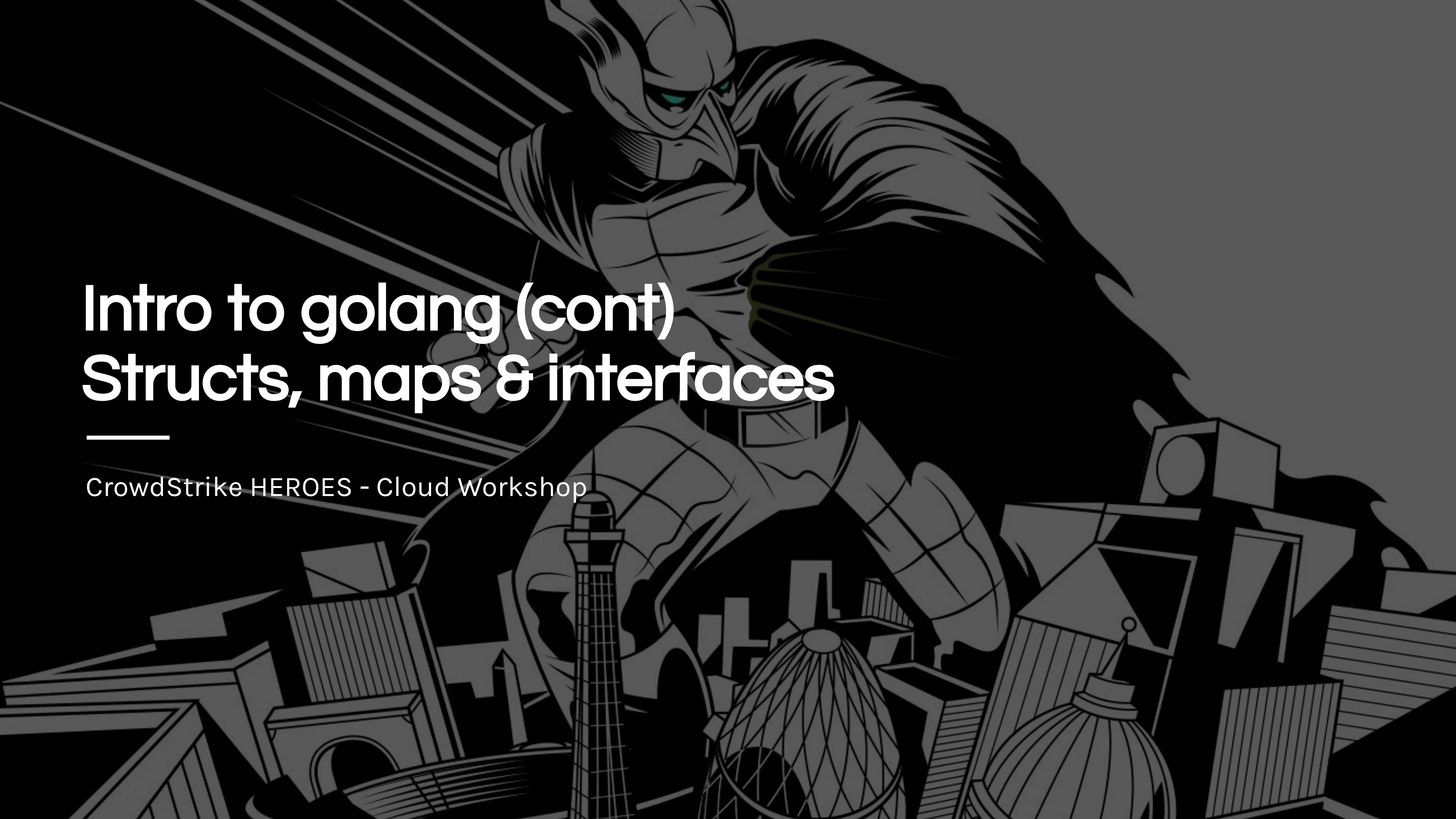
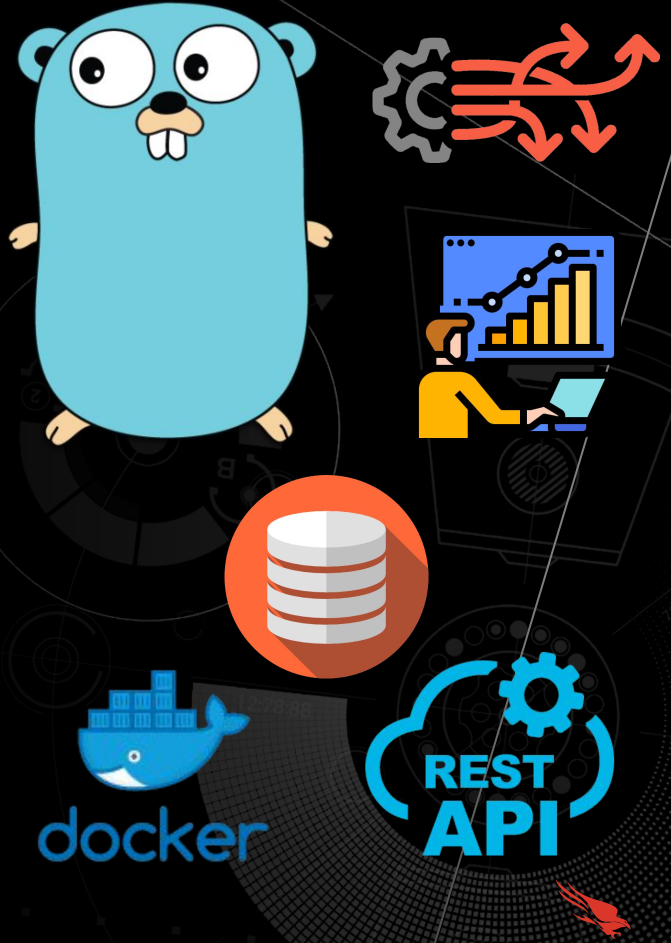


Intro to golang (cont)
Structs, maps & interfaces

CrowdStrike HEROES - Cloud Workshop

Date	Topic
July 25 - 16:00	Intro to golang
July 26 - 16:00	Intro to golang (continuation)
July 27 - 16:00	Multithreading
July 28 - 16:00	Rest API
July 29 - 16:00	Unit testing, logging and monitoring
August 1 - 16:00	Workshop and Q&A
August 2 - 16:00	Deployments/Docker
August 3 - 16:00	Databases
August 4 - 16:00	Databases extended
August 5 - 13:00	Microservices contest (4h with Awards)



Structs

- Collections of fields
- Keyword `struct`

```
type Vertex struct {
    X int
    Y int
}
```

- Keyword `type` can be used to define any new type

```
type myfunc func(int,int) int
```

- Accessing fields

```
v := Vertex{1, 2}
v.X = 4
```

- Access by '.' regardless of pointer type or simple object

```
v := Vertex{1, 2}
p := &v
p.X = 1e9
```

Initialising structs

```
type Vertex struct {
    X, Y int
    z bool
}
```

- `X, Y` are fields that can be used by other packages
- `z` can only be used inside the same package

```
var (
    v1 = Vertex{1, 2, true} // type Vertex
    v2 = Vertex{Z: true, X: 1}  // Y:0 is
implicit
    v3 = Vertex{}      // X:0 and Y:0 and
Z:false
    p  = &Vertex{1, 2, true} // has type
*Vertex
)
```

- Declaring structs without field names:
  - all fields must be specified
  - field order must be respected

Zero values for structs

- There are two formatting options for printing structs `‘%v’` and `‘%+v’`

```
var v1 Vertex

fmt.Printf("%v\n", v1)

> {0 0 false}

fmt.Printf("%+v\n", v1)

> {X:0 Y:0 z:false}
```

- `‘%+v’` also prints fields names

```
var p1 *Vertex

fmt.Printf("%v\n", p1)

> <nil>
```

- When defining a new pointer to a struct we get a empty pointer NOT an empty struct

```
x := p1.X

> panic...
```

- Initialising a pointer

```
p1 = new(Vertex)
```

Maps

- Declaring maps
  - `var m map[keyType]valueType`
  - Initial value is `nil`
- Initialised with the same keyword as arrays, `make`
  - `m := make(map[keyType]valueType)`
  - The initial size of a map is always zero
- To add values to a map there isn’t a built-in function
  - `m[newKey] = newValue`
- We can create a map with values already inserted (map literals)

```
temperatures := map[string]int{
     "Bucharest": 33,
     "Cluj": 28,
     "Iasi" 29,
}
```

Maps - Retrieving and deleting values

- Adding a new value to the same key overwrites the value
  - `m[oldKey] = newValue`
- Retrieving an element
  - `value := m[key]`
- Checking if an element exists
  - `value, exists := m[key]`
  - `exists` is a `bool` that is `true` if the element is present, `false` otherwise
  - If `key` is not in the map, then `value` is the zero value for `valueType`
  - We can just check existence without getting the value: `_ , exists := m[key]`
- Deleting an element
  - `delete(m, key)`

Methods

- Go does not have classes
- A method is a function with a special *receiver* argument

```
func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

- The *receiver* is the object on which the function is called

```
v := Vertex{3, 4}
v.Abs()
```

- Methods can also be defined on non-struct types

Pointer receivers

With a value receiver, the Abs method operates on a copy of the original Vertex value

```
func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

Methods with pointer receivers can modify the value to which the receiver points

```
func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}
```

We can call methods with either receiver type on any type, pointer or value.

Interfaces

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

```
type MyInterface interface {
    f1()
    f2(int, string) int
    f3(a int, b int)
}
```

Go figures it out at assigning an object to a interface if the object type implements it.

Until we write `var i MyInterface ; i = new(Vertex)` go does not care if Vertex is a MyInterface or not.

Implementing interfaces

There is no explicit declaration that we implement an interface, we just define the methods.

```
type I interface {
      M()
}

type T struct {
      S string
}
```

// This method means type T implements the interface I, but we don't need to explicitly declare that it does so.

```
func (t T) M() {
      fmt.Println(t.S)
}
```

The empty interface

The interface type that specifies zero methods is known as the empty interface:

```
interface{}
```

Empty interfaces are used by code that handles values of unknown type

```
var i interface{}
describe(i)

func describe(i interface{}) {
    fmt.Printf("(%v, %T)\n", i, i)
}
```

The %T format returns the underlying type of a value. The empty interface has the type `nil`

Function closures

- We can define anonymous functions
  - Much like lambdas in other languages

```
f := func (args) retType {
    // do stuff
}
ret := f(arg1, arg2)
```

- We can call it immediately by adding the arguments after the definition

```
res := func() int {return 0} ()
```

- As you can see we can have functions with no args or no return value

Variables in closures

Capturing outside variables

```
v := 3
f := func () int {
    return v
}
v = 4
fmt.Println(f())
> 4
```

Using values from surrounding context uses the value at the time of the call. All further changes to the objects can be seen in the function

Capturing variables by arguments

```
v := 3
res := func (v int) int {
    return v
}(v)
v = 4
fmt.Println(res)
> 3
```

Sending values as arguments takes the value at the time of the definition

CROWDSTRIKE

Any questions?