Let’s GO !

An introduction to the GO language.

Shawn Gwin

Manager of Software Engineering at Higher Logic

Higher Logic is hiring!

- Senior Software Engineer
- Software Engineer
- Cloud Engineer
- Data Engineer

Details @ https://www.higherlogic.com/about/company/jobs/

History of why GO was created

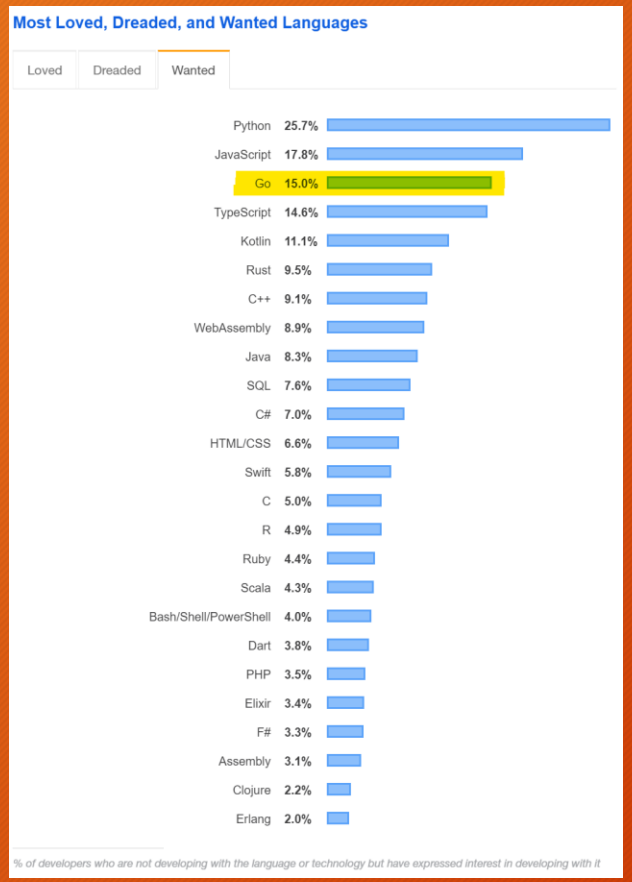
- Created by Ken Thompson, Rob Pike, Robert Griesemer at Google in 2007.
- They were frustrated with the complexity to use languages like C, C++, and Java and the lack of support for the new multicore processors.
- Go was created to address:
  - Ease of programming.
  - Efficient compilation.
  - Efficient execution (with concurrency and parallelism built in)

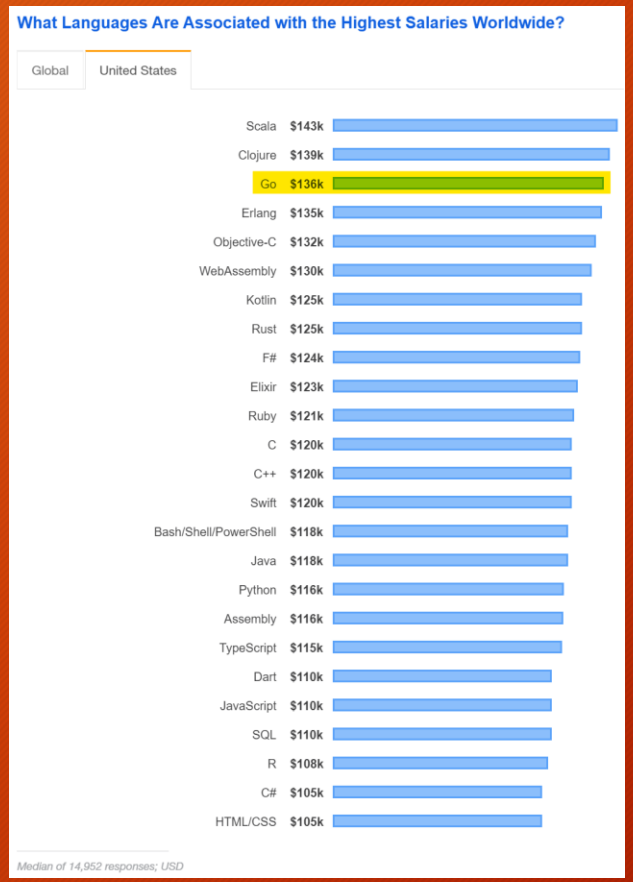
What is Go good at?

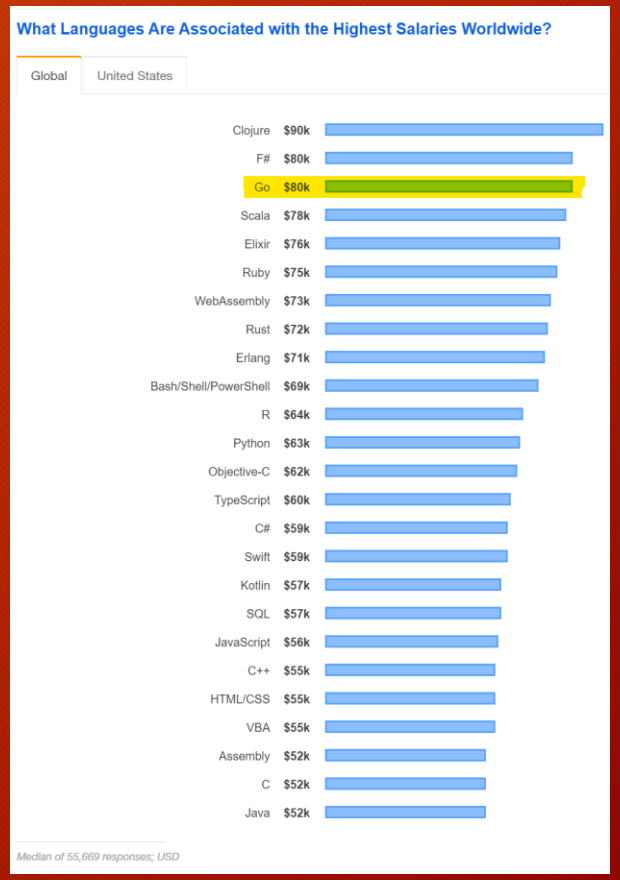
- Web Applications
- Networking (http, tcp, udp)
- Automation
- Cryptography
- Image Processing
- Micro-services
- Command-line tools
- Machine Learning
- Load Balancers/Network Servers
- File Compression
- File Encoding
- Mobile Applications
- … and much more!

Why would you want to learn Go as a developer?

- In addition to the previous reasons…

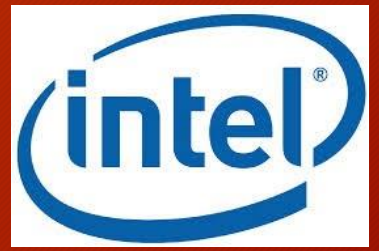






Stackoverflow – 2019 Developer Survey

Who uses Go?



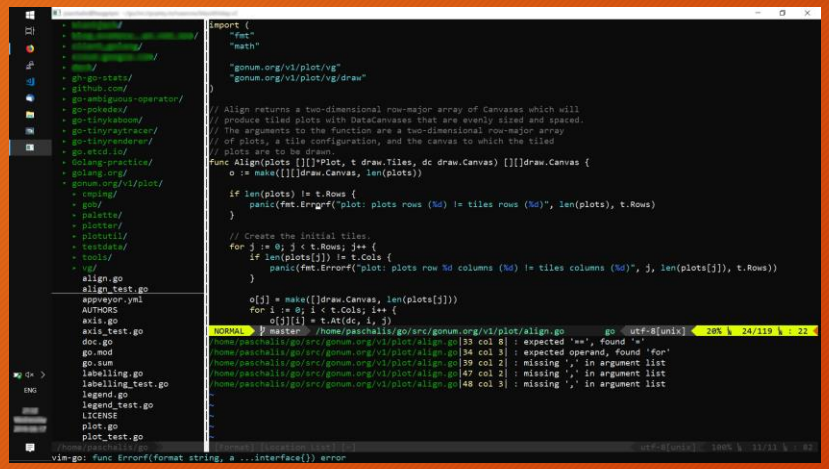
Updated list found here → https://github.com/golang/go/wiki/GoUsers

Up and running with Go

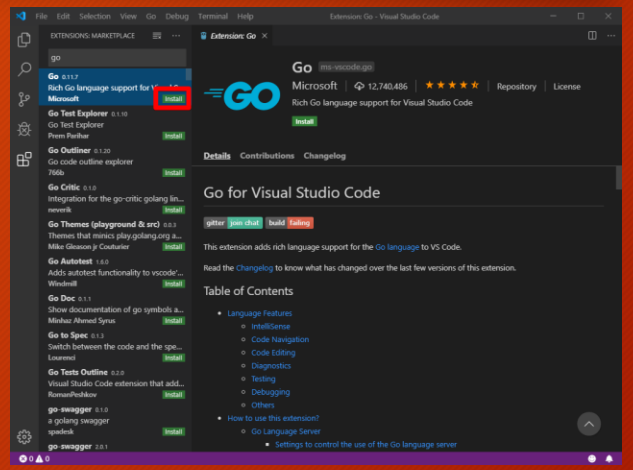
- play.golang.org – Immediately start working with the langage with this interactive playground. Only major limitations are that you do not get any intellisense and you cannot compile to an executable.
- golang.org– Click on the Download button and then follow the instructions for your operating sytem.
- hub.docker.com/_/golang - If you use Docker you can just grab this image and spin up a container.

IDE options

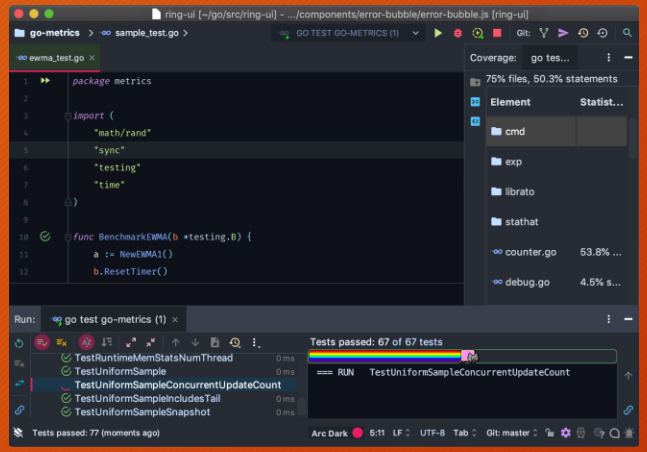
Vim



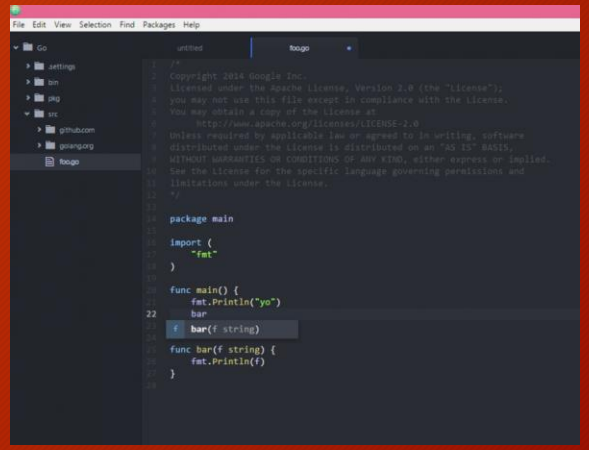
VS Code



GoLand



Atom



Hello World (because it is a requirement)

Packages are a way to organize your Go code

Use “import” keyword to access one or more libraries

“main” packages will compile as executable but all others are shared libraries

Each library name should be in quotes and on new line

Use “func” keyword to define a function

“main()” function is the entry point of our executable program

```
package main

import (
        "fmt"
)

func main(){
    fmt.Println("Hello, Let's Go!")
}
```

Imported library and function

Breaking the rules by not using “World!”

Variables and Conversions

- Explicitly typed integer with a default value of zero
  - var num int
- Explicitly type integer with a default value of 123
  - var num int = 123
- Inferred type of integer with default value of 456
  - var num = 456
- Declared and initialized variable
  - num := 7
- Convert Float to Int
  - var numFloat float32 = 1234.5678
  - var numInt int32 = int32(numFloat)

Strings

Built in String functions and manipulations:

- len(“my string”) → returns 9
- “Hello World”[6] → returns the ASCII number for “W” which is 87
- “Hello ” + “World” → concatenates the two strings and gives us “Hello World”
- “Hello\nWorld” → use special escape character combinations like the \n new line
- `Hello
- World` → use backticks ` to format multiple lines.
- In addition, there are multiple String libraries that can be utilized to manipulate strings.

Pointers

- Pointers allow you to pass a reference to a value into a functions.
- To indicate that you want a pointer to a value passed to your function use the *<type> syntax. For example, func add(iptr *int)
- When calling a function that is requesting a pointer prepend an ampersand to your variable. For example, add(&i). The will pass the memory address where the value of i is stored.
- Inside the add function, the value is obtained by dereferencing iptr by prepending an asterisk. For example, *iptr allows us to read and update the value that is in the memory address of i.

Arrays

- Arrays are specific length
- Declare:

  var test [7] int ← will create an array with seven zeros

  var lotto [5] int
  lotto[0] = 12
  lotto[1] = 17
  lotto[2] = 26
  lotto[3] = 33
  lotto[4] = 50

  c := [3]string{“apple”, “banana”, “coconut”}

- Read
  - println(lotto[2])
  - println(c)
- Length
  - len(lotto) will return 5

Slices

- Slices are built upon Go's Array, but a Slice does not have a particular length.
- Fixed array with 5 elements (length:5, capacity:5)
  - array := []string {"a", "b", "c", "d", "e"}
- Slice can be created with zero-valued elements and have more capacity than elements

  s := make([]byte, 3, 6)

  [0 0 0] - (length:3, capcity:6)

  s = append(s, 1,2,3) ← we can append elements to a slice

  [0 0 0 1 2 3]
- If you exceed the capacity of a slice a new one will be create in memory

Slices continued

- Slices can be copied from other slices or arrays

  s := []int{1,3,5,7,9}

  d := make([]int, len(s))

  copy(d, s) ← in the form (destination, source)

- Slices can be sliced

  e := s[1:4] ← Will copy 2nd up to but NOT including the 5th . e = [3 5 7]

  e := s[:3] ← Will copy the first 3 elements. e = [1 3 5]

  e := s[2:] ← Will copy all elements after the first 2. e = [5 7 9]

Structs

- Structs are a collection of fields

```
type Point struct {
        X int
        Y int
    }
p := Point{5, 9}
```

- Fields are accessed using dot.

```
p := Point{X:18, Y:3}
```
← you can implicitly define values using name: syntax.

```
p.X = 7
fmt.Println(p.X)
```

Maps

- Maps are key/value pairs
  - Define the mapping

    var m map[string]Coord ← map[key type]value type
  - Create the map

    m := make(map[string]Coord)
  - Assign values to the key

    m[“Albany_NY”] = Coord{42.6526, 73.7562}
  - Read values from map

    fmt.Println(m[“Albany_NY”])

    Output → {42.6526 73.7562}

If Else

- If (and Else) statements require brackets around what is to be exectuted if the statement is true.

```
if x > y {
        … do something
} else {
        … do something else
}
```

- If statement can start with a short statement

```
if x := rand.Intn(100); x%2==0{ ←x is only in scope until the end of the if/else statement
        … x is even
else {
        … x is odd
}
```

Switch

- Switch statements are a shorter way to write a sequence of if statements.
- The Go switch statement is similar to other languages. However, the “break” statement is implied with each case statement so only one case statement will be run that satisfies a condition.

```
switch x {
    case x > 2 :
        println(“x > 2”)
    case x = 2 :
        println(“x = 2”)
    Default:
        println(“x < 2”)
}
```

- You can use a fallthrough statement to transfer control to the next case.

For

- For is the only looping construct in Go.
- The for statement should look similar to other languages. However, Go does not have parentheses around the for statement conditions.

```
for i := 0; i < 10; i++ {
        … do something within the loop
}
```

- Can use break to exit the loop
- Can use continue to skip current condition and move to next

Range

- A range is used to iterate over elements in a data structure.
- For arrays and slices, both the element's index and value are returned for each entry.
- For maps, both the element's key and value are returned for each entry.

```
for i, v := range nums {
    println("index:", i, "value:", v)
}
```

Functions

- A function (func) in Go takes zero or more arguments
- A Go function can return multiple results
- Results must be explicitly returned to the caller using the return keyword.
- You can ignore a result by using the blank identifier “_” in the place of a variable.

```
func addMultiply(a int, b int)(int, int){
        return a+b, a*b
}
_, Y := addMultiply(5, 8) ← The addition value is ignored.
```

Goroutines

- Goroutines are a lightweight thread managed by the Go runtime
- You can simply use the “go” keyword in front of a function call to start a new goroutine

```
go addSeries(1, 10)
```

- Go does not automatically wait for a goroutine to finish before exiting the program. Therefore, setting all of your functions to run as goroutines will simply end your program.

Goroutine Channels

- Channels are a way to send and receive values between your goroutines.

  ch <- x ← data flows in the direction of the <-

  y := <-ch ← y gets the value assigned to the channel which is the value of x

- Just like slices and maps, you need to make your channels before they can be used.

  ch := make(chan int) ← channel type that contains integers

Resources

- golang.org – Where to download Go, access documentation, and the playground.
- pkg.go.dev – Search engine for all Go packages and documentation
- gobyexample.com – Many detailed examples for you to try out.
- forum.golangbridge.org – Great place to see answers to Go questions and ask your own.
- github.com/golang/go/wiki/GoUsers – Companies using Go