An Introduction to Go

Why and how to write good Go code

Radhouen Assakra

DevOps and Full stack engineer

```go
// HTMLEscape appends to dst the JSON-encoded src with <, >, &, U+2028 a
// characters inside string literals changed to \u003c, \u003e, \u0026,
// so that the JSON will be safe to embed inside HTML <script> tags.
// For historical reasons, web browsers don't honor standard HTML
// escaping within <script> tags, so an alternative JSON encoding must
// be used.
func HTMLEscape(dst *bytes.Buffer, src []byte) {
    // The characters can only appear in string literals,
    // so just scan the string one byte at a time.
    start := 0
    for i, c := range src {
        if c == '<' || c == '>' || c == '&' {
            if start < i {
                dst.Write(src[start:i])
            }
            dst.WriteString(`\u00`)
            dst.WriteByte(hex[c>>4])
            dst.WriteByte(hex[c&0xF])
            start = i + 1
        }
        // Convert U+2028 and U+2029 (E2 80 A8 a
        if c == 0xE2 && i+2 < len(src) && src[i+
            if start < i {
                dst.Write(src[start:i])
            }
            dst.WriteString(`\u202`)
            dst.WriteByte(hex[src[i+2]&0xF])
            start = i + 3
        }
    }
    if start < len(src) {
        dst.Write(src[start:])
    }
}
```

```go
func (e *MarshalerError) Error() string {
    return "json: error calling MarshalJSON for type " + e.Type.String() + ": " + e.Err
}

var hex = "0123456789abcdef"

// An encodeState encodes JSON into a bytes.Buffer.
type encodeState struct {
    bytes.Buffer // accumulated output
    scratch      [64]byte
}

var encodeStatePool sync.Pool

func  *encodeState {
    if v  encodeStatePool.Get(); v != nil {
         = (*encodeState)
        .Re()
```

just for

JustForFunc: Programming in Go

20,990 subscribers • 58 videos

Series of talk recordings and screencasts mainly about Go and the Google

The Go Programming Language

22,831 subscribers • 55 videos

Videos about working with the **Go Programming** Language.

Agenda

15 Hours	15 Hours	20 Hours
- Go basics - Type System	- Concurrency - Performance Analysis - Tooling - Advanced Topics - Q&A	- Go for Web Development

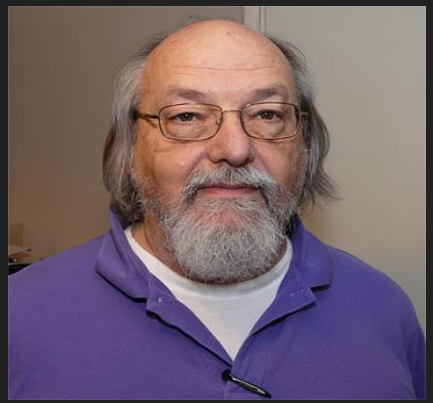
Chapter 1

GO PROGRAMMING-OVERVIEW

Plan

- Features of Go Programming
- Features Excluded Intentionally
- Go Programs
- Compiling and Executing Go Programs

Go Team



Ken Thompson

David Symonds

Robert Griesemer

Rob Pike

What is Go?

An open source (BSD licensed) project:

- Language specification,
- Small runtime (garbage collector, scheduler, etc),
- Two compilers (gc and gccgo),
- A standard library,
- Tools (build, fetch, test, document, profile,format),
- Documentation.

Language specs and std library are backwards compatible in Go 1.x.

What is Go?

Go is a compiled programming language. Before you run a program, Go uses a compiler to translate your code into the 1s and 0s that machines speak. It compiles all your code into a single executable for you to run or distribute. During this process, the Go compiler can catch typos and mistakes

Why Go?

We wanted a language with the safety and performance of **statically compiled** languages such as C++ and Java, but the lightness and fun of dynamically typed **interpreted languages** such as Python.

** Rob Pike **

Why Go?

- Like C, but with garbage collection, memory safety, and special mechanisms for concurrency
- Pointers but no pointer arithmetic
- No header files
- Simple, clean syntax
- Very fast native compilation (about as quick to edit code and restart as a dynamic language)
- Easy-to-distribute executables
- No implicit type coercions
- Simple built-in package system
- Simple tools
- Inferred types on variable declarations
- Slices and maps (feel like arrays and hashmaps in dynamic languages)
- Explicit error handling with error values and multiple return
- Interface-based polymorphism
- Interfaces implicitly implemented (allowing post-hoc interfaces for imported types)
- Goroutines (runtime managed lightweight threads)
- Channels for coordinating goroutines and sharing data between them (based on the theory of CSP)

Go History , THE PAST 2007–2009

Together these presentations provide a rationale for a new language, originally designed for Google's software development needs.
As it turns out—because we all need software—Go has become a pretty good fit for anyone writing large scale server software.
Because, at its core, the goal of Go is to improve developer productivity.

Go History , THE PRESENT 2009–2018

When Go was open sourced on the 11th of November 2009 it supported Linux, Mac OS X, on 386, amd64, and if you were running Linux, ARMv5 and v6.
By the time Go 1.0 was launched in March of 2012 we added support for Windows, FreeBSD, OpenBSD

THE PLATFORMS

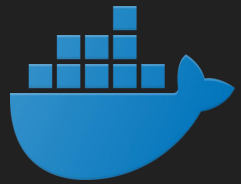
- In Go 1.3 we added support for FreeBSD, DragonfyBSD, OpenBSD, and NetBSD,Plan 9 on 386 and Native Client (NaCl), and Solaris on amd64

- Go 1.4 added support for cross compling to Android, NaCl on ARM, and Plan 9 amd64
- Go 1.5 added support for arm64 on Linux and OS X.
- Go 1.6 added support for 64bit MIPS on Linux, and Android on 386
- Go 1.7 added support for IBM System/z and 64 bit PowerPC
- Go 1.8 added support for 32 bit MIPS
- Go 1.11 added support for web assembly and plans are in the works for a RISC-V port

THE COMPANIES

Atlassian, Heptio, Digital Ocean, Netflix, Pulimi,Twitch, Google, Microsoft, Reddit, Cloudflare,MongoDB, InfluxDB, Datadog, bookings.com, Rakuten,GitHub, GitLab, Freelancer, Fastly, Netlify, Pivotal,Couchbase, Lyft, Monzo, Uber, Source{d}, srcgraph, ...

THE PROJECTS



Go History , THE futur 2018 - now

“Our goal for Go 2 is to fix the most significant ways Go fails to scale.”
—Russ Cox, GopherCon 2017

Top three pain points for Go developers:

- Dependency management - modules
- Error handling - check, handle, and error values
- Generics

Go History , THE futur 2018 - now

GO MODULES:
The first improvement is the addition of a new concept to the Go tool, a module.

A module is a collection of packages.

Just as we have .go source files grouped into a package, so too can a collection of packages with shared prefix be considered a module.

Now, this probably looks pretty close to a concept that you aready know, a git repository. But there is an important difference, modules have an explicit understanding of versions.

Go History , THE futur 2018 - now

WHY DO WE NEED G O MODULES ?

Prior to Go modules, go get only knew how to fetch whatever revision happened to be current in your repository at the time. If you already had a copy of a package in your $GOPATH then go get would skip it, so you might end up building against a really old version.
If you used the go get -u flag to force it to download a fresh copy, you might find that you now had a much newer version of a package than the author.

Go 1.x

Released in March 2012

A specification of the language and libraries supported for years. The guarantee: code written for Go 1.0 will build and run with Go 1.x. Best thing we ever did.

Hello, GoMyCode!

```
package main

import "fmt"  func main() {
    fmt.Println("Hello, GoMyCode")
}
```

Exercice:

True, false question ?

1. Go is an interpreted programming language ?
2. Go developed by facebook ?
3. Go is a statically typed ?

Chapter 2

GO PROGRAMMING-ENVIRONMENT SETUP

Plan

- Try it Option Online
- Local Environment Setup
- Text Editor
- The Go Compiler
- Installation on UNIX/Linux/Mac OS X, and FreeBSD
- Installation on Windows
- Verifying the Installation

Try it Option Online

The quickest way to get started with Go is to navigate to **play.golang.org**. At the Go Playground (figure 1.2) you can edit, run, and experiment with Go without needing to install anything. When you click the Run button, the playground will compile and execute your code on Google servers and display the result.

Local Environment Setup : editor

To play with Go locally , we need to install two necessary things :

- IDE or text editor
- Go Compiler

You will require a text editor to type your programs. Examples of

- **Text editors**: include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.
- **IDE**: Goland , eclipse , vscode ....

Local Environment Setup : Go compiler

The source code written in source file is the human readable source for your program. It needs to be compiled and turned into machine language so that your CPU can actually execute the program as per the instructions given.

The Go programming language compiler compiles the source code into its final executable program. Go distribution comes as a binary installable for FreeBSD (release 8 and above), Linux, Mac OS X (Snow Leopard and above), and Windows operating systems with 32-bit (386) and 64-bit (amd64) x86 processor architectures

Installation on UNIX/Linux/Mac

To install Go on your laptop that use unix/linux/macos or freebsd system you need :

1. Download the latest version of Go installable archive file Go Download,
2. Extract the download archive into the folder /usr/local, creating a Go tree in /usr/local/go. For example:

```
$ tar -C /usr/local -xzf go1.4.linux-amd64.tar.gz
```

Add **/usr/local/go/bin** to the PATH environment variable.

Installation on UNIX/Linux/Mac

OS	Output
Linux	go1.4.linux-amd64.tar.gz
Mac	go1.4.darwin-amd64-osx10.8.pkg
FreeBSD	go1.4.freebsd-amd64.tar.gz

Chapter 3

GO-PROGRAMMING
Introduction

Plan

- Syntax
- Base types
- Function & Variables Declarations
- Garbage Collection
- Running Go Code

Syntax:Program structure

```
package main //(1)

import "fmt" //(2)

func main() { //(3)
    fmt.Println("Hello,
    Golang")
}
```

(1) The main package is the starting point to run the program,

(2) The next line **import "fmt"** is a preprocessor command which tells the Go compiler to include the files lying in the package fmt.

(3) The next line func main() is the main function where the program execution begins. the function main will use the package fmt and here function Println to write “Hello, Golang” on the console when we run the program.

Syntax: comment

```
package main
// import packages (1)

import "fmt"

func main() {

/* here we will create variables,

constants, call functions ,,,

*/ (2)
     fmt.Println("Hello, Golang")
}
```

(1) Line comment
(2) Block comment

==> A comment is a line that will be ignored by the go compiler, Including comments in programs makes code more **readable** for humans as it provides some information or explanation about what each part of a program is doing.

Syntax: keywords

The following list shows the reserved words in Go. These reserved words may not be used as constant or variable or any other identifier names.

break	default	func	interface	select
case	defer	Go	map	Struct
chan	else	Goto	package	Switch
const	fallthrough	if	range	Type
continue	for	import	return	Var

Data Type

Type	Description
Boolean types	consists of the two predefined constants: **true,false**
Numeric types	consists of two types:integer,float types
String types	A string is a sequence of one or more characters (letters, numbers, symbols) that can be either a constant or a variable
Derived types	• **Aggregate Type:** Array and structs come under this category. • **Reference Type:** Pointers, slices, maps, functions, and channels come under this category. • **Interface Type**

Data Type: Integer

Go's integer types are: **uint8**, **uint16**, **uint32**, **uint64**, **int8**, **int16**, **int32** and **int64**. **8**, **16**, **32** and **64** tell us how many bits each of the types use. **uint means "unsigned integer"** while **int means "signed integer"**. Unsigned integers only contain positive numbers (or zero). In addition there two alias types: byte which is the same as uint8 and rune which is the same as int32. Bytes are an extremely common unit of measurement used on computers (1 byte = 8 bits, 1024 bytes = 1 kilobyte, 1024 kilobytes = 1 megabyte, …) and therefore Go's byte data type is often used in the definition of other types. There are also 3 machine dependent integer types: uint, int and uintptr. They are machine dependent because their size depends on the type of architecture you are using.

uint8	unsigned 8-bit integers (0 to 255)
uint16	unsigned 16-bit integers (0 to 65535)
uint32	unsigned 32-bit integers (0 to 4294967295)
uint64	unsigned 64-bit integers (0 to 18446744073709551615)
int8	signed 8-bit integers (-128 to 127)
int16	signed 16-bit integers (-32768 to 32767)
int32	signed 32-bit integers (-2147483648 to 2147483647)
int64	signed 64-bit integers (-9223372036854775808 to 9223372036854775807)

Generally if you are working with **integers** you should just use the **int** type.

Data Type: Integer(example)

```
package main

import "fmt"
func main()  {
        var x int32 = -5
        var y int8 = 58960 //constant
58960 overflows int8 (1)
        var s uint16 = -1 //constant -
1 overflows uint16 (2)
        fmt.Println(x)
        fmt.Println(y)
        fmt.Println(s)
}
```

(1) constant 58960 overflows int8 , while int8 is 8-bits integers so (-127 to 127) so this is handle an error overflows,

(2) the second error that -1 overflows uint16 because uint16 is for positive number only,

So we must be careful when we defined a type of variable (signed/not and the interval(N-bits)).

Data Type: Float and complex

Floating point numbers are numbers that contain a decimal component (real numbers).

Ken Thompson, one of the principal Go authors, wanted complex numbers in Go and so he added them to the Go language specification and implemented complex numbers for the Go gc compilers

Floats and complex numbers also come in varying sizes:

```
float32     IEEE-754 32-bit floating-point numbers
float64     IEEE-754 64-bit floating-point numbers
complex64   complex numbers with float32 real and imaginary parts
complex128  complex numbers with float64 real and imaginary parts
```

Data Type: Float,integer Operator

#	Operator
+	addition
-	subtraction
*	multiplication
/	division
%	remainder (for integr only)

```
package main

import (
            "fmt"
)

func main()  {
            fmt.Println("float example")
            var x float64 = 10
            var y float64 = 120
            var z float64 = x + y // invalid
operation (mismatched type float32 , float64)
            //to resolve this error replace float32
by float64
            fmt.Println("x + y =",x + y )
            fmt.Println("x - y =",x - y )
            fmt.Println("x * y =",x * y)
            fmt.Println("x / y =",x / y)
            fmt.Println("x % y =",x % y) // % is
not defined for float64
}
```

Data Type: String (example)

```
package main

import "fmt"

func main() {
    // Creating and initializing strings using var keyword
    var str1 string
    str1 = "Go"
    var str2 string
    str2 = "-MyCode !!!"
    // Concatenating strings  Using + operator
    fmt.Println( str1+str2)
    // Creating and initializing strings  Using shorthand declaration
    str3 := "Golang"
    str4 := "Course"
    result := str3 + " " + str4
    fmt.Println( result)

}
```

Data Type: String (example)

Print Whole String:
Because string is byte sequence, sometimes you want to:

- Print string as is.
- Print string with ASCII control chars as escape sequence: \t.
- Print string with NON-ASCII char as escape sequence: \u2665.
- Print string as hexadecimal to see the bytes.
- The fmt.Printf function has several verbs to help.

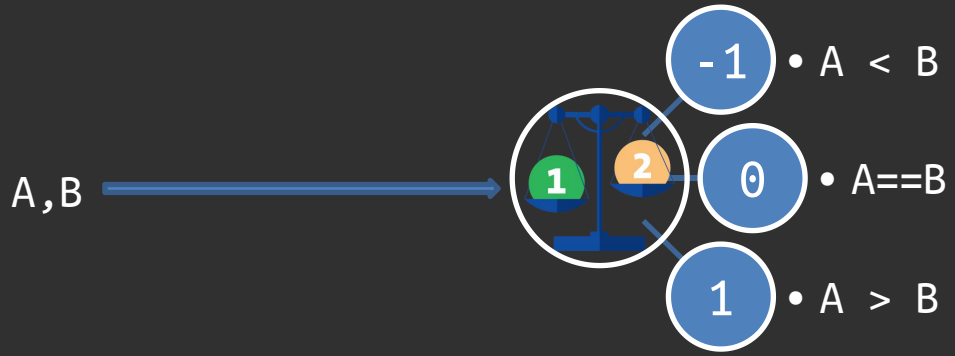
- **%s** → the uninterpreted bytes of the string or slice. If string contain arbitrary bytes or non-printable characters, this can cause problem in output.
- **%q** → output in golang string syntax, using backslash escape sequence for non-printable characters. (example: "♥\t😂")
- **%+q** → output in golang string syntax, using backslash escape for anything that's not printable ASCII. (example: "\u2665\t\U0001f602")
- **% x** → hexadecimal, with space between each 2 digits. (example: e2 99 a5 09 f0 9f 98 82) This is best if you want to see raw bytes. e.g. you want to see how the string is encoded in utf8.

Data Type: Strings package

Package strings implements simple functions to manipulate UTF-8 encoded strings.

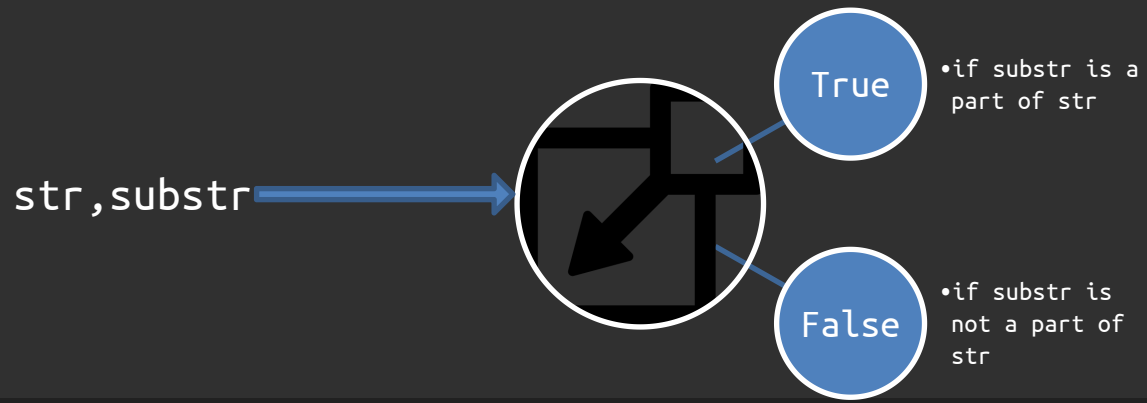
func Compare(a, b string) int
Compare returns an integer comparing two strings lexicographically. The result will be 0 if a==b, -1 if a < b, and +1 if a > b.

Compare is included only for symmetry with package bytes. It is usually clearer and always faster to use the built-in string comparison operators ==, <, >, and so on.



Data Type: Strings package

func Contains(s, substr string) bool
func Contains(s, substr string) bool



Data Type: Strings package

func Count:

```
func Count(s, substr string) int
```

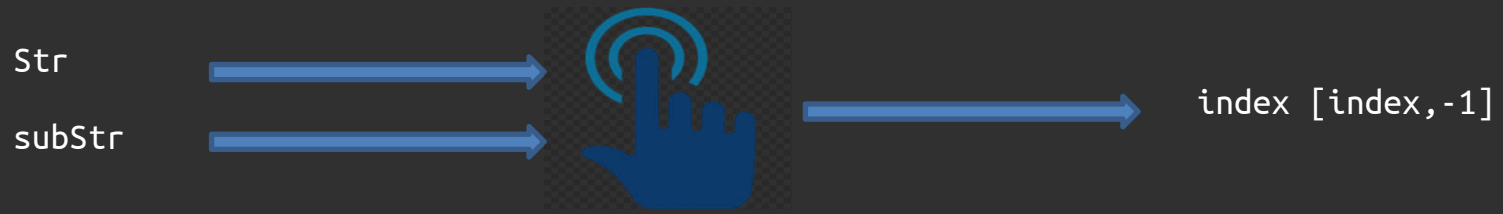
Count counts the number of non-overlapping instances of substr in s. If substr is an empty string, Count returns 1 + the number of Unicode code points in s.

Data Type: Strings package

func Index:

```
func Index(s, substr string) int
```

Index returns the index of the first instance of substr in str, or -1 if substr is not present in str.



Data Type: Strings package

func Join:

```
func Join(a []string, sep string) string
```

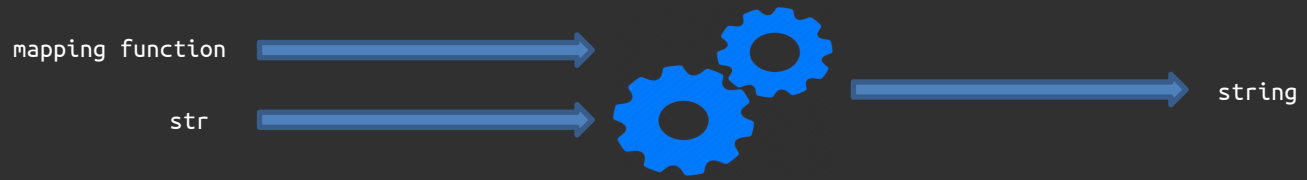
Join concatenates the elements of a to create a single string. The separator string sep is placed between elements in the resulting string.

Data Type: Strings package

func Map:

```
func Map(mapping func(rune) rune, s string) string
```

Map returns a copy of the string s with all its characters modified according to the mapping function. If mapping returns a negative value, the character is dropped from the string with no replacement.



Data Type: Strings package

func Replace:

```
func Replace(s, old, new string, n int) string
```

Replace returns a copy of the string s with the first n non-overlapping instances of old replaced by new. If old is empty, it matches at the beginning of the string and after each UTF-8 sequence, yielding up to k+1 replacements for a k-rune string. If n < 0, there is no limit on the number of replacements

Exercice:

1. Write a GO program to check whether a string is palindrome or not.
2. Write a GO program to replace first occurrence of a character with another in a string.
3. Write a Go program to remove all characters in a string except alphabets.
4. Write a Go program to count the total number of words in a string .
5. Write a program in Go to count total number of vowel or consonant in a string

Variables Declarations

- **What is a variable ?**

Variable is the name given to a memory location to store a value of a specific type. There are various syntaxes to declare variables in go.

- **Declaring single variable :**

```
var age int // variable declaration
```

- **Declaring a variable with initial value :**

```
var age int = 29 // variable declaration with initial value
```

- **Type inference :**

If a variable has an initial value, Go will **automatically** be able to infer the **type** of that **variable** using that initial value. Hence if a variable has an initial value, the type in the variable declaration can be omitted.

```
var age = 29 // type will be inferred
```

- **Multiple variable declaration :**

Multiple variables can be declared in a single statement.

```
var width, height int = 100, 50 //declaring multiple variables
```

Variables Declarations

- **Short hand declaration** :

Go also provides another concise way for declaring variables. This is known as short hand declaration and it uses **:=** operator.

```
name, age := "Radhouen", 27 //short hand declaration
 name, age := "Radhouen" //error
```

Variables Scope: local variable

- Variables that are declared inside a function or a block are termed as Local variables. These are not accessible outside the function or block.
- These variables can also be declared inside the for, while statement etc. inside a function.
- However, these variables can be accessed by the nested code blocks inside a function.
- These variables are also termed as the block variables.
- There will be a compile-time error if these variables are declared twice with the same name in the same scope.
- These variables don’t exist after the function’s execution is over.
- The variable which is declared outside the loop is also accessible within the nested loops. It means a global variable will be accessible to the methods and all loops. The local variable will be accessible to loop and function inside that function.
- A variable which is declared inside a loop body will not be visible to the outside of loop body.

Variables Scope: local variable

```
// Go program to illustrate the
// local variables
package main

import "fmt"

// main function
func main() { // from here local level scope of main function starts

 // local variables inside the main function
 var myvariable1, myvariable2 int = 89, 45

// Display the values of the variables
fmt.Printf("The value of myvariable1 is : %d\n",  myvariable1)

fmt.Printf("The value of myvariable2 is : %d\n", myvariable2)

} // here local level scope of main function ends
```

Variables Scope: Global variable

- The variables which are defined outside of a function or a block is termed as Global variables.
- These are available throughout the lifetime of a program.
- These are declared at the top of the program outside all of the functions or blocks.
- These can be accessed from any portion of the program.

Variables Scope: local variable

```
// Go program to illustrate the  global variables
package main
import "fmt"
  // global variable declaration
var myvariable1 int = 100
func main() { // from here local level scope starts
  // local variables inside the main function
var myvariable2 int = 200
  // Display the value of global variable
fmt.Printf("The value of Global myvariable1 is : %d\n", myvariable1)
  // Display the value of local variable
fmt.Printf("The value of Local myvariable2 is : %d\n",myvariable2)
// calling the function
display()
  } // here local level scope ends
  // taking a function
func display() { // local level starts
// Display the value of global variable
fmt.Printf("The value of Global myvariable1 is : %d\n", myvariable1)

} // local scope ends here
```

Exercice:

1. Select the right expresion for a comment:
   - /* ..... */
   - #
   - <?--- ?>
2. Choose the rigth variable decalaration:
   - var case = 1
   - firstChild := “Radhouen”
   - var 4G = “4G”
3. Response by True of False for this expression:

```
var a float64 = 5
var b int8 = 2
var c = a + b
```

===> c is equal to 7 ?

4. How we can define a Global variable in Go ?

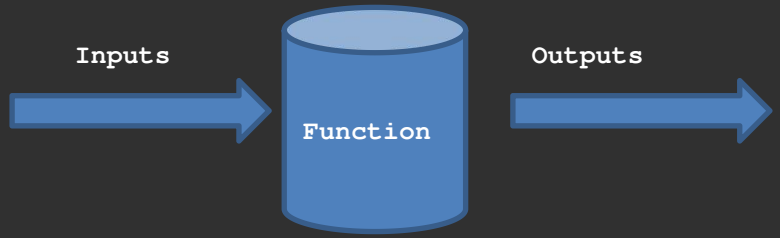
Exercice:

1. What are two ways to create a new variable?
2. What is the value of x after running:
   x := 5; x += 1 ?
3. What is scope and how do you determine the
   scope of a variable in Go?
4. What is the difference between var and const ?
5. Using the example program as a starting point,write a program that converts from Fahrenheit into Celsius. (C = (F - 32) * 5/9)
6. Write another program that converts from feet into meters. (1 ft = 0.3048 m)

Function

- **What is a function?**

A function is a block of code that performs a specific task. A function takes a input, performs some calculations on the input and generates a output.



- **Function declaration**

The general syntax for declaring a function in go is

```
func functionname(parametername type) returntype {
 //function body
}
```

Function: Simple function

The function declaration starts with a func keyword followed by the functionname. The parameters are specified between (and) followed by the returntype of the function. The syntax for specifying a parameter is parameter name followed by the type. Any number of parameters can be specified like (parameter1 type, parameter2 type). Then there is a block of code between { and } which is the body of the function.

The parameters and return type are optional in a function. Hence the following syntax is also a valid function declaration.

Example :

```
package main
import "fmt"
func add (a int, b int) int {
          somme := a + b
          return somme
}
func main()  {
          somme := add(3,5)
          fmt.Println("somme =", somme)
}
```

Function: Multiple return values

```
package main

import "fmt"

func add (a int, b int) int {
          somme := a + b
          return somme
}
func rectProps(length, width float64)(float64, float64) {
          var area = length * width
          var perimeter = (length + width) * 2
          return area, perimeter // return two parameters area, perimeter
}
func main()  {
          somme := add(3,5)
          fmt.Println("somme =", somme)

          area, perimeter := rectProps(10.8, 5.6)
          fmt.Printf("Area %f Perimeter %f", area, perimeter)
}
```

Function: Named return values

It is possible to return named values from a function. If a return value is named, it can be considered as being declared as a variable in the first line of the function.

The above rectProps can be rewritten using named return values as :

```
func rectProps(length, width float64)(area, perimeter float64) {
        var area = length * width
        var perimeter = (length + width) * 2
        return // no explicit return value
        //area and perimeter are the named return values in the above function.
}
```

Garbage Collection

It is possible to return named values from a function. If a return value is named, it can be considered as being declared as a variable in the first line of the function.

The above rectProps can be rewritten using named return values as :

```
func rectProps(length, width float64)(area, perimeter float64) {
        var area = length * width
        var perimeter = (length + width) * 2
        return // no explicit return value
        //area and perimeter are the named return values in the above function.
}
```

Running Go Code

we can run a Go program in different architecture and different operating systems,
Go support multiple a list of GOOS and GOARCH:

- **A list of valid GOOS values**

(Bold = supported by go out of the box, ie. without the help of a C compiler, etc.)

- android
- **darwin**
- **dragonfly**
- **freebsd**
- **linux**
- **nacl**
- **netbsd**
- **openbsd**
- **plan9**
- **solaris**
- **windows**
- zos

Running Go Code

- **A list of valid GOARCH values**

(Bold = supported by go out of the box, ie. without the help of a C compiler, etc.)

- **386**
- **amd64**
- **amd64p32**
- **arm**
- **armbe**
- **arm64**
- **arm64be**
- **ppc64**
- **ppc64le**
- **mips**
- **mipsle**
- **mips64**
- **mips64le**
- mips64p32
- mips64p32le
- ppc
- s390
- s390x
- sparc
- sparc64
- for more details visit : list-of-valid-goos-goarch-values

Running Go Code

- Run a mian,go example:

```
$ GOOS=darwin GOARCH=386 go build main.go
```

Exercice:code Compile

we want to calculate the newton some using this formula:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k} = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k,$$

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

```
package main
import "fmt"
func main() {
.....
}
func Somme() {
....
}
func factorial() {
...
}
func pow() {
.......
}
```

Exercice:Micro-project

1. Create a simple calculator that does this operations List:
    - Add
    - Mins
    - Multiplication
    - Subtraction
2. A simple Fibonacci sequence function. The Fibonacci sequence is the sequence of numbers where the value(n) = value(n-1) + value (n-2) with the seed values value(0) = 0 and value(1) = 1, Create a program to calculate Fibonacci(25), Fibonacci(5), Fibonacci(11).

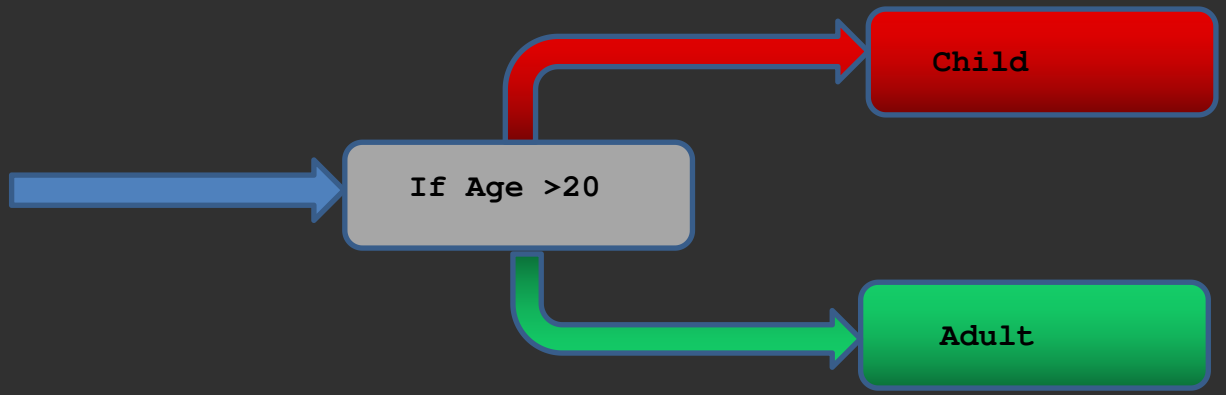
Chapter 4

GO-PROGRAMMING
Conditions and Loops

Plan

- If statement
- Switch case
- For Loop

If statement



If statement

The if statement looks as it does in C or Java, except that the ( ) are gone and the { } are required. Like for, the if statement can start with a short statement to execute before the condition. Variables declared by the statement are only in scope until the end of the if. Variables declared inside an if short statement are also available inside any of the else blocks.

- **If statement:**

```
if(boolean_expression) { // ---> { is mandatory
   /* statement(s) will execute if the boolean expression is true */
}
```

- **If..else if .. else statement:**

```
if(boolean_expression 1) {
   /* Executes when the boolean expression 1 is true */
} else if( boolean_expression 2) {
   /* Executes when the boolean expression 2 is true */
} else if( boolean_expression 3) {
   /* Executes when the boolean expression 3 is true */
} else {
   /* executes when the none of the above condition is true */
}
```

- **Nested if statements :**

It is always legal in Go programming to nest if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

If statement

- **Nested if statements :**

```
if( boolean_expression 1) {
   /* Executes when the boolean expression 1 is true */
   if(boolean_expression 2) {
      /* Executes when the boolean expression 2 is true */
   }
}
```

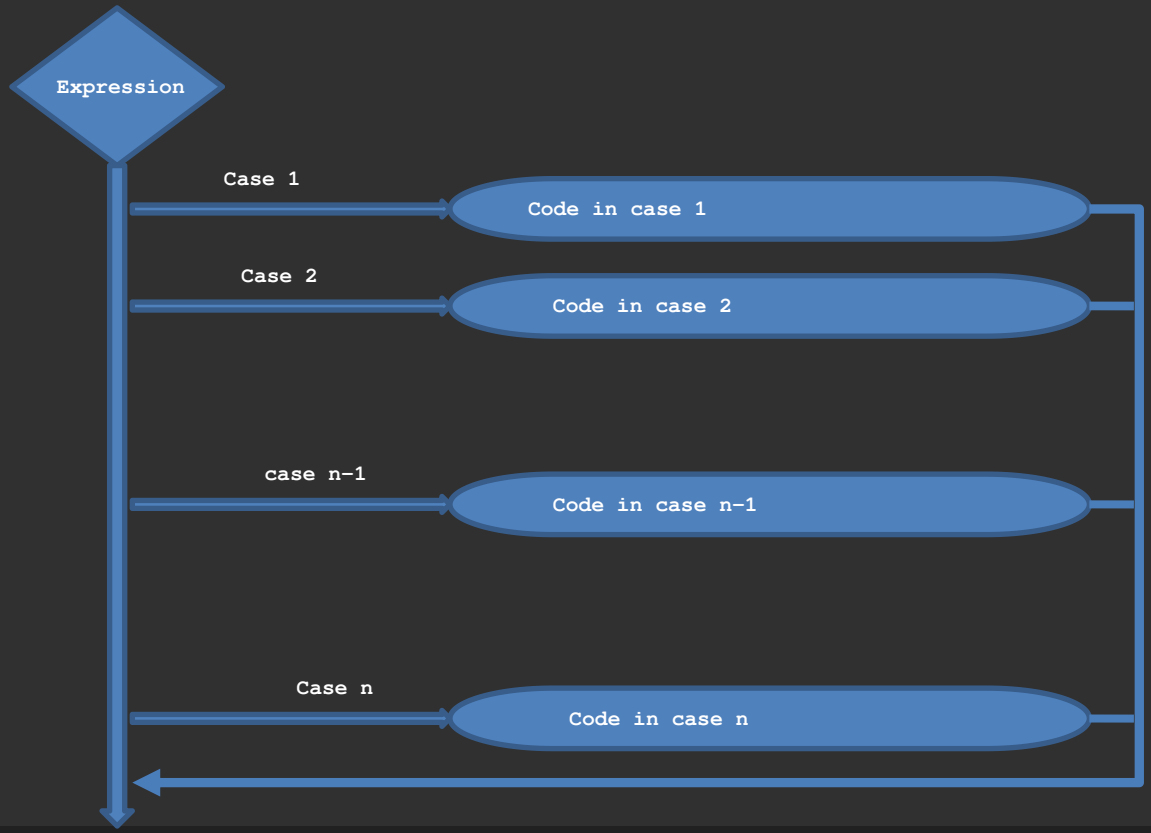
Exercice: code compile

1. We have 3 types of operating system (Linux, Unix and Windows), create a program that takes a string and return a message “this a windows/Linux/unix distribution “ in case of windows/Linux/Unix...

```
package main
import (
          "fmt"
          "strings"
       )
func main() {
fmt.Println(displayDistribution(“windows”))
fmt.Println(displayDistribution(“Linux”))
fmt.Println(displayDistribution(“unix ”))
fmt.Println(displayDistribution(“mac-os”))
}

fun displayDistribution(dist string) string {
// write code here
}
```

Switch Case


Expression
Case 1
Code in case 1
Case 2
Code in case 2
case n-1
Code in case n-1
Case n
Code in case n

Switch Case

- **Switch Case:**

```
score := 7
switch score {
case 0, 1, 3:
          fmt.Println("Terrible")
case 4, 5:
          fmt.Println("Mediocre")
case 6, 7:
          fmt.Println("Not bad")
case 8, 9:
          fmt.Println("Almost perfect")
case 10:
          fmt.Println("hmm did you cheat?")
default:
          fmt.Println(score, " off the chart")
}
```

Exercice:

1. Write a programming language that print the level for a developer **“You are a junior/senior,,, developer “**, we have **three degrees** by level, for example for junior developer, we have **j1** (one year of experience), **j2** and **j3**.
   We have multiple levels :
   1. Junior [j1,j2,j3]
   2. Confirmed[c1,c2,c3]
   3. Senior [s1,s2,s3]
   4. Manager [m1,m2,m3]
   5. Director[d1,d2,d3]

For Loop

A for loop is used for iterating over a sequence (that is either a slice, an array, a map, or a string).

As a language related to the C-family, Golang also supports for loop style control structures.

Golang has no while loop because the for loop serves the same purpose when used with a single condition.

- **Golang – traditional for Statement:**

The for loop is used when you **know in advance how many times the script should run**.

Consider the following example, display the numbers from 1 to 10 in three different ways.

```
for [condition |  ( init; condition; increment ) | Range] {
   statement(s);
}
```

For Loop

```
package main
import "fmt"
func main() {

          k := 1
          for ; k <= 10; k++ { // like for in other language
                    fmt.Println(k)
          }

          k = 1
          for k <= 10 { // look like while in other language
                    fmt.Println(k)
                    k++
          }

          for k := 1; ; k++ { // look like repeat until
                    fmt.Println(k)
                    if k == 10 {
                              break
                    }
          }
}
```

For Loop: break

The break statement in Go programming language has the following two usages –

- When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- It can be used to terminate a case in a switch statement.

If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

```
var a int = 10

/* for loop execution */
for a < 20 {
        fmt.Printf("value of a: %d\n", a);
        a++;
        if a > 15 {
                /* terminate the loop using break statement */
                break;
        }
}
```

For Loop: Continue

The **continue** statement in Go programming language works somewhat like a break statement. Instead of forcing termination, a continue statement forces the next iteration of the loop to take place, **skipping any code in between**.

In case of the for loop, continue statement causes the conditional test and increment portions of the loop to execute.

```
/* do loop execution */
for a < 20 {
          if a == 15 {
                    /* skip the iteration */
                    a = a + 1;
                    continue;
          }
          fmt.Printf("value of a: %d\n", a);
          a++;
}
```

Exercice:

1. We need to get the primary number between 1 and N(N>1) , A prime number is a positive number greater than 1 that divided only by 1 and itself.
   write a Go program to resolve this issue .
2. Write a go program that prints the number between 2 and N (N >2) multiple to (2 and 5) after (2,3 and 5).

==> code compile (add function main and a function name without impliment the code to be complited by the candidate,)

micro-project (about 10 function == 150 line end of super-skill)

workshop:

1. uses case about chapter , that mixt hard and soft skills (real case).

if we have a database table called Person (Name, UserName, Email, Age) and we want to display all Persons so we need to get all data from the table using SQL query and after we use a for loop to display it in front end page .

Name	UserName	Email	Age
Radhouen	Assakra	rassakra@gmail,com	27
John	Spada	jspada@gmail,com	32
Sami	Salhi	ssalhi@gmail.com	42

workshop:

so if we have a persons list so we can use for like this to display data in the console :

```
type Person  struct {
name string
userName string
email string
age age
}
var Persons  = []Person {
    {name : 'Radhouen', userName : "Assakra", email : 'rassakra@gmail.com', age : 26,},
    {name : 'John', userName : "Spada", email : 'jspada@gmail.com', age : 23,},
    {name : 'Sami', userName : "Salhi", email : 'ssalhi@gmail.com', age : 42,},
}

for _, person:= range Persons  {
          fmt.Println("person name :", person.name )
          fmt.Println("person userName :", person.userName )
          fmt.Println("person email :", person.email )

          }
```

one to one meeting:

1. Create at mins 10 questions/response about this chapter:

Q1.Documentation

1. Go's documentation can be read with the godoc program, which is included the Go distribution.

godoc hash gives information about the hash package:

```
% godoc hash
PACKAGE
package hash
...
...
...
SUBDIRECTORIES
adler32
crc32
crc64
fnv
```

With which godoc command can you read the documentation of fnv contained in hash?

one to one meeting:

Q1.String

1. Create a Go program that prints the following (up to 100 characters):

A
AA
AAA
AAAA
AAAAA
AAAAAA
AAAAAAA
...

2. Create a program that counts the number of characters in this string:
asSASA ddd dsjkdsjs dk
In addition, make it output the number of bytes in that string. Hint: Check out the utf8 package.

3. Extend/change the program from the previous question to replace the three runes at position 4 with 'abc'.

4. Write a Go program that reverses a string, so "foobar" is printed as "raboof". Hint: You will need to know about conversion; skip ahead to section "Conversions" on page 54.

one to one meeting:

1. Create at mins 10 questions/response about this chapter:

For...loop

1. Create a simple loop with the for construct. Make it loop 10 times and print out the loop counter with the fmt package.
2. Rewrite the loop from 1. to use goto . The keyword for may not be used.
3. Rewrite the loop again so that it fills an array and then prints that array to the screen.
4. Write a program that prints the numbers from 1 to 100. But for multiples of three print “Fizz” instead of the number and for the multiples of five print “Buzz”. For numbers which are multiples of both three and five print “FizzBuzz”.

Average

1. Write code to calculate the average of a float64 slice. In a later exercise (Q6 you will make it into a function.

Chapter 5

GO-PROGRAMMING
Data Structures

Plan

- Array
- Map
- Struct
- Interfaces
- Collection functions

Array

Go programming language provides a data structure called the **array**, which can store a fixed-size sequential **collection of elements of the same type**. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All arrays consist of **contiguous memory locations**. The lowest address corresponds to the first element and the highest address to the last element.

Array

- **Declaring Arrays:**

The type [n]T is an array of **n values** of **type T.** (n is an integer greater than 0)
Example : var tab[20] int ==> we declared an array called tab with 20 items of integer.

- **Initializing Arrays:**

```
var tab = [5] string { "Go" , "Course", "GoMyCode", "Labs" , "Session1"}
```

- **Accessing Array Elements:**

```
fmt.Println("array =", tab[0]) // will display GO
```

- **Two-Dimensional Arrays:**

var Mat [Size1][Size2] type
Example : var Mat[5][5] string // will define a Mat with 5 columns and 5 rows of string.

- **Initializing Two-Dimensional Arrays :**

```
var Mat = [2][4] string { {"Go" , "course", "GoMyCode", "Babs"},{"Go" , "course",
"GoMyCode", "Babs"}}
```

Exercice:

1. We Have an array of integer with a length N (N>10) , write a go program that can calculate the some of the array fields, after write anothor function that return the average of this array using the function some .
2. We have 15 employers, we need to calculate the salary of our company per month and per year, also we have other benefits like 10d for lunch per day and we work 20 days per month, also the salary is net, to calculate the Brut salary we applied this equation **(Brut = 1,3 * net)**.

1500	1200	2000	1800	2630	1590	1600	1550	1850	1400	1350	1250	1250	2400	1900

Map

Maps are somewhat similar to what other languages call “dictionaries” or “hashes”.

A map maps keys to values. Here we are mapping string keys (actor names) to an integer value (age).

The generic way to define a map is with:
map[<from type>]<to type>

Example:

```
monthdays := map[string]int{
"Jan": 31, "Feb": 28,"Mar":31,
"Apr": 30, "May": 31,"Jun":30,
"Jul": 31, "Aug": 31,"Sep":30,
"Oct": 31, "Nov": 30,"Dec":31,
}
```

Map

Note to use make when only declaring a map : monthdays := make(map[string]int)
For indexing (searching) in the map, we use square brackets. For example, suppose we want to print the number of days in December: fmt.Printf("%d\n", monthdays["Dec "])
If you are looping over an array, slice, string, or map a range clause will help you again, which returns the key and corresponding value with each invocation.

```
year := 0
for _, days := range monthdays {
← key unused, hence _, days
year += days
}
```

Map: Example

```
package main

import "fmt"

func main()  {
          monthdays := map[string]int{
                    "Jan": 31, "Feb": 28,"Mar":31,
                    "Apr": 30, "May": 31,"Jun":30,
                    "Jul": 31, "Aug": 31,"Sep":30,
                    "Oct": 31, "Nov": 30,"Dec":31,
          }
          fmt.Println(monthdays)
          fmt.Println("January", monthdays["Jan"])

          year := 0
          for _, days := range monthdays {
                    year += days
          }
          fmt.Println("year days = ", year)
}
```

Slice

- **Declare a Slice:**

An array has a **fixed size**. A slice, on the other hand, is a **dynamically-sized**, **flexible** view into the elements of an array. In practice, slices are much more common than arrays.

The type []T is a slice with elements of type T.

- **Subslicing**

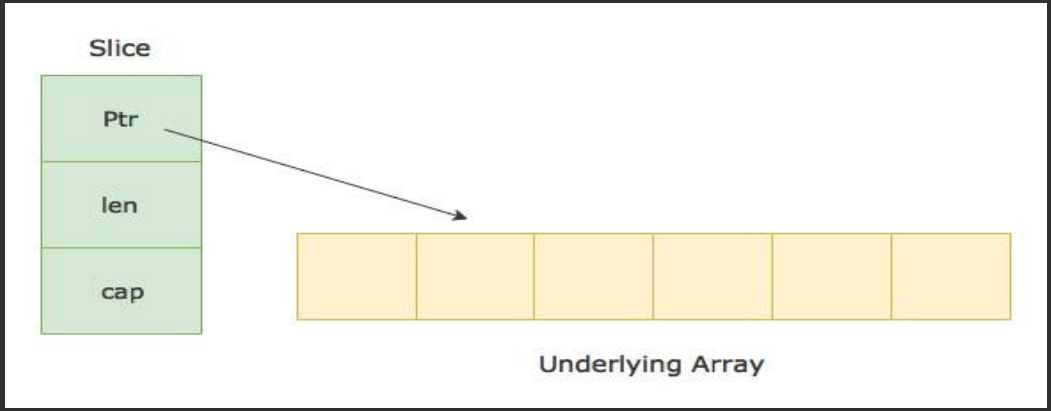
A slice is formed by specifying two indices, a **low** and **high** bound, separated by a **colon**: **a[low : high]**, This selects a half-open range which includes the first element, but excludes the last one.
The following expression creates a slice which includes elements 1 through 3 of a:
a[1:4]

Slice

- **Length and Capacity of a Slice:**

A slice consists of three things –

- A **pointer** (reference) to an underlying array.
- The **length** of the segment of the array that the slice contains.
- The **capacity** (the maximum size up to which the segment can grow).

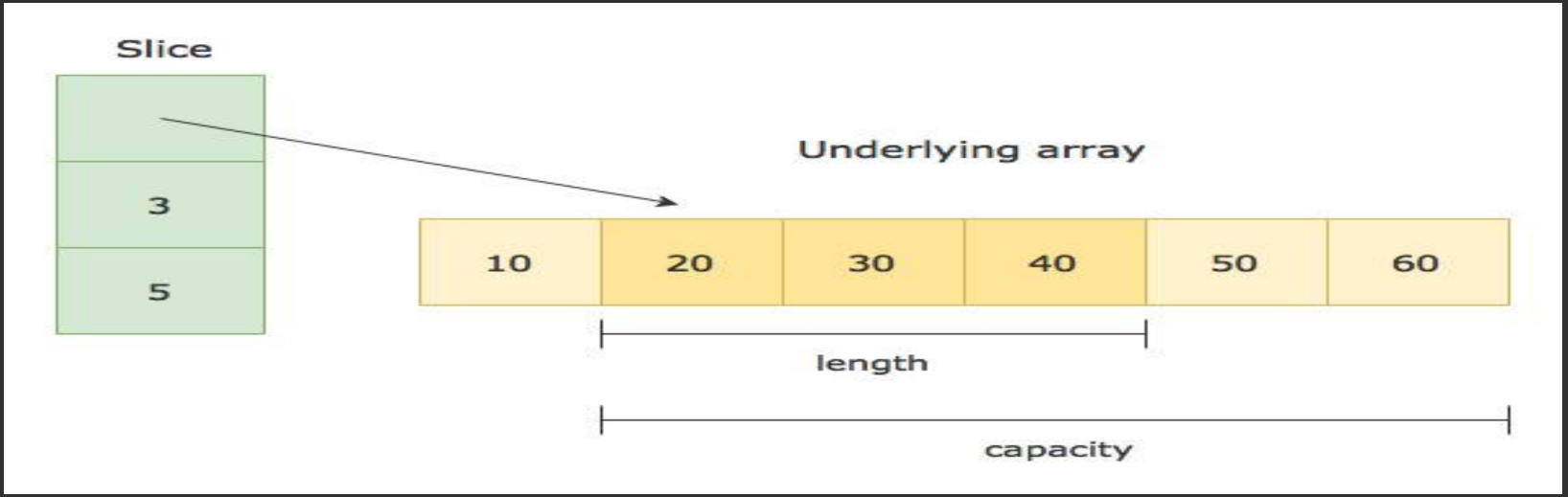


Slice

Let’s consider the following array and the slice obtained from it as an example –

```
var a = [6]int{10, 20, 30, 40, 50, 60}
var s = [1:4]
```

Here is how the slice s in the above example is represented –



Slice

The length of the slice is the number of elements in the slice, which is 3 in the above example.

The capacity is the number of elements in the underlying array starting from the first element in the slice. It is 5 in the above example.

You can find the length and capacity of a slice using the built-in functions **len()** and **cap()**

```
package main
import "fmt"

func main() {
            a := [6]int{10, 20, 30, 40, 50, 60}
            s := a[1:4]

            fmt.Printf("s = %v, len = %d, cap = %d\n", s, len(s), cap(s))
}
```

Slice: Create slice with “make”

You can create a slice with make. It lets you specify how many items to begin with, and capacity for growth.

- **make([]*type*, count_n)** → creates a slice of count_n items of type ***type***.
- **make([]*type*, count_n, capacity_m)** → with capacity of **capacity_m** items. capacity_m defaults to the value of **count_n**,
- **Example**: **make([]int, 10, 1000) → create a slice of int, 10 slots, but with total of 1000 slots for growth**.

- **Note**: capacity is not necessary for computation, because golang automatically grow the slice capacity when you append(…) beyond the capacity. However, capacity is there for efficiency reasons, because creating a new array with lots items is relatively slow. Best to always create a slice with expected growth.

Slice functions:

Slice of Slice:

- **s[a:b]** → returns a slice of s from index a to b. The a is included, The b is excluded. The result shares the same data with original.
- **s[:n]** → same as s[0:n]
- **s[n:]** → same as s[n:len(s)]

Append to Slice :

To append new items:

- **var new_slice = append(slice_x, new_item1, new_item2 etc)**
- **var new_slice = append(slice_x, slice_y ...)**
- **var new_slice = append(slice_x, string ...)**

Note the dot dot dot at the end. It turns a slice or string into items as function arguments.

Warning: append creates a new slice only when result length is greater than original slice slice_x's capacity.

Normally, append will always result a slice with capacity greater than 1st arg slice_x , therefore a new slice is created and slice_x is not modified. However, if slice_x is a slice of slice slice_w, and append result is less than capacity of slice_w, then slice_w is modified.

Slice functions: Copy

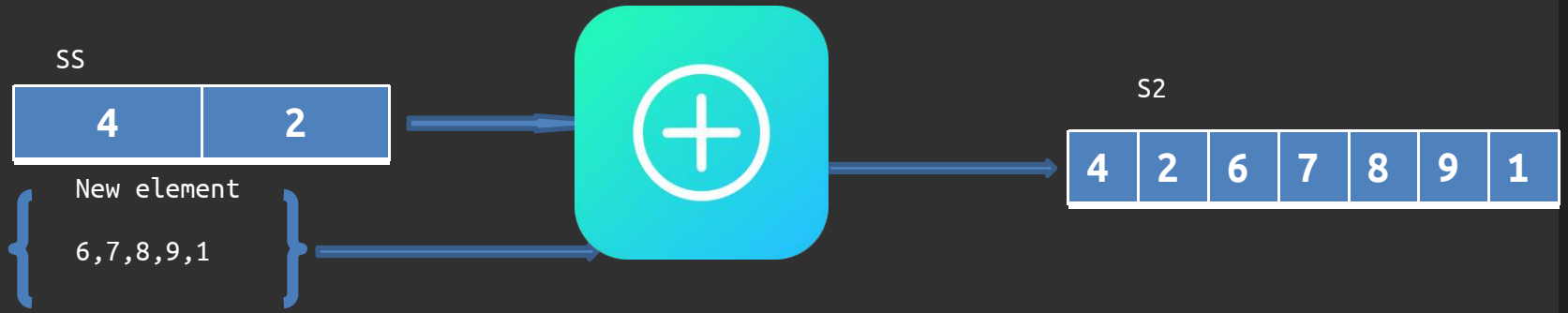
Copy Slice :
copy(dst, src) → copy elements of slice from src to dst. **The number of elements copied is the smaller of lengths of argument**, whichever is shorter. It wipes the values in the dst slice starting at index 0. Return the number of items copied.

arguments must be slice of the same type.
In this case the number of elements = min(length(dest), lenght(src)) = 3

Slice functions: Append

Append Slice :

```
var ss = []int{3, 5}
var s2 = append(ss, 8, 9)
```



Struct

Go arrays allow you to define variables that can hold several data items of the same kind. Structure is another user-defined data type available in Go programming, which allows you to combine data items of different kinds.

The declaration starts with the keyword type, then a name for the new struct, and finally the keyword struct. Within the curly brackets, a series of data fields are specified with a name and a type.

```
type identifier struct{
  field1 data_type
  field2 data_type
  field3 data_type
}
Example:
type Student struct {
firstName string
lastName string
age int
}
```

Struct

```
package main

import "fmt"

type Student struct { // create new student with three fields
          firstName string
          lastName string
          age int
}
func main()  {
          var S1 Student// All the struct fields are initialized with their zero value
// Initialize a struct by supplying the value of all the struct fields.
          S1.firstName = "Radhouen"
          S1.lastName = "Assakra"
          S1.age = 27
// we can use this to init a struct
// var S1 = Student {"Radhouen","Assakra",27}
//pS1 := new(Student ) // pS1 is a pointer to an instance of Student ,
          fmt.Println("Student Number 1=", S1)
}
```

Struct: Pointer

```
package main

import (
          "fmt"
          "reflect"
)
type Student struct {
          firstName string
          lastName string
          age int
}
func main()  {
pStudent := new(Student)
pStudent.firstName = "Radhouen"
pStudent.lastName = "Assakra"
fmt.Println("pointer to student", pStudent)

anotherStudent := &Student{"GoMyCode" , "Go course", 12}
fmt.Println("another pointer", anotherStudent)
fmt.Println(reflect.TypeOf(anotherStudent))// check the underlying type of a struct.
}
```

Interfaces

```
Interfaces are types that define a contract but not an implementation.
/* define an interface */
type interface_name interface {
   method_name1 [return_type]
   ...
   method_namen [return_type]
}
/* define a struct */
type struct_name struct {
   /* variables */
}
/* implement interface methods*/
func (struct_name_variable struct_name) method_name1() [return_type] {
   /* method implementation */
}
...
func (struct_name_variable struct_name) method_namen() [return_type] {
   /* method implementation */
}
```

Interfaces: Example

```
package main

import "fmt"

//define the interface

type Logger interface {
              Log() string
}

//define two struct

type MySqlConnection struct {
              dataBaseName string
              dataBaseUrl string
              dataBaseUserName string
              dataBasePassword string

}
type SShConnection struct {
              userName string
              url string
              password string
}
```

Interfaces: Example

```
// Impliment the interface log() functionfor MySqlConnection

func (Connection MySqlConnection) Log() string  {
if Connection.dataBaseName == "" || Connection.dataBaseUrl=="" || Connection.dataBaseUserName=="" || Connection.dataBasePassword=="" {
               return fmt.Sprintln("Please to create a secure connection , verify that the MySqlConnection data is not empty  ")
} else {
                 return fmt.Sprintln("we can launch a secure connection, congratulation", Connection)
      }

}

// Impliment the interface log() function  for SShConnection

func (Connection SShConnection) Log() string  {
if Connection.userName == "" || Connection.url=="" || Connection.password==""   {
               return fmt.Sprintln("Please to create a secure connection , verify that the SShConnection data is not empty  ")
               } else {
               return fmt.Sprintln("we can launch a secure connection, congratulation", Connection)
               }
}
```

Interfaces: Example

```
func main() {
		Connection := MySqlConnection{
				"gomycode",
				"https://www.127.168.10.10",
				"gomycode",
				"123654789/*-+",
		}
		sshConn := SShConnection{
				userName: "Radhouen",
				url:      "192.168.10.126",
				password: "gomycode123654/*-+",
		}
		fmt.Println(Connection.Log())
		fmt.Println(sshConn.Log())
}
```

Collection functions

We often need our programs to perform operations on collections of data, like selecting all items that satisfy a given predicate or mapping all items to a new collection with a custom function.

In some languages it's idiomatic to use generic data structures and algorithms. Go does not support generics; in Go it's common to provide collection functions if and when they are specifically needed for your program and data types.

for example iterating an array of string and change all items to UpperCase we have this:

```
func Map(vs []string, f func(string) string) []string {
    vsm := make([]string, len(vs))
    for i, v := range vs {
        vsm[i] = f(v)
    }
    return vsm
}
```

and then call it like so :

```
fmt.Println(Map(strs, strings.ToUpper))
```

workshop:

This section is very important because all projects needs this objects(slices,arrays,struct, map...) to represent data and organize our project .

Example : if we want to develop a web application for our university we need to create multiple struct like student, teachers, course, class, chapter .... .
if we have a list of classes so we need to define a slice of students if we don't know by default the size of the list , if else we can define an array .

one to one meeting:

Q1.Average

1. Write a function that calculates the average of a float64 slice.

Q2.Integer ordering

1. Write a function that returns its (two) parameters in the right, numerical (ascending) order:

f(7,2) → 2,7

f(2,7) → 2,7

Q3.What is wrong with the following program?

```
package main
import "fmt"
func main() {
for i := 0; i < 10; i++ {
fmt.Printf("%v\n", i)
}
fmt.Printf("%v\n", i)
}
```

one to one meeting:

Q4.Var args

Write a function that takes a variable number of ints and prints each integer on a separate line.

Q5.Fibonacci

The Fibonacci sequence starts as follows: 1, 1, 2, 3, 5, 8, 13, . . . Or in mathematical terms: x 1 = 1; x 2 = 1; x n = x n−1 + x n−2 ∀n > 2.

Write a function that takes an int value and gives that many terms of the Fibonacci sequence.

Q6.Minimum and maximum

1. Write a function that finds the maximum value in an int slice ([]int).
2. Write a function that finds the minimum value in an int slice ([]int).

Chapter 6

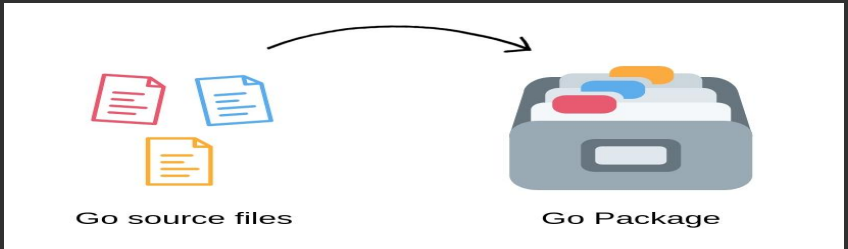
GO-PROGRAMMING
Packages and Imports

Plan

- Custom Packages
- Third-Party Packages : Go’s Commands, Go Standard Library, Archive and Compression Packages, Bytes and String-Related Packages . . Collection Packages
- File, Operating System, and Related Packages
- File Format-Related Packages
- Graphics-Related Packages
- Networking Packages
- The Reflect Package

Go Package : Overview

In the most basic terms, A package is nothing but a directory inside your Go workspace containing one or more Go source files, or other Go packages.



workspaces:

A workspace is a directory hierarchy with **two** directories at its root:

- src contains Go source files, and
- bin contains executable commands.

Go Package: benifits and main Package

Packaging functionalities in this way has the following benefits :

- It reduces naming conflicts. You can have the same function names in different packages. This keeps our function names short and concise.

- It organizes related code together so that it is easier to find the code you want to reuse.

- It speeds up the compilation process by only requiring recompilation of smaller parts of the program that has actually changed. Although we use the fmt package, we don't need to recompile it every time we change our program.

The main package and main() function

- Go programs start running in the **main package**. It is a special package that is used with programs that are meant to be executable.
- By convention, Executable programs (the ones with the main package) are called **Commands**. Others are called simply **Packages**.
- The **main()** function is a special function that is **the entry point of an executable program**.

Go Package: Import Package

Go's convention is that - the **package name** is the same as the last element of **import path**. **For example:** the name of the package imported as **math/rand is rand.** It is imported with path math/rand because It is nested inside the **math** package as a subdirectory.

```
package main

import (
    "fmt" // here we import the package fmt
    "math/rand" // here we import the package math/rand
)

func main() {
    fmt.Println(rand.Intn(30)) // here we can use the function defined inside rand package
    fmt.Println(rand.Intn(30))
    fmt.Println(rand.Intn(30))
    fmt.Println(rand.Intn(30))
    fmt.Println(rand.Intn(30))
}
```

Go Package: Custom Package

Let's create under src folder a new repository with a file_name.go :

$ mkdir /lab1/src/github.com/reverse-string/reverse.go

```
package reverse_string

func Reverse(s string) string  {
          b := []rune(s)
          for i :=0 ; i < len(b)/2 ; i++ {
                    j := len(b)-i-1
                    b[i],b[j] = b[j], b[i]
          }
          return string(b)
}
```

$ GOBIN=$GOPATH/bin go install

this command will create a new binary package under **$GOPATH/bin/github.com/reverse-string.a**

Go Package: Custom Package

Using this package:

```
package main

import "fmt"
import "lab1/src/reverse-string" // import custom package


func main() {
          fmt.Println(reverse_string.Reverse("GOLANG Course"))
          }
```

Go Commands

```
Go is a tool for managing Go source code.
$ go <<commands>> <<arguments>>
The commands are:
• bug         start a bug report
• build       compile packages and dependencies
• clean       remove object files and cached files
• doc         show documentation for package or symbol
• env         print Go environment information
• fix         update packages to use new APIs
• fmt         gofmt (reformat) package sources
• generate    generate Go files by processing source
• get         add dependencies to current module and install them
• install     compile and install packages and dependencies
• list        list packages or modules
• mod         module maintenance
• run         compile and run Go program
• test        test packages
• tool        run specified go tool
• version     print Go version
• vet         report likely mistakes in packages
```

Go Standard Library

```
Go is a tool for managing Go source code.
$ go <<commands>> <<arguments>>
The commands are:
• bug         start a bug report
• build       compile packages and dependencies
• clean       remove object files and cached files
• doc         show documentation for package or symbol
• env         print Go environment information
• fix         update packages to use new APIs
• fmt         gofmt (reformat) package sources
• generate    generate Go files by processing source
• get         add dependencies to current module and install them
• install     compile and install packages and dependencies
• list        list packages or modules
• mod         module maintenance
• run         compile and run Go program
• test        test packages
• tool        run specified go tool
• version     print Go version
• vet         report likely mistakes in packages
```

Go Standard Library: Archive

Go is a tool for managing Go source code.

Go Standard Library: Compression

```
Go is a tool for managing Go source code.
```

Go Standard Library: Collection

```
Go is a tool for managing Go source code.
```

Go Standard Library: File

```
Go is a tool for managing Go source code.
```

Go Standard Library: OS

```
Go is a tool for managing Go source code.
```

Go Standard Library: File Format

Go is a tool for managing Go source code.

Go Standard Library: Graphics-Related

```
Go is a tool for managing Go source code.
```

Go Standard Library: Networking

```
Go is a tool for managing Go source code.
```

Go Standard Library: Reflect

Go is a tool for managing Go source code.

Chapter 7

GO-PROGRAMMING
Templates

Plan

- Understanding templates
- Templating with concatenation
- Understanding package text/template: parsing & executing templates
- Passing data into templates
- Variables in templates
- Passing composite Data structures into templates
- Functions & pipelines in templates
- Predefined global functions in templates
- Nesting templates - modularizing your code

Understanding templates

Go templates are a powerful method to customize output however you want, whether you're creating a **web page**, **sending an e-mail**, working with Buffalo, Go-Hugo.

There're two packages operating with templates – **text/template** and **html/template**. Both provide the same interface, however the html/template package is used to generate HTML output safe against code injection.

Notice : Almost every programming language has a library implementing templating. In epoch of server side MVC dominance, templating was so important that it could determine language success or failure. Nowadays, however, when **single page applications** get momentum, **templates are used only occasionally**.

Understanding templates

Before we learn how to implement it, let's take a look at template's syntax. Templates are provided to the appropriate functions either as string or as "raw string". Actions represents the data evaluations, functions or control loops. They're delimited by **{{ }}**. Other, non delimited parts are left untouched.

- **Data evaluations**

Usually, when using templates, you'll bind them to some **data structure** (e.g. struct) from which you'll obtain data. To obtain data from a **struct**, you can use the **{{ .FieldName }}** action, which will replace it with **FieldName** value of given struct, on parse time. The struct is given to the **Execute** function, which we'll cover later.

There's also the **{{.}}** action that you can use to refer to a value of non-struct types.

- **Conditions**

You can also use if loops in templates. For example, you can check if FieldName non-empty, and if it is, print its value: **{{if .FieldName}} Value of FieldName is {{ .FieldName }} {{end}}**.

else and **else if** are also supported: **{{if .FieldName}} // action {{ else }} // action 2 {{ end }}**.

Passing data into templates

Loops
Using the range action you can loop through a slice. A range actions is defined using the {{range .Member}} ... {{end}} template.

If your slice is a non-struct type, you can refer to the value using the {{ . }} action. In case of structs, you can refer to the value using the {{ .Member }} action, as already explained.

Functions & pipelines in templates

- **Functions, Pipelines and Variables**

Actions have several built-in functions that're used along with pipelines to additionally parse output. Pipelines are annotated with | and default behavior is sending data from left side to the function on right side.

Functions are used to escape the action's result. There're several functions available by default such as, `html` which returns HTML escaped output, safe against code injection or js which returns JavaScript escaped output.

Using the with action, you can define variables that're available in that with block: `{{ with $x := <^>result-of-some-action<^> }} {{ $x }} {{ end }}`.

Throughput the article, we're going to cover more complex actions, such as reading from an array instead of struct.

Templating with concatenation

```
Go is a tool for managing Go source code.
```

parsing & executing templates

- **Parsing Templates:**

The three most important and most frequently used functions are:

- **New** – allocates new, undefined template,
- **Parse** – parses given template string and return parsed template,
- **Execute** – applies parsed template to the data structure and writes result to the given writer.

- **Verifying Templates:**
- **template** packages provide the **Must** functions, used to verify that a template is valid during parsing. The **Must** function provides the same result as if we manually checked for the error, like in the previous example.

- This approach saves you typing, but if you encounter an error, your application will panic. For advanced error handling, it's easier to use above solution instead of Must function.

- The **Must** function takes a template and error as arguments. It's common to provide New function as an argument to it:

Variables in templates

```
Go is a tool for managing Go source code.
```

Passing composite Data structures into templates

```
Go is a tool for managing Go source code.
```

Functions & pipelines in templates

```
Go is a tool for managing Go source code.
```

Predefined global functions in templates

```
Go is a tool for managing Go source code.
```

Nesting templates - modularizing your code

Go is a tool for managing Go source code.

Chapter 8

GO-PROGRAMMING
Introduction to Web Dev

Plan

- Understanding servers
- TCP server - Connection
- TCP server - Code a client
- TCP server - rot13 & in-memory database
- TCP server - HTTP request, method & multiplexer

- Net/Http package
- Understanding & using ListenAndServe
- Foundation of net/http: Handler, ListenAndServe, Request, ResponseWriter
- Retrieving form values - exploring *http.Request
- Retrieving other request values - exploring *http.Request
- Exploring http.ResponseWriter - writing headers to the response

Understanding servers

A web server is server software, or hardware dedicated to running said software, that can satisfy World Wide Web client requests. A web server can, in general, contain one or more websites. A web server processes incoming network requests over HTTP and several other related protocols.

An HTTP server serves data to clients using the HTTP protocol. It is also known as a web server.

The first http server was called CERN HTTPD, and it was written in 1990 by Tim Berners-Lee for the NeXTSTEP platform. Four years later, Berners-Lee initiated the creation of the World Wide Web Consortium (W3C) to regulate the development of HTTP, HTML, and other technologies related to the World Wide Web.

TCP server - Connection

The TCP server in Go that is concurrent in nature. This will enable this server to handle more than one connection. This server will also know if a client disconnects without asking the server to close the connection.

what we need to create a TCP server Connection:

- CONN_HOST
- CONN_Port
- CONN_type

Create a connection :

- **conn,err := net.Listen(CONN_TYPE, CONN_HOST+":"+CONN_PORT)**

TCP server - Connection

```
package main

import (
	"fmt"
	"net"
	"os"
)
const (
	CONN_HOST = "localhost"
	CONN_PORT = "3333"
	CONN_TYPE = "tcp"
)
func createConnection()  {
	con,err := net.Listen(CONN_TYPE,CONN_HOST+":"+CONN_PORT)
	if err != nil {
			fmt.Println("Error listening:", err.Error())
			os.Exit(1)
	}
	defer con.Close() // Close the listener when the application closes.
	fmt.Println("Listening on " + CONN_HOST + ":" + CONN_PORT)
	for   {
			req,err := con.Accept() // accept a connection
			if err != nil {

			}
			go handleRequest(req) // handle the connection will be concurrent
	}

}
```

TCP server - Connection : handel request

```
// Handles incoming requests.
func handleRequest(conn net.Conn) {
	// Make a buffer to hold incoming data.
	buf := make([]byte, 1024)
	// Read the incoming connection into the buffer.
	reqLen, err := conn.Read(buf)
	if err != nil {
		fmt.Println("Error reading:", err.Error())
	}
	// Send a response back to person contacting us.
	fmt.Println("reqlenght", reqLen)
	conn.Write([]byte("Message received."))
	// Close the connection when you're done with it.
	conn.Close()
}
```

If you build this and run it, you'll have a simple TCP server running on port 3333. To test your server, send some raw data to that port:

```
echo -n "test out the server" | nc localhost 3333
```

Response: message received

TCP server - Code a client

```
import "net"
import "fmt"
import "bufio" // Package bufio implements buffered I/O
import "os"

func main() {

  // connect to this socket
  conn, _ := net.Dial("tcp", "127.0.0.1:8081") // CONN_TYPE , CONN_HOST , CONN_PORT
  for {
    // read in input from stdin
    reader := bufio.NewReader(os.Stdin)
    fmt.Print("Text to send: ")
    text, _ := reader.ReadString('\n')
    // send to socket
    fmt.Fprintf(conn, text + "\n")
    // listen for reply
    message, _ := bufio.NewReader(conn).ReadString('\n')
    fmt.Print("Message from server: "+message)
  }
}
```

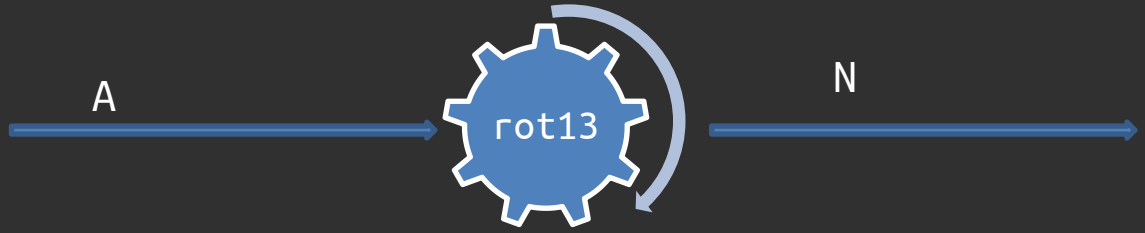
TCP server - rot13 & in-memory database

Golang ROT13 Method

This Go example uses the strings.Map method to apply the **rot13 cipher**.

ROT13:

This cipher obscures the characters in a string. With ROT13, characters are rotated 13 places in the alphabet. Other characters are not changed.



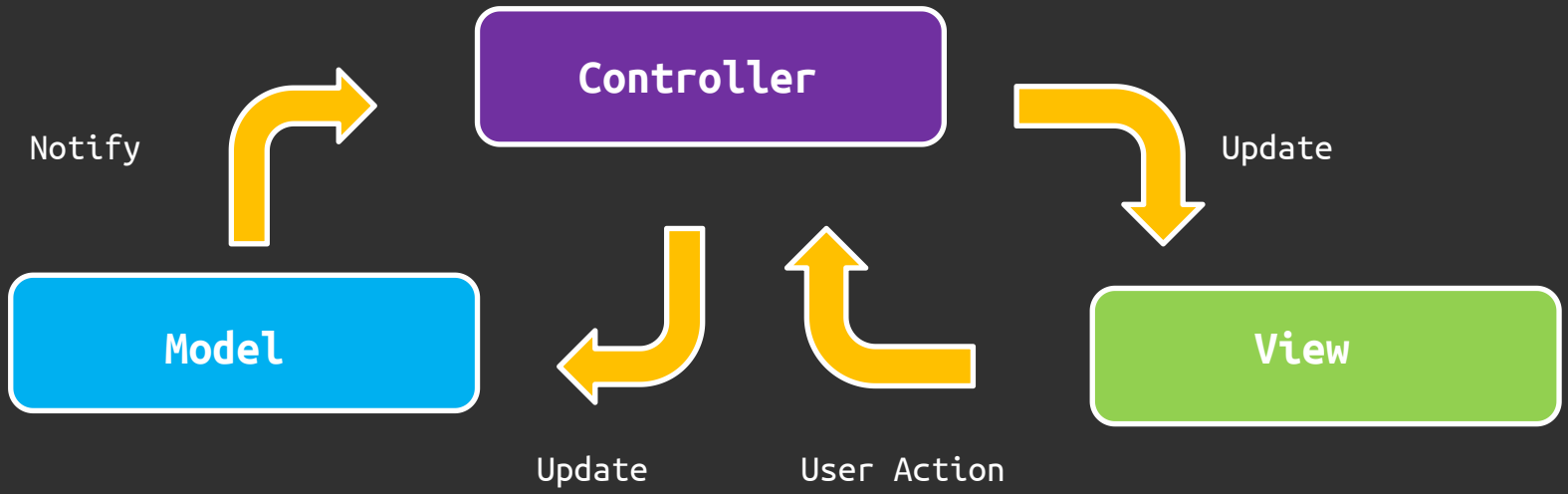
TCP server - rot13 & in-memory database

```
package main
import (
                "fmt"
                "strings"
)
func rot13(r rune) rune {
                fmt.Println("rune element :", r)
                if r >= 'a' && r <= 'z' {
                                if r >= 'm' {
                                                return r - 13
                                } else {
                                                return r + 13
                                }
                } else if r >= 'A' && r <= 'Z' {
                                if r >= 'M' {
                                                return r - 13
                                } else {
                                                return r + 13
                                }
                }
                return r
}
func main() {
                input := "Do you have any questions ?"
                mapped := strings.Map(rot13, input)
                fmt.Println(input)
                fmt.Println(mapped)
           }
```

TCP server - HTTP request, method & multiplexer

```
Go is a tool for managing Go source code.
```

Net/Http package



Net/Http package

A web server, also known as an HTTP server, uses the HTTP protocol to communicate with clients. All web browsers can be considered clients.

We can divide the web's working principles into the following steps:

- Client uses TCP/IP protocol to connect to server.
- Client sends HTTP request packages to server.
- Server returns HTTP response packages to client. If the requested resources include dynamic scripts, server calls script engine first.
- Client disconnects from server, starts rendering HTML.

This is a simple work flow of HTTP affairs-notice that the server closes its connections after it sends data to the clients, then waits for the next request.

Understanding & using Listen And Serve

ListenAndServe starts an HTTP server with a given **address** and **handler**. The **handler** is usually **nil**, which means to use **DefaultServeMux**. **Handle** and **HandleFunc** add handlers to DefaultServeMux:

```
package main

import (
            "fmt"
            "net/http"
)

func main()  {
            http.ListenAndServe("localhost:2600", nil)
}
```

If we run the project no thing happen because it miss a handler , the handler look like a call from the client or an action handled by the client to to something

Foundation of net/http:Request, ResponseWriter

Here we will create a new **mux** , a **mux** is like a **router** that give the direction or the way to acces to some fonctionality, after create a mux we will create a **handelFunc** that neet two parameters the first is a **http.ResponseWriter** and the second is a **reference of http.Request**,

```
package main

import (
            "fmt"
            "net/http"
)

func main()  {
            mux := http.NewServeMux()
            mux.HandleFunc("/", func(w http.ResponseWriter , r *http.Request) {
                        w.Write([]byte("Hello world"))
                        fmt.Println("request method", r.Method)
            })
            http.ListenAndServe("localhost:2600", mux)
}
```

Retrieving form values - exploring *http.Request

first of all we need to define a HTML form , don't forget we need an action to get the **endpoint/path** to call the back end function also the method **“POST”.**
After we need to add a name”username,password” to our inputs to use these keys in the go program, and of course an input with submit type to send the form .

```
<html>
    <head>
    <title></title>
    </head>
    <body>
        <form action="/login" method="post">
            Username:<input type="text" name="username">
            Password:<input type="password" name="password">
            <input type="submit" value="Login">
        </form>
    </body>
</html>
```

Retrieving form values – exploring *http.Request

Now what we need is to create a function with two parameters “ResponseWriter” to display the login form if the HTTP method is “GET” and the second parameters “ Http.Request” to get the data from the form.
we have two methods to get input value :

- r.FormValue(“INPUT_NAME”)
- r.ParseForm() and r.Form[“INPUT_NAME”]

```
func login(w http.ResponseWriter, r *http.Request) {
        fmt.Println("method:", r.Method) //get request method
        if r.Method == "GET" {
                t, _ := template.ParseFiles("login.gtpl")
                t.Execute(w, nil)
        } else {
                // use FormValue (1)
                fmt.Println("username", r.FormValue("username"))
                fmt.Println("password:", r.FormValue("password"))
                // logic part of login (2)
                r.ParseForm()
                fmt.Println("username:", r.Form["username"])
                fmt.Println("password:", r.Form["password"])
        }
}
```

Retrieving other request values - exploring *http.Request

There are anothor method for http.Request :

- Method
- Header
- Body
- ContentLength
- Url
- Close
- PostForm
- RemoteAddr
- RequestURI
- Cancel
- Response

https://golang.org/src/net/http/request.go

Retrieving other request values - exploring *http.Request

```
Http Status code 100 & 200:
          StatusContinue           = 100 // RFC 7231, 6.2.1
          StatusSwitchingProtocols = 101 // RFC 7231, 6.2.2
          StatusProcessing         = 102 // RFC 2518, 10.1
          StatusEarlyHints         = 103 // RFC 8297

          StatusOK                   = 200 // RFC 7231, 6.3.1
          StatusCreated              = 201 // RFC 7231, 6.3.2
          StatusAccepted             = 202 // RFC 7231, 6.3.3
          StatusNonAuthoritativeInfo = 203 // RFC 7231, 6.3.4
          StatusNoContent            = 204 // RFC 7231, 6.3.5
          StatusResetContent         = 205 // RFC 7231, 6.3.6
          StatusPartialContent       = 206 // RFC 7233, 4.1
          StatusMultiStatus          = 207 // RFC 4918, 11.1
          StatusAlreadyReported      = 208 // RFC 5842, 7.1
          StatusIMUsed               = 226 // RFC 3229, 10.4.1
```

Retrieving other request values - exploring *http.Request

```
Http Status code 300 (redirect):
          StatusMultipleChoices   = 300 // RFC 7231, 6.4.1
          StatusMovedPermanently  = 301 // RFC 7231, 6.4.2
          StatusFound             = 302 // RFC 7231, 6.4.3
          StatusSeeOther          = 303 // RFC 7231, 6.4.4
          StatusNotModified       = 304 // RFC 7232, 4.1
          StatusUseProxy          = 305 // RFC 7231, 6.4.5
          _                       = 306 // RFC 7231, 6.4.6 (Unused)
          StatusTemporaryRedirect = 307 // RFC 7231, 6.4.7
          StatusPermanentRedirect = 308 // RFC 7538, 3
```

Retrieving other request values - exploring *http.Request

```
Http Status code (400 code client):
                StatusBadRequest                   = 400 // RFC 7231, 6.5.1
                StatusUnauthorized                 = 401 // RFC 7235, 3.1
                StatusPaymentRequired              = 402 // RFC 7231, 6.5.2
                StatusForbidden                    = 403 // RFC 7231, 6.5.3
                StatusNotFound                     = 404 // RFC 7231, 6.5.4
                StatusMethodNotAllowed             = 405 // RFC 7231, 6.5.5
                StatusNotAcceptable                = 406 // RFC 7231, 6.5.6
                StatusProxyAuthRequired            = 407 // RFC 7235, 3.2
                StatusRequestTimeout               = 408 // RFC 7231, 6.5.7
                StatusConflict                     = 409 // RFC 7231, 6.5.8
                StatusGone                         = 410 // RFC 7231, 6.5.9
                StatusLengthRequired               = 411 // RFC 7231, 6.5.10
                StatusPreconditionFailed           = 412 // RFC 7232, 4.2
                StatusRequestEntityTooLarge        = 413 // RFC 7231, 6.5.11
                StatusRequestURITooLong            = 414 // RFC 7231, 6.5.12
                StatusUnsupportedMediaType         = 415 // RFC 7231, 6.5.13
                StatusRequestedRangeNotSatisfiable = 416 // RFC 7233, 4.4
                StatusExpectationFailed            = 417 // RFC 7231, 6.5.14
                StatusTeapot                       = 418 // RFC 7168, 2.3.3
                StatusMisdirectedRequest           = 421 // RFC 7540, 9.1.2
                StatusUnprocessableEntity          = 422 // RFC 4918, 11.2
                StatusLocked                       = 423 // RFC 4918, 11.3
                StatusFailedDependency             = 424 // RFC 4918, 11.4
                StatusTooEarly                     = 425 // RFC 8470, 5.2.
                StatusUpgradeRequired              = 426 // RFC 7231, 6.5.15
                StatusPreconditionRequired         = 428 // RFC 6585, 3
                StatusTooManyRequests              = 429 // RFC 6585, 4
                StatusRequestHeaderFieldsTooLarge  = 431 // RFC 6585, 5
                StatusUnavailableForLegalReasons   = 451 // RFC 7725, 3
```

Retrieving other request values - exploring *http.Request

```
Http Status code (500 server error) :
          StatusInternalServerError           = 500 // RFC 7231, 6.6.1
          StatusNotImplemented                = 501 // RFC 7231, 6.6.2
          StatusBadGateway                    = 502 // RFC 7231, 6.6.3
          StatusServiceUnavailable            = 503 // RFC 7231, 6.6.4
          StatusGatewayTimeout                = 504 // RFC 7231, 6.6.5
          StatusHTTPVersionNotSupported       = 505 // RFC 7231, 6.6.6
          StatusVariantAlsoNegotiates         = 506 // RFC 2295, 8.1
          StatusInsufficientStorage           = 507 // RFC 4918, 11.5
          StatusLoopDetected                  = 508 // RFC 5842, 7.2
          StatusNotExtended                   = 510 // RFC 2774, 7
          StatusNetworkAuthenticationRequired = 511 // RFC 6585, 6
```

Exploring http.ResponseWriter – writing headers to the response

A Header represents the key-value pairs in an HTTP header.

type Header map[string][]string

```
func saveHandler(w http.ResponseWriter, r *http.Request) {
    // allow cross domain AJAX requests
    w.Header().Set("Access-Control-Allow-Origin", "*")
   //
if origin := req.Header.Get("Origin"); origin != "" {
        rw.Header().Set("Access-Control-Allow-Origin", origin)
        rw.Header().Set("Access-Control-Allow-Methods", "POST, GET, OPTIONS, PUT, DELETE")
        rw.Header().Set("Access-Control-Allow-Headers",
            "Accept, Content-Type, Content-Length, Accept-Encoding, X-CSRF-Token, Authorization")
    }
}
```

Chapter 9

GO-PROGRAMMING
Routing and serving files

Plan

- Understanding ServeMux
- Disambiguation: func(ResponseWriter, *Request) vs. HandlerFunc
- Third-party servemux
- understanding files
- Serving & Creating a file
- log.Fatal & http.Error
- The http.NotFoundHandler

Chapter 10

GO-PROGRAMMING
States and sessions

Plan

- State overview
- Passing values
- Uploading a file, reading the file, creating a file on the server
- Encrypt
- Redirects
- Cookies

- Sessions Overview
- Universally unique identifier - UUID
- Sign-up / Encrypt password
- Login / Logout
- Permissions
- Expire session

State overview

An important topic in web development is providing a good user experience, but the fact that HTTP is a stateless protocol seems contrary to this spirit. How can we control the whole process of viewing websites for users? The classic solutions are using cookies and sessions, where cookies serve as the client side mechanism and sessions are saved on the server side with a unique identifier for every single user. Note that sessions can be passed in URLs or cookies, or even in your database (which is much more secure, but may hamper your application performance).

Cookies and Sessions are used to store information. Cookies are only stored on the client-side machine, while sessions get stored on the client as well as a server.

Session

A session creates a file in a temporary directory on the server where registered session variables and their values are stored. This data will be available to all pages on the site during that visit.

A session ends when the user closes the browser or after leaving the site, the server will terminate the session after a predetermined period of time, commonly 30 minutes duration.

State overview

Cookies and Sessions are used to **store** information. Cookies are only stored on the **client-side machine**, while sessions get stored **on the client as well as a server**.

- **Session**

A session creates a file in a temporary directory on the server where registered session variables and their values are stored. This data will be available to all pages on the site during that visit.

A session ends when the user closes the browser or after leaving the site, the server will terminate the session after a predetermined period of time, commonly 30 minutes duration.

- **Cookies**

Cookies are text files stored on the client computer and they are kept of use tracking purpose. Server script sends a set of cookies to the browser. For example name, age, or identification number etc. The browser stores this information on a local machine for future use.

When next time browser sends any request to web server then it sends those cookies information to the server and server uses that information to identify the user.

Passing values

Passing values:

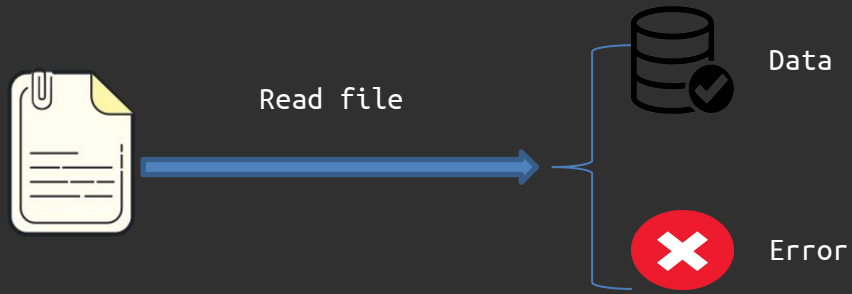
Uploading a file, reading the file, creating a file on the server

Within this tutorial, we are going to look at how you can effectively read and write to files within your filesystem using the go programming language.

The method we are going to use to read and write to these files will be file format-agnostic. What this means is that you'll be able to use the techniques we'll be covering in order to read and write, .txt, .csv, .xls and so on, the only thing that differs for these files is the structure of the data that you write to each of these file types.

- **Reading files :**

In order to read from files on your local filesystem, you'll have to use the `io/ioutil` module.
You'll first have to pull of the contents of a file into memory by calling `ioutil.ReadFile("/path/to/my/file.ext")` which will take in the **path to the file** you wish to read in as it's only parameter. This will return either the data of the file, or an err which can be handled as you normally handle errors in go.



Reading file:

Create two files main.go and localfile.txt

```
package main
// import the 2 modules we need
import (
    "fmt"
    "io/ioutil"
)
func main() {
    // read in the contents of the localfile.data
    data, err := ioutil.ReadFile("localfile.txt")
    // if our program was unable to read the file
    // print out the reason why it can't
    if err != nil {
        fmt.Println(err)
    }
    // if it was successful in reading the file then
    // print out the contents as a string
    fmt.Print(string(data))
}
```

Reading file:

localfile.txt

```
Hello world !!
Second line
third line
.....
```

```
$ go run main.go

Hello world !!
Second line
third line
.....
```

Encrypt

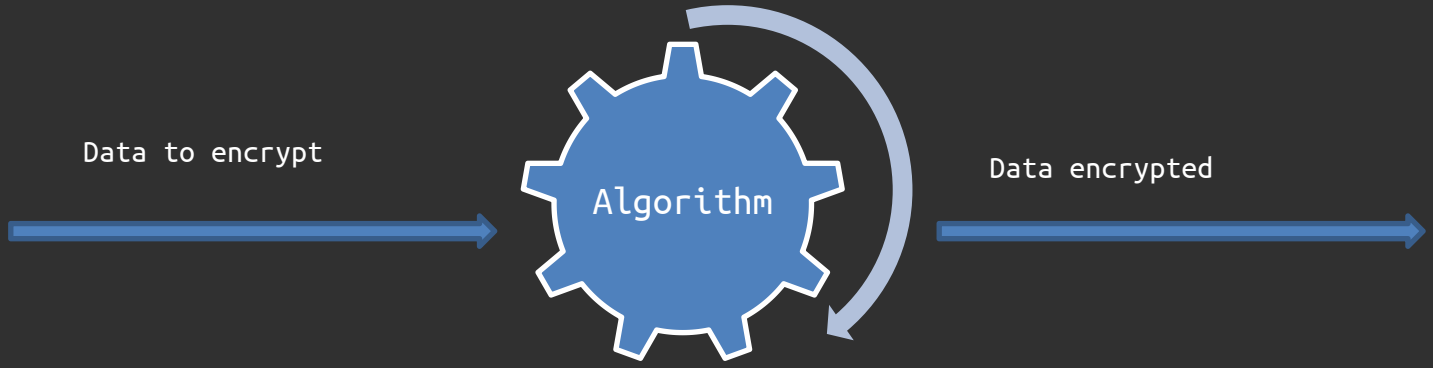
- **Definition:**

Encryption is the process through which data is encoded so that it remains hidden from or inaccessible to unauthorized users. It helps protect private information, sensitive data, and can enhance the security of communication between client apps and servers. In essence, when your data is encrypted, even if an unauthorized person or entity gains access to it, they will not be able to read it.

- **Algorithms:**

An algorithm is basically a **procedure** or a **formula** for solving a data snooping problem. An encryption algorithm is a set of mathematical procedure for performing encryption on data. Through the use of such an algorithm, information is made in the **cipher text and requires** the use of a key to transforming the data into its original form. This brings us to the concept of cryptography that has long been used in information security in communication systems.

Encrypt : Algorithm



Encrypt

- **Cryptography**

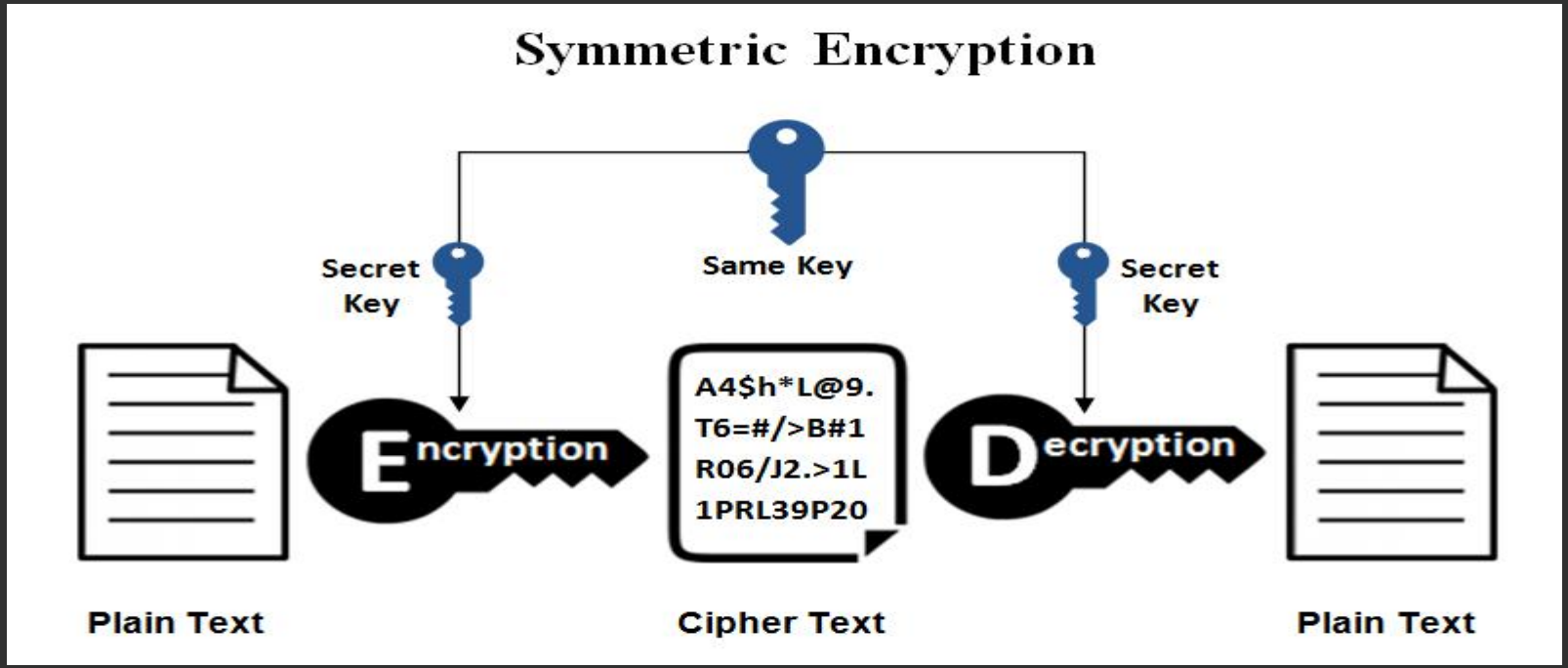
Cryptography is a method of **using advanced mathematical principles in storing and transmitting data** in a particular form so that only those whom it is intended can read and process it. **Encryption** is a key concept in cryptography - It is a process whereby a message is encoded in a format that cannot be read or understood by an eavesdropper. The technique is old and was first used by **Caesar** to encrypt his messages using Caesar cipher. A plain text from a user can be encrypted to a ciphertext, then send through a communication channel and no eavesdropper can interfere with the plain text. When it reaches the receiver end, the ciphertext is decrypted to the original plain text.

Encrypt

- **Cryptography Terms:**
  - **Encryption**: It is the process of locking up information using cryptography. Information that has been locked this way is encrypted.
  - **Decryption**: The process of unlocking the encrypted information using cryptographic techniques.
  - **Key**: A secret like a **password** used to **encrypt and decrypt** information. There are a few **different** types of keys used in cryptography.
  - **Steganography**: It is actually the science of hiding information from people who would snoop on you. The difference between steganography and encryption is that the would-be snoopers may not be able to tell there's any hidden information in the first place.

Encrypt : Symmetrical Encryption

- Symmetrical Encryption:



Encrypt : Symmetrical Encryption

- **Symmetrical Encryption:**

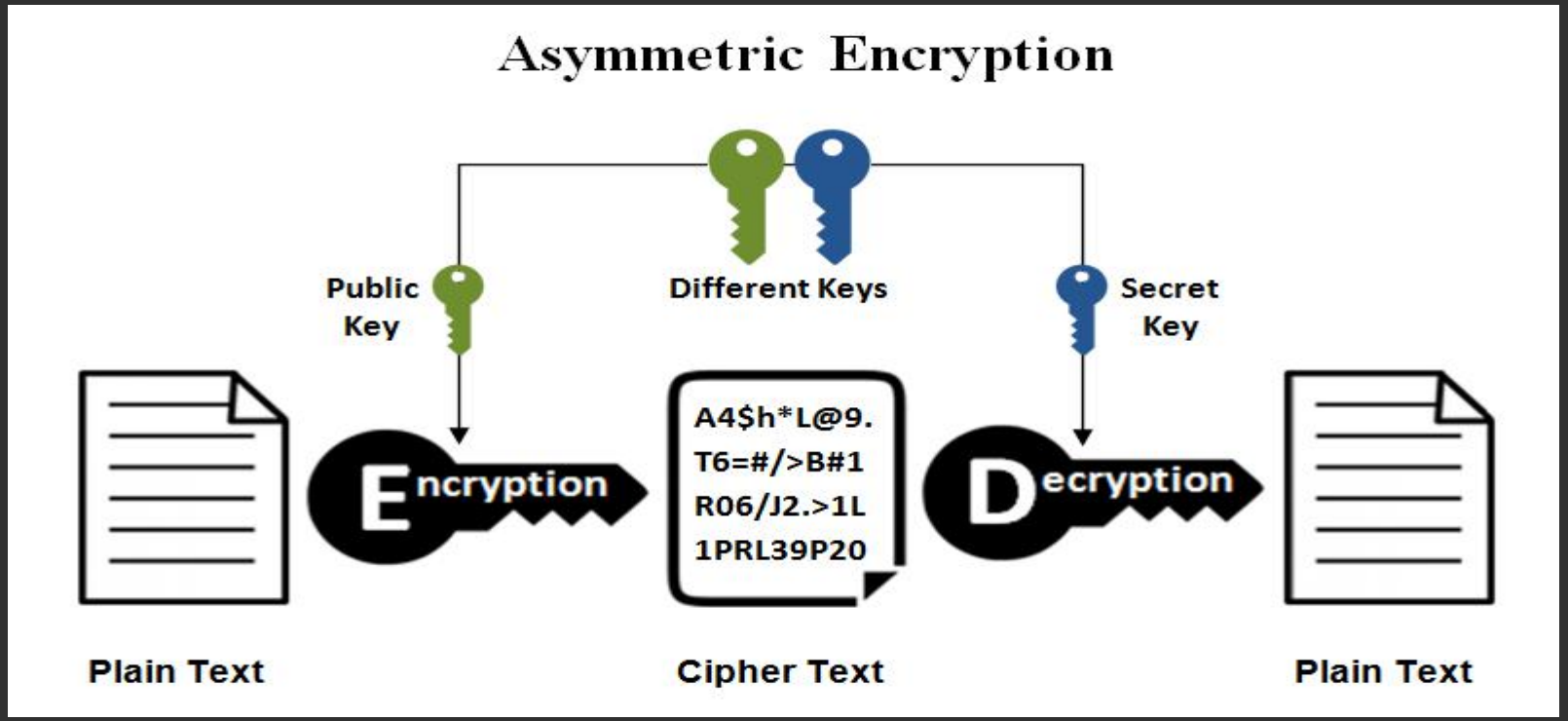
This is the simplest kind of encryption that involves **only one secret key to cipher and decipher information**. Symmetrical encryption is an old and best-known technique. **It uses a secret key that can either be a number, a word or a string of random letters**. It is a blended with the plain text of a message to change the content in a particular way. The sender and the recipient should know the secret key that is used to encrypt and decrypt all the messages.

Blowfish, AES, RC4, DES, RC5, and RC6 are examples of symmetric encryption. The most widely used symmetric algorithm is **AES-128, AES-192, and AES-256.**

The main disadvantage of the symmetric key encryption is that all parties involved have to exchange the key used to encrypt the data before they can decrypt it.

Encrypt : Asymmetrical Encryption

- Asymmetrical Encryption:



Encrypt : Asymmetrical Encryption

- **Asymmetrical Encryption:**

Asymmetrical encryption is also known as public key cryptography, which is a relatively new method, compared to symmetric encryption. Asymmetric encryption **uses two keys to encrypt a plain text**. **Secret keys are exchanged over the Internet or a large networ**k. It ensures that malicious persons do not misuse the keys. It is important to note that anyone with a secret key can decrypt the message and this is why asymmetrical encryption uses two related keys to boosting security. **A public key** is made freely available to anyone who might want to send you a message. The second private key is kept a secret so that you can only know.

A message that is **encrypted** using a **public key** can only be decrypted using a **private key**, while also, a message **encrypted** using a **private key** can be decrypted using a **public key**.

Security of the public key is not required because it is publicly available and can be passed over the internet. Asymmetric key has a far better power in ensuring the security of information transmitted during communication.

Asymmetric encryption is mostly used in day-to-day communication channels, especially over the Internet. Popular asymmetric key encryption algorithm includes **EIGamal, RSA, DSA, Elliptic curve techniques, PKCS**.

Encrypt : GO Implementation

Go Implementation:

The Go language supports symmetric encryption algorithms in its crypto package. Do not use anything except AES in GCM mode if you don't know what you're doing!
crypto/aes package: **AES (Advanced Encryption Standard)**, also known as Rijndael encryption method, is used by the U.S. federal government as a block encryption standard.
In the following example we demonstrate how to encrypt data using AES in GCM mode:

- **Define text and key :**
  - text := []byte("My name is Assakra Radhouen")
  - key := []byte("the-key-has-to-be-32-bytes-long!")
- **Use encrypt function to encrypt text:**
  - ciphertext, err := encrypt(text, key)**// we need to impliment these function**
- **Decrypt text using decrypt function:**
  - plaintext, err := decrypt(ciphertext, key)**// we need to impliment these function**
- **Use algorithm to impliment encrypt and decrypt:**
  - AES for symetric encryption : import “crypto/aes”
  - RSA for asymetric encryption : import "crypto/rsa"

Encrypt : GO AES

```
func encrypt(plaintext []byte, key []byte) ([]byte, error) {
            c, err := aes.NewCipher(key)
/*NewCipher creates and returns a new cipher.Block. The key argument should be the AES key,
either 16, 24, or 32 bytes to select AES-128, AES-192, or AES-256.*/
            if err != nil {
                        return nil, err
            }

            gcm, err := cipher.NewGCM(c)
            if err != nil {
                        return nil, err
            }

            nonce := make([]byte, gcm.NonceSize())
            if _, err = io.ReadFull(rand.Reader, nonce); err != nil {
                        return nil, err
            }

            return gcm.Seal(nonce, nonce, plaintext, nil), nil
}
```

Encrypt : GO AES

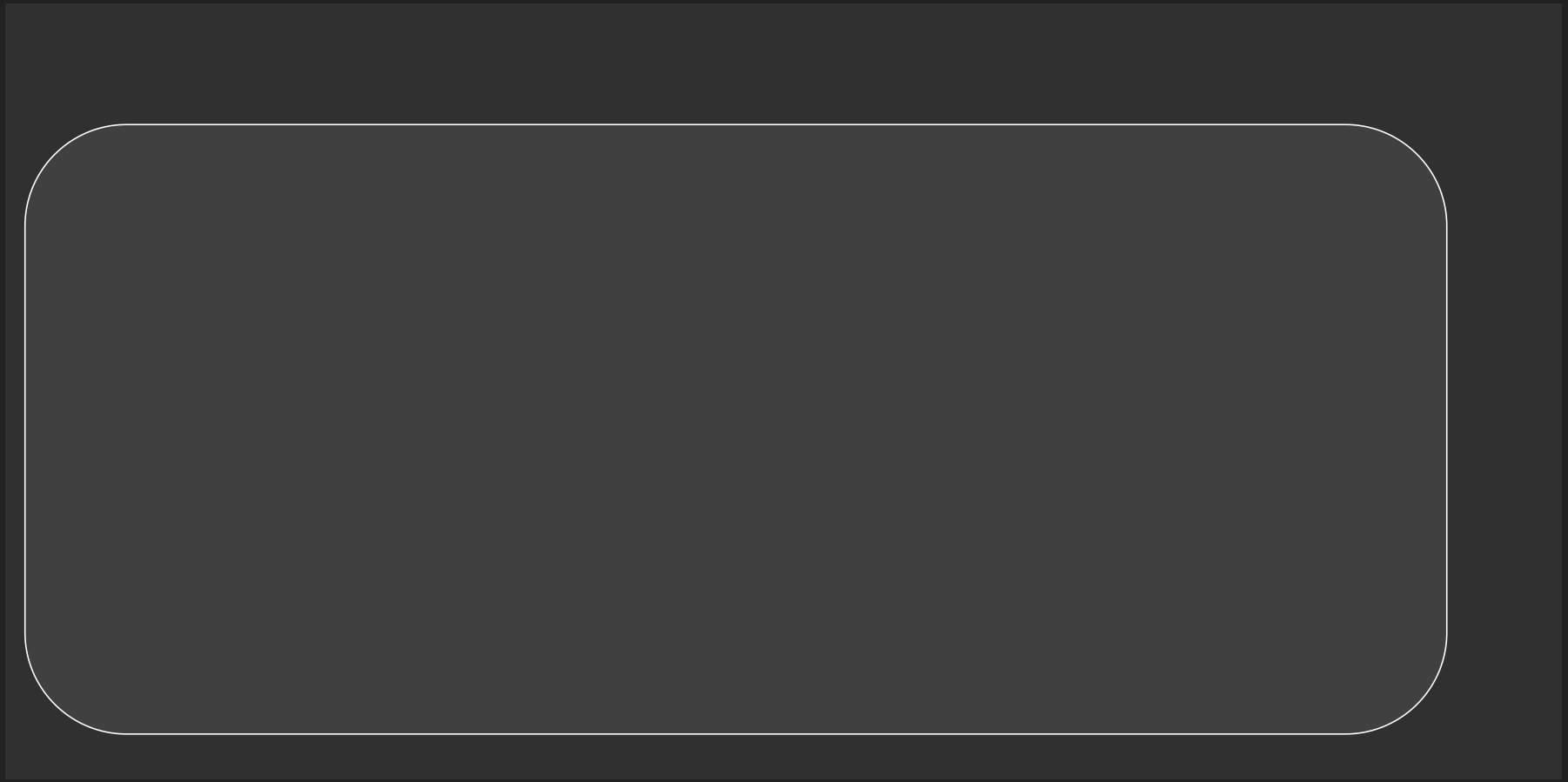
```
func decrypt(ciphertext []byte, key []byte) ([]byte, error) {
            c, err := aes.NewCipher(key)
            if err != nil {
                        return nil, err
            }

            gcm, err := cipher.NewGCM(c)
            if err != nil {
                        return nil, err
            }

            nonceSize := gcm.NonceSize()
            if len(ciphertext) < nonceSize {
                        return nil, errors.New("ciphertext too short")
            }

            nonce, ciphertext := ciphertext[:nonceSize], ciphertext[nonceSize:]
            return gcm.Open(nil, nonce, ciphertext, nil)
}
```

Encrypt : GO RSA



Redirects

Redirect replies to the request with a redirect to url, which may be a path relative to the request path.

The provided code should be in the **3xx** range and is usually StatusMovedPermanently, StatusFound or StatusSeeOther.

If the Content-Type header has not been set, Redirect sets it to "text/html; charset=utf-8" and writes a small HTML body. Setting the Content-Type header to any value, including nil, disables that behavior.

```
func Redirect(w ResponseWriter, r *Request, url string, code int)
```

Cookies

Set A cookies:

Go uses the SetCookie function in the net/http package to set cookies:

http.SetCookie(w ResponseWriter, cookie *Cookie)

w is the response of the request and cookie is a struct. Let's see what it looks like:

```
type Cookie struct {
        Name       string
        Value      string
        Path       string  // optional
        Domain     string  // optional
        Expires    time.Time  // optional
        RawExpires string  // for reading cookies only

          // MaxAge=0 means no 'Max-Age' attribute specified.
          // MaxAge<0 means delete cookie now, equivalently 'Max-Age: 0'
          // MaxAge>0 means Max-Age attribute present and given in seconds
        MaxAge   int
        Secure   bool
        HttpOnly bool
        Raw      string
        Unparsed []string // Raw text of unparsed attribute-value pairs
    }
```

Cookies

Set A cookies:
Here is an example of setting a cookie:

```
expiration := time.Now().Add(365 * 24 * time.Hour)
cookie := http.Cookie{Name: "username", Value: "astaxie", Expires: expiration}
http.SetCookie(w, &cookie)
```

Cookies

Fetch cookies in Go:
The above example shows how to set a cookie. Now let's see how to get a cookie that has been set:

```
cookie, _ := r.Cookie("username")
fmt.Fprint(w, cookie)
```

Here is another way to get a cookie:

```
for _, cookie := range r.Cookies() {
       fmt.Fprint(w, cookie.Name)
   }
```

Sessions Overview

- **Creating sessions**

The basic principle behind sessions is that a server maintains information for every single client, and clients rely on unique session id's to access this information. When users visit the web application, the server will create a new session with the following three steps, as needed:

  - Create a unique session id
  - Open up a data storage space: normally we save sessions in memory, but you will lose all session data if the system is accidentally interrupted. This can be a very serious issue if web application deals with sensitive data, like in electronic commerce for instance. In order to solve this problem, you can instead save your session data in a database or file system. This makes data persistence more reliable and easy to share with other applications, although the tradeoff is that more server-side IO is needed to read and write these sessions.
  - Send the unique session id to the client.

Universally unique identifier – UUID

Universally unique identifier – UUID:

The uuid package generates and inspects UUIDs based on RFC 4122 and DCE 1.1: Authentication and Security Services.

This package is based on the github.com/pborman/uuid package (previously named code.google.com/p/go-uuid). It differs from these earlier packages in that a UUID is a 16 byte array rather than a byte slice. One loss due to this change is the ability to represent an invalid UUID (vs a NIL UUID).

- **Install:**

go get github.com/google/uuid

- **Supported versions:**
    - Version 1, based on timestamp and MAC address (RFC 4122)
    - Version 2, based on timestamp, MAC address and POSIX UID/GID (DCE 1.1)
    - Version 3, based on MD5 hashing (RFC 4122)
    - Version 4, based on random numbers (RFC 4122)
    - Version 5, based on SHA-1 hashing (RFC 4122)

Universally unique identifier – UUID

```
package main

import (
        "fmt"
        "github.com/satori/go.uuid"
)
func main() {
        // Creating UUID Version 4
        // panic on error
        u1 := uuid.Must(uuid.NewV4())
        fmt.Printf("UUIDv4: %s\n", u1)

        // or error handling
        u2, err := uuid.NewV4()
        if err != nil {
                fmt.Printf("Something went wrong: %s", err)
                return
        }
        fmt.Printf("UUIDv4: %s\n", u2)

        // Parsing UUID from string input
        u2, err = uuid.FromString("6ba7b810-9dad-11d1-80b4-00c04fd430c8")
        if err != nil {
                fmt.Printf("Something went wrong: %s", err)
                return
        }
        fmt.Printf("Successfully parsed: %s", u2)
}
```

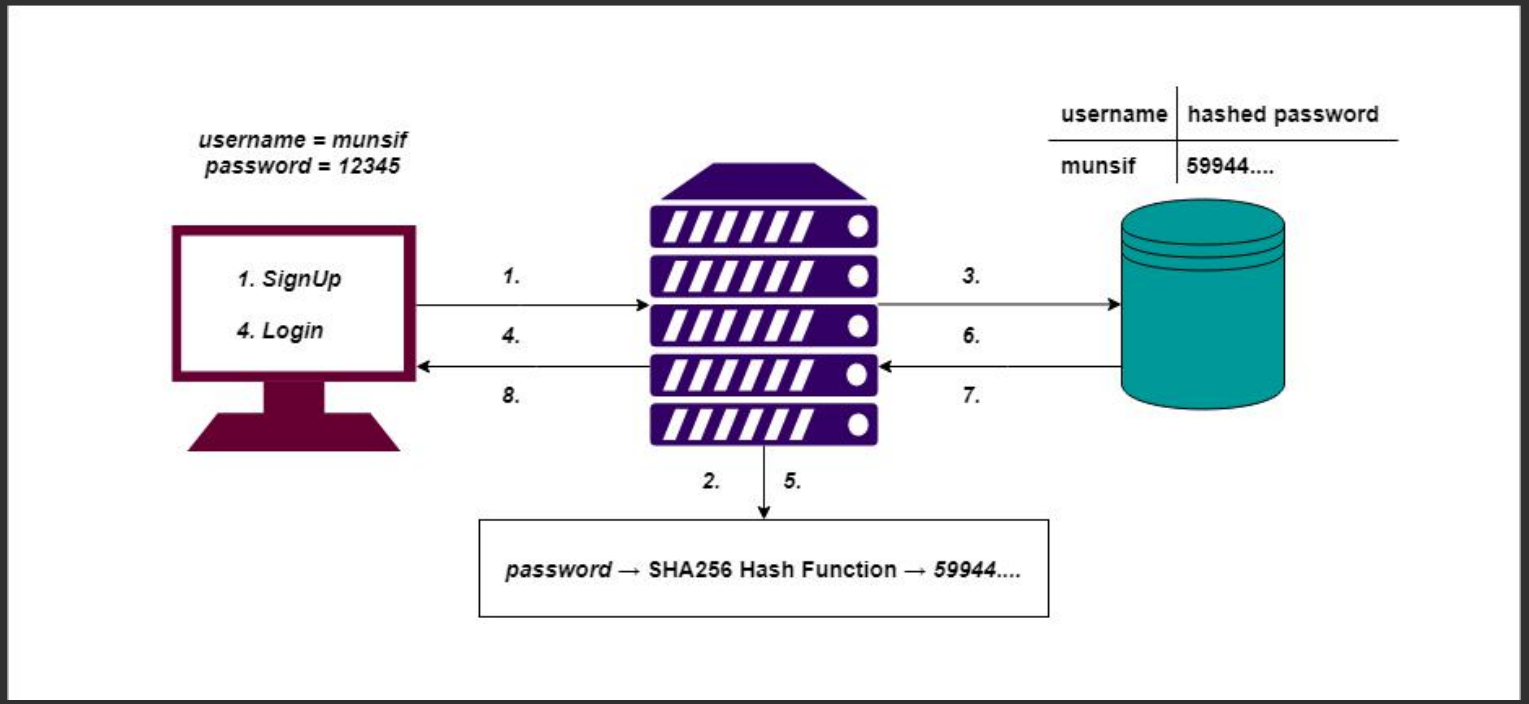
Universally unique identifier – UUID

- **Do you really need a UUID ?**
  - There are several reasons using a UUID as a PK would be great compared to auto-incrementing integers:

    At scale, when you have multiple databases containing a segment (shard) of your data, for example a set of customers, using a UUID means that one ID is unique across all databases, not just the one you're in now. This makes moving data across databases safe. Or in my case where all of our database shards are merged onto our Hadoop cluster as one, no key conflicts.
  - You can know your PK before insertion, which avoids a round trip DB hit, and simplifies transactional logic in which you need to know the PK before inserting child records using that key as its foreign key (FK)
  - UUIDs do not reveal information about your data, so would be safer to use in a URL, for example. If I am customer 12345678, it's easy to guess that there are customers 12345677 and 1234569, and this makes for an attack vector. (But see below for a better alternative).

Sign-up / Encrypt password

Sign-up / Encrypt password:



Sign-up / Encrypt password

Password Hashing (bcrypt)
This example will show how to hash passwords using bcrypt. For this we have to go get the golang bcrypt library like so:

```
$ go get golang.org/x/crypto/bcrypt
```

From now on, every application we write will be able to make use of this library.

Sign-up / Encrypt password

```
// passwords.go
package main

import (
    "fmt"
    "golang.org/x/crypto/bcrypt"
)

func HashPassword(password string) (string, error) {
    bytes, err := bcrypt.GenerateFromPassword([]byte(password), 14)
    return string(bytes), err
}

func CheckPasswordHash(password, hash string) bool {
    err := bcrypt.CompareHashAndPassword([]byte(hash), []byte(password))
    return err == nil
}

func main() {
    password := "secret"
    hash, _ := HashPassword(password) // ignore error for the sake of simplicity

    fmt.Println("Password:", password)
    fmt.Println("Hash:    ", hash)

    match := CheckPasswordHash(password, hash)
    fmt.Println("Match:   ", match)
}
```

Login / Logout

Login / Logout:

- To login we need two fields (username and password) so in deep login have these steps :
    - Get username and password from the Form or from another service
    - To Verify these data with a database query or hard coded(for test only)
    - Set a session .
    - Redirect to a specific page,
- Logout:
    - Clear Session
    - Redirect to Home Page

Login / Logout

```
// Handle Login
func loginHandler(response http.ResponseWriter, request *http.Request) {
        name := request.FormValue("name")
        pass := request.FormValue("password")
        redirectTarget := "/"
        if name != "" && pass != "" {
                // .. check credentials ..
                setSession(name, response)
                redirectTarget = "/internal"
        }
        http.Redirect(response, request, redirectTarget, 302)
}
// Set session
func setSession(userName string, response http.ResponseWriter) {
        value := map[string]string{
                "name": userName,
        }
        if encoded, err := cookieHandler.Encode("session", value); err == nil {
                cookie := &http.Cookie{
                        Name:  "session",
                        Value: encoded,
                        Path:  "/",
                }
                http.SetCookie(response, cookie)
        }
}
```

Login / Logout

```
// Clear Session
func clearSession(response http.ResponseWriter) {
              cookie := &http.Cookie{
                            Name:   "session",
                            Value:  "",
                            Path:   "/",
                            MaxAge: -1,
              }
              http.SetCookie(response, cookie)
}


// logout handler

func logoutHandler(response http.ResponseWriter, request *http.Request) {
              clearSession(response)
              http.Redirect(response, request, "/", 302)
}
```

Permissions

Roles:
Roles is an authorization library for Golang, it also integrates nicely with QOR Admin.
Permission Modes:
Permission modes are really the roles in Roles. Roles has 5 default permission modes:

- **roles.Read**
- **roles.Update**
- **roles.Create**
- **roles.Delete**
- **roles.CRUD** // CRUD means Read, Update, Create, Delete

You can use those permission modes, or create your own by defining permissions.

Expire session

Expire session:

Chapter 11

GO PROGRAMMING-Database

Plan

- MYSQL Databases
- Connect
- Insert
- Update
- Diplay
- Delete

MySql DataBase

MYSQL:

The **LAMP** stack has been very popular on the internet in recent years, and the M in LAMP stand for MySQL. **MySQL** is famous because it's open source and easy to use. As such, it has become the de-facto database in the back-ends of many websites.

MySQL drivers:

There are a couple of drivers that support MySQL in Go. Some of them implement the database/sql interface, and others use their own interface standards.

- https://github.com/go-sql-driver/mysql supports database/sql, written in pure Go.
- https://github.com/ziutek/mymysql supports database/sql and user defined interfaces, written in pure Go.

I'll use the first driver in the following examples and I also recommend that you use it for the following reasons:

- It's a new database driver and supports more features.
- It fully supports database/sql interface standards.
- Supports keep-alive, long connections with thread-safety.

Connect to a MySql DataBase

To connect to a data base we need 5 attributes : the database **HOST**, **PORT**, **USER**, **PASSWORD** and **NAME**.

```
type MySqlConnection struct {
            dataBaseName string // example test
            dataBasePort string // example 3306
            dataBaseUrl string  // example 127.0.0.1
            dataBaseUserName string // example root
            dataBasePassword string // example password1

}
Connection := MySqlConnection{
                        "gomycode",
                        "https://www.127.168.10.10",
                        "3306",
                        "gomycode",
                        "123654789/*-+",
            }
sql.Open("mysql", Connection.dataBaseUserName+":"+Connection.dataBasePassword+"
@tcp("+Connection.dataBaseUrl+":"+Connection.dataBasePort+")/"+Connection.dataBaseName")
Example: sql.Open("mysql", "root:password1@tcp(127.0.0.1:3306)/test")
```

Insert into an array Mysql

```
// insert
        stmt, err := db.Prepare("INSERT userinfo SET username=?,departname=?,created=?")
        checkErr(err)

        res, err := stmt.Exec("radhouen", "EL khadhra city", "2020-01-09")
        checkErr(err)

        id, err := res.LastInsertId()
        checkErr(err)

        fmt.Println(id)
```

Update an item

```
fun deleteElement() {
stmt, err = db.Prepare("update userinfo set username=? where uid=?")
        checkErr(err)

        res, err = stmt.Exec("astaxieupdate", id)
        checkErr(err)

        affect, err := res.RowsAffected()
        checkErr(err)

        fmt.Println(affect)
}
```

Display

```
// query
        rows, err := db.Query("SELECT * FROM userinfo")
        checkErr(err)

        for rows.Next() {
            var uid int
            var username string
            var department string
            var created string
            err = rows.Scan(&uid, &username, &department, &created)
            checkErr(err)
            fmt.Println(uid)
            fmt.Println(username)
            fmt.Println(department)
            fmt.Println(created)
        }
```

Delete

```
// delete
        stmt, err = db.Prepare("delete from userinfo where uid=?")
        checkErr(err)

        res, err = stmt.Exec(id)
        checkErr(err)

        affect, err = res.RowsAffected()
        checkErr(err)

        fmt.Println(affect)

        db.Close()
```

Chapter 12

GO PROGRAMMING-Database

Plan

- NoSQL Databases
- Go & MongoDB
- Aggregation
- JSON

NoSQL Databases

NoSQL encompasses a wide variety of different database technologies that were developed in response to the demands presented in building modern applications:

Developers are working with applications that create massive volumes of new, rapidly changing data types — structured, semi-structured, unstructured and polymorphic data.

Long gone is the twelve-to-eighteen month waterfall development cycle. Now small teams work in agile sprints, iterating quickly and pushing code every week or two, some even multiple times every day.

Applications that once served a finite audience are now delivered as services that must be always-on, accessible from many different devices and scaled globally to millions of users.

Organizations are now turning to scale-out architectures using open software technologies, commodity servers and cloud computing instead of large monolithic servers and storage infrastructure.

Relational databases were not designed to cope with the scale and agility challenges that face modern applications, nor were they built to take advantage of the commodity storage and processing power available today.

NoSQL Databases

NoSQL Database Types

- **Document databases** pair each key with a complex data structure known as a document. Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.

- **Graph stores** are used to store information about networks of data, such as social connections. Graph stores include Neo4J and Giraph.

- **Key-value stores** are the simplest NoSQL databases. Every single item in the database is stored as an attribute name (or 'key'), together with its value. Examples of key-value stores are Riak and Berkeley DB. Some key-value stores, such as Redis, allow each value to have a type, such as 'integer', which adds functionality.

- **Wide-column stores** such as Cassandra and HBase are optimized for queries over large datasets, and store columns of data together, instead of rows

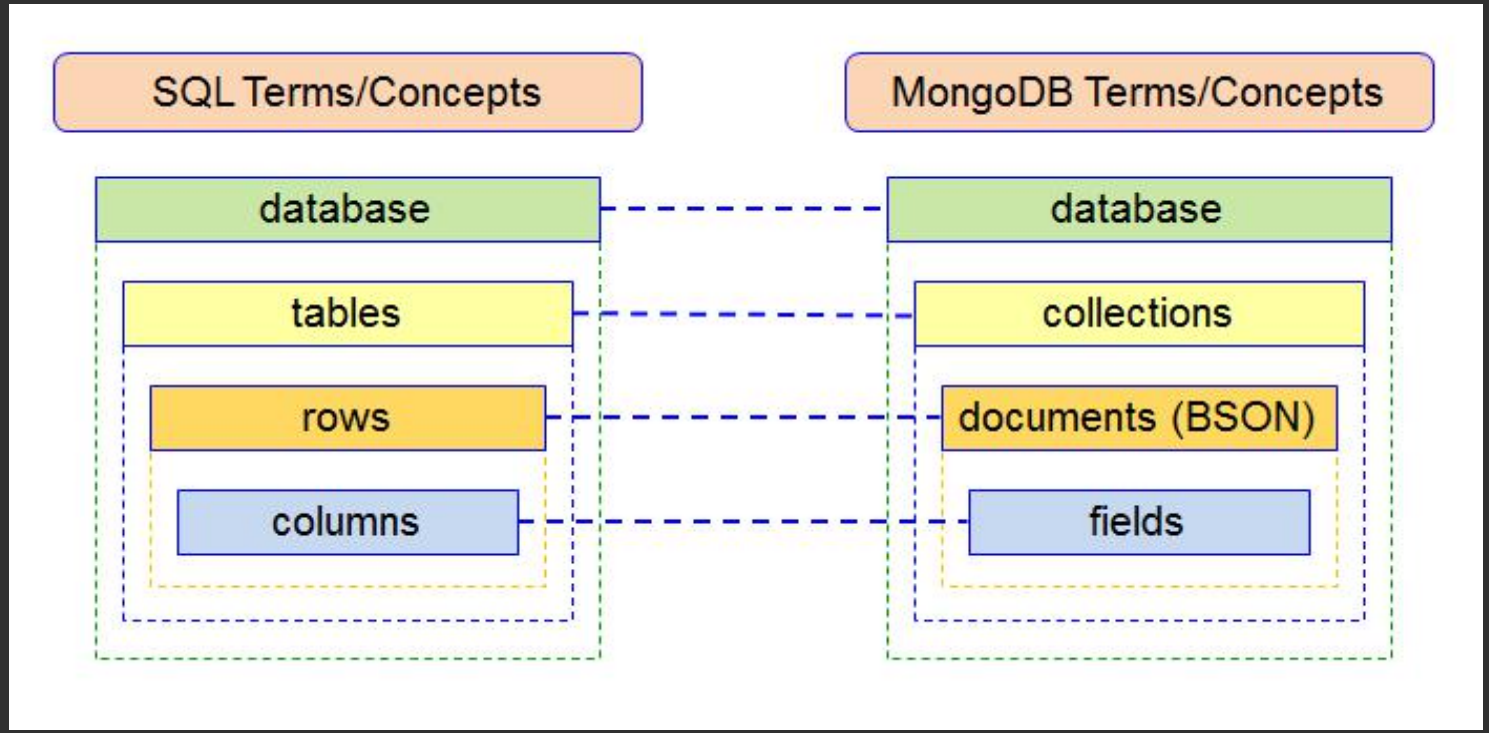
NoSQL Databases

The Benefits of NoSQL

When compared to relational databases, NoSQL databases are more scalable and provide superior performance, and their data model addresses several issues that the relational model is not designed to address:

- Large volumes of rapidly changing structured, semi-structured, and unstructured data
- Agile sprints, quick schema iteration, and frequent code pushes
- Object-oriented programming that is easy to use and flexible
- Geographically distributed scale-out architecture instead of expensive, monolithic architecture

NoSQL vs SQL


SQL Terms/Concepts
MongoDB Terms/Concepts
database
tables
rows
columns
database
collections
documents (BSON)
fields

Go & MongoDB : Prepare environement

While looking into working with mongodb using golang, I found it quite frustrating getting it up and running and decided to make a quick post about it.

What are we doing?

Examples using the golang driver for mongodb to connect, read, update and delete documents from mongodb.

Environment:

Provision a mongodb server in docker:

```
$ docker network create container-net
$ docker run -itd --name mongodb --network container-net -p 27017:27017 ruanbekker/mongodb
```

Drop into a golang environment using docker:

```
$ docker run -it golang:alpine sh
```

Get the dependencies:

```
$ apk add --no-cache git
```

Go & MongoDB : Prepare environement

Change to your project path:

```
$ mkdir $GOPATH/src/myapp
$ cd $GOPATH/src/myapp
```

Download the golang mongodb driver:

```
$ go get go.mongodb.org/mongo-driver
```

Connecting to MongoDB in Golang

First example will be to connect to your mongodb instance:

```
package main
import (
    "context"
    "fmt"
    "log"
    "go.mongodb.org/mongo-driver/mongo"
    "go.mongodb.org/mongo-driver/bson"
    "go.mongodb.org/mongo-driver/mongo/options"
)
type Person struct {
    Name string
    Age  int
    City string
}
func main() {
    clientOptions := options.Client().ApplyURI("mongodb://mongodb:27017")
    client, err := mongo.Connect(context.TODO(), clientOptions)
    if err != nil {
        log.Fatal(err)
    }
    err = client.Ping(context.TODO(), nil)

    if err != nil {
        log.Fatal(err)
    }
   fmt.Println("Connected to MongoDB!")
}
```

Go & MongoDB : Prepare environement

Running our app:

```
$ go run main.go
Connected to MongoDB!
```

Writing to MongoDB with Golang

Let’s insert a single document to MongoDB:

```
func main() {
    ..
    collection := client.Database("mydb").Collection("persons")

    ruan := Person{"Ruan", 34, "Cape Town"}

    insertResult, err := collection.InsertOne(context.TODO(), ruan)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("Inserted a Single Document: ", insertResult.InsertedID)
}
```

Running it will produce:

```
$ go run main.go
Connected to MongoDB!
Inserted a single document:  ObjectID("5cb717dcf597b4411252341f")
```

Writing to MongoDB with Golang

Writing more than one document:

```
func main() {
    ..
    collection := client.Database("mydb").Collection("persons")

    ruan := Person{"Ruan", 34, "Cape Town"}
    james := Person{"James", 32, "Nairobi"}
    frankie := Person{"Frankie", 31, "Nairobi"}

    trainers := []interface{}{james, frankie}

    insertManyResult, err := collection.InsertMany(context.TODO(), trainers)
    if err != nil {
      log.Fatal(err)
  }
    fmt.Println("Inserted multiple documents: ", insertManyResult.InsertedIDs)
}
```

Running it will produce:

```
$ go run main.go
Inserted Multiple Documents:  [ObjectID("5cb717dcf597b44112523420")
ObjectID("5cb717dcf597b44112523421")]
```

Updating Documents in MongoDB using Golang

Updating Frankie’s age:

```
func main() {
    ..
    filter := bson.D
    update := bson.D{
        {"$inc", bson.D{
            {"age", 1},
        }},
    }

    updateResult, err := collection.UpdateOne(context.TODO(), filter, update)
    if err != nil {
      log.Fatal(err)
  }
    fmt.Printf("Matched %v documents and updated %v documents.\n", updateResult.MatchedCount,
updateResult.ModifiedCount)
}
```

Running that will update Frankie’s age:

```
$ go run main.go
Matched 1 documents and updated 1 documents.
```

Reading Data from MongoDB

Reading the data:

```
funct main() {
    ..
    filter := bson.D
    var result Trainer

  err = collection.FindOne(context.TODO(), filter).Decode(&result)
  if err != nil {
      log.Fatal(err)
  }

  fmt.Printf("Found a single document: %+v\n", result)

  findOptions := options.Find()
    findOptions.SetLimit(2)

}
```

Running that will update Frankie’s age:

```
$ go run main.go
Found a single document: {Name:Frankie Age:32 City:Nairobi}
```

Reading Data from MongoDB

Finding multiple documents and returning the cursor:

```
func main() {
    ..
    var results []*Trainer
  cur, err := collection.Find(context.TODO(), bson.D, findOptions)
  if err != nil {
      log.Fatal(err)
  }

  for cur.Next(context.TODO()) {
      var elem Trainer
      err := cur.Decode(&elem)
      if err != nil {
          log.Fatal(err)
      }

      results = append(results, &elem)
  }

  if err := cur.Err(); err != nil {
      log.Fatal(err)
  }

  cur.Close(context.TODO())
    fmt.Printf("Found multiple documents (array of pointers): %+v\n", results)
}
```

Deleting Data from MongoDB:

Deleting our data and closing the connection:

```
func main(){
    ..
    deleteResult, err := collection.DeleteMany(context.TODO(), bson.D)
  if err != nil {
      log.Fatal(err)
  }

  fmt.Printf("Deleted %v documents in the trainers collection\n", deleteResult.DeletedCount)

  err = client.Disconnect(context.TODO())

  if err != nil {
      log.Fatal(err)
  } else {
      fmt.Println("Connection to MongoDB closed.")
  }
}
```

Running the example:

```
$ go run main.go
Deleted 3 documents in the trainers collection
Connection to MongoDB closed.
```

Aggregation

Aggregation operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. MongoDB provides three ways to perform aggregation: the aggregation pipeline, the map-reduce function, and single purpose aggregation methods.

JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

JSON is built on two structures:

A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.
These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

In JSON, they take on these forms:

An object is an unordered set of name/value pairs. An object begins with {left brace and ends with }right brace. Each name is followed by :colon and the name/value pairs are separated by ,comma.

JSON

JSON Package:

Package json implements encoding and decoding of JSON as defined in RFC 7159. The mapping between JSON and Go values is described in the documentation for the Marshal and Unmarshal functions.