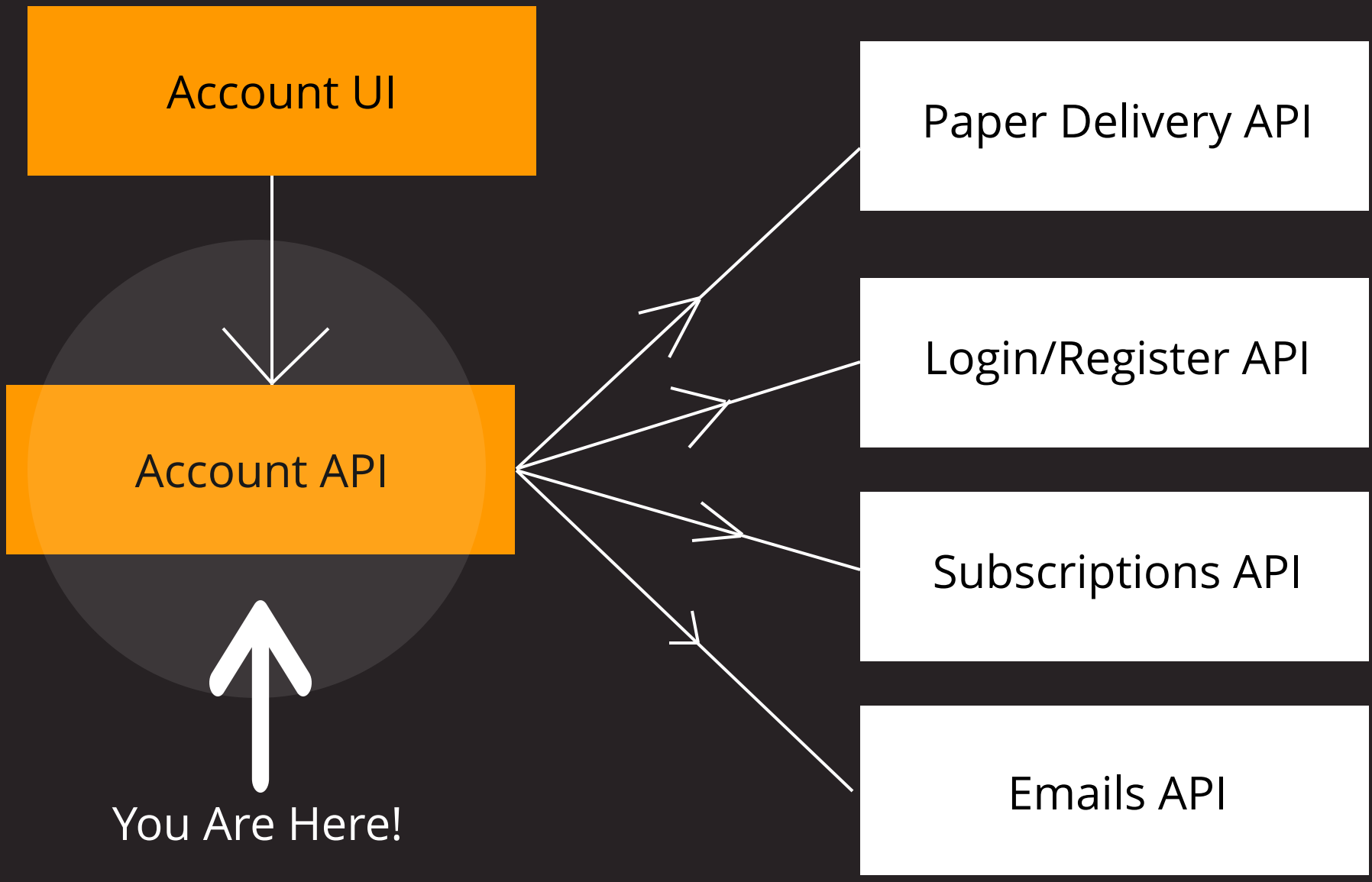
no

Further things you can do

- Categorize errors by severity.
- Categorize errors by type.
- Add application specific data.
- You can query all of the above.

Fast Forward 1 Year

The Architecture




TRADER
JOE'S


TRADER
JOE'S


TRADER JOE'S
WHOLE FOODS MARKET


TRADER JOE'S
WHOLE FOODS MARKET

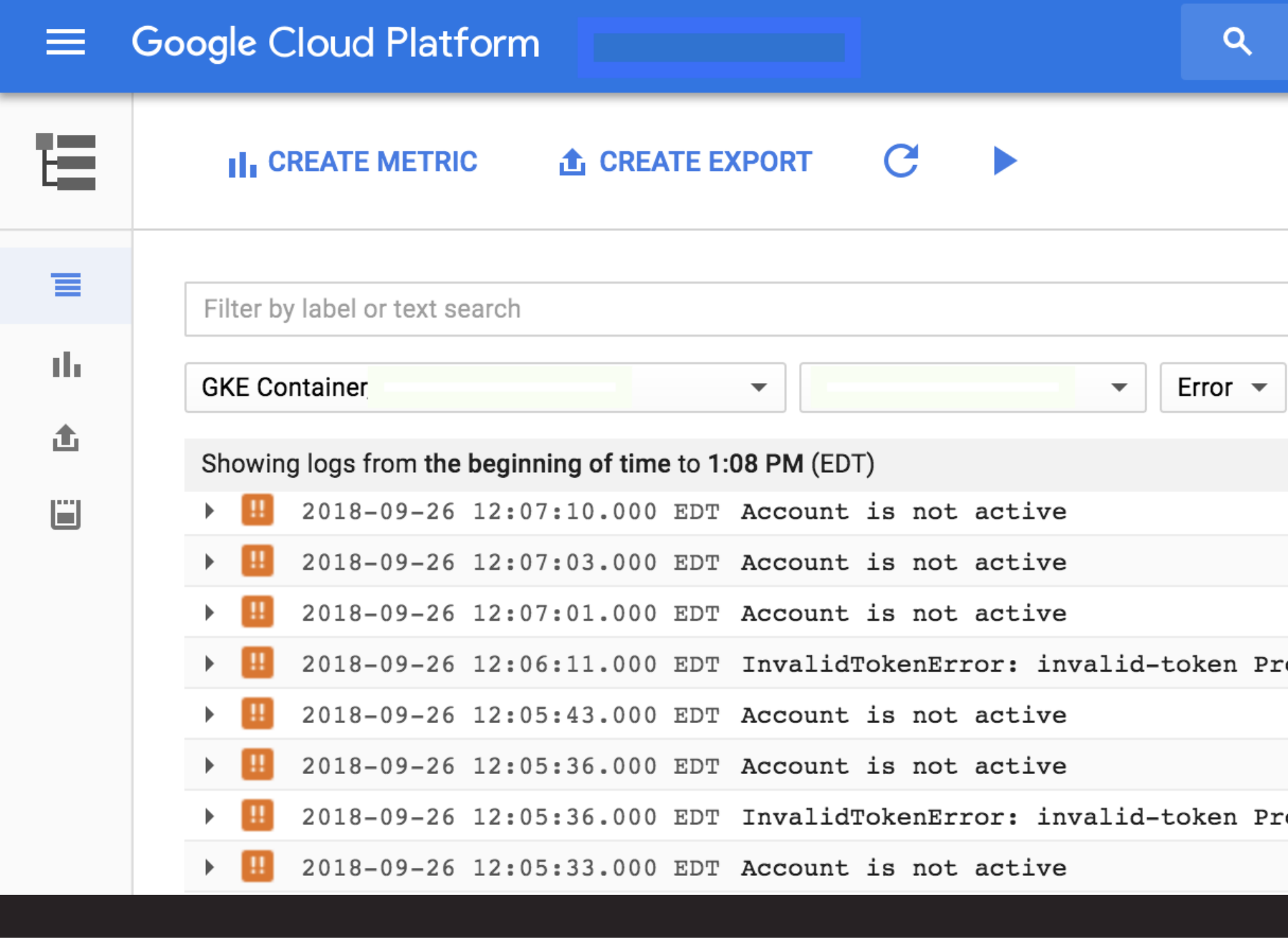

TRADER JOE'S
WHOLE FOODS MARKET

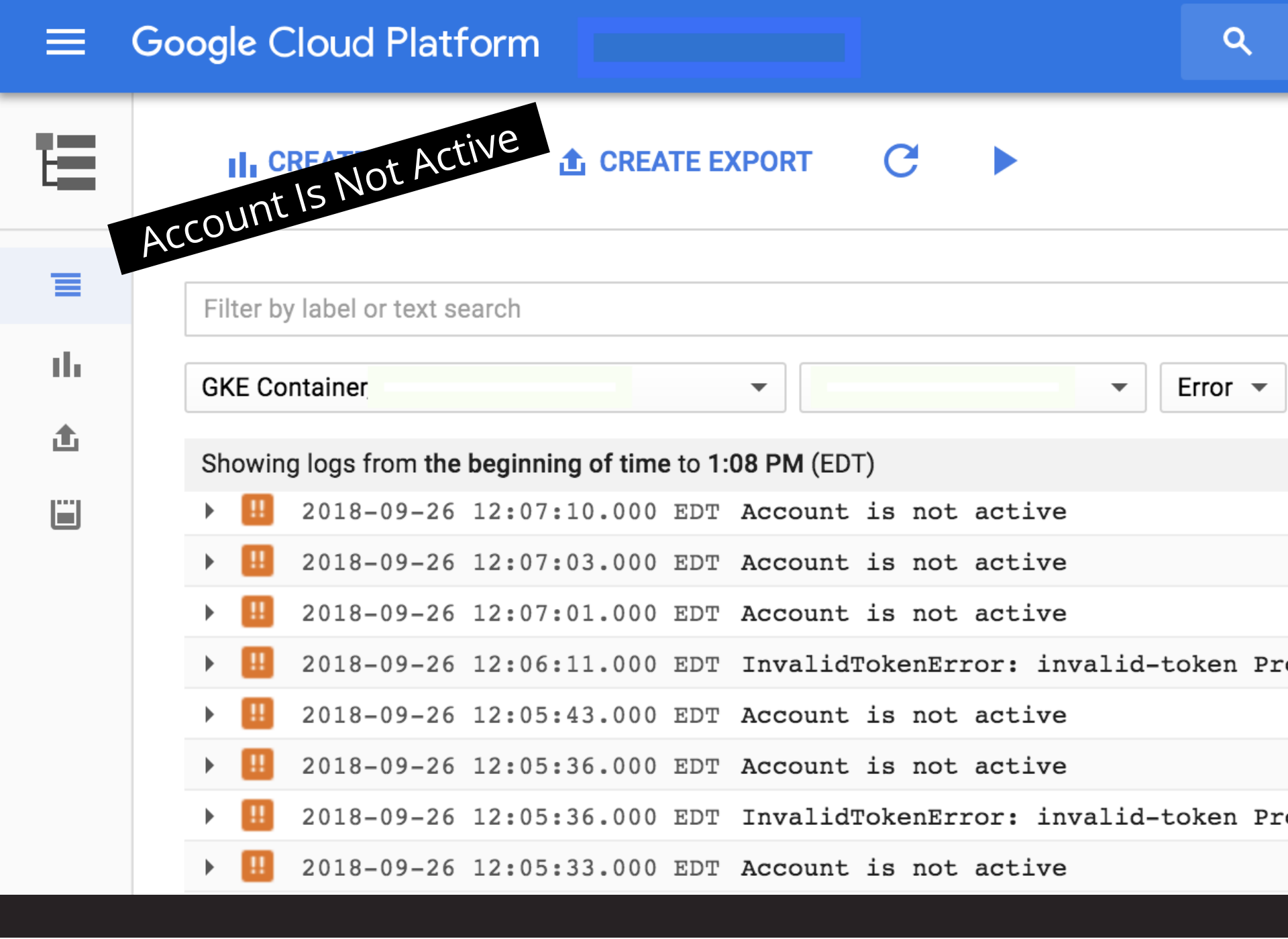
Not that different from making sandwiches

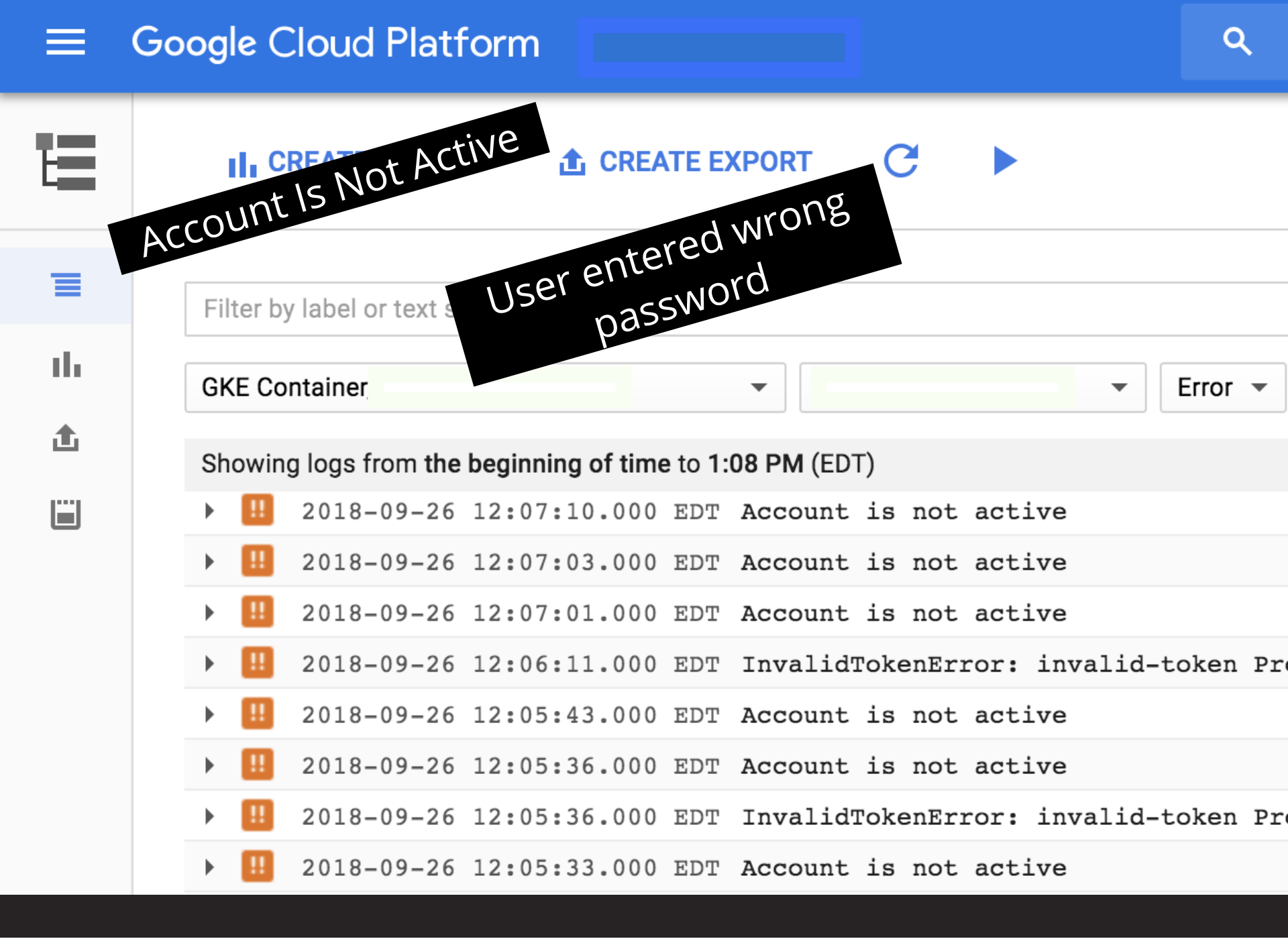
- One service that talks to a bunch of other services
- Instead of panicking, we log and monitor

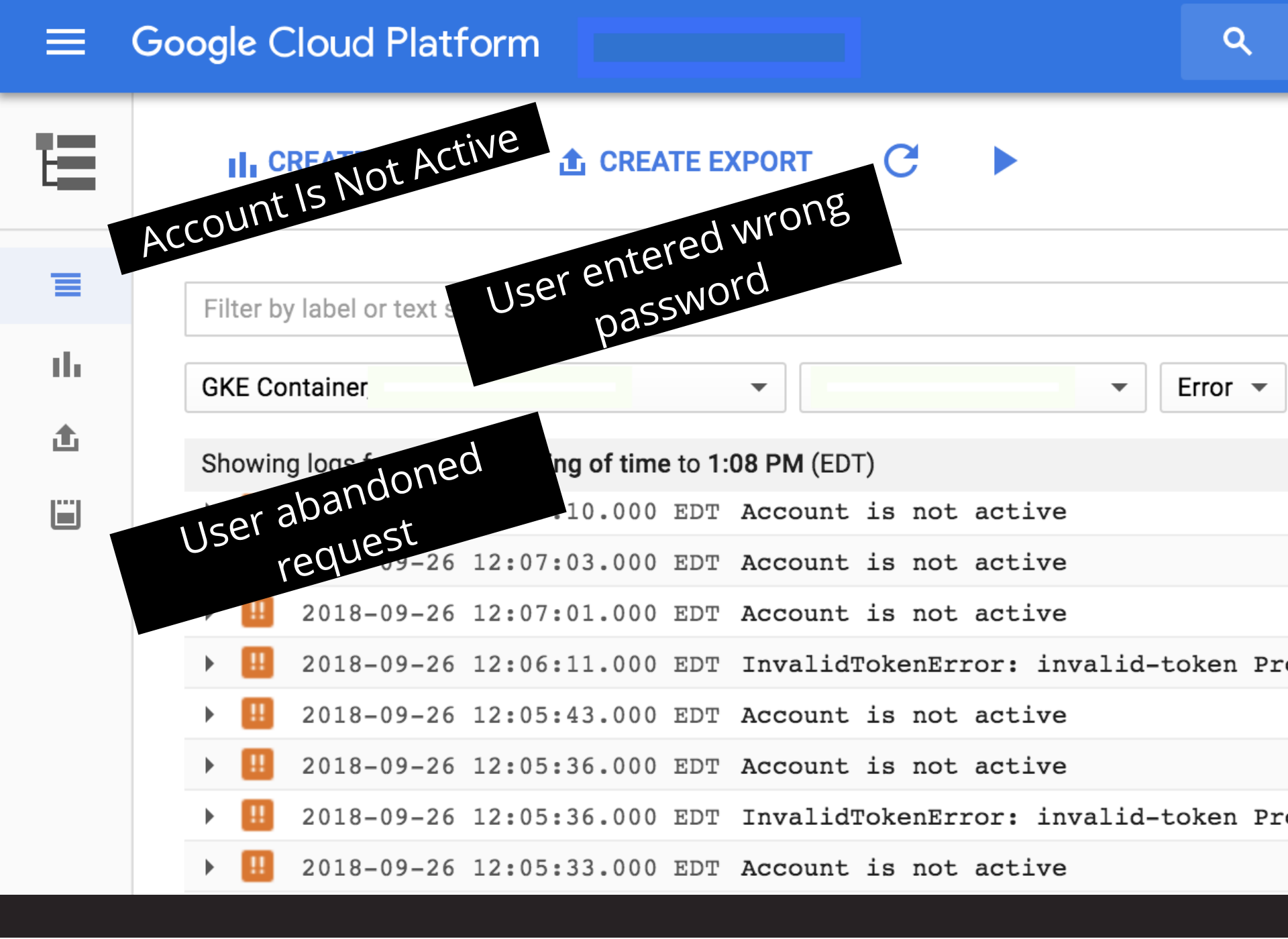
```
func getUser(userID string) (Subscription, time.Time, error) {
    err := loginService.Validate(userID)
    if err != nil {
        return err
    }

    subscription, err := subscriptionService.Get(userID)
    if err != nil {
        return err
    }

    deliveryTime, err := deliveryService.GetTodaysDeliveryTime(userID)
    if err != nil {
        return err
    }

    return subscription, deliveryTime, nil
}
```

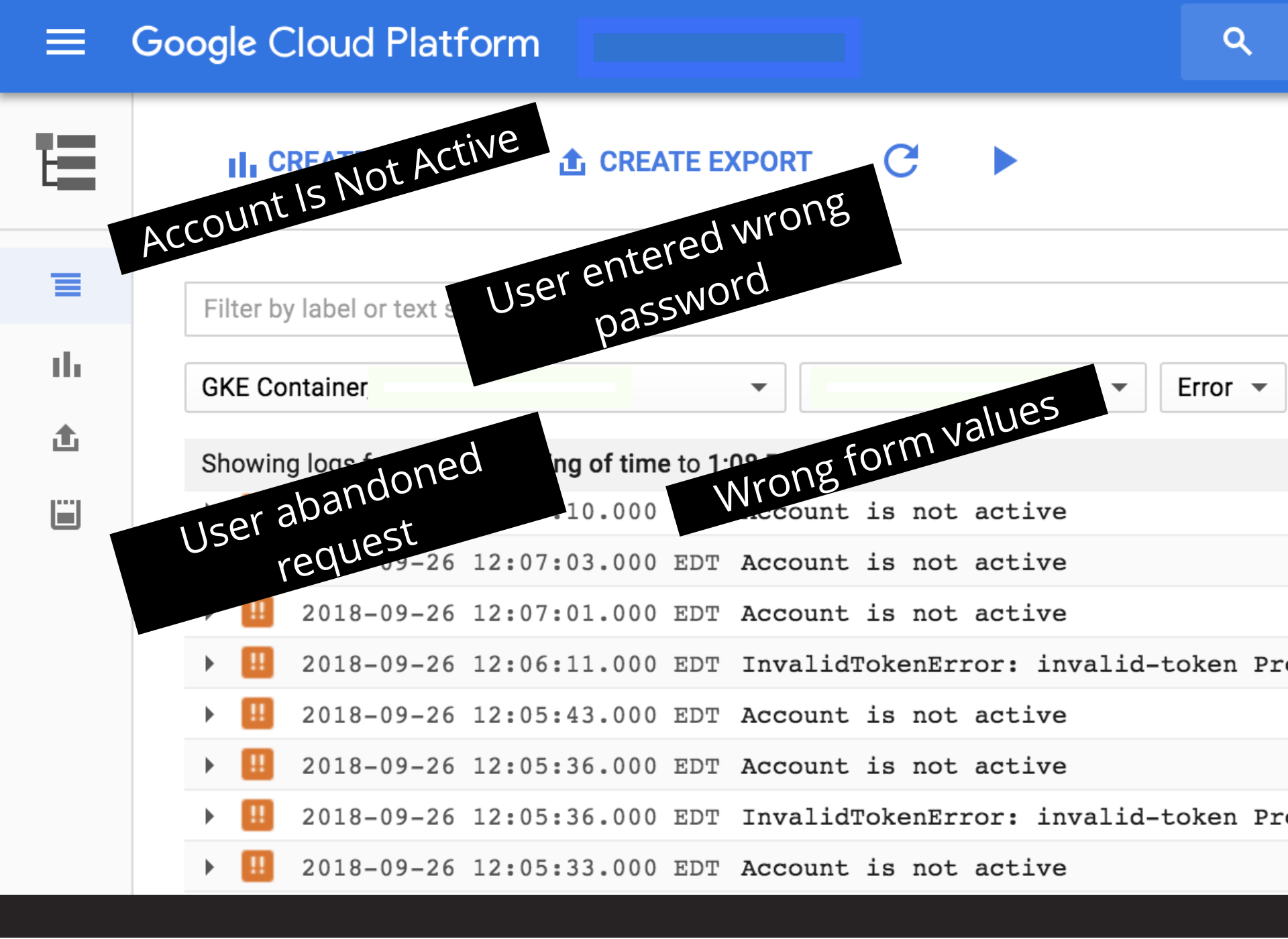
```
func getUserHandler(w http.ResponseWriter, r *http.Request) {
        // set up handler
        sub, deliveryTime, err := getUser(user)
        if err != nil {
                logger.Error(err)
                fmt.Fprint(w, "something went wrong")
                return
        }
        // return info to client
}
```

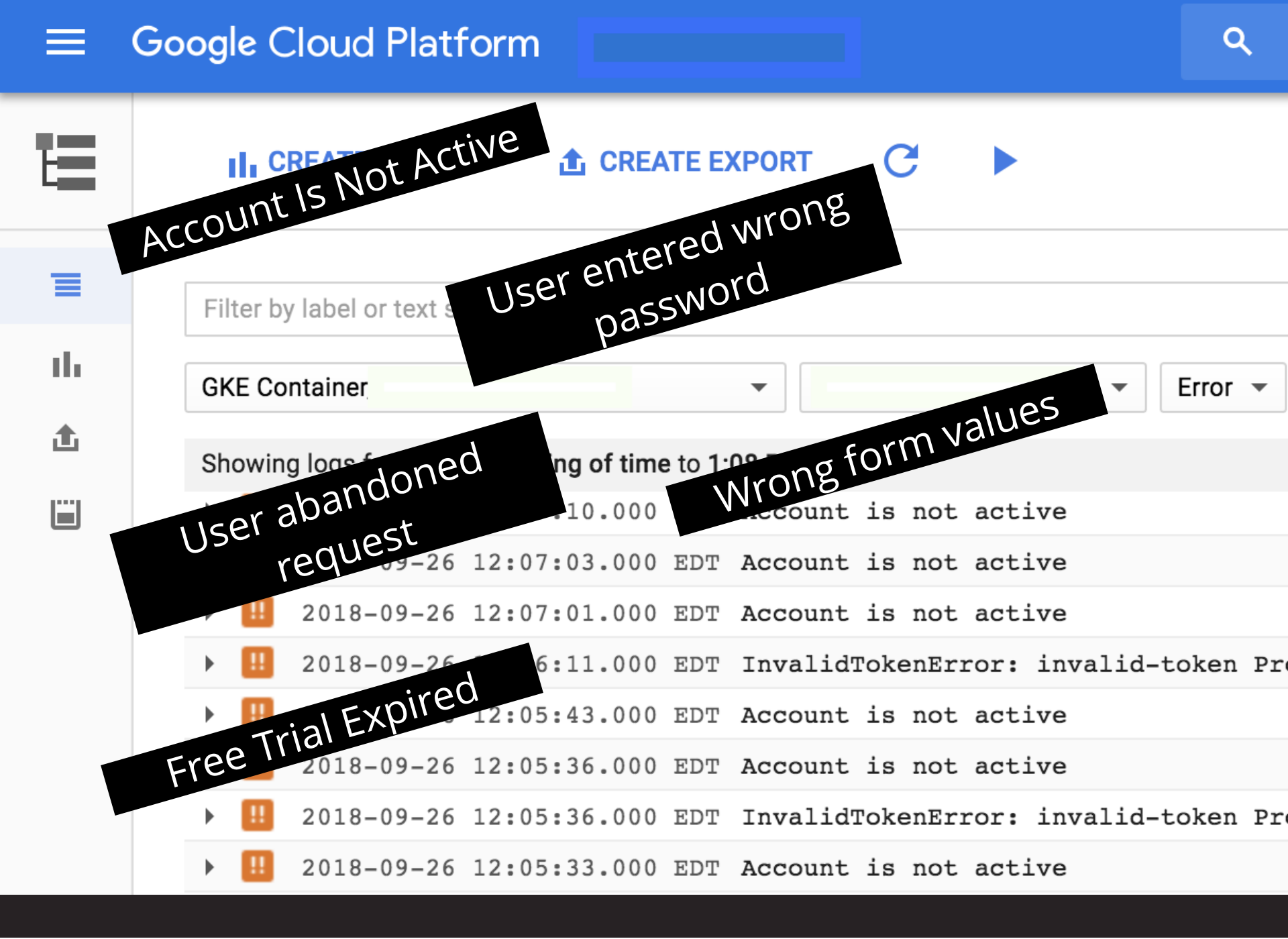

Google Cloud Platform
CREATE METRIC
CREATE EXPORT
Filter by label or text search
GKE Container
Error
Showing logs from the beginning of time to 1:08 PM (EDT)
2018-09-26 12:07:10.000 EDT Account is not active
2018-09-26 12:07:03.000 EDT Account is not active
2018-09-26 12:07:01.000 EDT Account is not active
2018-09-26 12:06:11.000 EDT InvalidTokenError: invalid-token Pr
2018-09-26 12:05:43.000 EDT Account is not active
2018-09-26 12:05:36.000 EDT Account is not active
2018-09-26 12:05:36.000 EDT InvalidTokenError: invalid-token Pr
2018-09-26 12:05:33.000 EDT Account is not active

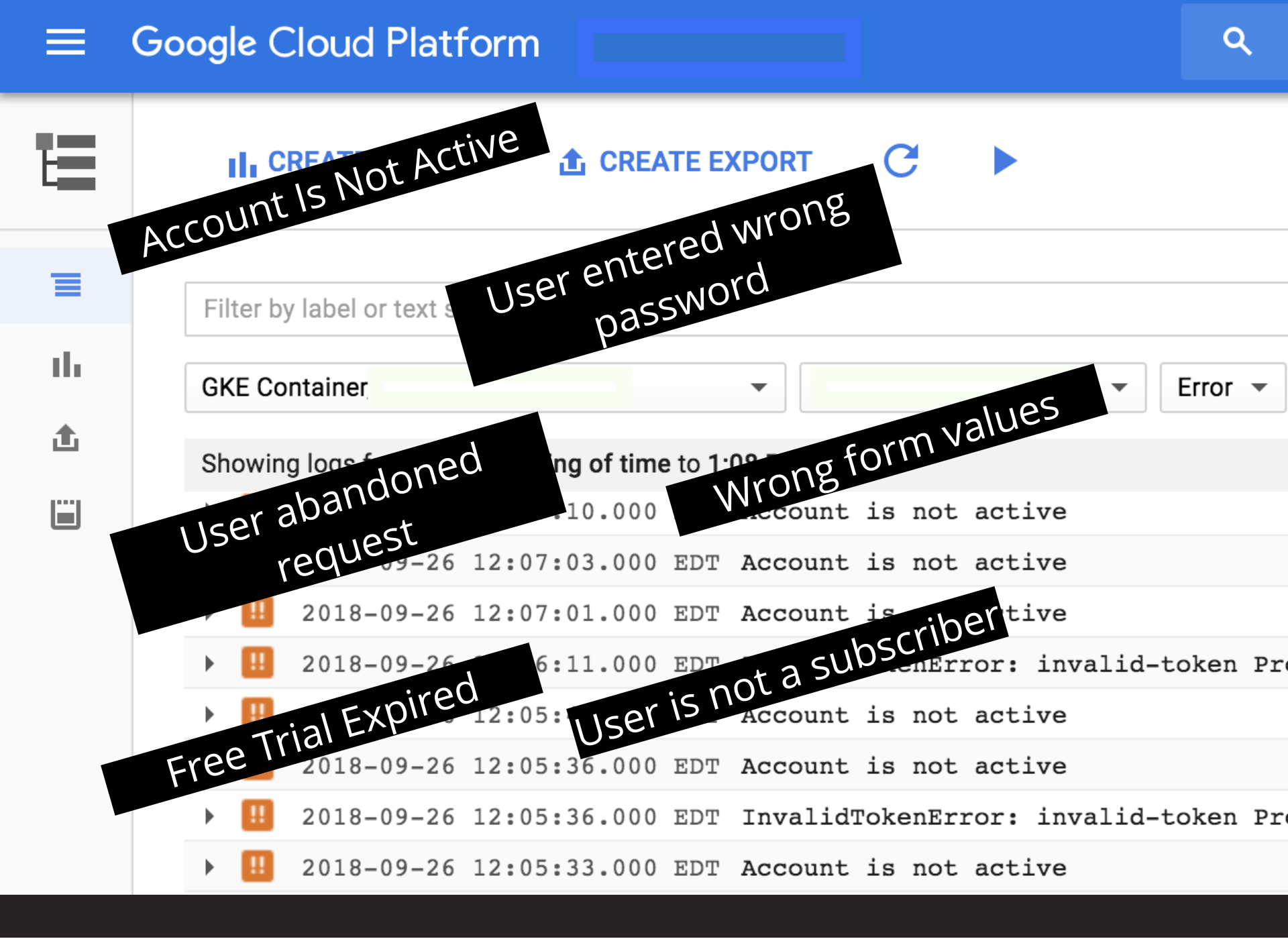

Google Cloud Platform
Account Is Not Active
CREATE EXPORT
Filter by label or text search
GKE Container
Error
Showing logs from the beginning of time to 1:08 PM (EDT)
2018-09-26 12:07:10.000 EDT Account is not active
2018-09-26 12:07:03.000 EDT Account is not active
2018-09-26 12:07:01.000 EDT Account is not active
2018-09-26 12:06:11.000 EDT InvalidTokenError: invalid-token
2018-09-26 12:05:43.000 EDT Account is not active
2018-09-26 12:05:36.000 EDT Account is not active
2018-09-26 12:05:36.000 EDT InvalidTokenError: invalid-token
2018-09-26 12:05:33.000 EDT Account is not active


Google Cloud Platform
CREATE EXPORT
Account Is Not Active
User entered wrong password
Filter by label or text s
GKE Container
Error
Showing logs from the beginning of time to 1:08 PM (EDT)
2018-09-26 12:07:10.000 EDT Account is not active
2018-09-26 12:07:03.000 EDT Account is not active
2018-09-26 12:07:01.000 EDT Account is not active
2018-09-26 12:06:11.000 EDT InvalidTokenError: invalid-token Pr
2018-09-26 12:05:43.000 EDT Account is not active
2018-09-26 12:05:36.000 EDT Account is not active
2018-09-26 12:05:36.000 EDT InvalidTokenError: invalid-token Pr
2018-09-26 12:05:33.000 EDT Account is not active


Google Cloud Platform
CREATE EXPORT
Account Is Not Active
User entered wrong password
User abandoned request
Filter by label or text s
GKE Container
Error
Showing logs f ng of time to 1:08 PM (EDT)
:10.000 EDT Account is not active
09-26 12:07:03.000 EDT Account is not active
2018-09-26 12:07:01.000 EDT Account is not active
2018-09-26 12:06:11.000 EDT InvalidTokenError: invalid-token Pr
2018-09-26 12:05:43.000 EDT Account is not active
2018-09-26 12:05:36.000 EDT Account is not active
2018-09-26 12:05:36.000 EDT InvalidTokenError: invalid-token Pr
2018-09-26 12:05:33.000 EDT Account is not active


Google Cloud Platform
CREATE EXPORT
Account Is Not Active
User entered wrong password
User abandoned request
Wrong form values
Filter by label or text s
GKE Container
Error
Showing logs f ng of time to 1:08
10.000 ccount is not active
09-26 12:07:03.000 EDT Account is not active
2018-09-26 12:07:01.000 EDT Account is not active
2018-09-26 12:06:11.000 EDT InvalidTokenError: invalid-token Pr
2018-09-26 12:05:43.000 EDT Account is not active
2018-09-26 12:05:36.000 EDT Account is not active
2018-09-26 12:05:36.000 EDT InvalidTokenError: invalid-token Pr
2018-09-26 12:05:33.000 EDT Account is not active


Google Cloud Platform
CREATE EXPORT
Account Is Not Active
User entered wrong password
User abandoned request
Wrong form values
Free Trial Expired
Filter by label or text s
GKE Container
Error
Showing logs f ng of time to 1:08
10.000 ccount is not active
09-26 12:07:03.000 EDT Account is not active
2018-09-26 12:07:01.000 EDT Account is not active
2018-09-26 6:11.000 EDT InvalidTokenError: invalid-token Pr
12:05:43.000 EDT Account is not active
2018-09-26 12:05:36.000 EDT Account is not active
2018-09-26 12:05:36.000 EDT InvalidTokenError: invalid-token Pr
2018-09-26 12:05:33.000 EDT Account is not active


Google Cloud Platform
CREATE EXPORT
Account Is Not Active
User entered wrong password
User abandoned request
Wrong form values
Free Trial Expired
User is not a subscriber
Filter by label or text s
GKE Container
Error
Showing logs f
ng of time to 1:08
10.000
Account is not active
09-26 12:07:03.000 EDT Account is not active
2018-09-26 12:07:01.000 EDT Account is
tive
2018-09-26
6:11.000 EDT
enError: invalid-token Pr
12:05:
Account is not active
2018-09-26 12:05:36.000 EDT Account is not active
2018-09-26 12:05:36.000 EDT InvalidTokenError: invalid-token Pr
2018-09-26 12:05:33.000 EDT Account is not active

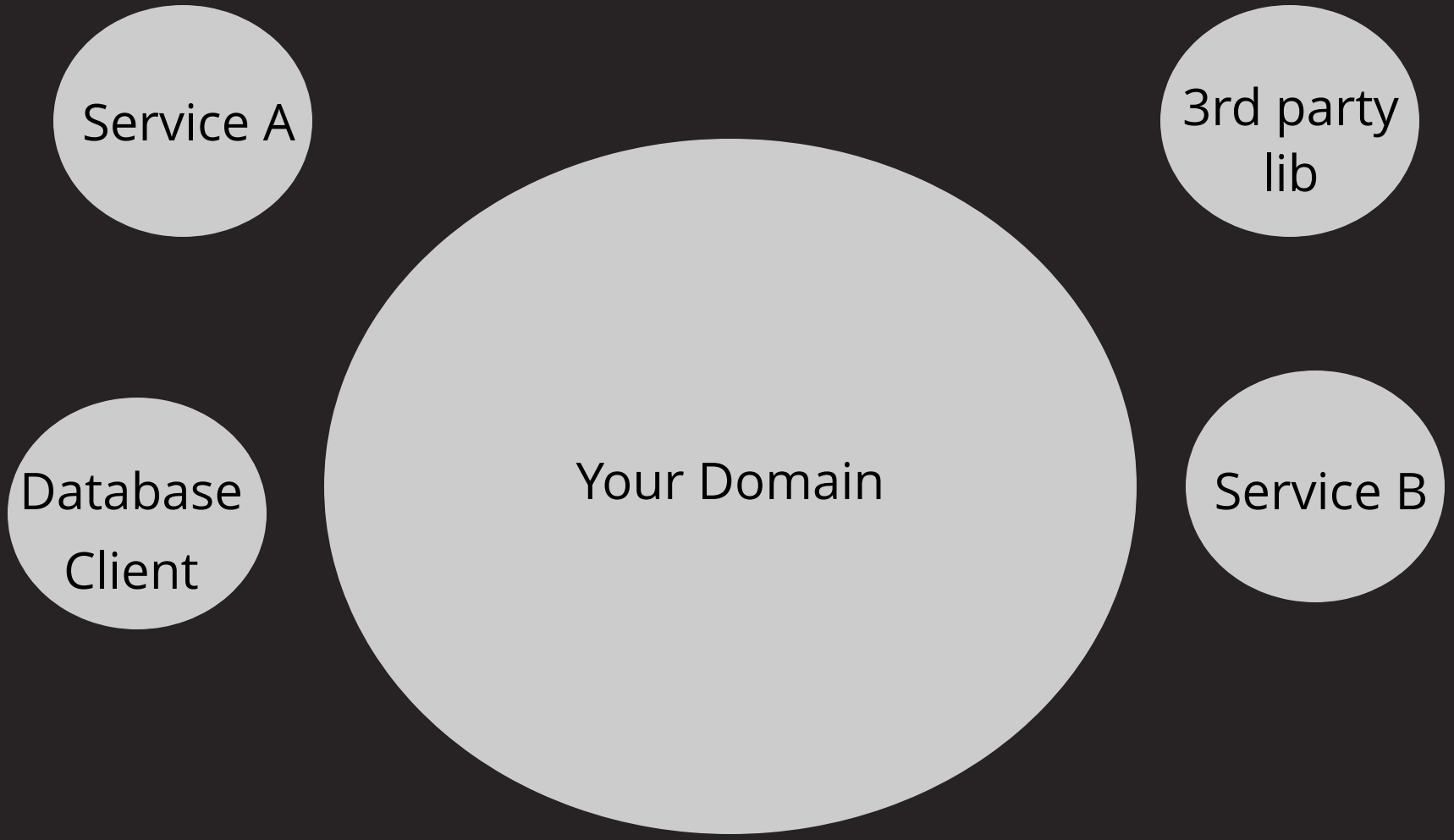
Refactor Goals:

- Can filter unexpected errors
- Group by error types
- Be able to answer specific questions

Looking for inspirations

1. Error Handling In Upsin - Andrew Gerrand/Rob Pike

2. Failure Is Your Domain - Ben Johnson


Service A
3rd party lib
Database Client
Your Domain
Service B

```
type DB interface {
  Get(id string) (*Record, error)
}
// Implementations: SQL, FS, MongoDB
```

```
func ForgotPassword(userID string, db DB) error {
    record, err := db.Get(userID)
    isNotFounnd := err == sql.ErrNoRows ||
        os.IsNotExist(err) ||
        mongo.ErrNoDocuments
    if isNotFounnd {
        return NotFoundError
    }
    // ...
}
```

```
package errors
```

```
package errors

type Error struct {

}
```

```
package errors

type Error struct {
    Op Op // operation
}
```

```
package errors

type Error struct {
    Op Op // operation
    Kind Kind // category of errors
}
```

```
package errors

type Error struct {
    Op Op // operation
    Kind Kind // category of errors
    Err error // the wrapped error
}
```

```
package errors

type Error struct {
    Op Op // operation
    Kind Kind // category of errors
    Err error // the wrapped error
    //... application specific fields
}
```

How do we construct an Error type?

```
if err != nil {
    return &errors.Error{
        Op: "getUser",
        Err: err,
    }
}
```

```
package errors

func E(args ...interface{}) error {
        e := &Error{}
        for _, arg := range args {
                switch arg := arg.(type) {
                case Op:
                        e.Op = arg
                case error:
                        e.Err = arg
                case Kind:
                        e.Kind = arg
                default:
                        panic("bad call to E")
                }
        }

        return e
}
```

```
if err != nil {
    return errors.E(op, err, errors.KindUnexpected)
}
```

What is an Operation?

```
type Op string
```

- A unique string describing a method or a function
- Multiple operations can construct a friendly stack trace.

```go
// app/account/account.go
package account

func getUser(userID string) (*User, err) {
    const op errors.Op = "account.getUser"
    err := loginService.Validate(userID)
    if err != nil {
        return nil, errors.E(op, err)
    }
    ...
}


// app/login/login.go
package login

func Validate(userID string) err {
    const op errors.Op = "login.Validate"
    err := db.LookUpUser(userID)
    if err != nil {
        return nil, errors.E(op, err)
    }
}
```

```go
// app/errors/errors.go
package errors

// Ops returns the "stack" of operations
// for each generated error.
func Ops(e *Error) []Op {
        res := []Op{e.Op}

        subErr, ok := e.Err.(*Error)
        if !ok {
                return res
        }

        res = append(res, Ops(subErr)...)

        return res
}
```

errors.Ops stack trace

```
["account.GetUser", "login.Validate", "db.LookUp"]
```

Classic stack trace

```
goroutine 19 [running]:
net/http.(*conn).serve.func1(0xc0000928c0)
        /usr/local/go/src/net/http/server.go:1746 +0xd0
panic(0x12459c0, 0x12eb450)
        /usr/local/go/src/runtime/panic.go:513 +0x1b9
db.LookUp(...)
        /Users/208581/go/src/app/db/lookup.go:25
login.Validate(...)
        /Users/208581/go/src/app/login/login.go:21
account.getUser(...)
        /Users/208581/go/src/app/account/account.go:17
main.getUserHandler(0x12ef4e0, 0xc000118000, 0xc000112000)
        /Users/208581/go/src/app/account/account.go:17
net/http.HandlerFunc.ServeHTTP(0x12bc838, 0x12ef4e0, 0xc000118000, 0xc000112000)
        /usr/local/go/src/net/http/server.go:1964 +0x44
net/http.(*ServeMux).ServeHTTP(0x149f720, 0x12ef4e0, 0xc000118000, 0xc000112000)
        /usr/local/go/src/net/http/server.go:2361 +0x127
net/http.serverHandler.ServeHTTP(0xc000098d00, 0x12ef4e0, 0xc000118000, 0xc000112000)
        /usr/local/go/src/net/http/server.go:2741 +0xab
net/http.(*conn).serve(0xc0000928c0, 0x12ef6e0, 0xc00009e240)
        /usr/local/go/src/net/http/server.go:1847 +0x646
created by net/http.(*Server).Serve
        /usr/local/go/src/net/http/server.go:2851 +0x2f5
```

Benefits of errors.Op

- A custom stack of your code only
- Easier to read
- Parsable and queryable.
- Not only can you know where something went wrong, but the impact it had on your entire application.

Query your stack trace

SELECT * FROM logs WHERE operations.include("login.Validate")

["account.getUser"],
["account.resetPassword"],
["homeDelivery.changeAddress"]

One final thought on Op

- You can make your stack even simpler by removing helper functions from the stack.

```
func helperFunction(op errors.Op, userID string) error {
    if somethingGoesWrong() {
        return errors.E(op, "something went wrong")
    }
}
```

Kind

- Groups all errors into smaller categories
- Can be predefined codes (http/gRPC)
- Or it can be your own defined codes

```
const (
    KindNotFound = http.StatusNotFound
    KindUnauthorized = http.StatusUnauthorized
    KindUnexpected = http.StatusUnexpected
)
```

Extracting a Kind from an error

```
func Kind(err error) codes.Code {
        e, ok := err.(*Error)
        if !ok {
                return KindUnexpected
        }

        if e.Kind != 0 {
                return e.Kind
        }

        return Kind(e.Err)
}
```

If your kind is an http status, then you can propagate an error back to the client

```
func getUserHandler(w http.ResponseWriter, r *http.Request) {
        // set up handler
        sub, deliveryTime, err := getUser(user)
        if err != nil {
                logger.Error(err)
                http.Error(w, "something went wrong", errors.Kind(err))
                return
        }
        // return info to client
}
```

The Severity Of Your Errors

```
type Error struct {
    ...
    Severity logrus.Level
}
```

```
func getUser(userID string) (*User, err) {
    const op errors.Op = "account.getUser"
    err := loginService.Validate(userID)
    if err != nil {
        return nil, errors.E(op, err, logrus.InfoLevel)
    }
    ...
}
```

```
func getUserHandler(w http.ResponseWriter, r *http.Request) {
        // set up handler
        sub, deliveryTime, err := getUser(user)
        if err != nil {
                logger.SystemErr(err)
                http.Error(w, "something went wrong", errors.Kind(err))
                return
        }
        // return info to client
}
```

```
func SystemErr(err error) {
        sysErr, ok := err.(*errors.Error)
        if !ok {
                logger.Error(err)
                return
        }

        entry := logger.WithFields(
                "operations", errors.Ops(sysErr),
                "kind", errors.Kind(err),
                // application specific data
        )

        switch errors.Level(err) {
        case Warning:
                entry.Warnf("%s: %v", sysErr.Op, err)
        case Info:
                entry.Infof("%s: %v", sysErr.Op, err)
        case Debug:
                entry.Debugf("%s: %v", sysErr.Op, err)
        default:
                entry.Errorf("%s: %v", sysErr.Op, err)
        }
}
```

Application Specific Data

```
type Error struct {
    // ... op, kind, etc
    Artist, Song string
}
```

```
type Error struct {
    // ... op, kind, etc
    ZipCode string
}
```

```
type Error struct {
    // ... op, kind, etc
    From, To string
}
```

```
type Error struct {
    // ... op, kind, etc
    OrderedAt time.Time
    RestaurantID string
}
```

Questions you can answer:

- Show me all paper delivery errors in zip code 22434
- Show me all food delivery errors by seafood restaurants
- Show me all errors that happened while trying to stream the latest Beyonce album

Takeaways:

- The error interface is intentionally simple.
- Design an errors package that makes sense to your application, and no one else

Conclusion

“ A big part of all programming, for real, is how you handle errors - Rob Pike

Thank You