The Go Programming Language

Quiz I

```
fruits := []string{"Apple", "Orage", "Plum", "Pomegranate", "Grape"}

some := fruits[1:3]

some = append(some, "Banana")

fmt.Println("fruits actual  : ", fruits)
```

Run

Exercise I

https://jsonplaceholder.typicode.com (https://jsonplaceholder.typicode.com)

```
/users
/albums
/photos
/todos
```

- commit & push to github
- structures

```
type User struct {
    ID       int    `json:"id"`
    Name     string `json:"name"`
    Username string `json:"username"`
    Email    string `json:"email"`
    Phone    string `json:"phone"`
}
```

Exercise I - structures

```
type Album struct {
    UserID int    `json:"userId"`
    ID     int    `json:"id"`
    Title  string `json:"title"`
}

type Photo struct {
    AlbumID      int    `json:"albumId"`
    ID           int    `json:"id"`
    Title        string `json:"title"`
    URL          string `json:"url"`
    ThumbnailURL string `json:"thumbnailUrl"`
}
```

Error handling

Eliminate error handling by eliminating errors

Count Lines: using Reader

```
func CountLines(r io.Reader) (int, error) {
    var (
        br    = bufio.NewReader(r)
        lines int
        err   error
    )

    for {
        _, err = br.ReadString('\n')
        lines++
        if err != nil {
            break
        }
    }

    if err != io.EOF {
        return 0, err
    }

    return lines, nil
}
```

Run

Count Lines: using Scanner

```
func CountLines(r io.Reader) (int, error) {
    sc := bufio.NewScanner(r)
    lines := 0

    for sc.Scan() {
        lines++
    }
    return lines, sc.Err()
}
```

Run

Count Lines: refactor

```
type Scanner struct {
    br  *bufio.Reader
    err error
}

func (s *Scanner) Scan() bool {...}

func (s *Scanner) Error() error {...}

func (s *Scanner) read() {
    _, s.err = s.br.ReadString('\n')
}

func New(r io.Reader) *Scanner {
    return &Scanner{br: bufio.NewReader(r)}
}
```

Exercise: WriteResponse

- result

```
HTTP/1.1 555 account not found.
Content-Type: application/json

this is a body
```

- structures

```
type Header struct {
    Key, Value string
}

type Status struct {
    Code   int
    Reason string
}
```

Exercise: WriteResponse

```
func WriteResponse(w io.Writer, st Status, headers []Header, body io.Reader) error {
    _, err := fmt.Fprintf(w, "HTTP/1.1 %d %s\r\n", st.Code, st.Reason)
    if err != nil {
        return err
    }
    // source code : https://bit.ly/2nrcVjK
    for _, h := range headers {
        _, err := fmt.Fprintf(w, "%s: %s\r\n", h.Key, h.Value)
        if err != nil {
            return err
        }
    }

    if _, err := fmt.Fprint(w, "\r\n"); err != nil {
        return err
    }

    _, err = io.Copy(w, body)
    return err
}
```

Exercise: WriteResponse

```
func main() {
    // souce code : https://bit.ly/2nrcVjK
    var buf bytes.Buffer
    st := Status{Code: 555, Reason: "account not found."}
    headers := []Header{
        {"Content-Type", "application/json"},
    }
    body := strings.NewReader("this is a body")

    err := WriteResponse(&buf, st, headers, body)

    fmt.Printf("buf: \n%s\n", buf)
    fmt.Println("error:", err)
}
```

Exercise: WriteResponse - create struct

```
type Header struct {
    Key, Value string
}

type Status struct {
    Code   int
    Reason string
}

type errWrite struct {
    io.Writer
    err       error
}

func (e *errWrite) Write(buf []byte) (int, error) {
    if e.err != nil {
        return 0, e.err
    }

    var n int
    n, e.err = e.Writer.Write(buf)
    return n, nil
}
```

Exercise: WriteResponse - using struct

```
func WriteResponse(w io.Writer, st Status, headers []Header, body io.Reader) error {
    ew := &errWrite{Writer: w}
    fmt.Fprintf(ew, "HTTP/1.1 %d %s\r\n", st.Code, st.Reason)

    for _, h := range headers {
        fmt.Fprintf(ew, "%s: %s\r\n", h.Key, h.Value)
    }

    fmt.Fprintf(ew, "\r\n")
    io.Copy(ew, body)
    return ew.err
}
```

Error in Go

Rob Pike

"A big part of all programming, for real, is how you handle errors"

Errors are value

Go Proverb

"Don't just check errors, handle them gracefully"

The error type

The error type is an interface type. An error variable represents any value that can describe itself as a string. Here is the interface's declaration:

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

error package

The most commonly-used error implementation is the errors package's unexported errorString type.

```
// errorString is a trivial implementation of error.
type errorString struct {
    s string
}

func (e *errorString) Error() string {
    return e.s
}
```

errors.New function

You can construct one of these values with the errors.New function. It takes a string that it converts to an errors.errorString and returns as an error value.

```
// New returns an error that formats as the given text.
func New(text string) error {
    return &errorString{text}
}
```

Here's how you might use errors.New:

```
func Sqrt(f float64) (float64, error) {
    if f < 0 {
        return 0, errors.New("math: square root of negative number")
    }
    // implementation
}
```

The fmt package formats an error value by calling its Error() string method.

A caller passing a negative argument to Sqrt receives a non-nil error value (whose concrete representation is an errors.errorString value). The caller can access the error string ("math: square root of...") by calling the error's Error method, or by just printing it:

```
f, err := Sqrt(-1)
if err != nil {
    fmt.Println(err)
}
```

It is the error implementation's responsibility to summarize the context.
The error returned by os.Open formats as "open /etc/passwd: permission denied," not just "permission denied."
The error returned by our Sqrt is missing information about the invalid argument.

the fmt package's Errorf

It formats a string according to Printf's rules and returns it as an error created by errors.New.

```
if f < 0 {
    return 0, fmt.Errorf("math: square root of negative number %g", f)
}
```

In many cases fmt.Errorf is good enough, but since error is an interface, you can use arbitrary data structures as error values, to allow callers to inspect the details of the error. For instance, our hypothetical callers might want to recover the invalid argument passed to Sqrt. We can enable that by defining a new error implementation instead of using errors.errorString:

```
type NegativeSqrtError float64

func (f NegativeSqrtError) Error() string {
    return fmt.Sprintf("math: square root of negative number %g", float64(f))
}
```

Exercise II

Implement your own error type
- error type name **BusinessError**
- print it by using fmt.Println
"business error: xxx"

Exercise II - example

```
type BusinessError string
func (b BusinessError) Error() string {
  return fmt.Sprintf("business error: %s", b)
}
```

Might be a **struct**

```
type BusinessError struct {
  s string
}

func (b BusinessError) Error() string {
  return fmt.Sprintf("business error: %s", b.s)
}
```

What's wrong with this code?

```
func AuthenticateRequest(r *Request) error {
  err := authen(r.User)
  if err != nil {
    return err
  }
  return nil
}
```

this is a result

```
"No such file or directory"
```

There is no information in the file or which line it error.

Annotating errors

```
func AuthenticateRequest(r *Request) error {
  err := authen(r.User)
  if err != nil {
    return fmt.Errorf("authenitcation failed: %v", err)
  }
  return nil
}
```

Ignoring an error

```
func Write(w io.Writer, buf []byte) {
  w.Write(buf)
}
```

Don't do this.

Handling an error twice

making more than one decision in response to a single error.
It also not a good thing.

```
func Write(w io.Writer, buf []byte) error {
  _ err := w.Write(buf)
    if err != nil {
        // annotated error goes to log file
        log.Println("unable to write:", err)

        // unannotated error returned to caller
        return err
    }

  return nil
}
```

error return to caller, who might also log it and return it.
we will get stack of all duplicate line in our log file.
the top of program we will get original error without any context.

errros Go1.13

The new function **As** finds the first error in a given error's chain (sequence of wrapped errors) that matches a given target's type, and if so, sets the target to that error value.

The new function **Is** reports whether a given error value matches an error in another's chain.

The new function **Unwrap** returns the result of calling Unwrap on a given error, if one exists.

You need to be prepared that errors you get may be wrapped.

Wrapping error

use fmt.Errorf with %w

```
var err error
var errOriginal error
errOriginal = &os.PathError{Op: "read", Path: "/a/b", Err: errors.New("read error")}

err = fmt.Errorf("create new error: %v", errOriginal)
fmt.Printf("error value: %T\n", err)
fmt.Println("error value:", err.Error())

err = fmt.Errorf("wrapped error: %w", errOriginal)
fmt.Printf("error value: %T\n", err)
fmt.Println("error value:", err.Error())
```

Unwrap error

use errros.Unwrap

```
errInner := errors.Unwrap(err)
```

```
fmt.Printf("error valuex: %T\n", errInner)
```

errors.Is

If you currently compare errors using ==, use **errors.Is** instead.
Example:

```
if err == io.ErrUnexpectedEOF
```

becomes

```
if errors.Is(err, io.ErrUnexpectedEOF)
```

- Checks of the form if err != nil need not be changed.
- Comparisons to *io.EOF* need not be changed, because *io.EOF* should never be wrapped

errors.As

If you check for an error type using a type assertion or type switch, use **errors.As** instead. Example:

```
if e, ok := err.(*os.PathError); ok
```

becomes

```
var e *os.PathError
if errors.As(err, &e)
```

- Also use this pattern to check whether an error implements an interface.

(This is one of those rare cases when a pointer to an interface is appropriate.)
- Rewrite a type switch as a sequence of if-elses.

Exercise

```
func insertInfo() error {
  // do work here.
  return sql.ErrTxDone
}

func updateInfoV1() error {
  // do work here.
  return fmt.Errorf("update info error: %v", sql.ErrTxDone)
}

func updateInfoV2() error {
  // do work here.
  return fmt.Errorf("update info error: %w", sql.ErrTxDone)
}

func readFile() error {
  return &os.PathError{Op: "read"}
}

func readFileV2() error {
  err := &os.PathError{Op: "read"}
  return fmt.Errorf("wrapped error: %w", err)
}
```

Error handling in Upspin

github upspin (https://github.com/upspin/upspin/blob/master/errors/errors.go)

Refactor your application

Go Modules and SemVer

- **Workshop**
- docker go.1.13
- create from scratch
- update/ add new dependency / revert dependency (go.sum)
- migrate to go mod

```
- go mod vendor for not have a internet
```

- go proxy

Semantic Versioning

Given a version number **MAJOR.MINOR.PATCH**, increment the:

- **MAJOR** version when you make incompatible API changes,
- **MINOR** version when you add functionality in a backwards compatible manner, and
- **PATCH** version when you make backwards compatible bug fixes.

Semantic Versioning

- **MAJOR** => Breaking
- **MINOR** => Features
- **PATCH** => Patch

Exercise

- api - service
- user profile - dependency

REST API

- refactor
- example project test every layer

Testing in Go

Exercise : Create call mongo db

- add users.Create
- add integratoin test
- add mongo
- refactor mongo

Mongo DB Driver package

mongo-go-driver (https://github.com/mongodb/mongo-go-driver/)

```
go get go.mongodb.org/mongo-driver@v1.0.0-rc2
```

Go Concurrency Patterns

What is concurrency?

Concurrency is the composition of independently executing computations.

Concurrency is a way to structure software, particularly as a way to write clean code that interacts well with the real world.

It is not parallelism.

Concurrency is not parallelism

Concurrency is not parallelism, although it enables parallelism.

If you have only one processor, your program can still be concurrent but it cannot be parallel.

On the other hand, a well-written concurrent program might run efficiently in parallel on a multiprocessor. That property could be important...

For more on that distinction, see the link below. Too much to discuss here.

golang.org/s/concurrency-is-not-parallelism (http://golang.org/s/concurrency-is-not-parallelism)

A model for software construction

Easy to understand.

Easy to use.

Easy to reason about.

You don't need to be an expert!

(Much nicer than dealing with the minutiae of parallelism (threads, semaphores, locks, barriers, etc.))

A boring function

We need an example to show the interesting properties of the concurrency primitives.

To avoid distraction, we make it a boring example.

```
func main() {
    c := make(chan string)
    go boring("boring!", c)

    for i := 0; i < 5; i++ {
        fmt.Printf("You day: %q\n", <-c)
    }

    fmt.Println("You're boring; I'm leaving.")
}

func boring(msg string, c chan string) {
    for i := 0; ; i++ {
        c <- fmt.Sprintf("%s %d", msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

Run

Running it

The say function runs on forever.

Make the intervals between messages unpredictable (still under a second).

```
func main() {
    say("Hello!")
}

func say(msg string) {
    for i := 0; ; i++ {
        fmt.Println(msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

Run

Ignoring it

The go statement runs the function as usual, but doesn't make the caller wait.

It launches a goroutine.

The functionality is analogous to the & on the end of a shell command.

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func main() {
    go say("Hello go!")
}
```

Run

Ignoring it a little less

When main returns, the program exits and takes the saying function down with it.

We can hang around a little, and on the way show that both main and the launched goroutine are running.

```
func main() {
    go say("boring!")
    fmt.Println("I'm listening.")
    time.Sleep(2 * time.Second)
    fmt.Println("You're boring; I'm leaving.")
}
```

Run

Goroutines

What is a goroutine? It's an independently executing function, launched by a go statement.

It has its own call stack, which grows and shrinks as required.

It's very cheap. It's practical to have thousands, even hundreds of thousands of goroutines.

It's not a thread.

There might be only one thread in a program with thousands of goroutines.

Instead, goroutines are multiplexed dynamically onto threads as needed to keep all the goroutines running.

But if you think of it as a very cheap thread, you won't be far off.

Communication

Our boring examples cheated: the main function couldn't see the output from the other goroutine.

It was just printed to the screen, where we pretended we saw a conversation.

Real conversations require communication.

Channels

A channel in Go provides a connection between two goroutines, allowing them to communicate.

```
// Declaring and initializing.
var c chan int
c = make(chan int)
// or
c := make(chan int)
```

```
// Sending on a channel.
c <- 1
```

```
// Receiving from a channel.
// The "arrow" indicates the direction of data flow.
value = <-c
```

Using channels

A channel connects the main and say goroutines so they can communicate.

```
func main() {
    c := make(chan string)
    go say("Hello!", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("You say: %q\n", <-c) // Receive expression is just a value.
    }
    fmt.Println("You're pretty; I love you.")
}
```

Run

```
func say(msg string, c chan string) {
    for i := 0; ; i++ {
        c <- fmt.Sprintf("%s %d", msg, i) // Expression to be sent can be any suitable value.
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

Synchronization

When the main function executes <–c, it will wait for a value to be sent.

Similarly, when the say function executes c <– value, it waits for a receiver to be ready.

A sender and receiver must both be ready to play their part in the communication. Otherwise we wait until they are.

Thus channels both communicate and synchronize.

An aside about buffered channels

Note for experts: Go channels can also be created with a buffer.

Buffering removes synchronization.

Buffering makes them more like Erlang's mailboxes.

Buffered channels can be important for some problems but they are more subtle to reason about.

We won't need them today.

The Go approach

Don't communicate by sharing memory, share memory by communicating.

Generator: function that returns a channel

Channels are first-class values, just like strings or integers.

```
    c := say("Yo!") // Function returning a channel.
    for i := 0; i < 5; i++ {
        fmt.Printf("You say: %q\n", <-c)
    }
    fmt.Println("Yo what's up.")
```

Run

```
func say(msg string) <-chan string { // Returns receive-only channel of strings.
    c := make(chan string)
    go func() { // We launch the goroutine from inside the function.
        for i := 0; ; i++ {
            c <- fmt.Sprintf("%s %d", msg, i)
            time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
        }
    }()
    return c // Return the channel to the caller.
}
```

Channels as a handle on a service

Our boring function returns a channel that lets us communicate with the boring service it provides.

We can have more instances of the service.

```
func main() {
    joe := say("Joe")
    ann := say("Ann")
    for i := 0; i < 5; i++ {
        fmt.Println(<-joe)
        fmt.Println(<-ann)
    }
    fmt.Println("You're both rich")
}
```

Run

Multiplexing

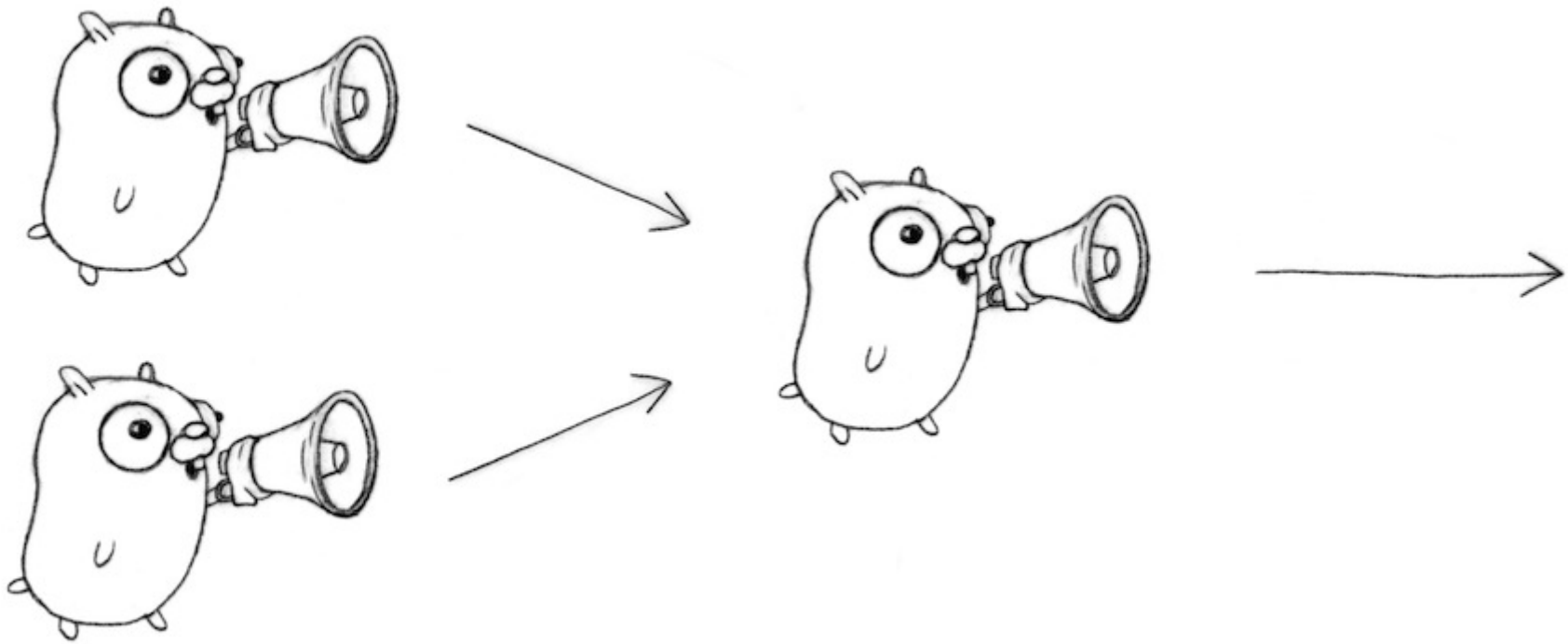
These programs make Joe and Ann count in lockstep.
We can instead use a fan-in function to let whosoever is ready talk.

```
func fanIn(input1, input2 <-chan string) <-chan string {
    c := make(chan string)
    go func() { for { c <- <-input1 } }()
    go func() { for { c <- <-input2 } }()
    return c
}
```

```
func main() {
    c := fanIn(boring("Joe"), boring("Ann"))
    for i := 0; i < 10; i++ {
        fmt.Println(<-c)
    }
    fmt.Println("You're both boring; I'm leaving.")
}
```

Run

Fan-in



Restoring sequencing

Send a channel on a channel, making goroutine wait its turn.

Receive all messages, then enable them again by sending on a private channel.

First we define a message type that contains a channel for the reply.

```
type Message struct {
    str string
    wait chan bool
}
```

Restoring sequencing.

Each speaker must wait for a go-ahead.

```
    for i := 0; i < 5; i++ {
        msg1 := <-c; fmt.Println(msg1.str)
        msg2 := <-c; fmt.Println(msg2.str)
        msg1.wait <- true
        msg2.wait <- true
    }
```

```
    waitForIt := make(chan bool) // Shared between all messages.
```

```
            c <- Message{ fmt.Sprintf("%s: %d", msg, i), waitForIt }
            time.Sleep(time.Duration(rand.Intn(2e3)) * time.Millisecond)
            <-waitForIt
```

Run

Select

A control structure unique to concurrency.

The reason channels and goroutines are built into the language.

Select

The select statement provides another way to handle multiple channels.
It's like a switch, but each case is a communication:
- All channels are evaluated.
- Selection blocks until one communication can proceed, which then does.
- If multiple can proceed, select chooses pseudo-randomly.
- A default clause, if present, executes immediately if no channel is ready.

```
    select {
    case v1 := <-c1:
        fmt.Printf("received %v from c1\n", v1)
    case v2 := <-c2:
        fmt.Printf("received %v from c2\n", v1)
    case c3 <- 23:
        fmt.Printf("sent %v to c3\n", 23)
    default:
        fmt.Printf("no one was ready to communicate\n")
    }
```

Fan-in again

Rewrite our original fanIn function. Only one goroutine is needed. Old:

```
func fanIn(input1, input2 <-chan string) <-chan string {
    c := make(chan string)
    go func() { for { c <- <-input1 } }()
    go func() { for { c <- <-input2 } }()
    return c
}
```

Fan-in using select

Rewrite our original fanIn function. Only one goroutine is needed. New:

```
func fanIn(input1, input2 <-chan string) <-chan string {
    c := make(chan string)
    go func() {
        for {
            select {
            case s := <-input1:  c <- s
            case s := <-input2:  c <- s
            }
        }
    }()
    return c
}
```

Run

Timeout using select

The time.After function returns a channel that blocks for the specified duration. After the interval, the channel delivers the current time, once.

```
func main() {
    c := boring("Joe")
    for {
        select {
        case s := <-c:
            fmt.Println(s)
        case <-time.After(1 * time.Second):
            fmt.Println("You're too slow.")
            return
        }
    }
}
```

Run

Timeout for whole conversation using select

Create the timer once, outside the loop, to time out the entire conversation.
(In the previous program, we had a timeout for each message.)

```
func main() {
    c := boring("Joe")
    timeout := time.After(5 * time.Second)
    for {
        select {
        case s := <-c:
            fmt.Println(s)
        case <-timeout:
            fmt.Println("You talk too much.")
            return
        }
    }
}
```

Run

Quit channel

We can turn this around and tell Joe to stop when we're tired of listening to him.

```
quit := make(chan bool)
c := boring("Joe", quit)
for i := rand.Intn(10); i >= 0; i-- { fmt.Println(<-c) }
quit <- true
```

```
        select {
        case c <- fmt.Sprintf("%s: %d", msg, i):
            // do nothing
        case <-quit:
            return
        }
```

Run

Receive on quit channel

How do we know it's finished? Wait for it to tell us it's done: receive on the quit channel

```
quit := make(chan string)
c := boring("Joe", quit)
for i := rand.Intn(10); i >= 0; i-- { fmt.Println(<-c) }
quit <- "Bye!"
fmt.Printf("Joe says: %q\n", <-quit)
```

```
        select {
        case c <- fmt.Sprintf("%s: %d", msg, i):
            // do nothing
        case <-quit:
            cleanup()
            quit <- "See you!"
            return
        }
```

Run

Systems software

Go was designed for writing systems software.
Let's see how the concurrency features come into play.

Summary

In just a few simple transformations we used Go's concurrency primitives to convert a

- slow
- sequential
- failure-sensitive

program into one that is

- fast
- concurrent
- replicated
- robust.

Go build

- build

```
go build
```

- build with specific file name

```
go build -o goapp
```

- build with specific OS and Architecture

```
GOOS=linux GOARCH=386 go build -o goapp
```

Go deployment (Heroku)

- Add Procfile to project

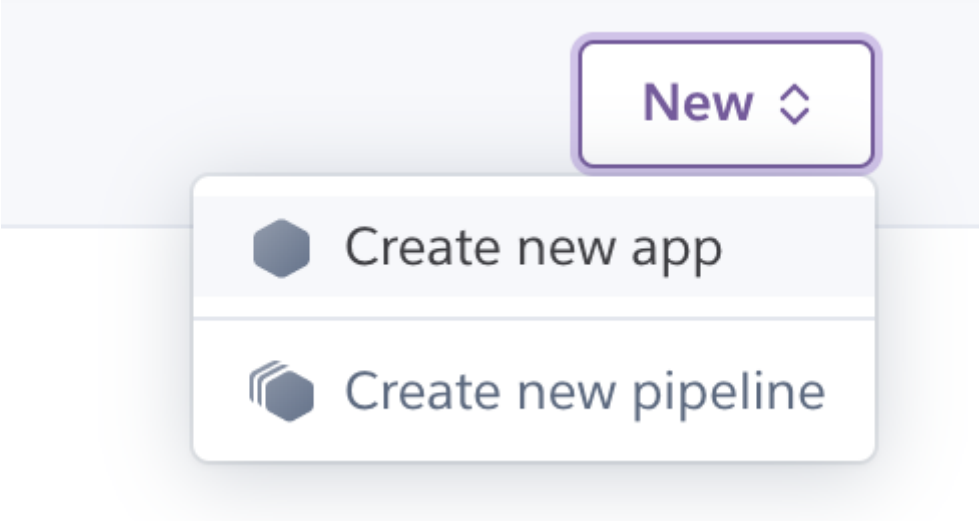
```
web: bin/<go-app-build-file>
```

- Ex:

```
web: bin/go-ajnong
```

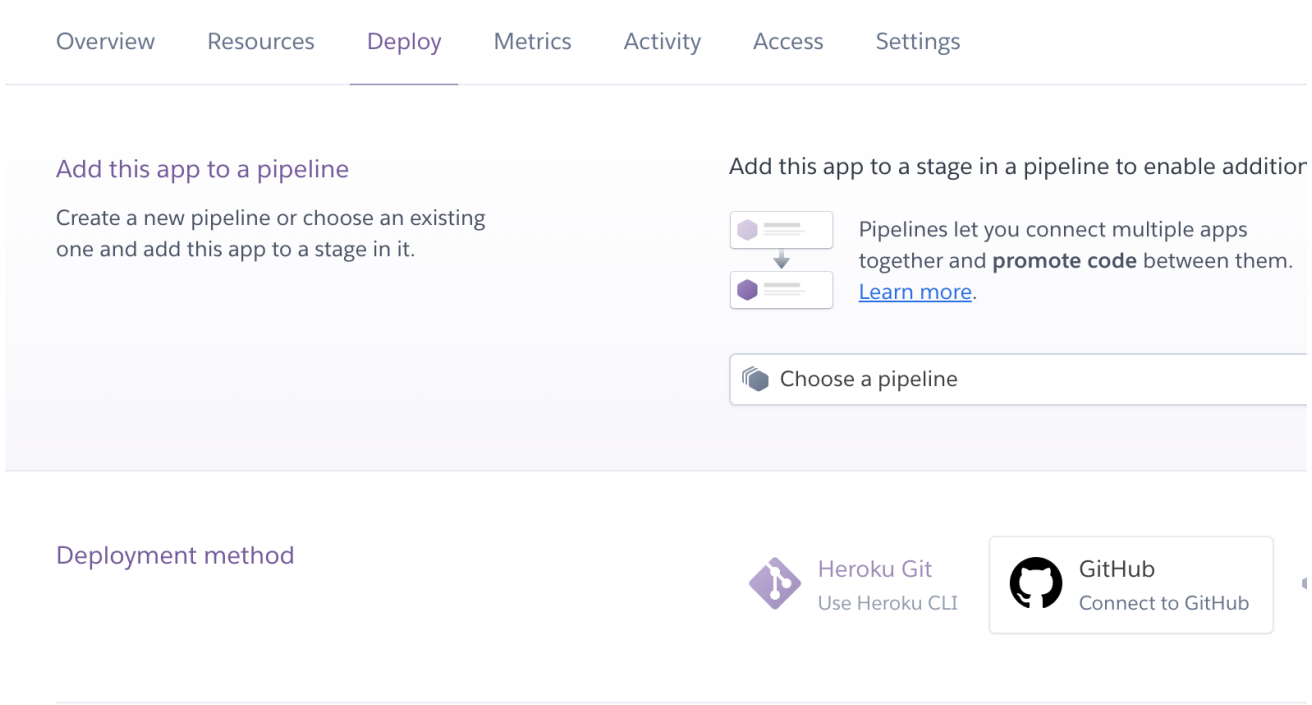
Go deployment (Heroku)

- Create heroku app



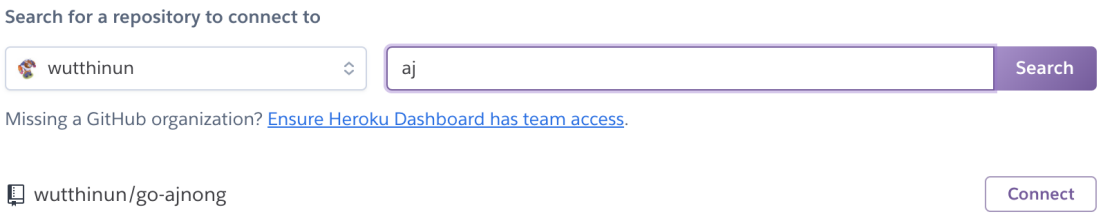
Go deployment (Heroku)

- Connect to Github



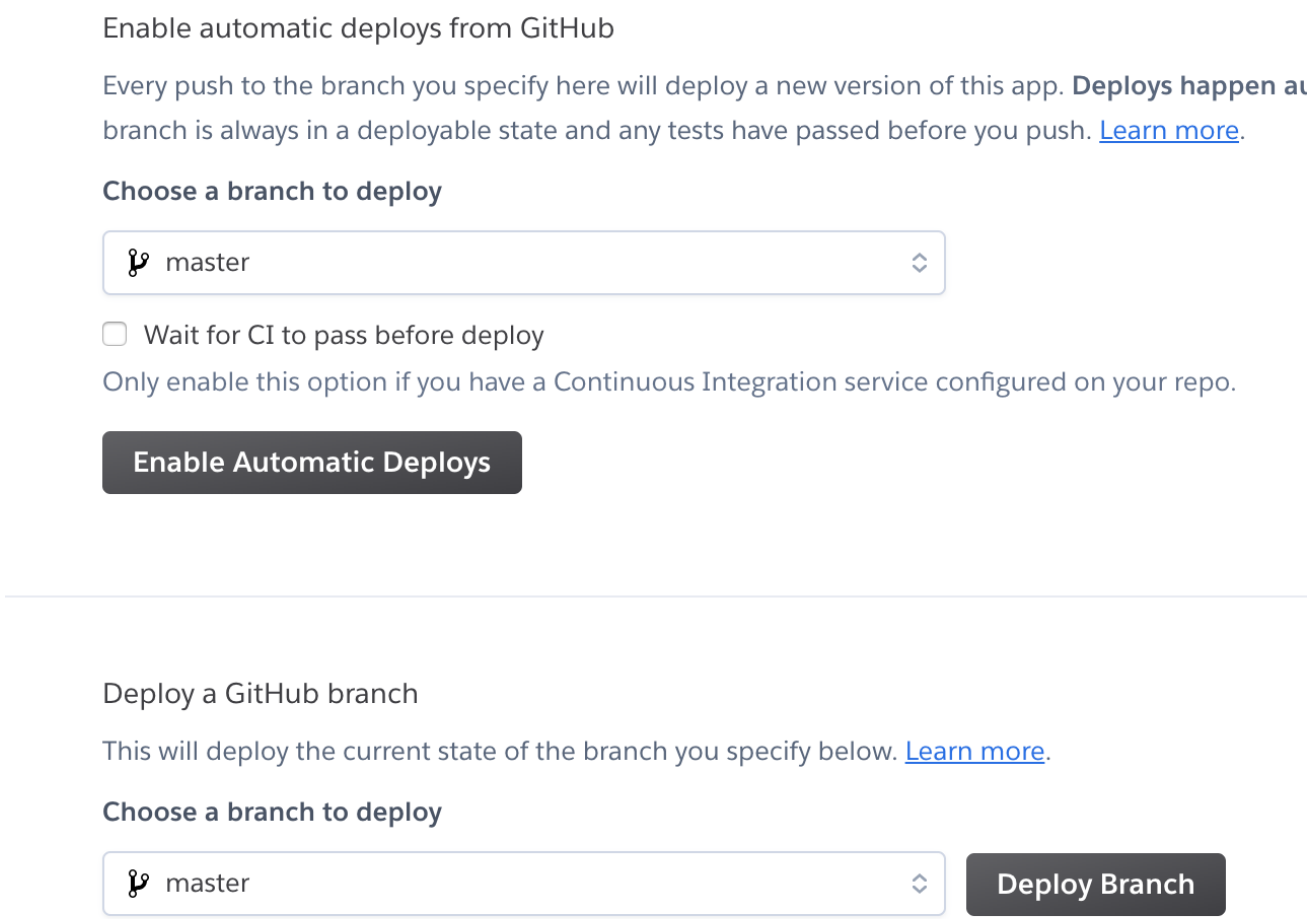
Go deployment (Heroku)

- Select repo



Go deployment (Heroku)

- Config auto deploy



Docker Go

- multi stage

```
FROM golang:1.11 AS build-env
ENV GO111MODULE=on
ADD . /src
RUN cd /src && GOOS=linux GOARCH=386 go build -o goapp

FROM alpine
ENV PORT=$PORT
WORKDIR /app
COPY --from=build-env /src/goapp /app/
CMD /app/goapp
```

Docker Go

- login heroku

```
heroku container:login
```

- create heroku app

```
heroku create
```

- config environment variable

```
heroku config:set MONGO_HOST=13.250.119.252 -a <app-name>
heroku config:set MONGO_USER=root -a <app-name>
heroku config:set MONGO_PASS=example -a <app-name>
```

- Build the image and push to Container Registry

```
heroku container:push web -a <app-name>
```

- Then release the image to your app

```
heroku container:release web -a <app-name>
```

- Now open the app in your browser

```
heroku open - <app-name>
```

Thank you