code.sd

November 2023 – April 2024

Migration to Go language

Introduction

Go is compiled, statically typed, concurrent, imperative, with OOP support, high-level programming language. Go has been developed and supported by Google, and version 1 has been released in 2012

Author

Motaz Abdel Azeem, I have graduated at Sudan university of science and technology in 1999, and I was developed using Delphi, then Java and now Go language. I have written many books for programming languages: Object Pascal, Java (an Arabic book) and now Go. I'm founder of Code for Computer software (code.sd)

Main objectives for this book

There are many reasons why I wrote this book, first I need to study again Go language in more academic way, and cover most of language details that I didn't used in production. Also to receive feedback from readers and reviewers for our methodology of using Go. Also this book is written to help spread - the good language - Go and add more learning books as our contribution to community and as pay-back after our benefit using Go. Also this book could be a guide and introduction for our new developers whom join Code.sd, to understand why we have choose Go over other languages, and our standard way of using it.

Book license:

Table of Contents

A Brief about us and used technologies:

Code (a Software Firm) is specialized in development in networking, telecom software, APIs, VOIP, and web applications. Previously Java and PHP were been used in development for all systems and modules, but after 4 to 6 years, problems has arises from both languages in the long run, these problems were:

a). **Java problems:**

1. Java consumes a lot of resources: CPU and Memory, specially when deploying in Tomcat server. Programs written in Java requires much memory in servers – compared with other technologies- for large amount of traffic. For development PCs also it consumes a lot of memory to run NetBeans (which is written in Java), also requires powerful CPU. It runs very slow in low resources developers PCs.

2. We encounter servers CPU load for Java web applications and web services, also crashes happens in Tomcat which results in stopping all deployed Java web applications and services.

3. JSP feature of Java encourages developers for bad practice: an anti-pattern of writing logic within presentation and sometimes data access in the same file (.jsp file).

4. Java JVM version differences: There are many servers with different releases of Ubuntu which has different versions of Java, starting from Java 7 to 11, also there are different versions of Tomcat and MySQL, in addition to developer different environment version, this makes incompatibility and errors when deploying with higher version and different version in targeted machines.

5. Oracle is licensing usage of Java SE, in spite of that OpenJDK requires no licensing, but it is not grantee that OpenJDK will be developed, improved, and comparable with new versions of Java.

6. Java run-time in deployment side requires updates on the server for security purposes. Old unpatched Java run-time could have vulnerabilities and expose server to intrusion

b). PHP problems

1. PHP is scripting language, so that it requires source to be deployed in server side, and in case of in-premises deployment, source code will be exposed in client customers and couldn't be protected against change and copyright

2. It requires PHP interpreter to be available on servers with specific version and with required packages

3. Upgrading developer PHP version requires upgrading servers version of PHP and all running applications on that server

4. Some developers are modifying code directly in clients servers, which results in different versions of files in servers than developers PCs and source control. This makes conflicts between main trunk version and deployed versions

5. PHP allows developers to write code and HTML in the same file, which breaks single responsibility principle and MVC. This makes code hard to read and to modify

6. Since PHP is not compiled, PHP source code with errors could be deployed or submitted to source control, also change could happen in deployment files that could generate syntax errors.

7. PHP has weak types definition and does not support unit testing and debugging, so that bugs and problems are hard to trace and to fix

8. Scripting languages are fragile in deployment environment, not like compiled and byte code languages that produces solid binaries or byte code that is more immune against change in deployed environments

9. PHP sites are subject to hackers intrusion, they will know that this site is built using PHP and they could find vulnerabilities. Hackers are always targeting PHP open source packages that are often deployed in public servers such as *phpMyAdmin*, and *wordpress*.

Go features and advantages

1. It is a compiled language, and it's executable binary is self-contained, which means that all packages and dependencies are linked into one single executable that is easy to deploy and run independently of platform version in target machines.

2. It supports cross-compilation: we can produce Linux 64 bit binary from windows OS and vice versa

3. Memory safety and garbage collection compared to similar compiled languages such as C and C++

4. Fast execution for produced binaries

5. Uses low resources: memory and CPU

6. Has simple and readable syntax, fast to learn

7. Module support: easy to import packages and it’s dependencies

8. Has built-in networking packages, this makes it suitable for networking, Internet, and communication applications.

9. Go-routines: an easy implementation of multi-tasking, light weight for CPU and memory compared to threads

10. Simplicity: Go language and tools are very simple, no frameworks are required. Standard language libraries are very rich, this reduces investment in study for the language and it’s basic libraries and packages, also provides new developers fast track to learn and start developing using Go.

11. No hacking target for deployment applications, because there is no platform, or libraries are required to deploy Go applications in servers, so that it is more immune against hacking compared to scripting and run-time based languages.

Go features according to Go site:

http://go.dev

- An open-source programming language supported by Google
- Easy to learn and great for teams
- Built-in concurrency and a robust standard library
- Large ecosystem of partners, communities, and tools

Drawback of Go language

As any programming language, each programming language has been designed to cover specific domain of programming, so that there is no one programming language could fit all domains. Go language is no exception, and as a newly introduced language it has it’s own problems and lacks in some aspects.

1. It has fewer developers compared to other programming languages that already has dominated development for decades. Our solution to this problem is to hire junior developers and let them learn Go, and we do benefit Go simplicity and ease of learning. Junior developers tend to accept new technologies more than seniors whom tend to resist and keep their old technologies that they already invested their time. In addition to seniors developers whom already face problems with other languages and has already convinced to migrate to Go.

2. Some customers requires to use specific languages that their internal developers use, in case of delivering software with source code, and they choose other languages that already famous, old, and has larger developers base, such as Java. Part of customers are asking for programming language for curiosity only. Most of customers requires delivery as black-box product
3. Go language is developed and controlled by single company : Google, so that license could be changed for future languages the as Oracle did for Java. Truly open source languages that has many organizations and companies sharing it's development is more immune against license change in future. In other side this also could be considered as strength point, to have one large successful company to produce such programming language and it's tools the same as commercial competitive products, and this company is one of largest users for Go language in their own systems and services.
4. Module Packages download must be downloaded through Google proxy, even if you host packages in your public server, if they close that proxy server from your country, you couldn't use Go with external packages

Our success case of using Go

We have started experimenting and re-writing small projects to Go language, then we have started a migration of critical and unstable Java modules. We get an immediate result by achieving stability with newly converted modules to Go, and one of examples is a web service for Asterisk that was unstable Java web service running in Tomcat which always crashes under heavy load. Go version has achieved stability and running for more than 5 years without any crash or stop. Here is the process in Linux server, check process date, it has started since 2018:

```
ps -eo lstart,cmd | grep goagent
Sun Jul  1 08:35:10 2018 ./goagent
```

This case becomes a reference point and solid result to adopt Go language in our critical modules.

An important thing that worth mentioning on this case, is that our unreliable Java web service and background services has been written after many years developing mostly in Java, compared to only two months of experimenting with Go language.

Successive migrations have been done after that module re-writing, and it always results in a better, stable and cleaner code. But note that we didn't translate Java code as-it-is in all cases, sometimes we have enhance internal structure for most of migrated code, to implement software engineering principles, but even when no enhancement in code logic and structure is done; we get benefits of performance, stability and low resources usage for all migrated projects and modules, also we could give junior developers a Java or a PHP module to translate it to Go.

Java modules that handles a moderate traffic and didn't require frequent modifications, are stable and working just fine, and there is no urgent need to migrate them, specially when were written using best practices. The main issue for such projects is that we need to have Java knowledge among team in order to maintain these modules. Our target is to minimize tools, technologies and

programming languages, so that if we manage to migrate all or most of projects and modules to one language and fewer technologies; we could achieve that target and support our running projects with a smaller team.

Go vs Java resources comparison

I have made simple console application timer in both Go and Java, which runs in an infinite loop to display current date and time in console to see how it consumes in resources:

1. **GoTimer source:**

```go
package main

import (
     "fmt"
      "time"
)

func main() {
   for {
        fmt.Println(time.Now().String())
        time.Sleep(time.Second)
   }
}
```

2. **JavaTimer source:**

```java
package javatimer;

import java.util.Date;

public class JavaTimer {
    public static void main(String[] args) throws InterruptedException {

        while (true){
            System.out.println(new Date().toString());
            Thread.sleep(1000);
        }
    }
}
```

Here are memory comparison when running both applications in Linux:

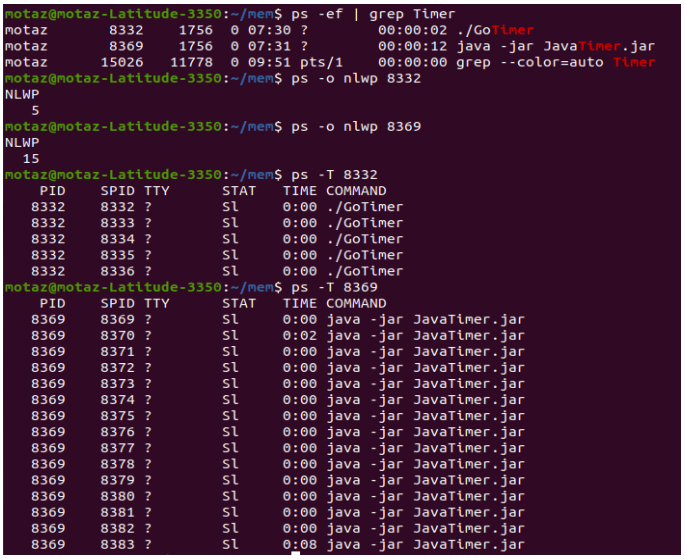
Note that *GoTimer* consume 1.9 Megabytes, while *JavaTimer* consumes 18.3 Megabytes:
Using *ps_mem* python program:

```
 Private  +   Shared  =  RAM used       Program

  1.9 MiB +   0.5 KiB =   1.9 MiB       GoTimer
--------------------------------
                1.9 MiB
================================

 Private  +   Shared  =  RAM used       Program

 11.6 MiB +   6.7 MiB =  18.3 MiB       java
--------------------------------
               18.3 MiB
================================
```

For consumed threads in OS:
using below command in Linux:

```
ps -o nlwp <pid>
```

GoTimer consumes 5 process threads while *JavaTimer* consumes 15 process threads:



To compare it to C, we have written below code in *ctimer.c* file:

```c
#include<stdio.h>
#include<time.h>

int main() {
    struct timespec waitt = { 1/*seconds*/, 0/*nanoseconds*/};
    while (1) {
      time_t t;
      time(&t);
      printf("Time: %s", ctime(&t));
      nanosleep(&waitt,NULL);
      fflush(stdout);
    }

    return 0;
}
```

Memory usage of C version of timer is only **109** Kilobytes

Private	+	Shared	=	RAM used	Program
96.0 KiB	+	13.5 KiB	=	**109.5** KiB	**a.out**

and it is only one process:

```
ps -o nlwp 44147
NLWP
   1
```

C is a winner here for resources consumption, but the code is less readable.

Also C is a low level programming language, and in our use case of Go and Java are higher level programming languages that can be used for enterprise software, so that C is not a candidate in our case.

Threads comparison:

Go has light weight threads called *go routines*, while Java has more heavy threads.
Here are previous Timer examples modified to have multiple threads, note that Go routine is more simple to write:

1. **GoTimer with threads: (20 lines)**

```go
package main

import (
    "fmt"
    "time"
)

func displayTime(id int) {
     for {
          fmt.Println(id, time.Now().String())
          time.Sleep(time.Second)
     }
}

func main() {
     fmt.Println("Display Time Go Routine")
     go displayTime(1)
     go displayTime(2)
     select {}
}
```

2. **JavaTimer with threads: (32 lines)**

```java
package javatimer;

import java.util.Date;

class JavaTimerThread extends Thread{
    public int ID;

    @Override
    public void run(){
        while (true){
            System.out.println(ID + ":" + new Date().toString());
            try {
                Thread.sleep(1000);
```

```
            } catch (InterruptedException ex) {
                System.err.println("Error: " + ex.toString());
            }
        }
    }
}

public class JavaTimer {
    public static void main(String[] args) throws InterruptedException {

        JavaTimerThread timer = new JavaTimerThread();
        timer.ID = 1;
        timer.start();

        JavaTimerThread timer2 = new JavaTimerThread();
        timer2.ID = 2;
        timer2.start();
    }
}
```

When we run command to check process internal threads we will notice that *GoTimer* remains with 5 threads while Java process threads has increased to 17:

```
motaz@motaz-Latitude-3350:~/go/src/GoTimer$ ps -Lefa | grep -e Timer -e NLWP
UID          PID    PPID     LWP  C NLWP STIME TTY          TIME CMD
motaz      22237   20041   22237  0   17 09:56 pts/3    00:00:00 java -jar dist/JavaTimer.jar
motaz      22237   20041   22238  0   17 09:56 pts/3    00:00:00 java -jar dist/JavaTimer.jar
motaz      22237   20041   22239  0   17 09:56 pts/3    00:00:00 java -jar dist/JavaTimer.jar
motaz      22237   20041   22240  0   17 09:56 pts/3    00:00:00 java -jar dist/JavaTimer.jar
motaz      22237   20041   22241  0   17 09:56 pts/3    00:00:00 java -jar dist/JavaTimer.jar
motaz      22237   20041   22242  0   17 09:56 pts/3    00:00:00 java -jar dist/JavaTimer.jar
motaz      22237   20041   22243  0   17 09:56 pts/3    00:00:00 java -jar dist/JavaTimer.jar
motaz      22237   20041   22244  0   17 09:56 pts/3    00:00:00 java -jar dist/JavaTimer.jar
motaz      22237   20041   22245  0   17 09:56 pts/3    00:00:00 java -jar dist/JavaTimer.jar
motaz      22237   20041   22246  0   17 09:56 pts/3    00:00:00 java -jar dist/JavaTimer.jar
motaz      22237   20041   22247  0   17 09:56 pts/3    00:00:00 java -jar dist/JavaTimer.jar
motaz      22237   20041   22248  0   17 09:56 pts/3    00:00:00 java -jar dist/JavaTimer.jar
motaz      22237   20041   22249  0   17 09:56 pts/3    00:00:00 java -jar dist/JavaTimer.jar
motaz      22237   20041   22250  0   17 09:56 pts/3    00:00:00 java -jar dist/JavaTimer.jar
motaz      22237   20041   22251  0   17 09:56 pts/3    00:00:00 java -jar dist/JavaTimer.jar
motaz      22237   20041   22252  0   17 09:56 pts/3    00:00:00 java -jar dist/JavaTimer.jar
motaz      22237   20041   22253  0   17 09:56 pts/3    00:00:00 java -jar dist/JavaTimer.jar
motaz      22256   19045   22256  0    5 09:56 pts/1    00:00:00 ./GoTimer
motaz      22256   19045   22257  0    5 09:56 pts/1    00:00:00 ./GoTimer
motaz      22256   19045   22258  0    5 09:56 pts/1    00:00:00 ./GoTimer
motaz      22256   19045   22259  0    5 09:56 pts/1    00:00:00 ./GoTimer
motaz      22256   19045   22260  0    5 09:56 pts/1    00:00:00 ./GoTimer
```

Executable size

Another important difference, but this time is on Java side: Java produces small size byte code executable (in kilobytes), while Go compiler produces self-contained large size binary executable, which contains run-time libraries and used packages, all of them will be inside executable that starts with about 2 megabytes in size.
Java is heavily rely on Java virtual machine or Java runtime (JRE), so that byte-code executable are very small and relies on shared JRE between all java applications on that machine Here are our previous comparison projects sizes:

Java
```
2.5K JavaTimer.jar
```

Go:
```
1.9M GoTimer
```
That was the minimum compiler output size, so that we need to check real projects for multi thousands lines:

C:
```
17K a.out
```
C binary has less statically-linked run-time size compared to Go, also C has no garbage collector.

Here is Go 6k lines that uses few packages:
```
12M CodeAccountingWS
```

Compared to Java 7k lines that uses few Java class libraries:
```
5.5M SMSPanel.war
```

This is not big issue for disk space on which we deploy executable, but it might be an issue when transferring binary files via network/Internet, it could take a time to in case of low speed network connections, so that it is better to compress binaries while copying.

Installing Go

For Linux, Go could be installed from repositories, but it is better to be downloaded directly from official Go site to get the latest versions:

https://go.dev/

After downloading suitable OS version we can follow the “installation instructions” depending on platform we are using.
This package contains Go compiler, debugger, run-time libraries, Go packages, and command line tools that used by developer and used by IDE that we will install later

This is an example of deploying Go package in Linux:

```
rm -rf /usr/local/go && tar -C /usr/local -xzf go1.20.linux-amd64.tar.gz
```

or as multiple commands and using sudo:

```
sudo rm -rf /usr/local/go
sudo tar -C /usr/local -xzf go1.20.linux-amd64.tar.gz
```

After that we need to check Go compiler has been deployed, and in case of installing new version we need to make sure new version is working now:

```
go version
```

Result could be something like this depending on version and target OS:

```
go version go1.20 linux/amd64
```

First Go application

After installing Go compiler, we could use any simple text editor to write first Go sample command line project:
First we should create folder by the same project name, for example “first”, then create *main.go* file inside “first” folder and write below code.
Then we can use *gofmt* tool to reformat text alignment to Go standards for more readability:

```
gofmt -w *.go
```

```
package main

import (
        "fmt"
)

func main() {

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

Then run first application:

```
go run main.go
```

This produces executable binary in a temporary directory and run that binary
Output:

```
Hello Go
```

To build go binary output, we can use *go build* command:

```
go build main.go
```

or just go build in case of multiple project files:

```
go build
```

After running *go build* command we will find executable in the same directory of source project code:

```
-rw-rw-r-- 1 motaz motaz  105 Feb 11 08:31 main.go
-rwxrwxr-x 1 motaz motaz 1.9M Feb 11 08:34 main
```

Previous sample files represents Linux environment, in Windows we will find *main.exe* file
We can run executable and distribute it to other machines

To run it in Linux:

```
./main
```

And in Windows

```
main.exe
```

or

```
main
```

As in native compiled languages, we will get a binary executable depending on OS and Platform, and that executable runs only on the same targeted Platform, to check executable in Linux we could use *file* command tool to check the file compiled target:

```
file main
```

The result will be something like this depending on OS:

```
main: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, Go
BuildID=T1xYAYiZPSUBtfa28x_N/Kko4il7vBW6OyEirX06W/9vAOZhoNUt7A7DP7tVTG/
Y_meKiaUFP4PphxQag_Q, with debug_info, not stripped
```

The advantage of Java here -compared to native compiled language- is that Java produced byte code will run on every machine that has compatible version of Java virtual machine, so that only one byte code is required to be produced, and could be run in different platforms in which JVM exists there.

Cross compilation

As Go available on all major platforms, we could compile that source in every platform the same like C and C++, but the advantage here in Go is that there is an easy cross-compilation, which means you can produce compiled binary for any supported platform from your development platform, for example if you are using Linux, you can produce Linux, Windows 64/32, Mac, ARM binaries from your Linux PC.

Here is an example of how to produce Windows 64 bit binary from Linux or any other platform:

```
GOOS=windows GOARCH=amd64 go build main.go
```

and now we will find *main.exe* file and if we checked it with *file main.exe* we will get:

```
file main.exe
main.exe: PE32+ executable (console) x86-64 (stripped to external PDB),
for MS Windows
```

Here another example to build ARM executable that could run on RaspberyPI:

```
env GOOS=linux GOARCH=arm GOARM=5 go build main.go
```

produced executable name will be *main*, so that your previous Linux binary will be overwritten, and you need to use *file* command to differentiate between different binaries of Linux, and Unix. Mac binary also has no extension the same like Linux:

```
file main
main: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked,
Go BuildID=Fk6aTEOgy7bWmOgkdLVd/KKZ5f1Qe8FS4O-NQHfYY/ce9Vr7NIvgeZAQWdT0HI/
FLMnwvzqn4tiG38gW_wB, with debug_info, not stripped
```

IDE

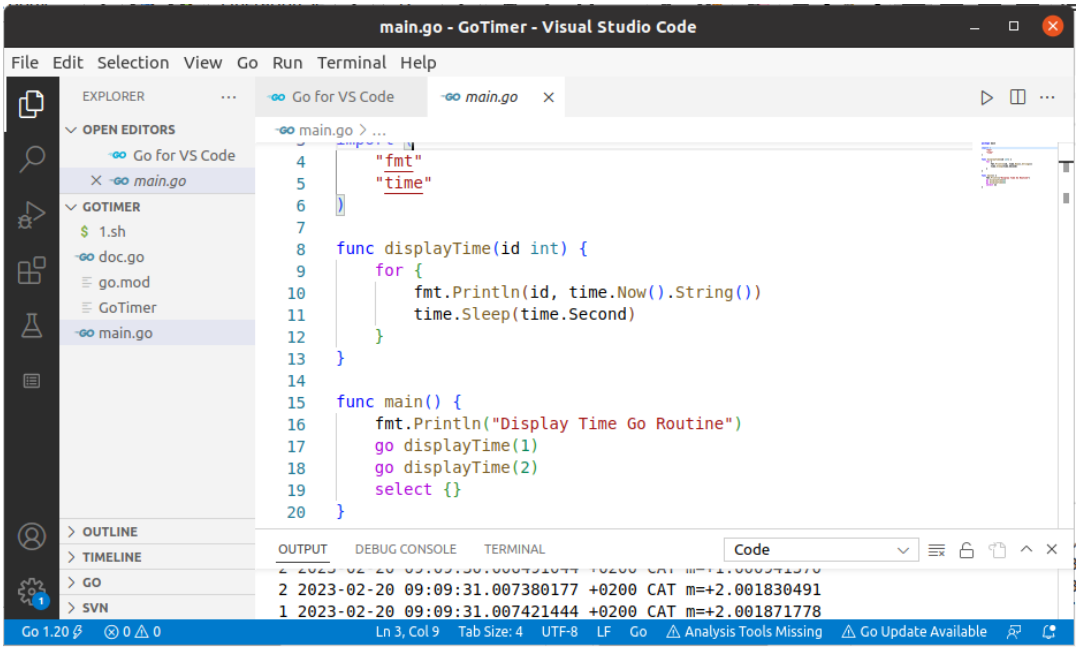
Go compiler and it's tools are independent of any development IDEs, it could be used with many IDEs, such as Visual Studio Code, and LiteIDE. These IDEs are using Go compiler and tools to provide syntax check, format, compilation, run, testing, building and debugging.

1. **Visual Studio Code**

It is one of famous development IDE, and it uses extensions in order to provide support for variety of programming languages.

After installing Visual Code, we can install Go extensions, or start writing Go project, then IDE will suggest extensions to be installed.

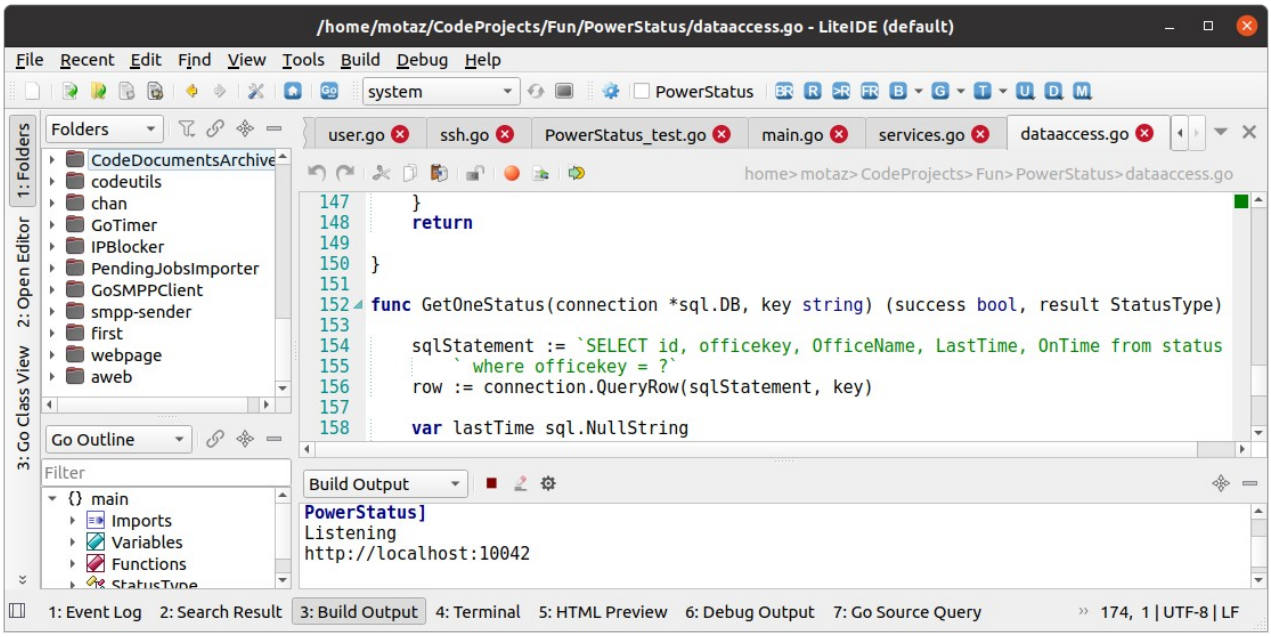
First extension will be: ***Go for Visual Studio Code***, then ***Go runner*** extension to manage running and stopping Go programs.



2. **LiteIDE**

LiteIDE is a specific Go IDE, so that it will be ready for Go projects, no extensions or special configuration is required, it will be ready out of the box for Go programs. It is ideal for starting with Go. It provides compilation, building, modules initialization, testing, and cross-compilation.

Also we can easily work in multiple projects in LiteIDE, but it lacks integration with source control, you need to use external tools or command line to import and commit projects in source control.



In our illustration we will use LiteIDE because of previously mentioned reasons, but we will mention command line method for running and building which is used by Visual Code.

First Go sample with LiteIDE

To create new Go project you will find multiple choices, we will choose: ***Go new command project*** option.

This creates project in default go/src directory, if you want to create project in another directory you could select ***Go new command project (Anywhere)*** option

We named it ***first-sample***, then we will find below ready written source template:

```
// first-sample project main.go
package main

import (
	"fmt"
)

func main() {
	fmt.Println("Hello World!")
}
```

If we try to run it using build and run button (BR) or Control+R, we will get below message:

```
go: go.mod file not found in current directory or any parent directory; see 'go
help modules'
Error: process exited with code 1.
```

LiteIDE enforces to use Go module in spite of that this simple project does not require external package, but we could initialize module for future use of external packages.

Module could be initialized using (M) button menu, then select ***Go module Init***.

For Visual code we don't have to initialize module unless we need an external package, and in this case we could initialize module using below command line in project directory:

```
go mod init
```

go.mod file will be created containing below text:

```
module first-sample
go 1.20
```

Then we can run again the project and see the result, also an executable will be produced after building project, either ***first-sample*** filename in Linux, Unix, and Mac environment or ***first-sample.exe*** in Windows environment

We can add another line to *main.go* source file to display current date and time by typing:

```
fmt.Println(time.
```

Then LiteIDE will suggests to add time package automatically, and code will be:

```
// first-sample project main.go
package main

import (
     "fmt"
     "time"
)

func main() {
     fmt.Println("Hello World!")
     fmt.Println(time.Now().String())
}
```

this requires build and run again to get below result:

```
Hello World!
2023-02-28 08:41:40.236956669 +0200 CAT m=+0.000123066
Success: process exited with code 0.
```

In below command, we can do temporary build for this program and run it:

```
go run .
```

This command will not provide executable in current directory, and if we need to produce executable without running the application we can use below command:

```
go build
```

then we can run produced binary using:

```
./first-sample
```

in Linux, Unix, and Mac

or *first.sample* in Windows

```
first-sample
```

We can right-click project in LiteIDE then select ***Open terminal here***

We can assign current time to a variable then use that variable for different purposes such as to display time or to store date and time in a log file, using one of below variable declarations:

1. Using explicit declaration using *var* keyword:

```
        var atime time.Time
        atime = time.Now()
        fmt.Println(atime.String())
```

2. Second method is to declare variable by direct assign of value using (:=) operator:

```
        atime := time.Now()
        fmt.Println(atime.String())
```

In this example we will use explicit declaration of *atime* type, then assign value in the same line:

```
        var atime time.Time = time.Now()
        fmt.Println(atime.String())
```

This could also be for any other types, such as integers and strings:

```
        var area int = 1200
        diameter := 1.2
        unit := "Kilometers"
        fmt.Printf("Area is %d %s\n", area, unit)
        fmt.Printf("Diameter is %f %s\n", diameter, unit)
```

Note that explicit declaration of variable type makes it more readable to define to readers the type of variable instead of searching function returned value type or guessing assigned value type. IDE also could help showing function returned value by moving mouse pointer to function name as in

Visual Code, or pressing Control key and pointing by mouse as in LiteIDE, but sometimes we need to review code using browser when accessing source control, so that explicit delegation will help in this case.

Note that there is no semicolon (;) at the end of lines or statements, which eliminates *; expected* compiler errors, such as in other programming languages, but it could be used for multi-statement lines such as:

```
        diameter := 1.2; unit := "Kilometers"
```

If we click save on LiteIDE or used *gofmt, they* will convert it to two lines and removes ;

Arrays and Slices:

Arrays in Go has fixed length of specific type, here is example of array of string:

```
    var list [3]string
    list[0] = "First"
    list[1] = "Second"
    list[2] = "Third"
    fmt.Println(list)
```

Output:

```
[First Second Third]
```

It could be iterated using ***for .. range*** statement:

```
    for i, item := range list {
         fmt.Println(i, item)
    }
```

Output:

```
0 First
1 Second
2 Third
```

First variable of *for .. range* (i) will hold index of current iterated item of array while second variable (item) will hold current iterated array element. We can omit index if we need only item by using underscore:

```
for _, item := range list {
```

Slices are dynamic, it requires initialization either with zero-length then append it later with items, or initialized with initial size and also it could be appended later:

```
    var list []string
```

```
	list = make([]string, 3)
	list[0] = "First"
	list[1] = "Second"
	list[2] = "Third"
	list = append(list, "Fourth")
	fmt.Println(list)
```

output:

```
[First Second Third Fourth]
```

Maps

Maps in Go is an implementation of hash table, which stores data in a *key-value* pair, as in below example:

```
func main() {
	personMap := make(map[string]string)
	personMap["name"] = "Motaz"
	personMap["address"] = "Khartoum,Sudan"
	personMap["title"] = "Developer"
	personMap["dob"] = "1975-11-16"
	fmt.Println("Name = ", personMap["name"])
	fmt.Println(personMap)
	delete(personMap, "dob")
	fmt.Println(personMap)
}
```

output:

```
Name = Motaz
map[address:Khartoum,Sudan dob:1975-11-16 name:Motaz title:Developer]
map[address:Khartoum,Sudan name:Motaz title:Developer]
```

Here we have initialized map of string that their keys are string also, and we could use other types such as *map[string]int*, that their values are *integer* and keys as *string*, or *map[int]string:* values are *string* and keys as *integer*.

We can iterate through map using *for .. range* statement as in below sample:

```
	for key, value := range personMap {
		fmt.Println(key, "=", value)
	}
```

output:

```
name = Motaz
address = Khartoum,Sudan
title = Developer
dob = 1975-11-16
```

Passing variables by reference

When calling a function we could call it using constant parameters such as:

```
        displaySum(2, 5)
```

or passing parameters using variable by copy:

```
        a:= 5
        b:= 6
        displaySum(a, b)
```

Third option is to pass parameter by reference, in which we need to let function to be able to change variable value, for example swap function:

In this case we have to define parameters as pointers:

```
func swapNumbers(a *int, b *int) {

   *a, *b = *b, *a
   return
}
```

When calling that function we have to pass address instead of values (&a, &b):

```
     swapNumbers(&a, &b)
```

Here is another example using string type: a function to change spaces to underscore in names:

```
func toUnderscore(name *string) {

     *name = strings.ReplaceAll(*name, " ", "_")
     return
}
```

*name = means the contents of pointer (name)

We can call it by passing variable address (&x):

```
      name := "go language"
      toUnderscore(&name)
      fmt.Println(name)
```

We can make a general swap function using *interface{}* type or it's new alias type (*any)* :

```
func swap(a, b *any) {
        *a, *b = *b, *a
        return
}
```

We can call it as:

```
        var a, b any
        a = "a"
        b = "b"
        swap(&a, &b)
```

```
        fmt.Println(a, b)
```

Generics

Generics has been added to Go in version 1.18, instead of using dynamic empty *interface{}* or it's *any* alias type that is checking type at run-time and could generate panic error which could close application, a new method has been introduced, in this method compiler could check types at compile time and prevents errors. Here is an example of add function that could be used to add integers, float or concatenate strings:

```
func add[T int | float64 | string](a T, b T) (result T) {
        return a + b
}
```

This generic *add* function could be called as in below example:

```
        fmt.Println(add(2, 3))
        fmt.Println(add(1.2, 2.1))
        fmt.Println(add("Hello", " World"))
```

Using this *generic* method is more safer and faster than *interface{}*, the compiler will generate multiple instances of add function for each type. This represents static and strong typing discipline in Go language

Writing to text file

Here is an example of how to write string in text file, if file does not exist; *writeToFile* function will create it, and if file already exists it will append to it:

```
package main

import (
     "fmt"
     "os"
     "time"
)

func main() {
     fmt.Println("Writing to text file")
     writeToFile("text.txt", "Time in server is: "+time.Now().String())
}

func writeToFile(filename string, text string) (err error) {

     var file *os.File
     file, err = os.OpenFile(filename,
                 os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
     if err == nil {
          defer file.Close()
          file.WriteString(text + "\n")
     }
```

```
	return
}
```

We have used point to type *os.File* in “os” package, so that we have added it to import section.

In *writeToFile* function we have called ***os.OpenFile*** with create and append options, if opening/creating file has succeeded and *err* is nil, it will write text to file, but note that we have used *defer* keyword statement in front of closing file (*f.Close()*). Defer do two things: 1: delays execution of given statement to end of current block or function, and 2: ensures execution of that statement in all cases, such as error or panic, except if explicit *Exit* command is called.

File mode parameter that contains *0644* is a Unix permission type giving full read-write permission to owner, and read-only for groups and others.

Reading text file

In this example we have written two ways to read from text file: first method (*readLinesFromFile*) is to read line by line, and second method (*readEntireFile*) that reads all contents at once :

```
package main

import (
	"bufio"
	"fmt"
	"io/ioutil"
	"os"
)

func main() {
	filename := "main.go"
	fmt.Println("Reading from text file: ", filename)
	//readLinesFromFile(filename)
	readEntireFile(filename)
}

func readLinesFromFile(filename string) (err error) {

	var file *os.File
	file, err = os.OpenFile(filename, os.O_RDONLY, 0644)
	if err == nil {
		defer file.Close()
		scanner := bufio.NewScanner(file)
		for scanner.Scan() {
			line := scanner.Text()
			fmt.Println(line)
		}
	} else {
		fmt.Println("Error: ", err.Error())
	}
	return
}

func readEntireFile(filename string) (err error) {

	var file *os.File
	file, err = os.OpenFile(filename, os.O_RDONLY, 0644)
	if err == nil {
```

```
            defer file.Close()
            var contents []byte
            contents, err = ioutil.ReadAll(file)
            if err != nil {
                    fmt.Println("Error reding file : ", err.Error())
            }
            fmt.Print(string(contents))
    } else {
            fmt.Println("Error: ", err.Error())
    }
    return
}
```

I think the code is clear and needs no more clarifications, it reads all file contents in an array of byte, then we could typecast if to string to show it in console.

We can read filename from command line arguments by modifying main function to:

```
func main() {

    if len(os.Args) != 2 {
          fmt.Println("Usage:")
          fmt.Println("readtextfile <filename>")

    } else {
          filename := os.Args[1]
          fmt.Println("Reading from text file: ", filename)
          //readLinesFromFile(filename)
          readEntireFile(filename)
    }
}
```

Note that first argument (os.Args[0]) is program executable name itself with it's path

We could call it from command line in Linux and Max as:

```
./readtextfile main.go
```

And in Windows:

```
readtextfile main.go
```

Go routines

Go routines is a way of implementing parallel running for code and using multi-core CPU. Go routine is lighter than Java thread, here is sample using of Go routines:

```
package main

import (
    "fmt"
    "time"
)
```

```
func main() {

     go displayTime("1")
     go displayTime("2")
     go displayTime("3")
     for {
     }
}

func displayTime(id string) {

    for i := 0; i < 3; i++ {
        fmt.Printf("Routine : %s time in server is: %s\n", id,
            time.Now().String()[:19])
        time.Sleep(time.Second)
    }
}
```

Output:

```
Routine : 1 time in server is: 2023-04-11 08:27:34
Routine : 2 time in server is: 2023-04-11 08:27:34
Routine : 3 time in server is: 2023-04-11 08:27:34
Routine : 1 time in server is: 2023-04-11 08:27:35
Routine : 2 time in server is: 2023-04-11 08:27:35
Routine : 3 time in server is: 2023-04-11 08:27:35
Routine : 3 time in server is: 2023-04-11 08:27:36
Routine : 1 time in server is: 2023-04-11 08:27:36
Routine : 2 time in server is: 2023-04-11 08:27:36
```

We have only added *go* keyword to run *displayTime* function in background as that simple.

Three routines will run simultaneously

Infinite *for* loop is important to keep main program wait that execution, otherwise application will exit before executing any routine. We could also add timer instead of infinite loop, because using infinite loop will not exist unless it has been closed using Ctrl-C or ending it from tasks or processes manager

```
     go displayTime("1")
     go displayTime("2")
     go displayTime("3")
     time.Sleep(time.Second * 3)
```

Wait Group

If we have Go routines or multiple Go routines that we want to run in parallel and waiting their results before proceed to next statements, we can use ***WaitGroup*** in *sync* package.

We have modified previous go routine sample that has three times for loop to use *WaitGroup* instead of infinite *for loop* at end of program:

```

```

```go
package main

import (
   "fmt"
   "sync"
   "time"
)

func main() {
     var wg sync.WaitGroup
     wg.Add(3)
     go displayTime(&wg, "1")
     go displayTime(&wg, "2")
     go displayTime(&wg, "3")
     wg.Wait()
     fmt.Println("All routines has finished")
}

func displayTime(wg *sync.WaitGroup, id string) {

   for i := 0; i < 3; i++ {
         fmt.Printf("Routine : %s time in server is: %s\n", id,
                  time.Now().String()[:19])
         time.Sleep(time.Second)
   }
   wg.Done()
}
```

In this sample program we have declared *wg* variable as *sync.WaitGroup* type, and have initialized it for three routines using *Add* method (*wg.Add(3)*) and we will wait until all these three routines to finish. We have passed *wg* variable by reference (*&wg*) to *displayTime* function to call *Done()* method of wait group.

At main function after calling go routines we have called *wg.**Wait()*** to wait all routines to finish before proceed with next statements of program, and we could get result back for another process, such as fetching data from web in parallel then do process for that data

Output:

```
Routine : 3 time in server is: 2023-04-11 09:13:20
Routine : 1 time in server is: 2023-04-11 09:13:20
Routine : 2 time in server is: 2023-04-11 09:13:20
Routine : 2 time in server is: 2023-04-11 09:13:21
Routine : 1 time in server is: 2023-04-11 09:13:21
Routine : 3 time in server is: 2023-04-11 09:13:21
Routine : 2 time in server is: 2023-04-11 09:13:22
Routine : 3 time in server is: 2023-04-11 09:13:22
Routine : 1 time in server is: 2023-04-11 09:13:22
All routines has finished
```

Note that it is better to call *Done()* at the top of function with *defer* keyword to ensure calling it at any cases, and not let *Wait()* wait forever in case of crash inside function before reaching *Done()* method:

```
func displayTime(wg *sync.WaitGroup, id string) {

   defer wg.Done()
   for i := 0; i < 3; i++ {
       fmt.Printf("Routine : %s time in server is: %s\n", id,
          time.Now().String()[:19])
       time.Sleep(time.Second)
   }
}
```

Mutex

One of important thing in Go routines and threads is shared resources access, for example if we look back to our *write to text file* sample and tried to write on that file using multiple go routines, a race condition will occur, which means trying to access resource exclusively, in this case to write on file, such race condition could corrupt file or crashes the program. To prevent race condition, we have to synchronize writing to that file, by enabling only one thread/go routine to write to that file and lock the resource while writing, then release that lock after finish writing, while enforce other routines to wait first routine until finish.

Mutex (Mutual Exclusion) in Go enables lock and unlock for any resource, here write to file sample program rewritten using go routines and mutex :

```
package main

import (
     "fmt"
     "os"
     "sync"
     "time"
)

var mutex *sync.Mutex
var counter int = 0

func main() {

     mutex = &sync.Mutex{}
     fmt.Println("Writing to text file")
     for i := 0; i < 10; i++ {
             go writeToFile("text.txt",
     fmt.Sprintf("routine # %d: Time in server is: %s",
                     i+1, time.Now().String()))
     }
     time.Sleep(time.Second * 5)
     mutex.Lock()
     defer mutex.Unlock()
      fmt.Println("Written counter: ", counter)
}

func writeToFile(filename string, text string) (err error) {
        mutex.Lock()
        defer mutex.Unlock()
        counter++
        var file *os.File
        file, err = os.OpenFile(filename, os.O_APPEND|os.O_CREATE|
           os.O_WRONLY, 0644)
```

```
        if err == nil {
                defer file.Close()
                file.WriteString(text + "\n")
        }
        return
}
```

Now we have two shared resources: external text file, and counter global variable, which they should be accessed in synchronized method for go routines.

We have defined *mutex* as pointer of type *sync.Mutex* globally in program and initialized it at main function

We used Lock() method of mutex before access shared resource, and release lock using Unlock() method after writing/reading from that resource:

```
        mutex.Lock()
        fmt.Println("Written counter: ", counter)
        mutex.Unlock()
```

We can test our programs against race condition using below command line:

```
go run -race main.go
```

or

```
go build --race main.go
```

then run it as main executable normally

In our case we will get normal execution and result:

```
Writing to text file
Written counter:  10
```

If we commented out *mutex.Lock()* and *mutext.Unlock()* in *writeToFile* method

```
        //mutex.Lock()
        //defer mutex.Unlock()
```

and build the application then run *go run -race main.go* again, we will get race condition:

```
==================
WARNING: DATA RACE
Read at 0x0000005f8780 by main goroutine:
  main.main()
      /home/motaz/……/samples/textfiles/main.go:24 +0x266

Previous write at 0x0000005f8780 by goroutine 15:
  main.writeToFile()
      /home/motaz/……/samples/textfiles/main.go:31 +0x84
  main.main.func1()
      /home/motaz/……/samples/textfiles/main.go:19 +0x53

Goroutine 15 (finished) created at:
  main.main()
      /home/motaz/……/samples/textfiles/main.go:19 +0xe4
==================
```

```
Written counter:  10
Found 3 data race(s)
exit status 66
```

Race condition happens on below lines:

```
        counter++
```

and

```
        fmt.Println("Written counter: ", counter)
```

Note that race condition isn't detected in writing to text file, only on global variable (counter) writing and reading while others are writing. This might mean that writing to a file in Go is thread-safe, but it is not guaranteed in all cases and all platforms, we are here to demonstrate the need and mechanism of using Mutex

HTTP Web Sample

Go supports HTTP web applications (backend) easily in standard networking packages. HTTP application could be either web application or web service, here is our sample application for web app:

We have created a new project as command line project with name *first-web*, then we have initialized module in LiteIDE.

When using Visual Code or other editor we will not need to initialize module, because we are using only standard packages in this sample:

```
// first-web project main.go
package main

import (
     "fmt"
     "net/http"
     "time"
)

func main() {
     http.HandleFunc("/", index)
     fmt.Println("Listening on \nhttp://localhost:9090")
     err := http.ListenAndServe(":9090", nil)
     if err != nil {
          fmt.Println("Error while listening to port 9090: " + err.Error())
     }
}

func index(w http.ResponseWriter, r *http.Request) {

     w.Header().Set("content-type", "text/html")
     fmt.Fprintf(w, "<h2>Go Web Page</h2>")
     fmt.Fprintf(w, "<font color=blue>Main index page</font>")
     fmt.Fprintf(w, "<br/>Today is %s", time.Now().String())
}
```

Main important function here is *http.**ListenAndServe**(":9090", nil)*, which will work as an embedded HTTP web server, listening on port (9090 in this example), and this lets program holds and keep working, and makes it a multi-threaded application server listening and serving all requests on that port.

If port is used by other process, or no permission to use (ports under 1024 requires admin permission), *ListenAndServe* will exit and returns error in *err* object and our command application will closes in this case.

Second important method is the handler: *index()* method that we have written to receive HTTP client requests (such as browser request through URL, curl or any HTTP client)

We have linked HTTP path request of (/) to index() handler using *HandleFunc* method:

```
	http.HandleFunc("/", index)
```

We could add another path handler such as ***/hello*** after writing *hello()* method handler and link it using *HandleFunc*:

```
	http.HandleFunc("/hello", hello)
```

```
func hello(w http.ResponseWriter, r *http.Request) {
	w.Header().Set("content-type", "text/html")
	fmt.Fprintf(w, "<h2>Hello Go</h2>")
}
```

This is a simple method for writing web pages in Go, but in real development we use templates that we will talk about it later.

Web services

Web service is using the same HTTP package and similar to web page in Go, except it has no user interface and is not targeting browsers, so that there is no HTML here, instead; it targets calling from other applications, but it is still can be called for - testing purposes - from browsers, command line tools, and browser web services plugins.

Here is a simple web service that returns server's date and time in JSON format:

```
// first-service project main.go
package main

import (
	"encoding/json"
	"fmt"
	"net/http"
	"time"
)

func main() {
```

```
     http.HandleFunc("/time", gettime)
     fmt.Println("Listening on \nhttp://localhost:9090")
     err := http.ListenAndServe(":9090", nil)
     if err != nil {
        fmt.Println("Error while listening to port 9090: " + err.Error())
     }
}

type TimeResponseType struct {
     Time string
}

func gettime(w http.ResponseWriter, r *http.Request) {

     w.Header().Set("content-type", "application/json")
     var atime TimeResponseType
     atime.Time = time.Now().String()
     res, _ := json.Marshal(atime)
     w.Write(res)
}
```

Note that the first difference here is ***content-type*** type, which is ***application/json*** in web services instead of *text/html* in web pages. Also we have used struct type to return JSON response object:

```
type TimeResponseType struct {
     Time string
}
```

We call the web service using below request URL:

http://localhost:9090/time

We will get below JSON response:

```
{"Time":"2023-03-10 09:32:39.072559162 +0200 CAT m=+4.697942371"}
```

We can add any other type as fields on this struct object (***TimeResponseType***), but we have to capitalize first letter of field names to make it public, if we write first letter of struct field name in small letter it will be private and will not be displayed in JSON response.

If we want to display it in JSON format as lower-case or in another name we can add JSON formatted as below:

```
type TimeResponseType struct {
     Time string `json:"time"`
}
```

Another example :

```
   Time string `json:"server-time"`
```

This will make field naming in JSON independent of Go field naming

Go HTML templates

Go has templates processors, which enables web developers to isolate HTML design from Go code to achieve single responsibility principle and MVC in web development.

Not like PHP and JSP which allows developers to write code inside presentation (HTML) files, Go language has a simple HTML tags notation that could be used inside HTML to do presentation functions, such as displaying variables, iterating through slice to fill a table, and limited *if conditions*. This notation is a bridge between Go code which represents controller and presentation HTML pages. This notation is called Go template language.

We have created for templates sample a new project called (webpage) and created new sub-folder inside project folder called (templates) in which we can put HTML pages, and we have created new file called *index.html* with below contents:

```
<html>
<head>
<title>Go Template</title>
<body>
  <h3>Go HTML Template web application</h3>
  <p>
    Time in server is: <b> {{.Time}} </b>
    <br/>
    Your IP address is: <b> {{.IP}} </b>
  </p>
</body>
</html>
```

Note that *{{.Time}}* and *{{.IP}}* tags will be replaced later by values passed from Go code.

Here *main.go* file:

```
package main

import (
     "fmt"
     "html/template"
     "net/http"
     "strings"
     "time"
)

var mytemplate *template.Template

func main() {
     mytemplate = template.Must(template.ParseGlob("templates/*"))
     fmt.Println("http://localhost:9090")
     http.HandleFunc("/", index)
     err := http.ListenAndServe(":9090", nil)
     if err != nil {
           fmt.Println("Error while listening to port 9090")
     }
}

type TemplateData struct {
     IP   string
     Time string
```

```
}

func index(w http.ResponseWriter, r *http.Request) {
     var data TemplateData
     data.IP = r.RemoteAddr
     if strings.Contains(data.IP, ":") {
          data.IP = data.IP[:strings.Index(data.IP, ":")]
     }
     data.Time = time.Now().String()[:19]
     mytemplate.Execute(w, data)
}
```

We have defined *mytemplate* global variable as point to *template.Template* type, then initialized it to load all HTML files in (*templates*) directory:

```
      mytemplate = template.Must(template.ParseGlob("templates/*"))
```

We have used our own *struct* type (*TemplateData*) to encapsulate all our required variables and data to be displayed in HTML page

Then we called it inside index handler to display HTML contents and passing required variables/tags to it:

```
      mytemplate.Execute(w, data)
```

r.RemoteAddr returns client IP address with port number such as: *27.0.0.1:43060* and we want to remove the port using strings package:

```
      data.IP = r.RemoteAddr
      if strings.Contains(data.IP, ":") {
             data.IP = data.IP[:strings.Index(data.IP, ":")]
      }
```

We can copy part of string using [:] for instance: copy first two characters:

```
        name := "Ahmed"
        name = name[:2]
```

and last two characters as:

```
        name = name[len(name)-2:]
```

and two letters from the middle as:

```
        name = name[2:4]
```

here is the output page in browser:

http://localhost:9090/

Go HTML Template web application

Time in server is: **2023-04-10 07:26:24**
Your IP address is: **127.0.0.1**

Static contents in Web applications

We can add front-end static contents to Go web application such as: images, css, and java script in directories and make that directory accessible to clients, from our HTTP web server using below as in our below sample.

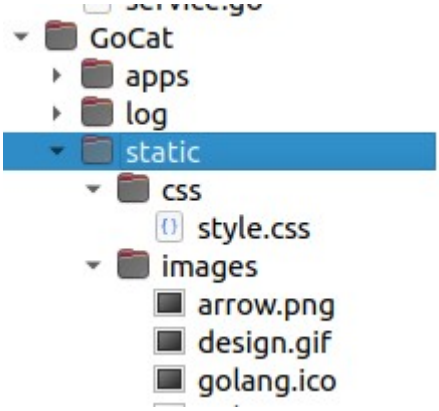
Below sample is part of GoCat project (which is similar to Tomcat), this web application is used for Go web applications and Web services to deploy and mange that applications, complete source exist in this repository:

https://github.com/motaz/gocat

In *main.go* file before calling http.Listen..

```
    fs := SetCacheHeader(http.FileServer(http.Dir("static")))
       http.Handle("/gocat/static/", http.StripPrefix("/gocat/static/", fs))
```

We have a directory called ***static*** contains *css* sub-folder and *images* sub-folder:



We can access static contents from HTML template as in below HTML header tags:

```
<link rel="shortcut icon" href="/gocat/static/images/golang.png" />
<link href="/gocat/static/css/style.css" rel="stylesheet" type="text/css">
```

Deploying Web apps and Web services

In production it is not practical to have many Go web applications and web services with different ports and enable/open that ports in Firewall. Support that we have 3 apps in the same server listening on ports 9090, 9091, and 9092, and more apps could be added later that requires opening new ports if deployment server is behind a firewall, instead we could use web server such as Apache or Nginx to host our applications. Nginx is more easy and ready to work as reverse proxy, here is sample of entry in sites-enabled/default file for one of our Go web apps or web services listening on port 9090:

```
     location /timer/ {
          proxy_pass http://localhost:9090/;
     }
```

We can access it using default port 80 for HTTP or port 443 for HTTPS depending on which section we have defined our reverse proxy entry, we can access it from browser using below URL:

http://localhost/timer

We have to use relative path in our web applications, or we can define a constant path prefix such as:

```
        http.HandleFunc("/goweb", index)
        http.HandleFunc("/goweb/time", timer)
        http.HandleFunc("/goweb/login", login)
        http.HandleFunc("/goweb/logout", logout)
        http.HandleFunc("/goweb/users", users)
        err := http.ListenAndServe(":9090", nil)
```

In this case we could define our reverse proxy in Nginx as:

```
      location /goweb/ {
            proxy_pass http://localhost:9090/goweb/;
      }
```

We can access it in browser or any client side as:

http://localhost/goweb

http://server-ip/goweb

Nginx could also be used as load balancer, we could host Go web applications and web services in multiple machines and configure load balancer and configure reverse proxy for them

HTTP client

We can write HTTP client using *http* package that we have already used in our previous web service sample and web app examples, also it can be used to call that web service.

In this sample we will call IP location web service http://ip2c.org giving it public IP address to return country code and name that given IP belongs to:

```
// http-client project main.go
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
)

func main() {
     ip := "41.209.90.89"
     result := callHTTP("https://ip2c.org/" + ip)
     fmt.Println(result)
}
```

```
func callHTTP(url string) (result string) {

   resp, err := http.Get(url)
   if err != nil {
        fmt.Println("Error: ", err.Error())
   } else {
       body, err := ioutil.ReadAll(resp.Body)
       if err != nil {
           fmt.Println("Error in reading result: ", err.Error())
        } else {
           resp.Body.Close()
        }
        result = string(body)
   }
   return
}
```

Note that we have write new function (*callHTTP*) that accepts URL, sends HTTP Get request, then returns response in *result* parameter.

This is the sample result of running above example:

```
1;SD;SDN;Sudan
```

We can add more features to *callHTTP* function, to include timeout, request content-type, to return content-type, and HTTP status. Here is modified code:

```
package main

import (
      "fmt"
      "io/ioutil"
      "net/http"
      "time"
)

func main() {
     ip := "41.209.90.89"
     status, result := callHTTP("https://ip2c.org/" + ip)
     fmt.Println("Response:")

     fmt.Println(status)
     fmt.Println(result)
}

func callHTTP(url string) (status, result string) {

     timeout := time.Duration(10 * time.Second)
     client := http.Client{
             Timeout: timeout,
     }
     req, _ := http.NewRequest("GET", url, nil)
     req.Header.Set("Content-Type", "text/plain")
     resp, err := client.Do(req)
     if err != nil {
           fmt.Println("Error: ", err.Error())
     } else {
           status = resp.Status
```

```
            fmt.Println("Header Items")
            for key, item := range resp.Header {
                  fmt.Println(key, ": ", item[0])
            }

            body, err := ioutil.ReadAll(resp.Body)
            if err != nil {
                  fmt.Println("Error in reading result: ", err.Error())
            } else {
                  resp.Body.Close()
            }
            result = string(body)
    }
    return
}
```

Here is execution result:

```
Header Items
Content-Type : text/html; charset=UTF-8
Access-Control-Allow-Origin : *
Server : nginx
Date : Tue, 04 Apr 2023 04:31:18 GMT
Response:
200 OK
1;SD;SDN;Sudan (the)
```

We have set timeout of calling that web service to 10 seconds, then we have set *Content-Type* header value to *text/plain*, but this is used with POST method to indicate content type that been sent to web app/service.

Note that our function now is returning two values, and this is a remarkable features in Go language; is to return multiple values in functions. We can ignore any of returning values using underscore character (_), for example:

```
    _, result := callHTTP("https://ip2c.org/" + ip)
```

We couldn't assign value to a variable without using it in Go, so we have to ignore returning values that we will not use when calling functions.

Response header is define as map of string :

```
type Header map[string][]string
```

and their values are array of string:

We do iteration in the map using *for .. range* statement:

```
   for key, item := range resp.Header {
         fmt.Println(key, ": ", item[0])
   }
```

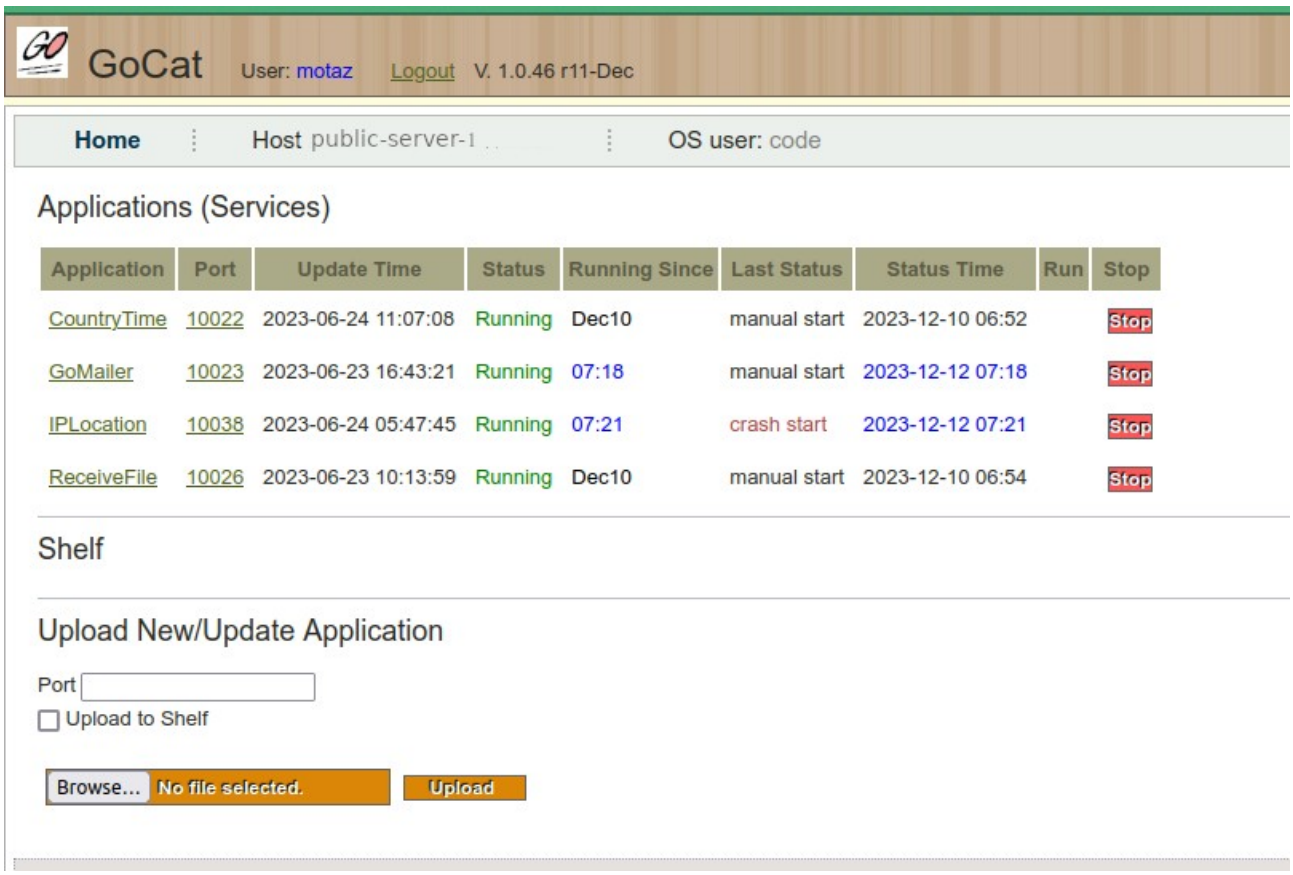
GoCat manager

We have developed GoCat manager which is similar to Java's Apache Tomcat, which allows deployment and managing Go web services and web apps, but the difference is that GoCat is not a web container and not a web server, we still need Nginx as we have described previously to access Go web services using standard HTTP port 80 or HTTPS port 443, and to manage load balancing.

GoCat helps to upload, update and monitor Go services and web apps easily.

For GoCat download please visit below page:

https://github.com/motaz/GoCat



MySQL connection and third party packages

In this sample we will connect to MySQL database, and this requires an import of additional package which is not part of standard Go libraries and packages, so that we need to initialize Go module in this sample project to let the additional package be installed from Internet.

Note that this project will be found in sample projects in this book page, with name: *mysql-sample*

We have used two additional packages in this project:

```
1. github.com/go-sql-driver/mysql
```
which is mysql connection library for Go

```
2. github.com/motaz/codeutils
```

codeutils is a package that contains functions to read from configuration file (.ini) and to write into log file

we put them in module, and this is source of *go.mod* file after initializing Go module using LiteIDE Module menu button:

Go module Init

or using command line:

```
go mod init
```

We will get below text in *go.mod* file:

```
module mysql-sample

go 1.20
```

After that we can import that packages from their location using below command, in LiteIDE we can click on Module menu button then select

Go module Tidy

or using command line when using other IDEs:

```
go mod tidy
```

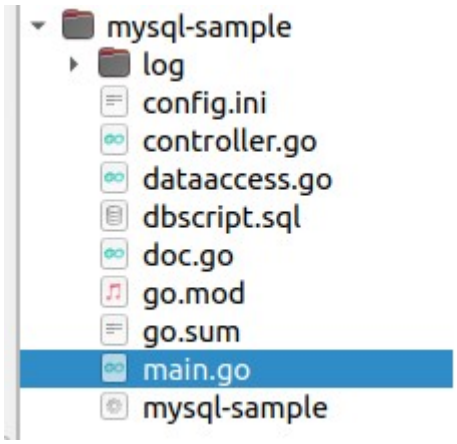
Required packages will be imported from net and ***go.mod*** file will be updated to:

```
module mysql-sample

go 1.20

require (
      github.com/go-sql-driver/mysql v1.7.1
      github.com/motaz/codeutils v1.0.20
)
```

Imported packages will be cached locally in Go package folders (GoPath/pkg/mod) according to their version.

Version is an important tag of Go package, if you write application and depend on specific version of any package, other developers whom need to compile or modify your source will get the exact version of required package when run *go mod tidy* command, this ensures that your application behavior will not change if new version of package has been released.

Here are *mysql-sample* project folder files:



We have divided code into three Go source files to implement single responsibility principle:

1. **main.go** which contains start of project code. Code here is very few because main responsibility of *main.go* code is to start project and it show be minimum to possible for easy understanding of project.
2. **dataaccess.go:** this source file contains mysql database access function, like connection, reading from tables, and updating data.
3. **controller.go:** instead of calling database access functions directly from *main.go* file, it is better to have middleware layer (business layer/controller layer) to have logical and abstract functions, for example if we use Login method, in this layer we could check username validity, and do hashing for password and make sure that this user is not restricted, then call lower level functions in access layer such as getUserInfo. Actual authentication and authorization logic will happen in controller layer not in data access layer, also external authentication could be used here by calling remote web service or LDAP server, or in another access layer file but not in MySQL access layer file.

Other files are:

1. **go.mod**: the file that contains required packages and project name. This file is important for project and should be included in source control

```
module mysql-sample
go 1.20
require (
        github.com/go-sql-driver/mysql v1.7.1
        github.com/motaz/codeutils v1.0.20
)
```

2. **go.sum**: automatically generated file after using tidy tool, this file contains information about locally cached versions of required packages. If cache already exists, then no need to import from remote repository. This file shouldn't be included in source control, because it will be generated automatically after caching packages in each developer machine.

```
github.com/go-sql-driver/mysql v1.7.1 h1:lUIinVbN1DY0xBg0eMOzmmtGoHwWBbvnWubQUrtU8EI=
github.com/go-sql-driver/mysql v1.7.1/go.mod h1:OXbVy3sEdcQ2Doequ6Z5BW6fXNQTmx+9S1MCJN5yJMI=
github.com/motaz/codeutils v1.0.20 h1:fQ6wdkGwUQwsqOyoGocreF9leDHhQzNlPRPtMNV3LL4=
github.com/motaz/codeutils v1.0.20/go.mod h1:3Nwz4asoggDXCVd0Z+5Wf4BPKa2lXZt3tjniSlILN40=
```

As a hacking method if there is no Internet while creating new project, this file (*go.sum*) could be copied manually to new project folder in case of using the same packages, but packages should be already importer before for previous projects, and tidy will not be required because it requires Internet access to download from remote repository

3. **config.ini:** this is a configuration file and it is not part of source. We used *codeutils.GetConfigValue* function to read user name and password of database instead of fixing them inside code, to allow portability when deploying in different MySQL server. This file shouldn't be included in source control, each developer should have different local file to represents local MySQL configuration.

   This is a sample of configuration data inside *config.ini*

```
dbuser=mysqluser
dbpassword=mysecretpass
```

4. **dbscript.sql**: this contains database schema which is used by this project and it is part of project source and should be included in source control, note that it is not part of Go project, but it is required to create database before running application.

```
CREATE TABLE `sample`.`users` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `username` VARCHAR(45) NULL,
  PRIMARY KEY (`id`));
```

Last file *mysql-sample* - which is not part of project source code and **shouldn't be included in source control** - the output of complication which is binary executable for Unix like systems, and in Windows it will be *mysql-sample.exe*

Back to Go source files, we will start with lower level files, which means are close to OS, resources, Databases and far from user interface.

1. dataaccess.go:

```go
package main

import (
	"database/sql"
	"fmt"
	"strings"
	_ "github.com/go-sql-driver/mysql"
	"github.com/motaz/codeutils"
)

func GetConfigurationParameter(param, defaultValue string) string {

     value := codeutils.GetConfigValue("config.ini", param)
     if value == "" {
        value = defaultValue
     }
     return value
}

func WriteLog(event string) {

     codeutils.WriteToLog(event, "log")
}

func SQLConnection() (db *sql.DB, err error) {

	var databaseServer, databaseUser, database, password string
	databaseServer = GetConfigurationParameter("dbserver", "localhost")
	databaseUser = GetConfigurationParameter("dbuser", "")
	database = GetConfigurationParameter("database", "sample")
	password = GetConfigurationParameter("dbpassword", "")
	connectionString := fmt.Sprintf("%v:%v@tcp(%s:3306)/%v?parseTime=true",
		databaseUser, password, databaseServer, database)
	db, err = sql.Open("mysql", connectionString)
	if err != nil {
		WriteLog("Error in SQLConnection: " + err.Error())
	}
	return
}

type UserType struct {
     ID       int
     Username string
}

func ListUsers(db *sql.DB) (users []UserType, err error) {

     sqlStatement := `select id, username from users order by id`
     rows, err := db.Query(sqlStatement)
     users = make([]UserType, 0)
     if err == nil {
          defer rows.Close()
          for rows.Next() {
              var user UserType
              err = rows.Scan(&user.ID, &user.Username)
                  if err == nil {
                         users = append(users, user)
                  }
          }
     } else {
          WriteLog("Error in ListUsers: " + err.Error())
```

```
     }
     return
}

func InsertUser(db *sql.DB, username string) (success bool, err error) {

     sqlStatement := `INSERT INTO users (username) values (?)`
     username = strings.TrimSpace(username)
     _, err = db.Exec(sqlStatement, username)
     success = err == nil
     if !success {
          WriteLog("Error in InsertUser:  " + err.Error())
     }
     return
}
```

At the top of *dataaccess.go* file, we have two functions: *GetConfigurationParameter* and *WriteLog*, these are functions wrappers for similar functions in *codeutils* package *GetConfigValue* and *WriteToLog*. First one reads configuration value from ini file (*config.ini*) and second one writes event log, such as error message in log file. Calling function wrapper is more easy, convenient, more customized and specific for current project, also we could introduce new features that not exist in original functions of package, such as *default* value in *GetConfigurationParameter* function.

There are three database functions:

a. **SQLConnection:** This connects to MySQL server using parameters in *config.ini* file, and returns a pointer to database connection object (db *sql.DB)

b. **ListUsers**: This method retrieves all users information as a list (slice), and it retrieves them from *users* table.

c. **InsertUser**: This method inserts new user into *users* table.

2. controller.go : This Go source file is a higher level than data access layer, it calls *dataaceess.go* functions and it could add business logic and provides new functions with that business logic for higher level layer (presentation layer), which is in our case *main.go*.

Here is *controller.go* code:

```
package main

import (
    "bufio"
    "database/sql"
    "errors"
    "fmt"
    "os"
    "strings"
)

func readAndInsertUser(db *sql.DB) (err error) {
```

```
     fmt.Print("Please enter username: ")
     var username string
     in := bufio.NewReader(os.Stdin)
     username, err = in.ReadString('\n')
     if err == nil && strings.TrimSpace(username) == ""
        err = errors.New("Empty username")
     }
     if err == nil {
         _, err = InsertUser(db, username)
     }
     if err == nil {
        fmt.Println("User: ", username, " Has been added")
     } else {
        fmt.Println("Error in getAndInsertUser: " + err.Error())
     }
    return
}

func showUsers(db *sql.DB) (err error) {

     list, err := ListUsers(db)
     if err == nil {
        for _, user := range list {
               fmt.Printf("User #%d: %s\n", user.ID, user.Username)
        }
     } else {
         fmt.Println("Error in showUsers: " + err.Error())
     }
     return
}

func InsertAndShowUsers() {

     db, err := SQLConnection()
     if err == nil {
            defer db.Close()
            getAndInsertUser(db)
            showUsers(db)
     }
}
```

This file contains three functions:

a. **readAndInsertUser:** This function asks user to enter a name and inserts it to database by calling *InsertUser* method in *dataaccess.go* file.

b. **showUsers**: This function calls *ListUsers* function in *dataaccess.go* layer, and retrieves users list to displays users in console.

Note that above both function names starts with lower case(*read*,, *show*,) that means they are private functions and couldn't be accessed outside current package (**main**), only Go source files that belongs to *main* package could access these private functions, but in our current example all files belongs to the same package (*main*), and we will change this in next example.

c. **InsertAndShowUsers**: This function is public, since it is started with capital letter (*Insert*..) and it could be accessed outside package, and we need higher level layer to access only this method which represents interface of this file (*controller.go*), that means higher level shouldn't access other methods directly, but since higher level file belongs to the same package, we couldn't prevent such access.

3. main.go: This is the starter file in Go because it contains *main* function, and it represents presentation layer in console application. It simply calls *InsertAndShowUsers* in *controller.go*.

This is the source of *main.go* file:

```
// mysql-sample project main.go
package main

func main() {
     InsertAndShowUsers()
}
```

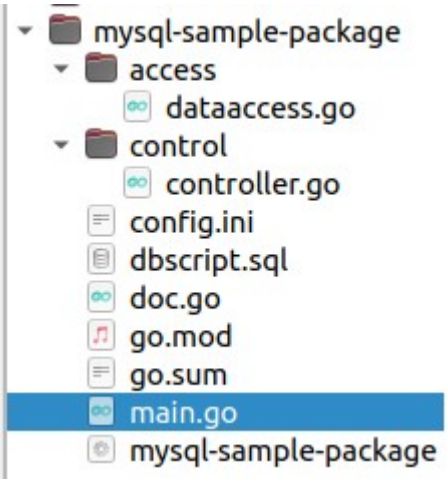
Note that it contains simple code, no much detailed code is there, no business logic and no data access code here. This represents single responsibility and to make project easier to be studied.

MySQL-Sample with packages

Here is the same previous example divided into packages with the same project. Packages are sub-directories inside project, and we have changed the main package name to new packages that has the same name is parent directory, for example inside *access* directory we renamed package name for all files in that directory to *access*.

Here are the new hierarchy of application, we have introduced two new directories: ***access***, and ***control***, which means two additional packages in addition to standard package name (***main***):

Below is new application directory contents (*mysql-sample-package*) which will be found with sample projects with this book:



There are slight changes happens from previous project:

1. dataaccess.go source file package name has been renamed to *access* according to the parent sub-directory of this source file, also has been moved to *access* sub-directory:

```
package access

import (
     "database/sql"
     "fmt"
     "strings"
     _ "github.com/go-sql-driver/mysql"
     "github.com/motaz/codeutils"
)
```

Remaining of source of *dataaccess.go* is the same.

Note that we could have another access files in the same directory with package name (*access*). These files could access their private variables and functions in *access* package as well as public functions.

2. controller.go: There are three changes happen in *controller.go* file, in addition to moving it to *control* folder

```
package control

import (
        "bufio"
        "database/sql"
        "errors"
        "fmt"
        "mysql-sample-package/access"
        "os"
        "strings"
)

func readAndInsertUser(db *sql.DB) (err error) {

        fmt.Print("Please enter username: ")
        var username string
        in := bufio.NewReader(os.Stdin)
        username, err = in.ReadString('\n')
        if err == nil && strings.TrimSpace(username) == "" {
                err = errors.New("Empty username")
        }
        if err == nil {
                _, err = access.InsertUser(db, username)
        }
...
```

First: package has been renamed to ***control***

Second: below line has been added to *import* section to be able to access *access* package:

```
 "mysql-sample-package/access"
```

Third change is that we need to add *(access.)* prefix before accessing any function in *access* layer. In LiteIDE we could start writing (*access.*) And wait for import suggestion, then press enter for adding that package in *import* section.

3. main.go: in main file we have done two changes: adding below package name of controller to import:

```
"mysql-sample-package/control"
```

Then we have added (*control.*) Prefix before calling any function inside *controller.go* file, here is the complete source of *main.go*:

```
// mysql-sample-package project main.go
package main

import (
    "mysql-sample-package/control"
)

func main() {
     control.InsertAndShowUsers()
}
```

Allocating Go files into sub-folder packages is good for large and medium applications for more modularity and to achieve information hiding principle through hiding detailed functions and variables and expose only interfaces that should only be accessed outside their packages such as functions.

Unit testing in Go

Unit testing is important in big projects, specially when working with multiple packages, we need to make sure every function inside package is working properly, and we don't want to wait until higher level functions has written and call that function.

To write unit testing file, we can write it inside package sub-directory and name it with the same package name and suffix it with **_test.go**, for example for *access* package, testing package name will be ***access_test.go***, and we should write a test function start with *Test* prefix in *access_test.go* file, and we can write multiple test functions, example:

access_test.go:

```
package access

import (
     "fmt"
     "testing"
)

func TestUsersList(t *testing.T) {

     db, err := SQLConnection()
     if err == nil {
        list, err := ListUsers(db)
```

```
        if err == nil {
                for _, item := range list {
                        fmt.Printf("%+v\n", item)
                }
         } else {
                fmt.Println(err.Error())
         }
    } else {
         fmt.Println(err.Error())
    }
}

func Test2(t *testing.T) {
      fmt.Println("Test2")
}
```

To run this test file we have to press Ctrl+T in LiteIDE, or click on *Test* button, - our cursor have to be in any file in *access* directory- or using command line inside ***access*** directory

```
go test
```

here is output sample:

```
.../samples/mysql-sample-package/access$ go test
{ID:1 Username:Motaz}
{ID:2 Username:Khalid}
{ID:3 Username:ahmed}
{ID:4 Username:Mohamed Khalid}
{ID:5 Username:Test}
{ID:6 Username:Mohamed}
{ID:7 Username:Ali}
{ID:8 Username:SUhaib}
Test2
PASS
ok    mysql-sample-package/access   0.005s
```

MySQL-Sample with OOP

Go is not an OOP language, but it does support most of OOP features. Our previous projects were normal structured programs that rely on functions without OOP.

We can implement OOP in Go language using *struct* type, we have modified *dataaccess.go* source file that part of *access* package. We have added *struct* type called *DAO* (Data Access Object), this type contains a private field called (*connection*) which holds MySQL connection pointer, we don't want other packages to access this field

dataaccess.go:

```
type DAO struct {
    connection *sql.DB
}
```

This is the main part of OOP which represents class definition, all members (methods) will be linked with this struct.

Initialization or **instantiation** (as in OOP, to create instance of that class), it requires constructor function outside class, because we couldn't access members of object that not initialized yet.

Here is constructor function:

```
func InitDAO() (dao *DAO, err error) {
        dao = new(DAO)
        dao.connection, err = sqlConnection()
        return
}
```

We have initialized an instance of DAO type using ***new*** keyword to allocate memory space and return it's pointer, then we have initialized MySQL connection and have returned MySQL connection in *dao.connection* private field

We have modified *ListUsers* function to become part of DAO class as:

```
func (dao *DAO) ListUsers() (users []UserType, err error)
```

This passes variable (dao) when calling *ListUsers* in controller like this:

```
dao.ListUsers()
```

Also there is another two methods: *InsertUser* and *Close* to close MySQL connection

Here is complete ListUsers method:

```
func (dao *DAO) ListUsers() (users []UserType, err error) {

     sqlStatement := `select id, username from users order by id`
     rows, err := dao.connection.Query(sqlStatement)
     users = make([]UserType, 0)
     if err == nil {
          defer rows.Close()
          for rows.Next() {
              var user UserType
              err = rows.Scan(&user.ID, &user.Username)
              if err == nil {
                    users = append(users, user)
              }
          }
     } else {
          WriteLog("Error in ListUsers: " + err.Error())
     }
     return
}
```

Change here is accessing MySQL connection pointer inside *dao* object:

```
dao.connection.Query(..
```

Here is complete *dataaccess.go* which contains DAO definition, initialization and it's methods :

```
package access

import (
        "database/sql"
```

```
        "fmt"
        "strings"
        _ "github.com/go-sql-driver/mysql"
        "github.com/motaz/codeutils"
)

type DAO struct {
     connection *sql.DB
}

func InitDAO() (dao *DAO, err error) {

     dao = new(DAO)
     dao.connection, err = sqlConnection()
     return
}

func GetConfigurationParameter(param, defaultValue string) string {

     value := codeutils.GetConfigValue("config.ini", param)
     if value == "" {
          value = defaultValue
     }
     return value
}

func WriteLog(event string) {
     codeutils.WriteToLog(event, "log")
}

func sqlConnection() (db *sql.DB, err error) {

      var databaseServer, databaseUser, database, password string
      databaseServer = GetConfigurationParameter("dbserver", "localhost")
      databaseUser = GetConfigurationParameter("dbuser", "")
      database = GetConfigurationParameter("database", "sample")
      password = GetConfigurationParameter("dbpassword", "")
      connectionString := fmt.Sprintf("%v:%v@tcp(%s:3306)/%v?parseTime=true",
              databaseUser, password, databaseServer, database)
      db, err = sql.Open("mysql", connectionString)
      if err != nil {
              WriteLog("Error in SQLConnection: " + err.Error())
      }
      return
}

type UserType struct {
     ID       int
     Username string
}

func (dao *DAO) ListUsers() (users []UserType, err error) {

      sqlStatement := `select id, username from users order by id`
      rows, err := dao.connection.Query(sqlStatement)
      users = make([]UserType, 0)
      if err == nil {
            defer rows.Close()
            for rows.Next() {
                  var user UserType
                  err = rows.Scan(&user.ID, &user.Username)
```

```
                    if err == nil {
                          users = append(users, user)
                    }
              }
       } else {
             WriteLog("Error in ListUsers: " + err.Error())
       }
       return
}

func (dao *DAO) InsertUser(username string) (success bool, err error) {

     sqlStatement := `INSERT INTO users (username) values (?)`
     username = strings.TrimSpace(username)
     _, err = dao.connection.Exec(sqlStatement, username)
     success = err == nil
     if !success {
         WriteLog("Error in InsertUser:  " + err.Error())
     }
     return
}




func (dao *DAO) Close() {
        dao.connection.Close()
}
```

In *controller.go* we have modified initialized *dao* object and add *dao.* prefix when calling functions in *access* package that already part of DAO class :

```
func InsertAndShowUsers() {

     dao, err := access.InitDAO()
     if err == nil {
           defer dao.Close()
           readAndInsertUser(dao)
           showUsers(dao)
     } else {
           fmt.Println("Error in InsertAndShowUsers: " + err.Error())
     }
}
```

We have initialized *dao* object and then pass it as method receiver to be used to call *InsertUser*, *ListUsers* and *Close*.

Here is *showUsers* private function in controller when calling *ListUsers* in *dataaccess*:

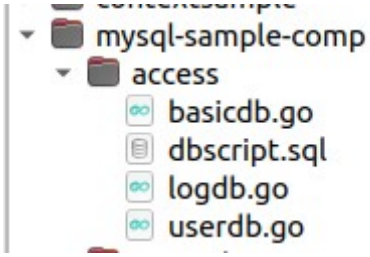
```
func showUsers(dao *access.DAO) (err error) {

     list, err := dao.ListUsers()
     if err == nil {
        for _, user := range list {
              fmt.Printf("User #%d: %s\n", user.ID, user.Username)
        }
     } else {
         fmt.Println("Error in showUsers: " + err.Error())
```

```
    }
    return
}
```

Composition

Go has composition feature instead of inheritance, composition is loosely coupling for classes compared to inheritance that is making tightly coupling of sub-classes with super-classes. Loosely coupling is a good practice and important principles in software engineering.

We have modified last project and renamed it *mysql-sample-comp* to implement this feature. We have added log table to log events and errors for the program. Instead of having one *dataacess.go* file we have abstracted basic class to implement database connection. New file name is *basicdb.go* which contains *DAO* struct type.



This new source file contains *InitDAO*, *sqlConnection*, and *Close* functions/methods.

Here are new code of basicdb.go file:

```
type DAO struct {
        connection *sql.DB
}

func InitDAO() (dao *DAO, err error) {
        dao = new(DAO)
        dao.connection, err = sqlConnection()
        return
}

func (dao *DAO) Close() {
        dao.connection.Close()
}
```

Other user table functions has been moved to *userdb.go* file, which contains *UserDB* struct/class type, this new class contains *Dao* field of type *DAO* class, which implements composition to reuse *DAO* functions:

userdb.go:

```
type UserDB struct {
        Dao *DAO
}
```

Also it contains *InitUserdb* function to initialize new object as well as *Dao* object:

```
func InitUserdb() (userObj *UserDB, err error) {
        userObj = new(UserDB)
        userObj.Dao, err = InitDAO()
        return
}
```

Also *ListUsers*, and *InsertUser* methods has been moved in this file.

This file of access package is *logdb.go* file which contains *LogDB* class and new functions and methods:

InitLogdb to initialize instance of *LogDB* type:

logdb.go:

```
func InitLogdb() (logObj *LogDB, err error) {
        logObj = new(LogDB)
        logObj.Dao, err = InitDAO()
        return
}
```

Also it contains new methods: *ListLastLog*, and *InsertLog*.

Other modifications has done in *main.go* file and *controller.go* to use Log:

main.go file:

```

package main

import (
        "fmt"
        "mysql-sample-comp/access"
        "mysql-sample-comp/control"
        "runtime"
)

func main() {
     logObj, err := access.InitLogdb()
     if err == nil {
             defer logObj.Dao.Close()
             control.LogAction(logObj, "start", "Program started on: "+
                     runtime.GOOS+"."+runtime.GOARCH)
             control.InsertAndShowUsers(logObj)
             control.ShowLog(logObj)
     } else {
          fmt.Println("Unable to initialize log: ", err.Error())
     }
}
```

Note that we could access method inside composed object such as *Close()* method in *DAO*:

```
logObj.Dao.Close()
```

Also we could do multiple composition for different struct types inside new class struct.

Here are new functions in *controller.go* file to access log methods:

```
func LogAction(logObj *access.LogDB, action, details string) {

     logObj.InsertLog(action, details)
}
```

```
func ShowLog(logObj *access.LogDB) {

     fmt.Println("------------")
     fmt.Println("Last log:")
     logRecords, err := logObj.ListLastLog(10)
     if err == nil {
        for _, log := range logRecords {
               fmt.Printf("#%d: %s: %s: %s\n", log.ID,
                  log.ActionTime, log.Action, log.Details)
        }
     } else {
        fmt.Println("Error in ShowLog: " + err.Error())
     }
}
```

Anonymous field composition

We could define struct in sub-struct without field name, for example we have changed *LogDB* struct definition from:

```
type LogDB struct {
        Dao  *DAO
}
```

To:

```
type LogDB struct {
        DAO
}
```

We have removed instance name (*Dao*) so that we could access and inherit *DAO* fields and methods directly from *LogDB* instance, but in this case we could not use initialize for DAO (*InitDAO*) because it requires object reference for DAO, instead we have initialized connection field of *DAO* from sub-struct LogDB:

```
func InitLogdb() (logObj     LogDB, err error) {

    logObj = new(LogDB)
    logObj.connection, err = sqlConnection()
    return
}
```

Instead of accessing connection through Dao instance:

```
logObj.Dao.connection.Query(..
```

We could access it directly:

```
logObj.connection.Query(..
```

Also we could access methods directly:

```
logObj.Close()
```

In addition to not been able to use initialize function of base class, we could face naming conflicts between base and sub-class, we have to avoid similar field and method names, also in multiple composition we could have the same issue.
Defining struct field name is more convenient and more readable than anonymous field for accessing specific base class properties, so that it is better to returned it to:

```
type LogDB struct {
        Dao  *DAO
}
```