Welcome

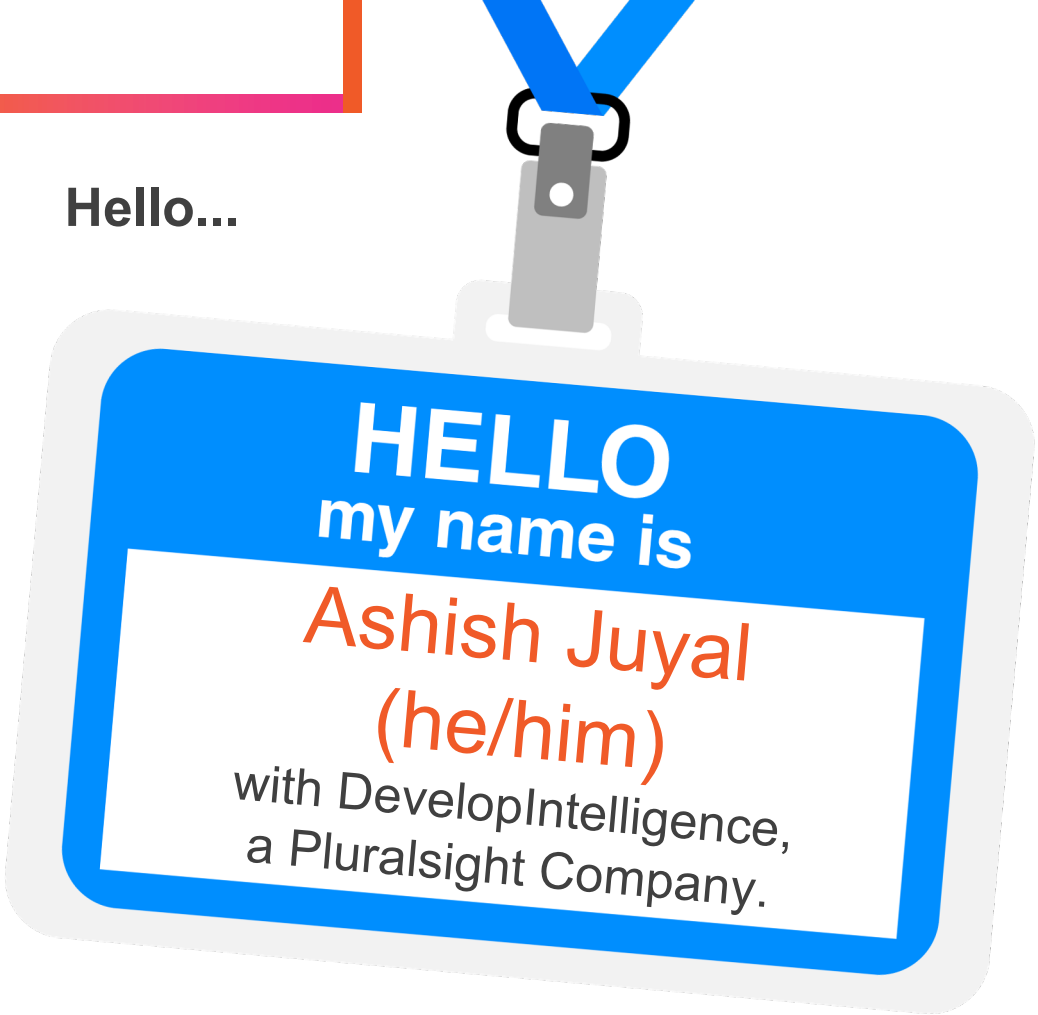
Go Intermediate

DevelopIntelligence
A PLURALSIGHT COMPANY

You?

- Name
- Total experience
- Background in Go
- Goals

Hello...



About me…

- 20+ years experience
- Architect & TDD Coach
- Consultancy
- Distributed architecture
- Microservices
- DevOps
- Java / Scala / Go

Prerequisites

This course assumes you should have preliminary knowledge of Go language:

- Have experience developing with Go
- Primitive types, constants, pointers, structs
- Control, scopes, visibility
- Errors
- Composition
- Array, slice, Map
- Functions, built-in functions and methods
- General programming experience
- Standard data structures
- Basic shell knowledge

We teach over 400 technology topics

You experience our impact on a daily basis!

Objectives

At the end of this course you will be able to:

- Write idiomatic Go using principles such as embedding and interfaces.
- Understand how simple design leads to testable code (unit testing).
- Understand and use "Dependency Injection" in Go.
- Understand and write HTTP server
- Spot common coding pitfalls in Go and correct them.
- Understand concurrency model in Go using Go routines and channels
- Structured logging
- Containerization

Agenda - Day 1

- Introduction
- Refresher
- Pointers
- Interface
- Testable code

- Higher order functions
- Dependency injection

Agenda - Day 2

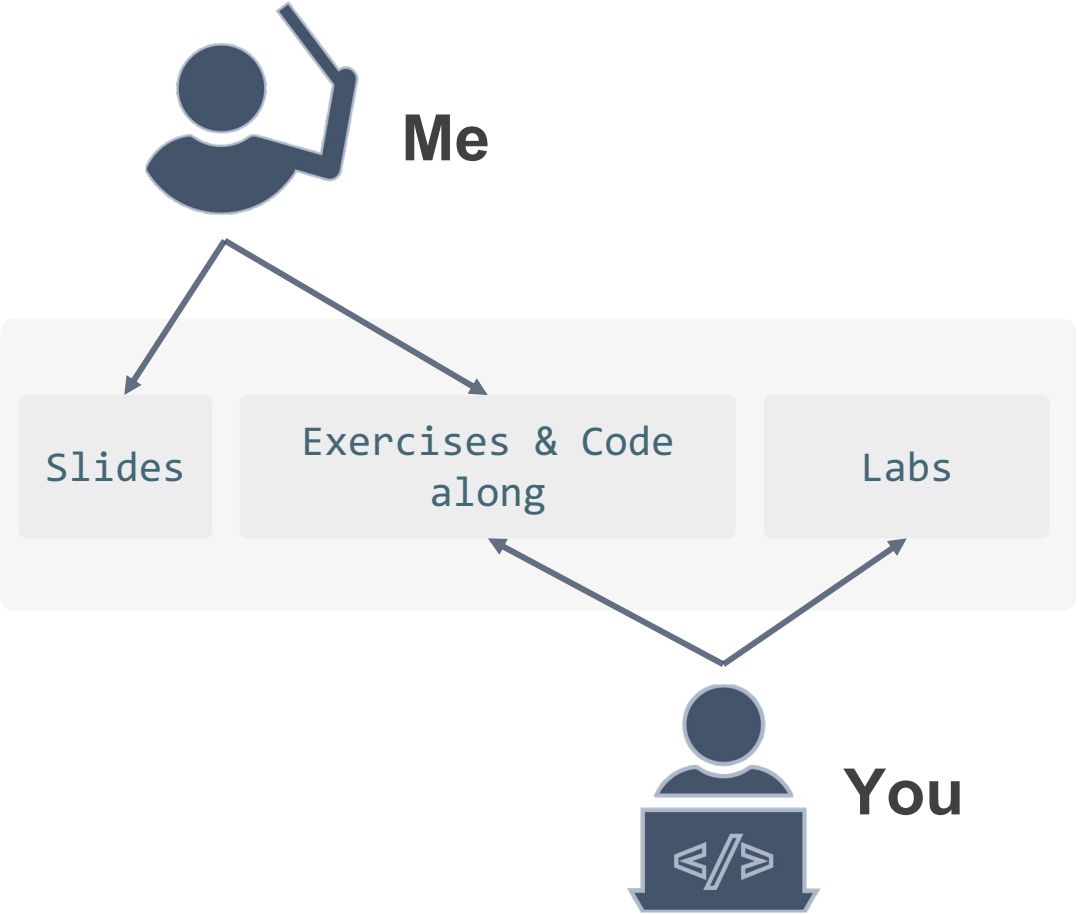
- Concurrency
- Go routines
- Go scheduler
- Runtime

- Go routines
- Channels
- Mutex
- HTTP multiplexer

Agenda - Day 3

- Structured logging
- Unit testing
- Mocking
- Containerization

How we're going to work together



Expectations

Me

- 👷 Be interactive
- ⏱️ Timed breaks
- ✋ Ask questions
- 🗣️ Ensure everyone can speak

- 🕘 Be on time
- ⚠️ Be *in the room*
- 🎙️ Mute your mic
- ✋ Ask questions

You

Breaks

10 mins break every hour

Lunch break at 12:30 PM for an hour

Housekeeping

📞 If I drop, I'll be back

🖐🏾 Let's practice Zoom reactions

Working with virtual machines

We will be doing most of our work in virtual machines

1. Please login to https://labs.datacouch.io/pluralsight/ in an incognito window in Chrome
   - Just do File -> New Incognito Window
2. Note that to copy/paste to/from the VM you will need to use CTRL+SHIFT+C and CTRL+SHIFT+V on the mac
3. Note that Firefox and VSCode are already installed

Let’s GO

How Go was made: https://talks.golang.org/2015/how-go-was-made.slide#1

discussion

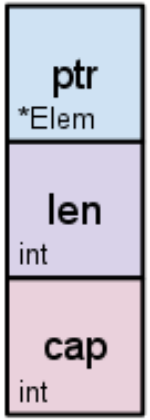
How Go is different from other programming languages ?

Refresher

- Pointers
- Array and slices
- Interface
- Interface composition
- Functions as first citizen
- Higher order functions
- Concurrency & Parallelism

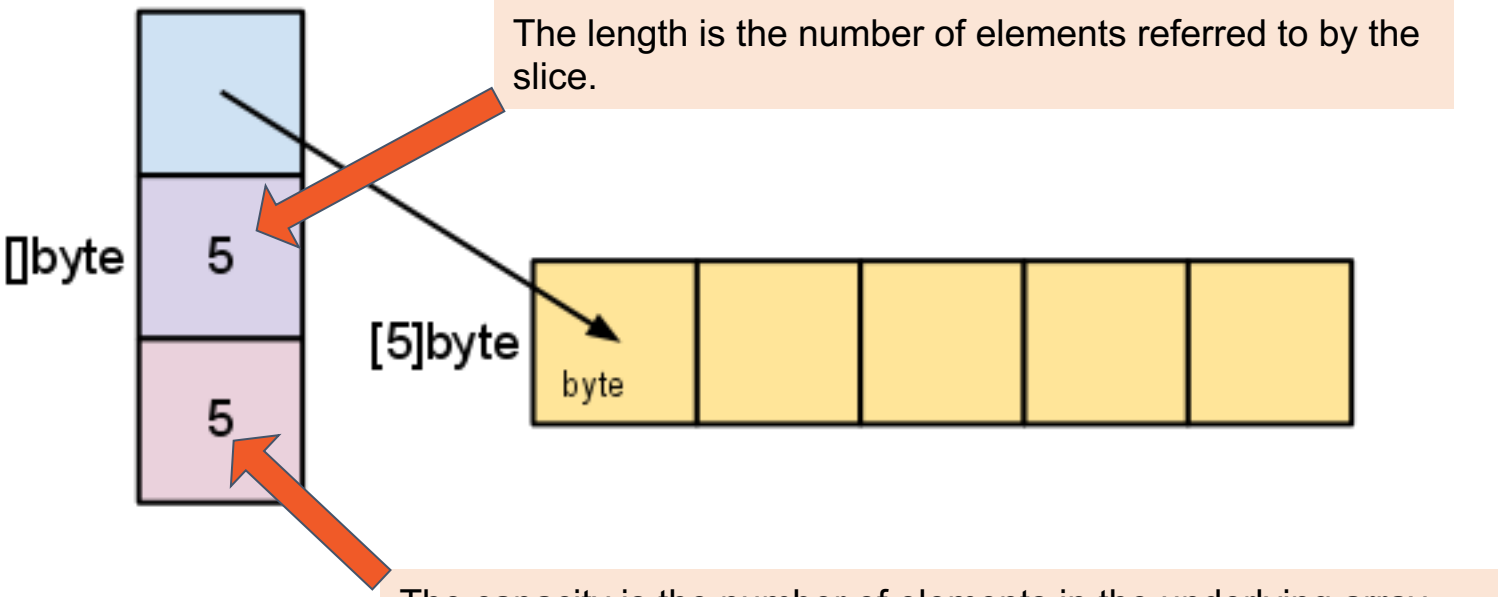
Pointers

Slice internals



A slice is a descriptor of an array segment. It consists of a pointer to the array, the length of the segment, and its capacity (the maximum length of the segment).

Slice internals

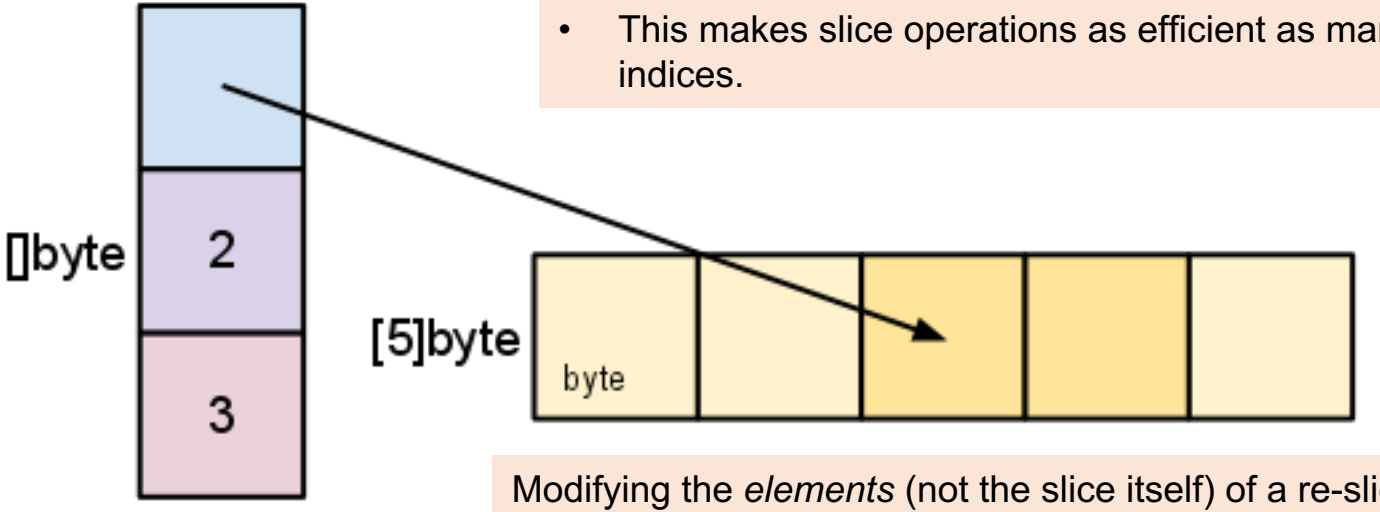


The length is the number of elements referred to by the slice.

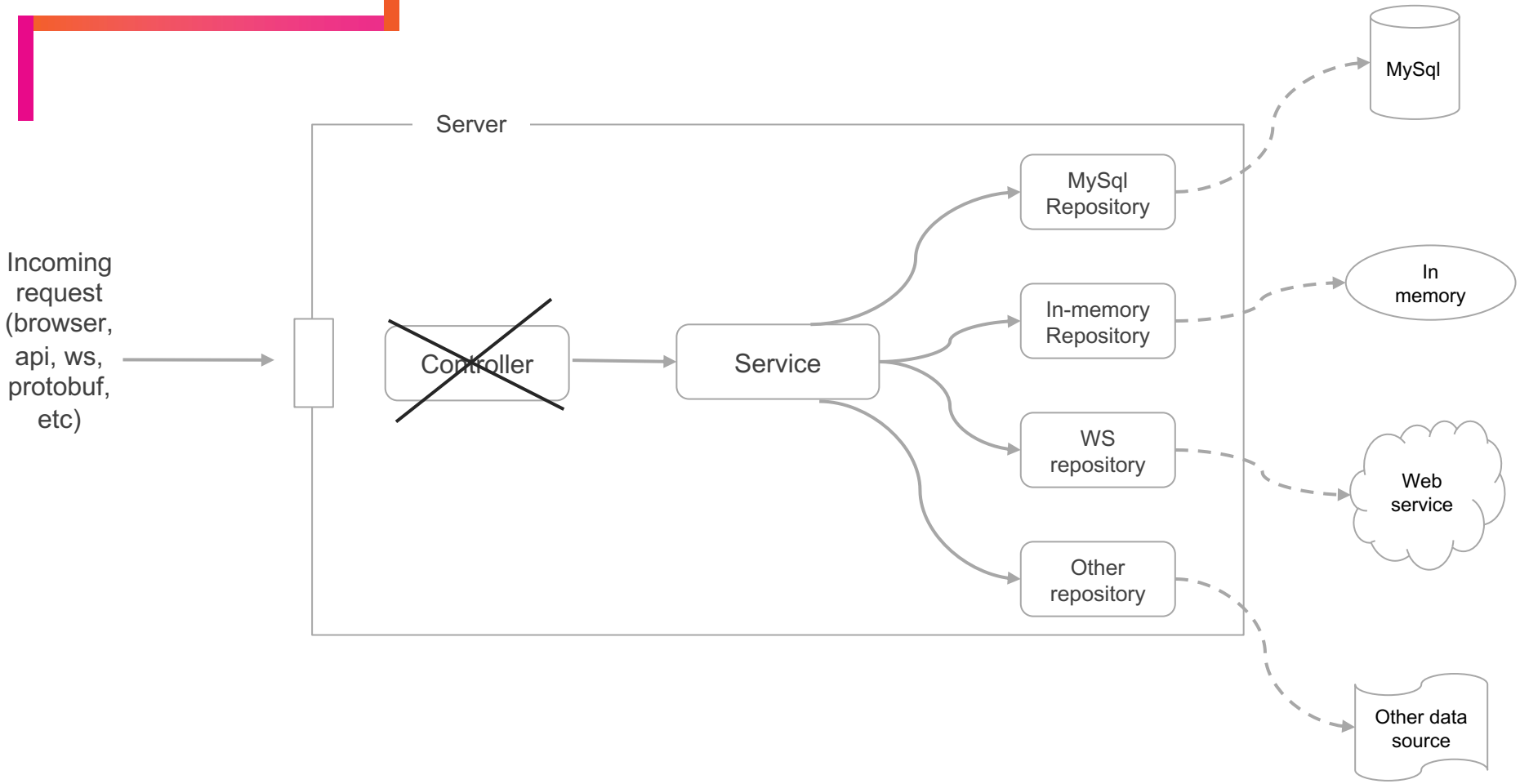
The capacity is the number of elements in the underlying array (beginning at the element referred to by the slice pointer).

Slice internals

- Slicing does not copy the slice's data.
- It creates a new slice value that points to the original array.
- This makes slice operations as efficient as manipulating array indices.



Modifying the *elements* (not the slice itself) of a re-slice modifies the elements of the original slice


Server
Incoming request (browser, api, ws, protobuf, etc)
Controller
Service
MySql Repository
In-memory Repository
WS repository
Other repository
MySql
In memory
Web service
Other data source

Interface

Interfaces are used to define abstractions.

How is it different in Go?

In Go interfaces are implicitly implemented.

What does it mean?

Let’s see it using an example

Interface

Are we just saving ourself from writing two words?

```
implements <interface name>
```

Abstractions are difficult to define, identifying them is the biggest challenge

Interface - abstractions

- Why do we need abstractions?
- When do we need abstractions? Can we identify them early in the game?
- Should we do big design upfront and find abstractions?
- Abstractions are best identified based on their need

Interface

An interface type is defined as a set of method signatures.

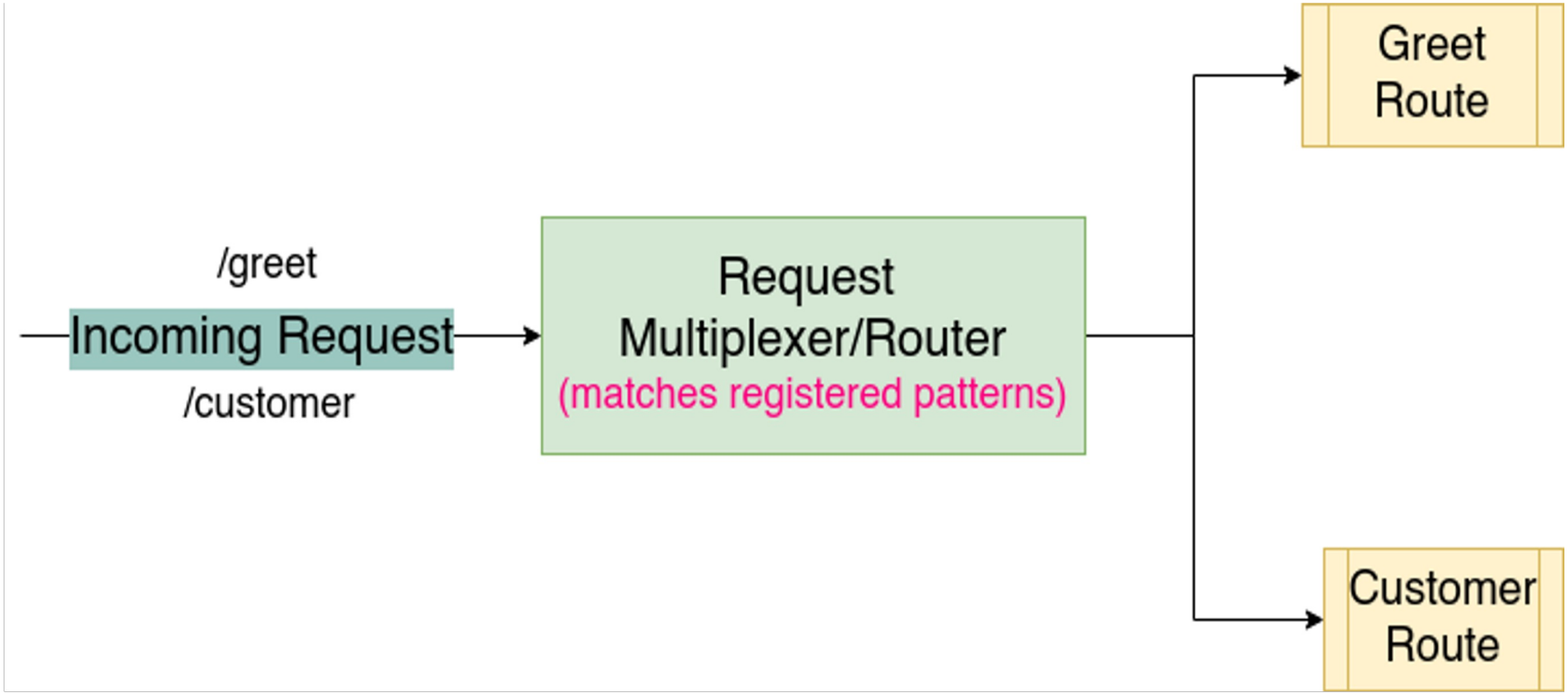
- *if something can do this, it can be used here*
- Duck typing
- `interface{}` / `any`

HTTP

Standard library

HTTP

- Mechanism of HTTP web server
- Handler Functions and Request Multiplexer (Router)
- Request and Response Headers
- Marshaling data structures to JSON and XML representations


/greet
Incoming Request
/customer
Request
Multiplexer/Router
(matches registered patterns)
Greet
Route
Customer
Route

Register handler func

```go
http.HandleFunc("/greet", func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello World"))
})

log.Fatal(http.ListenAndServe("localhost:8080", nil))
```

Writing response

```go
func writeResponse(w http.ResponseWriter, code int, data interface{}) {
    w.Header().Add("Content-Type", "application/json")
    w.WriteHeader(code)
    if err := json.NewEncoder(w).Encode(data); err != nil {
        w.Write([]byte("not able to write response"))
    }
}
```

URL Query

```
// Get gets the first value associated with the given key.
Query parameters
id := r.URL.Query().Get("id")
```

HTTP

gorilla/mux

gorilla/mux

```go
r := mux.NewRouter()

r.HandleFunc("/greet", func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello World"))
})

log.Fatal(http.ListenAndServe("localhost:8080", r ))
```

Vars

```go
// Vars returns the route variables for the current request, if any.
vars := mux.Vars(r)
id := vars["id"]
productId, err := strconv.Atoi(id)
```

NotFoundHandler

```go
r := mux.NewRouter()

// Configurable Handler to be used when no route matches.
r.NotFoundHandler = http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Content-Type", "application/json")
    w.WriteHeader(404)
    w.Write([]byte("invalid route\n"))
})

// custom html page
r.NotFoundHandler = http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    http.ServeFile(w, r, "notfound.html")
})
```

Middleware

```go
r := mux.NewRouter()
r.Use(loggingMiddleware)


func loggingMiddleware(next http.Handler) http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
      t1 := time.Now()
      next.ServeHTTP(w, r)
      fmt.Println(fmt.Sprintf("Incoming request %s completed in %v: ", r.URL.Path,
           time.Since(t1)))
   })
}
```

discussion

What is concurrency?

The art of doing several things at the “same time”

Go routine

- In Go, concurrency is achieved by using Goroutines
- A function that executes simultaneously with other goroutines in a program
- Are lightweight threads managed by Go
- Takes about 2kB of stack space to initialize

What does “At the same time” mean?

Writing a text document

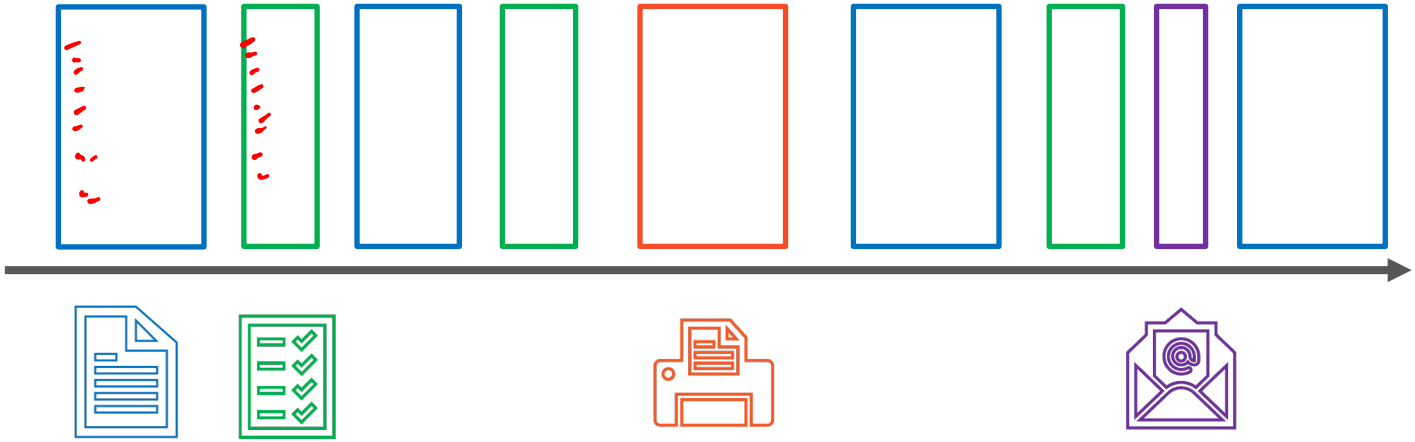
Running the spell check

Printing document

Receiving mails

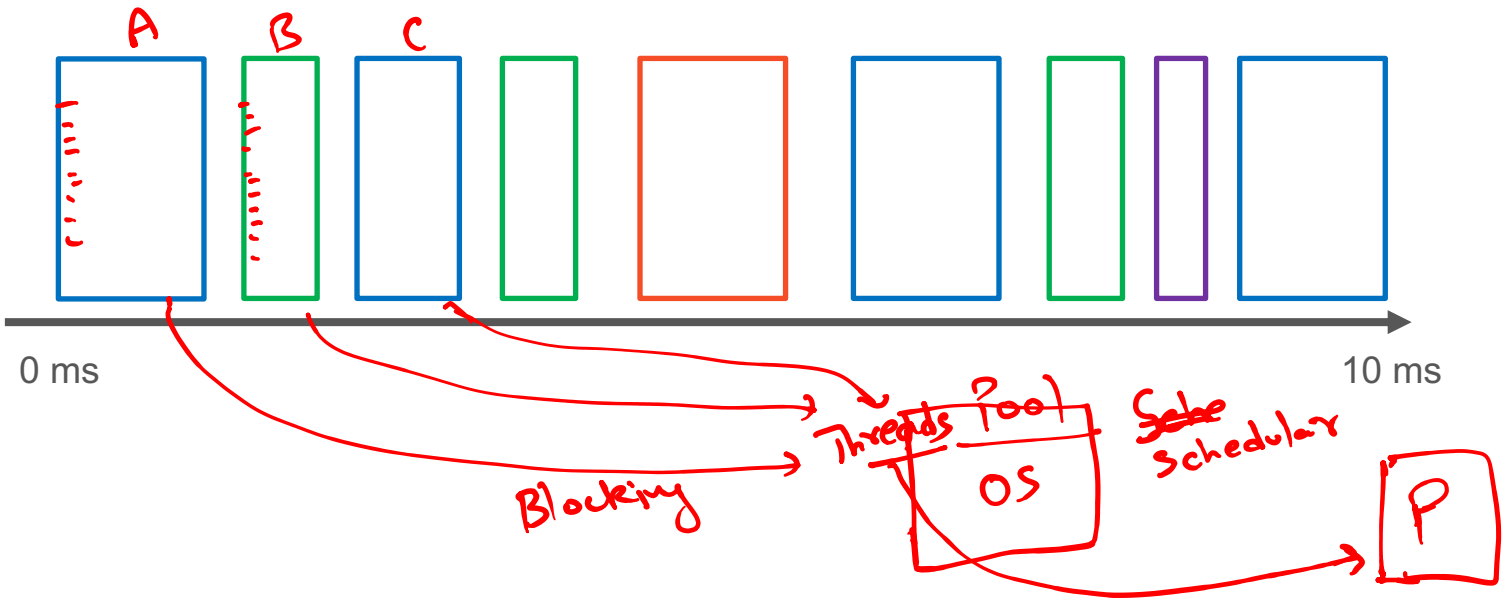
What is happening at CPU level

1st case: CPU with only one core

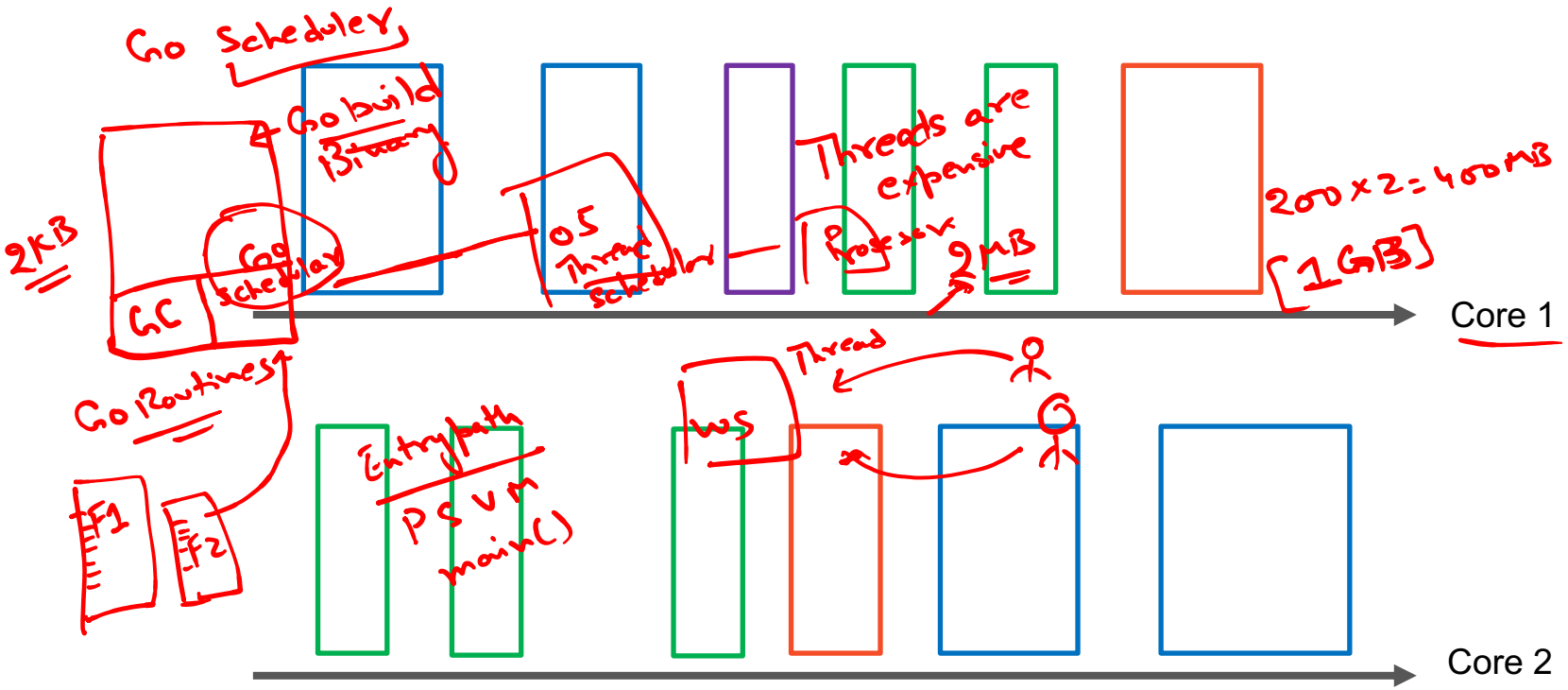


Why does it feels like everything is happening at the same time?

Because things are happening fast



2nd case: CPU with multiple cores



Only on a multicore CPU things are really happening “at the same time”

Parallel

Go routine lab

Wait group

- Used to wait for the program to finish all goroutines launched from the main function
- Block until the WaitGroup counter goes back to 0; all the workers notified they’re done.
- If a WaitGroup is explicitly passed into functions, it should be done by pointer.

Wait group lab

Channel

- Channels are the pipes that connect concurrent goroutines.
- You can send values into channels from one goroutine and receive those values into another goroutine.

```
ch <- v    // Send v to channel ch.
v := <-ch  // Receive from ch, and
           // assign value to v.
```

The data flows in the direction of the arrow.

Channel

- Like maps and slices, channels must be created before use

```
ch := make(chan int)
```

- By default, sends and receives block until the other side is ready.
- This allows goroutines to synchronize without explicit locks or condition variables.

Channel direction

- When using channels as function parameters, you can specify if a channel is meant to only send or receive values.
- This specificity increases the type-safety of the program.

Channel direction

- This function only accepts a channel for sending values. It would be a compile-time error to try to receive on this channel.

```
func producer(stream Stream, out chan<- Message)
```

- This function only accepts a channel for receiving values.

```
func consumer(in <-chan Message)
```

Select statement

select

- Go’s select lets you wait on multiple channel operations.
- Combining goroutines and channels with select is a powerful feature of Go.

Thanks

If you have additional questions, please reach out to me at: ashish@datacouch.io

DevelopIntelligence
A PLURALSIGHT COMPANY