Go for PHP devs

rethinking long running tasks

#ViennaPHP

Johannes Pichler

- Web Developer since 2006
- working @ karriere.at
- doing backend stuff in PHP

karriere.at

- biggest job platform in Austria

karriere.at - devs

Long running tasks in PHP?

Problems with long running tasks

- memory*limit and max*execution_time limitations
- what happens if the process dies?

Potential solutions

- set limits to maximum 😵
- limit execution time and reschedule task
- use a more appropriate programming language

Considering other languages

- (enterprise) Java
- python, ruby …
- C/C++ 😱
- GOlang

Get GOing

The basics - package declaration

```
package main

import "fmt"

func main() {
  fmt.Printf("Hello World!")
}
```

The basics - imports

```
package main

import "fmt"

func main() {
  fmt.Printf("Hello World!")
}
```

The basics - main function

```
package main

import "fmt"

func main() {
   fmt.Printf("Hello World!")
}
```

Variables

```
var abc string = "a string variable"

var number int // number = 0
var emptyString string // emptyString = ""

short := "a string variable" // var short string = "a string variable"
```

Variables

```
var abc string = "a string variable"

var number int // number = 0
var emptyString string // emptyString = ""

short := "a string variable" // var short string = "a string variable"
```

Variables

```
var abc string = "a string variable"

var number int // number = 0
var emptyString string // emptyString = ""

short := "a string variable" // var short string = "a string variable"
```

Arrays

```
// have a fixed size
types := [5]int // [0 0 0 0 0]

// values can be changed and retrieve
types[2] = 3 // [0 0 3 0 0]
val := types[2] // 3

// can be initialized on delaration
types := [5]int{1, 2, 3, 4, 5} // [1 2 3 4 5]

// can have more dimensions
more := [2][3]int
```

Arrays

```
// have a fixed size
types := [5]int // [0 0 0 0 0]

// values can be changed and retrieve
types[2] = 3 // [0 0 3 0 0]
val := types[2] // 3

// can be initialized on delaration
types := [5]int{1, 2, 3, 4, 5} // [1 2 3 4 5]

// can have more dimensions
more := [2][3]int
```

Arrays

```
// have a fixed size
types := [5]int // [0 0 0 0 0]

// values can be changed and retrieved
types[2] = 3 // [0 0 3 0 0]
val := types[2] // 3

// can be initialized on delaration
types := [5]int{1, 2, 3, 4, 5} // [1 2 3 4 5]

// can have more dimensions
more := [2][3]int
```

Arrays

```
// have a fixed size
types := [5]int // [0 0 0 0 0]

// values can be changed and retrieve
types[2] = 3 // [0 0 3 0 0]
val := types[2] // 3

// can be initialized on delaration
types := [5]int{1, 2, 3, 4, 5} // [1 2 3 4 5]

// can have more dimensions
more := [2][3]int
```

Slices

```
slice1 := make([]string, 3) // ["" "" ""]
slice2 := []string{"a", "b", "c"} // ["a" "b" "c"]

slice1[0] = "a" // ["a" "" ""]

var length = len(slice1) // 3

slice2 = append(slice2, "d") // ["a" "b" "c" "d"]
```

Slices

```
slice1 := make([]string, 3) // ["" "" ""]
slice2 := []string{"a", "b", "c"} // ["a" "b" "c"]

slice1[0] = "a" // ["a" "" ""]

var length = len(slice1) // 3

slice2 = append(slice2, "d") // ["a" "b" "c" "d"]
```

Slices

```
slice1 := make([]string, 3) // ["" "" ""]
slice2 := []string{"a", "b", "c"} // ["a" "b" "c"]

slice1[0] = "a" // ["a" "" ""]

var length = len(slice1) // 3

slice2 = append(slice2, "d") // ["a" "b" "c" "d"]
```

Slices

```
slice1 := make([]string, 3) // ["" "" ""]
slice2 := []string{"a", "b", "c"} // ["a" "b" "c"]

slice1[0] = "a" // ["a" "" ""]

var length = len(slice1) // 3

slice2 = append(slice2, "d") // ["a" "b" "c" "d"]
```

Loops - classic while

```
i := 0
for i <= 3 {
    i = i + 1
}
```

Loops - classic for

```
for i := 0; i <= 3; i++ {
    fmt.Println(i)
}
```

Loops - range

```
numbers := []int{1, 2, 3}
sum := 0

for i, val := range numbers {
    fmt.Println("index:", i)
    sum += val
}

fmt.Println("sum:", sum)
```

Functions

```
func sum(a int, b int) int {
    return a + b
}

// or

func sum(a, b int) int {
    return a + b
}
```

Multiple return values

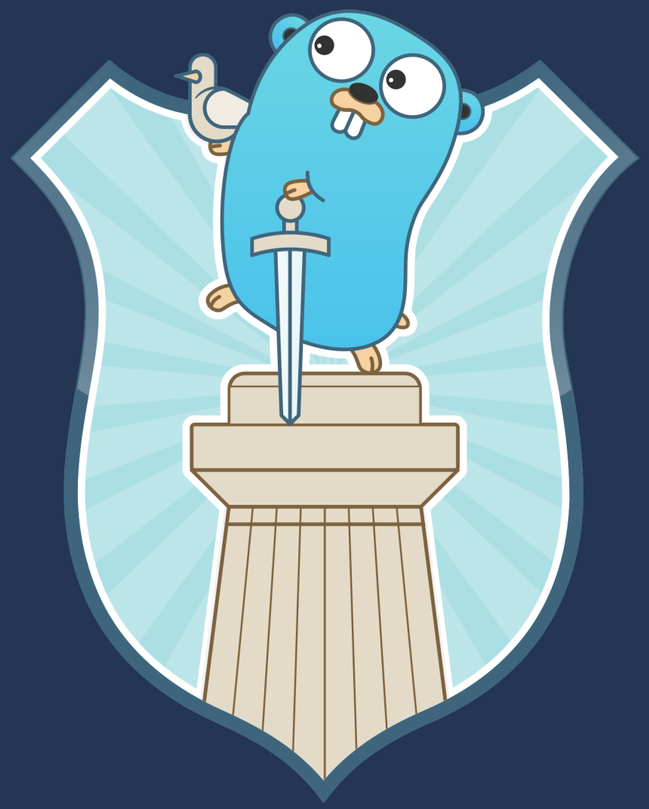
```
func fancySum(a int, b int) (int, bool) {
    ok := true

    // some error checks

    return a + b, ok
}

func main() {
    sum, ok := fancySum(1, 2)

    if !ok {
        // fail with error
    }
}
```



Demo Time #1

A simple calculator

Structs

```
type person struct {
    firstname string
    lastname  string
    age       int
}

person1 := person{firstname: "John", lastname: "Doe", age: 28}
person2 := person{"Jane", "Doe", 28}

fmt.Println(person1.age)
```

Interfaces

```
type animal interface {
    color() string
}
```

Interfaces

```
type cat struct {
    name string
}

type mouse struct {
    name string
}
```

Interfaces

```
func (c cat) color() string {
    if c.name == "Kitty" {
        return "black"
    }

    return "white"
}

func (m mouse) color() string {
    return "grey"
}
```

Interfaces

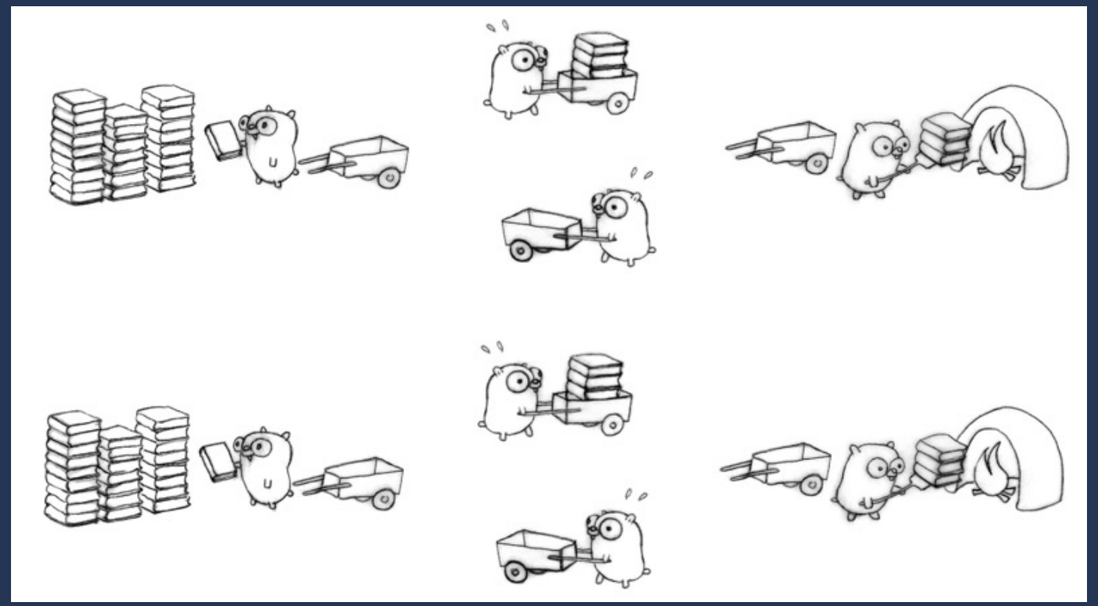
```
func print(a animal) {
    fmt.Println(a.color())
}

func main() {
    c1 := cat{"Kitty"}
    c2 := cat{"Miau"}
    m := mouse{"Pinky"}

    print(c1) // black
    print(c2) // white
    print(m)  // grey
}
```

Goroutines

- concurrent execution of code
- synchronization needed for shared resources (mutex, lock)



Goroutines

```
func doSomeWork() {
    for i := 0; i < 5; i++ {
        fmt.Println("Work index:", i)
    }
}

func main() {
    go doSomeWork()

    // continue with main thread flow
}
```

Channels

- used for communication between goroutines
- sender sends messages, receiver reads from channel
- channels are basically blocking except they are buffered

Channels

```
// create a channel
messages := make(chan string)

// send a message
messages <- "a message"

// read a message
msg := <- messages
```

Channels

```
// create a channel
messages := make(chan string)

// send a message
messages <- "a message"

// read a message
msg := <- messages
```

Channels

```
// create a channel
messages := make(chan string)

// send a message
messages <- "a message"

// read a message
msg := <- messages
```

Demo Time #2

Goroutines

Testing in Go

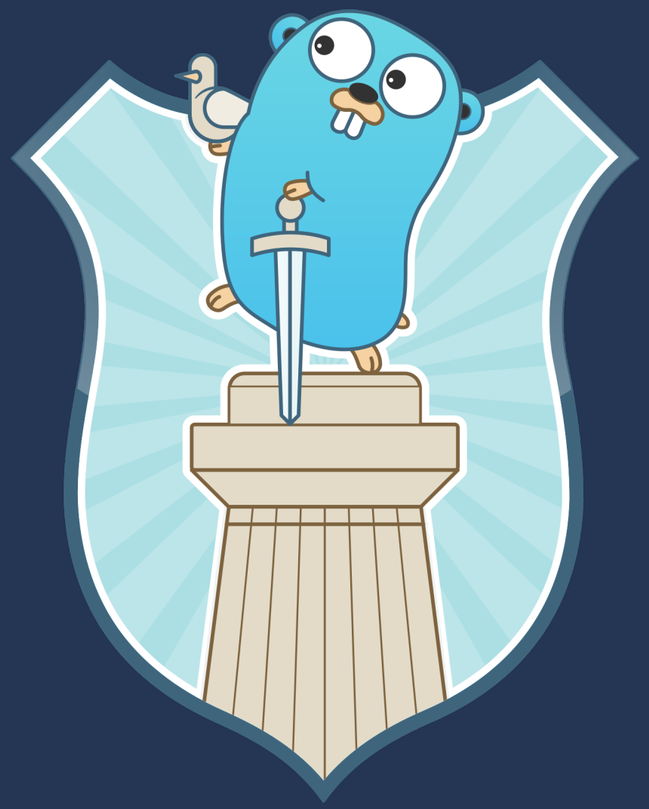
```
import "testing"

...

func TestTestName(t *testing.T) {
    // test code
}
```

Testing in Go

1. define your test set
2. iterate over your test set and validate the function under test
3. fail in case of an error



Demo Time #3

Testing the calculator

Tooling for Go

- code formating
- linting
- testing
- benchmarking
- documentation
- profiling

my golang favorites

- strong type system
- error handling/multiple return values
- implicit interfaces
- built-in tooling
- concurrency
- speed

Helpful resources

- A Tour of Go
  https://tour.golang.org
- Go by Example
  https://gobyexample.com
- List of Go Books
  https://github.com/dariubs/GoBooks

Thanks for listening

@fetzi_io