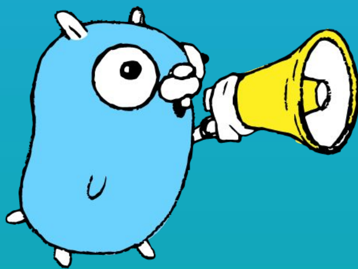
HAN Arnhem (NL), 27 jan 2023

Writing a tcp server for embedded devices in Golang within hours

Jerry Jacobs
Adimec
Development Engineer Firmware

Randy Wijnants
STRATACACHE
Lead Software Engineer



Agenda

About us	01
Golang introduction	02
Exercise explanation	03
Golang hands-on	04
Solution	05
Q&A	06

SECTION ONE

About us

GO

whois Jerry Jacobs

$ whoami

Jerry, 33 years @ xor-gate.org
Development Engineer Firmware at

Adimec
Advanced image systems bv (Eindhoven, NL)

Studied HAN ESE 2010-2013

whois Randy Wijnants

$ whoami

Randy, 34 years @ linkedin.com/in/randywijnants
Lead Software Engineer at

STRATACACHE
Digital Signage (Dayton, USA)

Studied HAN ESE 2011-2014

GO

SECTION TWO

Golang introduction

GO

Intro: Why a new language?

- Faster build times (no make / cmake)
- Simple dependency / package management
- Better usage of multi-core systems
- Modern language designed from the ground up
- More secure by design

Modern software practices

The language and toolchain contains support for modern programming practices. A good ecosystem around the language defines its success:

- Code formatting (gofmt)
- Documentation (go doc)
- Unit testing (go test)
- Race condition testing (go test -race)
- Performance benchmarking (go test -bench)
- Static analysis (go vet)
- Code generation (go generate)

Deployment with peace of mind

- Compiles to monolithic application (“backend.exe”)
  a. No need to ship dependencies
  b. Go runtime is built in to the application
- Golang runtime written in Golang (no insecure C/C++!)
- Cross-compiles easily to UNIX / Linux / macOS / Windows…

```
$ go tool dist list
```

Deployment with peace of mind

```
android/386
android/amd64
android/arm
android/arm64
darwin/386
darwin/amd64
darwin/arm
darwin/arm64
dragonfly/amd64
freebsd/386
freebsd/amd64
freebsd/arm
linux/386
linux/amd64
linux/arm
linux/arm64
linux/mips
linux/mips64
linux/mips64le
linux/mipsle
linux/ppc64
linux/ppc64le
linux/s390x
nacl/386
nacl/amd64p32
nacl/arm
netbsd/386
netbsd/amd64
netbsd/arm
openbsd/386
openbsd/amd64
openbsd/arm
plan9/386
plan9/amd64
plan9/arm
solaris/amd64
windows/386
windows/amd64
```

Deployment with peace of mind

```
android/386
android/amd64
android/arm
android/arm64
darwin/386
darwin/amd64
darwin/arm
darwin/arm64
dragonfly/amd64
freebsd/386
freebsd/amd64
freebsd/arm
linux/386
linux/amd64
linux/arm
linux/arm64
linux/mips
linux/mips64
linux/mips64le
linux/mipsle
linux/ppc64
linux/ppc64le
linux/s390x
nacl/386
nacl/amd64p32
nacl/arm
netbsd/386
netbsd/amd64
netbsd/arm
openbsd/386
openbsd/amd64
openbsd/arm
plan9/386   <-
plan9/amd64
plan9/arm
solaris/amd64
windows/386
windows/amd64
```

Deployment with peace of mind

```
android/3        m
android/a        86
android/a        md64
android/a        rm
darwin/38
darwin/am        64
darwin/ar
darwin/ar        md64
dragonfly        86
freebsd/3        md64
freebsd/a
freebsd/a
linux/386
linux/amd
```

The language

- Simple compared to C & C++:
    a. Code is structured in packages
    b. No header files
    c. No macros
    d. No classes (but methods on a type)
    e. No inheritance (but object composition)
    f. No exceptions (but error return type)
    g. Garbage collected language

The language - Data types

- Basic: Numbers, Boolean, String
- Aggregate: Array and struct
- Reference: Pointer, slice, map, function, channel
- Interface type

The language - Method type

Create a rectangle type

```
type Rectangle struct {
    width, height int
}
```

This `Area` method has a *receiver type* of `*Rectangle`.

```
func (r *Rectangle) Area() int {
    return r.width * r.height
}
```


GO

SECTION THREE

Exercise explanation

Golang hands-on exercise

Writing a TCP-server but how?

- Starting a TCP-server
- Starting multiple “simulated IoT” TCP-clients
- Doing all this work in parallel with **Go**routines
- What you must do to complete the exercise
  a. Figure out which package (imports) are used and import
  b. Starting the server and clients as goroutines
  c. Send the DISCONNECT message from server to client

SECTION FOUR

Golang hands-on

Golang hands-on workshop

For instructions go to

xor-gate.org/talks/golang-han-2023

SECTION FIVE

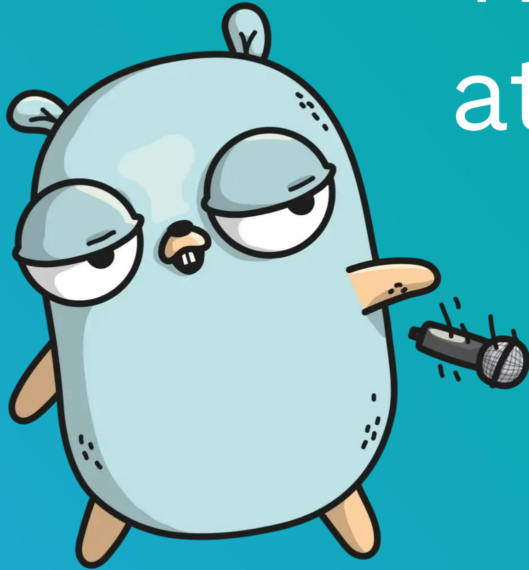
Solution

SECTION SIX

Q&A

GO

Thank you for your attention!



SECTION X

Bonus slides

Golang introduction - Generics

- Go 1.18 introduces support for generic code using parameterized types (think as C++ templates)
- Lodash-like libs now available:
  (MapBy, Filter, FilterMap, Subset, Union, etc.)

Golang introduction - Slices

Slices are intuitive “dynamic size” array implementation:

```go
// Slice
var slice []int

// Array (fixed size)
var array [5]int
```

Golang introduction - Slices

Slices can be ..drum-roll.. **sliced** !
Think python syntactic sugar:

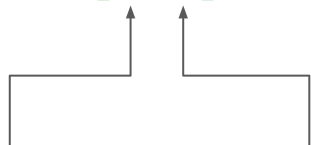
```go
var slice []int{3, 4, 5, 6, 7, 8}

slice[2:5] // yields [5, 6, 7]
```

Slices can be ..drum-roll.. **sliced** !
Think python syntactic sugar:

```go
var slice []int{3, 4, 5, 6, 7, 8}

slice[2:5] // yields [5, 6, 7]
```



Inclusive **Exclusive**

Golang introduction - Slices

Everything in go is **passed by value**, slices too.

However, a **slice's value** is a `SliceHeader struct` behind the scenes, which contains a pointer to the actual array:

```
var slice []int{3, 4, 5, 6, 7, 8}

func doSomething(slice []int) {
    slice[2] = 50 // Modifies original slice
}
```


GO

Golang introduction - Slices

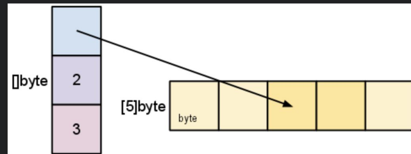
Interesting read: Slice internals

Good for overall knowledge

https://go.dev/blog/slices-intro

As we slice `s`, observe the changes in the slice data structure and their relation to the underlying array:

```
s = s[2:4]
```

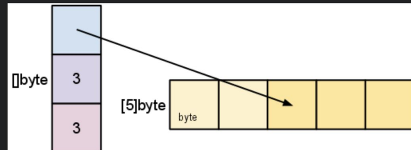


Slicing does not copy the slice's data. It creates a new slice value that points to the original array. This makes slice operations as efficient as manipulating array indices. Therefore, modifying the *elements* (not the slice itself) of a re-slice modifies the elements of the original slice:

```
d := []byte{'r', 'o', 'a', 'd'}
e := d[2:]
// e == []byte{'a', 'd'}
e[1] = 'm'
// e == []byte{'a', 'm'}
// d == []byte{'r', 'o', 'a', 'm'}
```

Earlier we sliced `s` to a length shorter than its capacity. We can grow s to its capacity by slicing it again:

```
s = s[:cap(s)]
```



GO

Golang introduction - Code visibility

Capitalized Public functions

```go
func PublicFunction() {}

func privateFunction() {}
```

Golang introduction - Interface

- Go has a built-in interface type
- Objects implicit satisfy an interface
- Compile time and runtime interface checking possible

Golang introduction - Interface

Interfaces specify behaviors.

An interface type defines a set of methods:

```
type Abser interface {
  Abs() float64
}
```

A type that implements those methods implements the interface:

```
func PrintAbs(a Abser) {
  fmt.Printf("Absolute value: %.2f\n", a.Abs())
}

PrintAbs(MyFloat(-10))
PrintAbs(Point{3, 4})
```

Types implement interfaces implicitly. There is no “implements” declaration.

GO