Jump to Index

Event
Go - Introductory upskilling

- **Area**: Software development - Programming language
- **Level**: Beginners
- **Provided as:** Webinar
- **Length:** 2.5h over 4 days
- **Teacher:** Davide D'Innocente
- **Materials:** Path on Pluralsight, slides
- **Organizer:** Advanced Training Center

Description

Go is an open-source programming language supported by Google, easy to learn and great for teams. It provides built-in concurrency and a robust standard library and it has a large ecosystem of partners, communities and tools. In this course, which assumes the knowledge of the fundamental elements of the language, we will discuss the fundamental elements of the Go language. To improve understanding, the different modules of the course are immediately put into practice in hands-on sessions in which students and teachers can interact directly on simple but significant concrete problems proposed in the exercise.

Skills

By the end of the course each student should be able to:

- write a Go program/module
- understand the fundamentals practices for programming applications in Go

Agenda

Day	Time	Lenght	Topic
We 18/01/2023	5.00-6.00 pm	1 h	Introduction to Go
Fr 20/01/2023	5.30-6.00 pm	0.5 h	...
We 25/01/2023	5.30-6.00 pm	0.5 h	...
Fr 27/01/2023	5.30-6.00 pm	0.5 h	...

Target Audience

Programmers who want to use Go to write and manage software applications.

Pre-requisites

To be patient, polite and cooperative.

Any Question?

Please contact the tutor of this course for technical questions on the topics. Ask the Advanced Training Center for organizational and logistic details.

nexum

Indice

Presentazione

Salve a tutti sono Davide D'Innocente un giovane sviluppatore Java. Scrivo questo documento per agevolare lo sviluppo e l'apprendimento dei rudimenti del linguaggio di programmazione **GOLANG** ai miei colleghi. Svilupperemo insieme, passo per passo, almeno un micro servizio con delle API REST, vedremo inoltre come configurare scrittura e lettura di almeno un database, gestione degli errori, best practice e tanto altro. Vedremo inoltre come sfruttare questo potente linguaggio per scrivere Cloud Function, molto utilizzate in progetti con architettura Event Driven che prediligono ambienti di computing Serverless ad alta scalabilità orizzontale. Spero che il mio lavoro sarà di vostro gradimento, i feedback costruttivi da parte vostra sono essenziali per accrescere la nostra conoscenza collettiva.

Introduzione

Go è un linguaggio di programmazione cross-platform e open source, può essere utilizzato per creare applicazioni con performance elevate. E' un linguaggio compilato, "fortemente" e staticamente tipato e che si comporta in modo simile a linguaggi con tipizzazione dinamica e interpretati. Go è stato sviluppato da Robert Griesemer, Rob Pike e Ken Thompson nel 2007 e la sua sintassi è simile al C++.

Go è utilizzato principalmente per Web development (server-side), developing network-based programs, developing cross-platform enterprise applications, cloud-native development.

Perchè Go: divertente e facile da studiare, “*fast run time and compilation time*”, supporta la concorrenza, ottima gestione della memoria (garbage collection), lavora su differenti piattaforme (Windows, Mac, Linux, Raspberry Pi, etc...).

Prime configurazioni dell’ambiente di sviluppo

Per iniziare a sviluppare con Golang, per prima cosa installare Go sul proprio computer, a questo scopo potete consultare il link e scaricare l’installer appropriato al sistema operativo. Verificare l’installazione con il comando “go version”. Come seconda cosa assicuriamoci che l’installazione abbia creato le variabili d’ambiente GoLand e GOPATH. A questo punto andiamo a creare un repository su Git/GitLab/GitHub.

Per questa demo ho creato un repository remoto sul GitLab di Nexum ed il branch dev al seguente link . A questo punto creiamo una cartella locale dove intendiamo clonare il repository, apriamo la PowerShell in questa directory ed eseguiamo il comando:

```
git clone -b dev https://gitlab.com/d.dinnocente/golang-users.git
```

A questo punto siamo pronti ad aprire un IDE compatibile con Golang, io ho scelto Visual Studio Code, e per prima cosa andrò ad installare i plugin/estensioni con publisher “Go Team at Google” che sono “Go” e “Go Nightly”. Una volta installate le estensioni necessarie andiamo ad aprire il folder del progetto che abbiamo clonato.

Verosimilmente abbiamo aperto con Visual Studio un folder che contiene solo il file README automaticamente generato da GitLab. Creiamo quindi il primo file per verificare il funzionamento:

main.go

```
package main

import "fmt"

func main() {

	fmt.Println("Hello World")
}
```

A questo, se il progetto non si trova in una sottocartella rispetto a quella indicata dalla variabile d'ambiente GOPATH vuol dire che stiamo utilizzando i moduli, andiamo quindi a creare un modulo inserendo nella PawerShell, aperta all'interno del folder che contiene quindi il file main.go, nel mio caso vedrò una cosa del genere:

PS C:\GOLAG\Golang Users\golang-users> go mod init gitlab.com/d.dinnocente/golang-users

PS C:\GOLAG\Golang Users\golang-users> go mod tidy
go get .
go build
go install

Questo secondo comando permetterà al modulo di scaricare automaticamente le dipendenze che stiamo utilizzando nel nostro progetto. A questo punto se non abbiamo ottenuto errori siamo in grado di startare il nostro progetto con il comando:

PS C:\GOLAG\Golang Users\golang-users> go run .

OSS:
1- Il punto nel comando "go run" sta a significare che il file main.go si trova nella directory C:\GOLAG\Golang Users\golang-users e si chiama "main.go" come di default per Go. Nel caso non siamo in questo caso possiamo sostituire il punto con path + nome-file (che contiene il metodo main).
2- Verranno creati altri moduli all'interno dello stesso progetto utilizzando un comando del tipo: go mod init gitlab.com/d.dinnocente/golang-users/new-module
Non c'è da preoccuparsi di tenere a mente tale path perchè il comando go mod <path> genererà all'interno del progetto un file go.mod che conterrà le specifiche di tutti i moduli e versioni utilizzati nel progetto stesso.

Se abbiamo ottenuto la stampa su console Hello World vuol dire che tutto sta funzionando e siamo pronti a sviluppare in Go.

Nozioni di sintassi

Per acquisire le nozioni di base sulla sintassi di questo linguaggio di programmazione,

è consigliato seguire il Go Tour1, Go Tour2, oppure A Tour of Go(online version), dando un'occhiata veloce ad entrambi i link è possibile rendersi conto della piccola differenza fra i due.

Cmq sia inserirò qui di seguito alcune nozioni utili.

Per prima cosa i **commenti** sono uguali a Java ('//' per il commento in linea, '/* ...commento... */' per i blocchi commentati su una o più linee).

Tipi primitivi (basic types)

- int, int8, int16, int32, int64 (integers)
  rune // alias for int32
- uint, uint8, uint16, uint32, uint64 (unsigned integers)
  byte // alias for uint8
- float32 (range ,-3.4e+38 to 3.4e+38), float64 (range -1.7e+308 to +1.7e+308)
  Le variabili di questo tipo hanno due tipi di espressioni:
  var x float32 = 123.78
  var y float32 = 3.4e+38

- complex64, complex128
- string
- bool (true, false)

Dichiarazione e inizializzazione di variabili
Per la dichiarazione delle variabili è utilizzato il costrutto **var**,
che può essere anche utilizzato contestualmente all'inizializzazione della variabile stessa,
ma in questo secondo caso non è necessario specificare il tipo. vediamo qualche esempio:

```
var name string
var surname string  =  "D'Innocente"
var nameSurname  =  "Davide D'Innocente" //type is inferred (dedotto)
```

Solo se la variabile è dichiarata e contestualmente inizializzata all'interno di un metodo o di una funzione è possibile l'utilizzo del costrutto **":="** vediamo quindi un esempio:

```
package main
import ("fmt")
var a int
var b int = 2
var c = 3
y := 1 -> ./prog.go:5:1: syntax error: non-declaration statement outside
function body
func main() {
  a = 1
  x := 2 //type is inferred
  fmt.Println(a)
  fmt.Println(b)
  fmt.Println(c)
  fmt.Println(x)
}
```

Il valore delle variabili definite con i costrutti che abbiamo visto (var e :=) può essere modificato. Se una variabile deve avere un valore che non deve poter essere modificato è possibile l'utilizzo del costrutto **const** il quale prevede che la variabile venga dichiarata e inizializzata in modo simile := e il tipo può essere dedotto. Tali variabili sono comunemente chiamate costanti e possono essere definite dentro e fuori metodi e funzioni. Al di fuori dei metodi è anche possibile definire un gruppo di costanti:

```
const (
  A int = 1
  B = 3.14
  C = "Hi!"
)
//come anche di variabili non costanti:
var i,j string = "Hello","World" //anche all'interno di un metodo
```

Funzione Printf()

Soprattutto per gli sviluppatori acerbi in questo linguaggio vediamo come utilizzare questa funzione che permette di stampare diverse informazioni delle variabili sulla consolle:

%v stampa il valore della variabile nel formato di default

%#v stampa il valore nel formato Go-syntax

%T stampa il tipo della variabile (o valore)

%% stampa %

Vediamo alcuni esempi:

```
var i = 15.5
 var txt = "Hello World!"

 fmt.Printf("%v\n", i)            15.5
 fmt.Printf("%#v\n", i)           15.5
 fmt.Printf("%v%%\n", i)          15.5%
 fmt.Printf("%T\n", i)            float64

 fmt.Printf("%v\n", txt)          Hello World!
 fmt.Printf("%#v\n", txt)         "Hello World!"
 fmt.Printf("%T\n", txt)          string
```

Per altri tips vi invito a consultare questa scheda.

Array

Come per le variabili è possibile dichiarare e inizializzare gli array con i costrutti che abbiamo visto.

```
var array_name = [length]datatype{values} // here length is defined
var array_name = [...]datatype{values} // here length is inferred
array_name := [length]datatype{values} // here length is defined
array_name := [...]datatype{values} // here length is inferred
var arr1 = [3]int{1,2,3}
arr2 := [5]int{4,5,6,7,8} oppure
var arr1 = [...]int{1,2,3}
arr2 := [...]int{4,5,6,7,8}
```

Note:

The length specifies the number of elements to store in the array. In Go, arrays have a fixed length. The length of the array is either defined by a number or is inferred (means that the compiler decides the length of the array, based on the number of values).

Possiamo anche definire un array di una certa lunghezza specificando solo determinati elementi:

```
arr1 := [5]int{   1:10, //assegno 10 al primo elemento
```

```
2:40  //assegno 40 al primo elemento
}
```

Slices

Gli slices sono simili agli array ma più potenti e flessibili, sono in grado di contenere valori multipli in una singola variabile, ma a differenza degli array la loro dimensione può variare. Vediamo la sintassi per definire gli slices:

```
slice_name := []datatype{values}
myslice := []int{}
myslice := []int{1,2,3}
```

Abbiamo a disposizione due funzioni per ottenere rispettivamente lunghezza e capacità degli slices **len()** e **cap()**.
Possiamo inoltre creare uno slice a partire da un array:

```
var myarray = [length]datatype{values} // An array
myslice := myarray[start:end] // A slice made from the array
arr1 := [6]int{10, 11, 12, 13, 14,15}
myslice := arr1[2:4]
```

La funzione **make()** permette di creare uno slice:

```
slice_name := make([]type, length, capacity)
myslice1 := make([]int, 5, 10)
myslice2 := make([]int, 5)// with omitted capacity

Per accedere ad un valore di uno slice e modificarlo:
prices := []int{10,20,30}
fmt.Println(prices[0])
prices[2] = 50
Per aggiungere elementi si utilizza la funzione append():
slice_name = append(slice_name, element1, element2, ...)
myslice1 := []int{1, 2, 3, 4, 5, 6}
myslice1 = append(myslice1, 20, 21)
myslice2 := []int{4,5,6}
```

```
myslice3 := append(myslice1, myslice2...)
```

Per l'ottimizzazione della memoria, possiamo copiare un array di grandi dimensioni in uno slice più piccolo sfruttando la funzione **copy(src,dest)**:

```
numbers := []int{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
    // Original slice
    fmt.Printf("numbers = %v\n", numbers)
    fmt.Printf("length = %d\n", len(numbers))
    fmt.Printf("capacity = %d\n", cap(numbers))

// Create copy with only needed numbers
    neededNumbers := numbers[:len(numbers)-10]
    numbersCopy := make([]int, len(neededNumbers))
    copy(numbersCopy, neededNumbers)
```

Accenniamo un "**for-each**":

```
pow := make([]int, 10)
      for i := range pow {
            pow[i] = 1 << uint(i) // == 2**i
      }
      for _, value := range pow {
            fmt.Printf("%d\n", value)
      }
```

Operatori

Operatori aritmetici:

x+x, x-y , x*y, x/y, x%y, x++, yy--

Operatori di assegnazione:

x = 5

x += 3 (x = x + 3)

x -= 3

x *= 3

x /= 3

x |= 3 (x = x | 3)

x ^= 3 (x = x^3)

x >>= 3

x <<= 3

Operatori di comparazione:

== , !=, <, >, >=, <=

Operatori logici:

&& (logical and), || (logical or), ! (logical not)

Operatori Bitwise (usati con numeri (binari)):

& (AND), | (OR), ^ (XOR),

<< (zero fill left shift)

>> (signed right shift)

nexum

Statements

Le “Go Conditions” supportano gli operatori di comparazione e logici.

```
if condition { /* code to be executed if condition is true */ }

if condition {
  // code to be executed if condition is true
} else {
  // code to be executed if condition is false
}

if condition1 {
   // code to be executed if condition1 is true
} else if condition2 {
   // code to be executed if condition1 is false and condition2 is true
} else {
   // code to be executed if condition1 and condition2 are both false
}

if condition1 {
   // code to be executed if condition1 is true
  if condition2 {
     // code to be executed if both condition1 and condition2 are true
  }
}


switch expression {
case x:
   // code block
case y:
   // code block
case z:
```

```
...
default:
   // code block
}

switch expression {
case x,y:
   // code block if expression is evaluated to x or y
case v,w:
   // code block if expression is evaluated to v or w
case z:
...
default:
   // code block if expression is not found in any cases
}

for i:=0; i < 5; i++ {
    fmt.Println(i)
  }

for i:=0; i < 5; i++ {
    if i == 3 {
      break
    }
   fmt.Println(i)
  }
```

Possiamo anche omettere primo e terzo argomento:

```
sum := 1
for ; sum < 1000; {
      sum += sum
}
for sum < 1000 {
            sum += sum
      }
```

```
for index, value := array|slice|map {
   // code to be executed for each iteration
}
fruits := [3]string{"apple", "orange", "banana"}
  for idx, val := range fruits {
     fmt.Printf("%v\t%v\n", idx, val)
  }
for _, val := range fruits {
     fmt.Printf("%v\n", val)
  }
for idx, _ := range fruits {
     fmt.Printf("%v\n", idx)
  }
```

Funzioni

Le funzioni in Go possono avere zero o più argomenti e zero o più risultati.

In Go ci sono anche i metodi, che a differenza delle funzioni, prendono come parametro un puntatore, per questo motivo le funzioni “vere e proprie” non sono molto frequenti di tipo “void”.

Se la funzione ha più argomenti dello stesso tipo, è possibile definirli di seguito specificando il tipo solo per l’ultimo parametro:

```
func add(x, y int) int {
	return x + y
}
```

OSS(best practice): Anticipo ciò che vedremo più approfonditamente più tardi in numerosi casi, se uno dei parametri è un errore verrà definito come ultimo parametro. Questa cosa ovviamente vale sia per le funzioni che per i metodi e vale anche per i valori in output.

Per ritornare più di un valore semplicemente si utilizza la virgola:

```
func swap(x, y string) (string, string) {
	return y, x
}

func main() {
	a, b := swap("hello", "world")
	fmt.Println(a, b)
}
```

nexum

Function values

Le funzioni sono anche valori e possono essere passate come parametro di un'altra funzione.
Le function values possono essere utilizzate come argomenti e valori di ritorno.

```
package main

import (
	"fmt"
	"math"
)

func compute(fn func(float64, float64) float64) float64 {
	return fn(3, 4)
}

func main() {
	hypot := func(x, y float64) float64 {
		return math.Sqrt(x*x + y*y)
	}
	fmt.Println(hypot(5, 12))

	fmt.Println(compute(hypot))
	fmt.Println(compute(math.Pow))
}
```

Function closures
Una closure è il valore di una funzione che referenzia variabili definite all'esterno del suo body.
La funzione può accedere ed assegnare le variabili referenziate,
in questo senso la funzione è limitata alle sue variabili.

```
package main

import "fmt"

func adder() func(int) int {
	sum := 0
	return func(x int) int {
		sum += x
		return sum
	}
}

func main() {
	pos, neg := adder(), adder()
	for i := 0; i < 10; i++ {
		fmt.Println(
			pos(i),
			neg(-2*i),
		)
```

```
    }
}
```

Puntatori

Siamo arrivati ad uno degli argomenti forse più importanti e caratterizzanti di questo linguaggio. Vedremo al riguardo qualche significativo esempio, se al termine di questa sezione non avrete tutto ben chiaro, sappiate che approfondiremo questo argomento in successive sezioni riguardanti strutture e metodi. Se la pazienza non è il vostro forte... questo è il vero problema!

Un puntatore mantiene la memoria dell'indirizzo del valore.
Un tipo ***T** è un puntatore al valore T.
Il suo valore nullo è **nil**.
L'operatore **&** genera un puntatore, "**referencing**", che punta al suo operando:

```
var p *int
i := 42
p = &i

i := 42
var p *int = &i
```

L'operatore * denota il valore sottostante del puntatore:

```
fmt.Println(*p)    // read i through the pointer p
*p = 21            // set i through the pointer p
```

Conosciuto come "**dereferencing**" o "**indirecting**".

Vediamo insieme questo esempio:

```go
package main

import "fmt"

func zeroval(ival int) {
    ival = 0
}

func zeroptr(iptr *int) {
    *iptr = 0
}

func main() {
    i := 1
    fmt.Println("initial: ", i) // initial: 1
    zeroval(i)
    fmt.Println("zeroval: ", i) // initial: 1
    zeroptr(&i)
    fmt.Println("zeroptr: ", i) // initial: 0
    fmt.Println("pointer: ", &i) //pointer: 0x42131100
}
```

Questo esempio mostra che una funzione (void) ha la capacità di modificare il valore di una variabile solo se modifica il valore del suo puntatore.

Potete anche consultare questo link per un ulteriore esempio anche dell'operatore **new(type)**.

Mappe

Le mappe sono utilizzate per dati in coppie chiave:valore, non hanno ordinamento, sono modificabili e non ammettono duplicati. La lunghezza di una mappa è pari al numero dei suoi elementi e coincide con il valore della funzione **len()**. Il valore di default di una mappa è **nil**.
Le mappe contengono riferimenti a una " hash table" sottostante.
Vediamo come creare una mappa definita e inizializzata:

```
var a = map[KeyType]ValueType{key1:value1, key2:value2,...}
b := map[KeyType]ValueType{key1:value1, key2:value2,...}
var a = map[string]string{"brand": "Ford", "model": "Mustang", "year": "1964"}
b := map[string]int{"Oslo": 1, "Bergen": 2, "Trondheim": 3, "Stavanger": 4}
```

Oppure attraverso la funzione **make()** che ci permette di creare una mappa senza inizializzarla(vuota):

```
var a = make(map[KeyType]ValueType)
b := make(map[KeyType]ValueType)
var a map[KeyType]ValueType //EMPTY MAP
```

Vediamo qualche esempio di utilizzo:

```
var a = make(map[string]string) // The map is empty now
  a["brand"] = "Ford"
  a["model"] = "Mustang"// a is no longer empty
fmt.Printf(a["brand"])
```

```
b := make(map[string]int)
  b["Oslo"] = 1
  b["Bergen"] = 2
```

Le **chiavi** ammesse per una mappa sono quelle per cui è definito oppure è possibile definire un operatore di uguaglianza (“==”) ed includono:
booleani, numeri, stringhe, array, puntatori, strutture e interfacce.
Non ci sono vincoli sui **valori** ammessi, possono essere di qualsiasi tipo.

Rivediamo le sintassi delle varie operazioni:

.Accedere ad un elemento — value = map_name[key]
.Aggiunta e modifica — map_name[key] = value
.Rimozione — delete(map_name,key)
.Verifica di uno specifico elemento — _, **ok** := map_name[key]

```
if ok != nil {
      //do something
      }
```

.Verifica + accesso — val, ok := map_name[key]

Iterare in ordine casuale

```
a := map[string]int{"one": 1, "two": 2, "three": 3, "four": 4}
for k, v := range a {
  fmt.Printf("%v : %v, ", k, v)
}
```

Result: `two : 2, three : 3, four : 4, one : 1,`

Iterare in uno specifico ordine
Le mappe sono strutture di dati senza ordinamento proprio.
Se c’è la necessità di utilizzare un ordinamento specifico è necessario avere un array (ordinato) che contiene le chiavi nell’ordine desiderato, in questo modo possiamo recuperare i valori della mappa iterando sull’array.

```
a := map[string]int{"one": 1, "two": 2, "three": 3, "four": 4}
```

```
  var b = []string             // defining the order
  b = append(b, "one", "two", "three", "four")

  for k, v := range a {        // loop with no order
    fmt.Printf("%v : %v, ", k, v)
  }

  fmt.Println()

  for _, element := range b {  // loop with the defined order
    fmt.Printf("%v : %v, ", element, a[element])
  }
```

Strutture

Una struttura in Go è simile ad una classe in Java, è una collezione di campi.
I campi di una struttura sono accessibili attraverso un puntatore della struttura stessa (**struct pointer**).

```
package main

import "fmt"

type Vertex struct {
      X int
      Y int
      // oppure    X, Y int
}

func main() {
      fmt.Println(Vertex{1, 2})
      v := Vertex{1, 2}
      v.X = 4
      fmt.Println(v.X)
      p := &v
      p.X = 1e9
      fmt.Println(v)
```

```
	v2 = Vertex{X:1} // Y:0 is implicit
	v3 = Vertex{} // X:0 and Y:0
	p = &Vertex{1,2} //has type *Vertex
}
```

Vediamo un esempio di slice di tipo “custom”:

```
package main

import "fmt"

func main() {
	q := []int{2, 3, 5, 7, 11, 13}
	fmt.Println(q)

	r := []bool{true, false, true, true, false, true}
	fmt.Println(r)

	s := []struct {
		i int
		b bool
	}{
		{2, true},
		{3, false},
		{5, true},
		{7, true},
		{11, false},
		{13, true},
	}
	fmt.Println(s)
```

```
}

OUTPUT:
[2 3 5 7 11 13]
[true false true true false true]
[{2 true} {3 false} {5 true} {7 true} {11 false} {13 true}]
```

Map literals

Map literals are like struct literals, but the keys are required.

```
package main

import "fmt"

type Vertex struct {
	Lat, Long float64
}

var m = map[string]Vertex{
	"Bell Labs": Vertex{
		40.68433, -74.39967,
	},
	"Google": Vertex{
		37.42202, -122.08408,
	},
}
/* Map literals continued,
var m = map[string]Vertex{
	"Bell Labs": {40.68433, -74.39967},
	"Google":    {37.42202, -122.08408},
}
*/
```

```
func main() {
	fmt.Println(m)
}
```

Metodi - value receiver

Go non ha classi ma è comunque possibile definire sui tipi (strutture).
Un metodo è una funzione con un argomento “speciale” chiamato ***receiver***.
Il receiver appare tra il nome della funzione e le keyword func.

```
package main

import (
	"fmt"
	"math"
)
type Vertex struct {
	X, Y float64
}
func (v Vertex) Abs1() float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
func Abs2(v Vertex) float64{
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
func main() {
	v := Vertex{3, 4}
	//v.X,v.Y
	fmt.Println(v.Abs1())
	Abs2(v)
}
```

Methods continued
I metodi possono anche essere dichiarati per tipi che non sono strutture.

OSS: non si possono dichiarare metodi con receiver il cui tipo è definito in un altro package rispetto al metodo.

```
package main

import (
	"fmt"
	"math"
)
type MyFloat float64

func (f MyFloat) Abs() float64 {
	if f < 0 {
		return float64(-f)
	}
	return float64(f)
}

func main() {
	f := MyFloat(-math.Sqrt2)
	fmt.Println(f.Abs())
}
```

Metodi - Pointer receivers

Possono essere definiti metodi il cui receiver è un puntatore, questi metodi possono modificare il valore a cui il receiver punta, per questa caratteristica questo tipo di metodi è maggiormente utilizzato.

```
package main

import (
	"fmt"
	"math"
)

type Vertex struct {
	X, Y float64
}
// value receiver method
func (v Vertex) Abs() float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
// pointer receiver method
func (v *Vertex) Scale(f float64) {
	v.X = v.X * f
	v.Y = v.Y * f
}
```

```
func main() {
	v := Vertex{3, 4}
	v.Scale(10)
	fmt.Println(v.Abs())
}
```

Pointers and functions

Le funzioni in Go possono prendere come argomento un puntatore (pointer).

```
package main

import (
	"fmt"
	"math"
)
type Vertex struct {
	X, Y float64
}
func Abs(v Vertex) float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
func Scale(v *Vertex, f float64) {
	v.X = v.X * f
	v.Y = v.Y * f
}
func main() {
	v := Vertex{3, 4}
	Scale(&v, 10)
	fmt.Println(Abs(v))
}
```

La funzione Scale in questo caso modifica il valore di &v, che quindi modifica v,

infatti abbiamo
(3,4)->Scale(3,4)=(30,40)->Abs(30,40)=(30^2+40^2)^(½) = (2500)^(½) = 50 = Abs(Scale(Vertex{3,4}))

OSS: Se provassimo ad eliminare l'asterisco, a definire quindi la funzione Scale(v Vertex, f float64), essa non sarebbe in grado di modificare il valore di v, quindi otterremmo lo stesso risultato che è possibile ottenere eliminando completamente la funzione Scale.

Interfacce

Un ***interface type*** è definita come un insieme di firme di metodi.
Un valore di tipo di interfaccia (interface type) può contenere qualsiasi valore che implementa tali metodi.

```
package main
import (
	"fmt"
	"math"
)
type Abser interface {
	Abs() float64
}
func main() {
	var a Abser
	f := MyFloat(-math.Sqrt2)
	v := Vertex{3, 4}
	a = f  // a MyFloat implements Abser
	a = &v // a *Vertex implements Abser
	// In the following line, v is a Vertex (not *Vertex)
	// and does NOT implement Abser.
	//a = v
	fmt.Println(a.Abs())
}
type MyFloat float64
```

```
func (f MyFloat) Abs() float64 {
      if f < 0 {
            return float64(-f)
      }
      return float64(f)
}
type Vertex struct {
      X, Y float64
}
func (v *Vertex) Abs() float64 {
      return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

Le interfacce sono implicitamente implementate

Un tipo implementa un'interfaccia implementando i suoi metodi.
Non c'è una dichiarazione esplicita dell'implementazione dell'interfaccia stessa (come in Java).
Le interfacce implicite disaccoppiano la definizione di un'interfaccia dalla sua implementazione, che può essere presente il qualsiasi package.

```
package main

import "fmt"

type I interface {
      M()
}
type T struct {
      S string
}
// Questo metodo indica che il tipo T implementa l'interfaccia I
func (t T) M() {
      fmt.Println(t.S)
}
func main() {
      var i I = T{"hello"} // una sorta di polimorfismo
      i.M()
}
```

Interface values

I valori dell'interfaccia possono essere pensati come una tupla di un valore e un tipo concreto: (value, type)

Un valore di interfaccia contiene un valore di un tipo concreto sottostante specifico.

La chiamata di un metodo su un valore di interfaccia esegue il metodo con lo stesso nome sul relativo tipo sottostante.

Questo lo verifichiamo nell'esempio che segue, nel quale la stessa interfaccia è implementata da più di un tipo.

```
package main

import (
	"fmt"
	"math"
)
type I interface {
	M()
}
type T struct {
	S string
}
func (t *T) M() { //A
	fmt.Println(t.S)
}
type F float64

func (f F) M() { //B
	fmt.Println(f)
}
func main() {
	var i I
	i = &T{"Hello"}
	describe(i)
	i.M() //A
```

```
	i = F(math.Pi)
	describe(i)
	i.M() //B
}
func describe(i I) {
	fmt.Printf("(%v, %T)\n", i, i)
}
```

La funzione **describe** è una funzione generica, accetta un qualsiasi tipo che implementa l'interfaccia I.

Interface values with nil underlying values
Se il valore concreto contenuto nell'interfaccia è **nil**, il metodo sarà chiamato con un **nil receiver**. In alcuni linguaggi(java) questo genera un **null pointer exception**, ma in Go viene generato un errore del tipo:
`panic: runtime error: invalid memory address or nil pointer dereference`
`[signal 0xc0000005 code=0x0 addr=0x0 pc=0x45fc9a]`
che causa l'interruzione dell'applicazione.
Buona norma è scrivere metodi che gestiscono il valore del receiver.

OSS: il valore di un'interfaccia che contiene un *nil concrete value* è *non-nil*.

```
package main
import "fmt"

type I interface {
	M()
}
type T struct {
	S string
}
func (t *T) M() {
	if t == nil {
		fmt.Println("<nil>")
```

```
            return
        }
        fmt.Println(t.S)
}
func main() {
        var i I
        var t *T
        i = t
        describe(i)
        i.M()
        i = &T{"hello"}
        describe(i)
        i.M()
}
func describe(i I) {
        fmt.Printf("(%v, %T)\n", i, i)
}
```

Type switches

Un type switch è un costrutto che permette diverse asserzioni di tipo in serie, è come un regolare switch statement ma i casi sono tipi e non valori, tali “valori” vengono confrontati con il tipo del valore contenuto nel valore dell’interfaccia specificato.

```
 package main
import "fmt"

func do(i interface{}) {
        switch v := i.(type) {
        case int:
                fmt.Printf("Twice %v is %v\n", v, v*2)
        case string:
                fmt.Printf("%q is %v bytes long\n", v, len(v))
        default:
                fmt.Printf("I don't know about type %T!\n", v)
        }
}
func main() {
        do(21)
```

```
	do("hello")
	do(true)
}
```

Stringer Interface

Una delle interfacce più diffuse è Stringer definita dal pacchetto **fmt**.
Uno Stringer è un tipo che può descrivere se stesso come una stringa. Il pacchetto fmt (e molti altri) cerca questa interfaccia per stampare i valori.

```
type Stringer interface {
    String() string
}
```

```
package main
import "fmt"

type Person struct {
	Name string
	Age  int
}
// Person implement Stringer
func (p Person) String() string {
	return fmt.Sprintf("%v (%v years)", p.Name, p.Age)
}

func main() {
	a := Person{"Arthur Dent", 42}
```

```
	z := Person{"Zaphod Beeblebrox", 9001}
	fmt.Println(a, z)
}
```

Error Interface

Le applicazioni in Go esprimono gli stati di errore con valori di tipo **error**.
Il tipo *error* è un'interfaccia built-in simile a fmt.Stringer

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

Spesso le funzioni ritornano un **error value**, il codice chiamante deve gestire gli errori testando se l'errore è uguale a **nil**.

```
package main

import (
	"fmt"
	"time"
)
type MyError struct {
	When time.Time
	What string
}
func (e *MyError) Error() string {
	return fmt.Sprintf("at %v, %s",
		e.When, e.What)
```

```
}
func run() error {
	return &MyError{
		time.Now(),
		"it didn't work",
	}
}
func main() {
	if err := run(); err != nil {
		fmt.Println(err)
	}
}
```

Gin Gonic Web Framework

https://gin-gonic.com/
https://gin-gonic.com/docs/examples/
https://github.com/gin-gonic/gin
https://go.dev/doc/tutorial/web-service-gin
https://pkg.go.dev/github.com/gin-gonic/gin@v1.8.2

Router

```
import(
   "fmt"
   “net/http”
   "github.com/gin-gonic/gin"
)

func main() {
    router = gin.Default()

    router.GET("/ping", controller.Ping)
    router.GET("/user/:name", controller.GetUser)
    router.GET("/users", controller.GetAllUser)
```

```
    router.POST("/user", controller.CreateUser)

    fmt.Println("Application Started")
    router.Run(":8080")

}
```

Validazioni

Post - Input Validation

```
type User struct {
    Name     string    `json:"name"`
    Surname  string    `json:"surname"`
    DateTime time.Time `json:"datetime"`
}

func CreateUser(c *gin.Context) {
    var user model.User
    err := c.ShouldBindJSON(&user)
    if err != nil {
        c.JSON(http.StatusBadRequest, err.Error())
        return
    }
    c.JSON(http.StatusCreated, &user)
}
```

Get - Path Variable

```
func GetUser(c *gin.Context) {
    name := c.Param("name")
    …
```

Uri Validation

```
var person Person
if err := c.ShouldBindUri(&person); err != nil {
      c.JSON(400, gin.H{"msg": err})
      return
}
```

Query Param Validation

```
type Person struct {
      Name     string    `form:"name"`
      Address  string    `form:"address"`
      Birthday time.Time `form:"birthday" time_format:"2006-01-02"
time_utc:"1"`
}
func main() {
      route := gin.Default()
      route.GET("/testing", startPage)
      route.Run(":8085")
}

func startPage(c *gin.Context) {
      var person Person
      // If `GET`, only `Form` binding engine (`query`) used.
      // If `POST`, first checks the `content-type` for `JSON` or `XML`, then
uses `Form` (`form-data`).
      // See more at
```

```
https://github.com/gin-gonic/gin/blob/master/binding/binding.go#L48
	if c.ShouldBind(&person) == nil {
		log.Println(person.Name)
		log.Println(person.Address)
		log.Println(person.Birthday)
	}
	c.String(200, "Success")
}

Test it with:
$ curl -X GET
"localhost:8085/testing?name=appleboy&address=xyz&birthday=1992-03-15"
```

Approfondimenti
https://gin-gonic.com/docs/examples/binding-and-validation/

Concetti Fondamentali
Ora vedremo alcuni dei concetti fondamentali per la creazione di un microservizio Golang con API REST. Questa è solo una prima bozza, approfondiremo tali concetti durante il corso.

I **metodi del Controller** (o dei controller), hanno sempre come parametro formale (parametro/input) un puntatore all'interfaccia **gin.Context**, hanno come valori di output **un puntatore all'oggetto di output** e un **errore** oppure un modello di errore custom.
A tali metodi inoltre sono delegate tre funzioni principali: **gestione degli input** che possono essere parametri in uri e in query string oppure oggetti (request body) nel caso di API POST o PUT. Nel caso dell'oggetto si verifica la corretta sintassi JSON e la corrispondenza con le chiavi attese attraverso il metodo pointer receiver `c.ShouldBindJSON(&obj)` dell'interfaccia Context già citata. A valle delle validazioni degli input, **il controller richiama un metodo dello strato di servizio** (che a sua volta tornerà un oggetto e un errore) e gestisce l'errore ritornato. Infine ritorna l'oggetto oppure l'oggetto di errore serializzato con il metodo **JSON** presente anch'esso nell'interfaccia Context. Ricordatevi che il metodo c.JSON(status,obj) non provoca l'interruzione del metodo, effettua solamente un'assegnazione al puntatore c, quindi utilizzare il **return** dopo tale istruzione.

I **metodi dello strato di servizio** implementano la logica di business dell'API, comprese interrogazioni al database... e ritornano un puntatore all'oggetto generato e un oggetto di errore.

L'utilizzo dei puntatori è fondamentale per l'ottimizzazione della memoria. Inoltre l'utilizzo delle interfacce ci da la possibilità di confrontare i tipi (o gli oggetti) con il **<nil>**. Se non riuscite a confrontare un oggetto o un oggetto di errore con quest'ultimo significa che non state programmando secondo la logica Go.

In generale i metodi devono sempre avere come valori di ritorno una coppia (*obj,error), nell'utilizzo del costrutto **return** tenere sempre a mente che i valori **ammessi** sono solamente due:

- **return nil, error**
- **return &obj, nil**

Non sono assolutamente ammessi i valori di ritorno del tipo:

- **return &obj, error**
- **return nil, nil**

Questi ultimi creano ambiguità nell'esistenza di un errore, la non osservazione di questa pratica crea gli strati dell'applicazione non completamente isolati, visto che l'altro deve conoscere una logica di ritorno particolare del metodo chiamato.

Swagger

Per documentare le API esposte dal servizio Golang con framework Gin Gonic, per prima cosa importiamo le dipendenze nel progetto:

```
go install github.com/swaggo/swag/cmd/swag@latest
```

successivamente andiamo a documentare gli attributi che andranno a configurare la Swagger UI nel progetto:

```
package main

import (
    _ "golang-users/docs" //NON OPZIONALE

    "github.com/gin-gonic/gin"
    swaggerFiles "github.com/swaggo/files"
    ginSwagger "github.com/swaggo/gin-swagger"

    "golang-users/app"
    "golang-users/utils/logger"
)

var (
```

```
    router = gin.Default()
)

// swagger embed files

// @title           USERS API
// @version         1.0
// @description     This is a sample server celler server.
// @termsOfService  http://swagger.io/terms/

// @contact.name   API Support
// @contact.url    http://www.swagger.io/support
// @contact.email  support@swagger.io

// @license.name  Apache 2.0
// @license.url   http://www.apache.org/licenses/LICENSE-2.0.html

// @schemes http
// @host      localhost:8080
// @BasePath  /
func main() {
    app.MapUrls(router)
    url := ginSwagger.URL("http://localhost:8080/swagger/doc.json")
    router.GET("/swagger/*any", ginSwagger.WrapHandler(swaggerFiles.Handler,
url))
    logger.Info("about to start the application...")
    router.Run(":8080")
}
```

Abbiamo aggiornato il metodo mapUrls()-> MapUrls(*gin.Engine):

```
package app

import (
    "golang-users/controllers/ping"
```

```
    "golang-users/controllers/users"

    "github.com/gin-gonic/gin"
)

func MapUrls(router *gin.Engine) {
    router.GET("/ping", ping.Ping)

    router.POST("/users", users.Create)
    router.GET("/users/:user_id", users.Get)
    router.GET("/users/gorm/email/:email", users.GetByEmail)
    router.POST("/users/gorm", users.CreateWithGorm)
    router.PUT("/users/:user_id", users.Update)
    router.PATCH("/users/:user_id", users.Update)
    router.DELETE("/users/:user_id", users.Delete)
    router.GET("/internal/users/search", users.Search)
    router.POST("/users/login", users.Login)
}
```

Documentare i metodi dei Controller definiti in MapUrls(*gin.Engine) come nell'esempio:

```
//      @Summary    GetUserById
//      @Param        user_id    path     string  true  "search by user_id"
Format(int)
//      @Schemes
//      @Description    description
//      @Tags           users
//      @Accept         json
//      @Produce        json
//      @Success        200 {object}    users.User
//      @Router         /users/{user_id} [get]
func Get(c *gin.Context) {

    userId, idErr := getUserId(c.Param("user_id"))
    if idErr != nil {
```

```
        c.JSON(idErr.Status(), idErr)
        return
    }
    user, getErr := services.UsersService.GetUser(userId)
    if getErr != nil {
        c.JSON(getErr.Status(), getErr)
        return
    }
```

Avviare l'applicazione (go run .) e visitare il link:
http://localhost:8080/swagger/index.html

Per maggiori informazioni sulle annotation disponibili e i parametri configurabili, visitare la documentazione:

GitHub - swaggo/swag: Automatically generate RESTful API documentation with Swagger 2.0 for Go.

Persistenza

MySQL

Vediamo una prima semplice implementazione della configurazione del database MySQL dal nostro servizio Golang. Come prima cosa importiamo la dipendenza `"database/sql".`
Per prima cosa dobbiamo sapere che la **func init()** è una funzione che viene **eseguita prima** della **func main(),** una ed una sola volta. Questa funzione è utilizzata per gestire, come in questo caso, la connessione ad un database, ma anche per Kafka, e altri servizi simili.

```
package main
import "database/sql"

var Client *sql.DB
var dataSourceName  string =
fmt.Sprintf("%s:%s@tcp(%s)/%s?charset=utf8","root", "admin", "localhost",
"db_users",)

func init() {

    var err error
    Client, err = sql.Open("mysql", dataSourceName)
    if err != nil {
        panic(err)
    }
    if err = Client.Ping(); err != nil {
        panic(err)
    }
    fmt.Println("database successfully configured")
```

```
}
```

Una volta creata la connessione al database, scriviamo le query:

```
const (
    queryInsertUser             = "INSERT INTO users(id,first_name, last_name,
email, date_created, status, password) VALUES(?, ?, ?, ?, ?, ?, ?);"
    queryGetUser                = "SELECT id, first_name, last_name, email,
date_created, status FROM users WHERE id=?;"
    queryUpdateUser             = "UPDATE users SET first_name=?, last_name=?,
email=? WHERE id=?;"
    queryDeleteUser             = "DELETE FROM users WHERE id=?;"
    queryFindByStatus           = "SELECT id, first_name, last_name, email,
date_created, status FROM users WHERE status=?;"
    queryFindByEmailAndPassword = "SELECT id, first_name, last_name, email,
date_created, status FROM users WHERE email=? AND password=? AND status=?"
)
```

Abbiamo la struttura User:

```
type User struct {
    Id          int64
    FirstName   string
    LastName    string
    Email       string
    DateCreated string
    Status      string
    Password    string
}
type Users []User
```

Andiamo a vedere come sfruttare i metodi della struttura ***sql.DB** per effettuare i vari tipi di scrittura e lettura:

```
func (user *User) Get() error {
    stmt, err := Client.Prepare(queryGetUser)
    if err != nil {
        return err
    }
    defer stmt.Close()
```

```
    result := stmt.QueryRow(user.Id)

    if getErr := result.Scan(&user.Id, &user.FirstName, &user.LastName,
&user.Email, &user.DateCreated, &user.Status); getErr != nil {
        return getErr
    }
    return nil
}
```

Questa è una get by id, non passiamo parametri perché questo è un metodo pointer receiver, il risultato va a modificare l'oggetto a cui il puntatore sta puntando, è quindi richiamato con una sintassi del tipo:

```
dao := &User{Id: userId}
if err := dao.Get(); err != nil {
   return nil, err
}
```

```
func (user *User) Save() error {

    stmt, err := users_db.Client.Prepare(queryInsertUser)
    if err != nil {
        return err
    }
    defer stmt.Close()

    _, saveErr := stmt.Exec(user.Id, user.FirstName, user.LastName, user.Email,
user.DateCreated, user.Status, user.Password)
    if saveErr != nil {
        return saveErr
    }

    return nil
}
```

```
func (user *User) Update() error {
    stmt, err := users_db.Client.Prepare(queryUpdateUser)
    if err != nil {
```

```
        return err
    }
    defer stmt.Close()

    _, err = stmt.Exec(user.FirstName, user.LastName, user.Email, user.Id)
    if err != nil {
        return err
    }
    return nil
}
```

```
func (user *User) Delete() error {
    stmt, err := users_db.Client.Prepare(queryDeleteUser)
    if err != nil {
        return err
    }
    defer stmt.Close()

    if _, err = stmt.Exec(user.Id); err != nil {
        return err
    }
    return nil
}
```

```
func (user *User) FindByStatus(status string) ([]User, error) {
    stmt, err := users_db.Client.Prepare(queryFindByStatus)
    if err != nil {
        return nil, err
    }
    defer stmt.Close()

    rows, err := stmt.Query(status)
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    results := make([]User, 0)
    for rows.Next() {
```

```
        var user User
        if err := rows.Scan(&user.Id, &user.FirstName, &user.LastName,
&user.Email, &user.DateCreated, &user.Status); err != nil {
            return nil, err
        }
        results = append(results, user)
    }
    if len(results) == 0 {
        return nil, err
    }
    return results, nil
}
```

Defer

Abbiamo visto l'utilizzo, in tutti i metodi che interagiscono con il database, dello statement **defer stmt.Close()**. Questa keyword permette di definire un'istruzione che verrà eseguita dopo il return del metodo nel quale viene utilizzata. Nel caso in cui più istruzioni abbiano il prefisso defer, esse verranno eseguite nell'ordine inverso rispetto al quale sono inserite. Vediamo due semplici esempi:

```
package main
import "fmt"
func main() {
      defer fmt.Println("world")
      fmt.Println("hello")
}

Output:
hello
world
```

```
package main
import "fmt"
func main() {
      fmt.Println("counting")
      for i := 0; i < 10; i++ {
            defer fmt.Println(i)
```

```
        }
        fmt.Println("done")
}

Output:
counting
done
9
8
7
6
5
4
3
2
1
0
```

Un’altro esempio o caso d’uso è quello dell’apertura di un file:

```
f, _ := os.Open(filename)
defer f.Close()
```

nexum

Panic, Recover

La funzione built-in **panic(string)** è in grado di interrompere l'esecuzione dell'applicazione.
La funzione **recover()** è in grado di "fermare" panic ma deve essere utilizzata con defer.
Vediamo un esempio:

Non funziona:

```
package main

import "fmt"

func main() {
  panic("PANIC")
  str := recover()
  fmt.Println(str)
}
```

Funziona:

```
package main

import "fmt"

func main() {
  defer func() {
    str := recover()
    fmt.Println(str)
  }()
```

```
  panic("PANIC")
}
```

Gorm

```
package gorm

import (
    "fmt"
    "log"
    "math/rand"

    "gorm.io/driver/mysql"
    "gorm.io/gorm"
)

var dataSourceName =
fmt.Sprintf("%s:%s@tcp(%s)/%s?charset=utf8mb4&parseTime=True&loc=Local",
    "root", "admin", "127.0.0.1:3306", "db_users",
)

type User struct {
    Id          int64  `gorm:"column:id"`
    FirstName   string `gorm:"column:first_name"`
    LastName    string `gorm:"column:last_name"`
    Email       string `gorm:"column:email"`
    DateCreated string `gorm:"column:date_created"`
    Status      string `gorm:"column:status"`
    Password    string `gorm:"column:password"`
```

```
}

type UserDTO struct {
    //Id          int64  `gorm:"column:id"`
    FirstName   string `gorm:"column:first_name"`
    LastName    string `gorm:"column:last_name"`
    Email       string `gorm:"column:email"`
    DateCreated string `gorm:"column:date_created"`
    Status      string `gorm:"column:status"`
    //Password    string `gorm:"column:password"`
}



type Tabler interface {
    TableName() string
}

// TableName overrides the table name used by Album to `album`
func (User) TableName() string {
    return "users"
}
func conn() (gorm.DB, error) {
    db, err := gorm.Open(mysql.Open(dataSourceName), &gorm.Config{})
    if err != nil {
        return *db, err
    }
    log.Println("Connection Open: " + db.Name())
    return *db, nil
}

func GetUserByEmail(email *string) ([]User, error) {
    db, err := conn()
    if err != nil {
        return nil, err
    }
    const per string = "%"
    param := per + *email + per
    rows, err := db.Model(&User{}).Where("email LIKE ?",
param).Find(&UserDTO{}).Rows()
    if err != nil {
```

```
        return nil, err
    }
    defer rows.Close()
    users := make([]User, 0)
    var user User
    for rows.Next() {
        errror := db.ScanRows(rows, &user)
        if errror != nil {
            return nil, errror
        }
        users = append(users, user)
    }
    return users, nil
}

func Create(user *User) error {
    user.Id = rand.Int63()
    db, err := conn()
    if err != nil {
        return err
    }
    result := db.Create(&user)
    return result.Error
}

func Update(user *User) error {
    user.Id = rand.Int63()
    db, err := conn()
    if err != nil {
        return err
    }
    result := db.Save(&user)
    return result.Error
}
```

Gestione Errori

Vediamo un esempio di gestione degli errori provenienti dal driver di MySql che stiamo utilizzando nella nostra applicazione:

```
package mysql_utils

import (
    "errors"
    "golang-users/utils/rest_errors"
    "strings"

    "github.com/go-sql-driver/mysql"
)

const (
    ErrorNoRows = "no rows in result set"
)

//http://go-database-sql.org/errors.html

func ParseError(err error) rest_errors.RestErr {
    sqlErr, ok := err.(*mysql.MySQLError)
    if !ok {
        if strings.Contains(err.Error(), ErrorNoRows) {
            return rest_errors.NewNotFoundError("no record matching given id")
        }
        return rest_errors.NewInternalServerError("error parsing database
response", err)
```

```
    }

    switch sqlErr.Number {
    case 1062:
        return rest_errors.NewBadRequestError("invalid data")
    }
    return rest_errors.NewInternalServerError("error processing request",
errors.New("database error"))
}
```

Per approfondire e gestire gli altri codici di errore **sqlErr.Number** vi invito a consultare la documentazione:
http://go-database-sql.org/errors.html
In caso di problemi di connessione potete consultare il forum:
https://github.com/go-sql-driver/mysql/pull/302