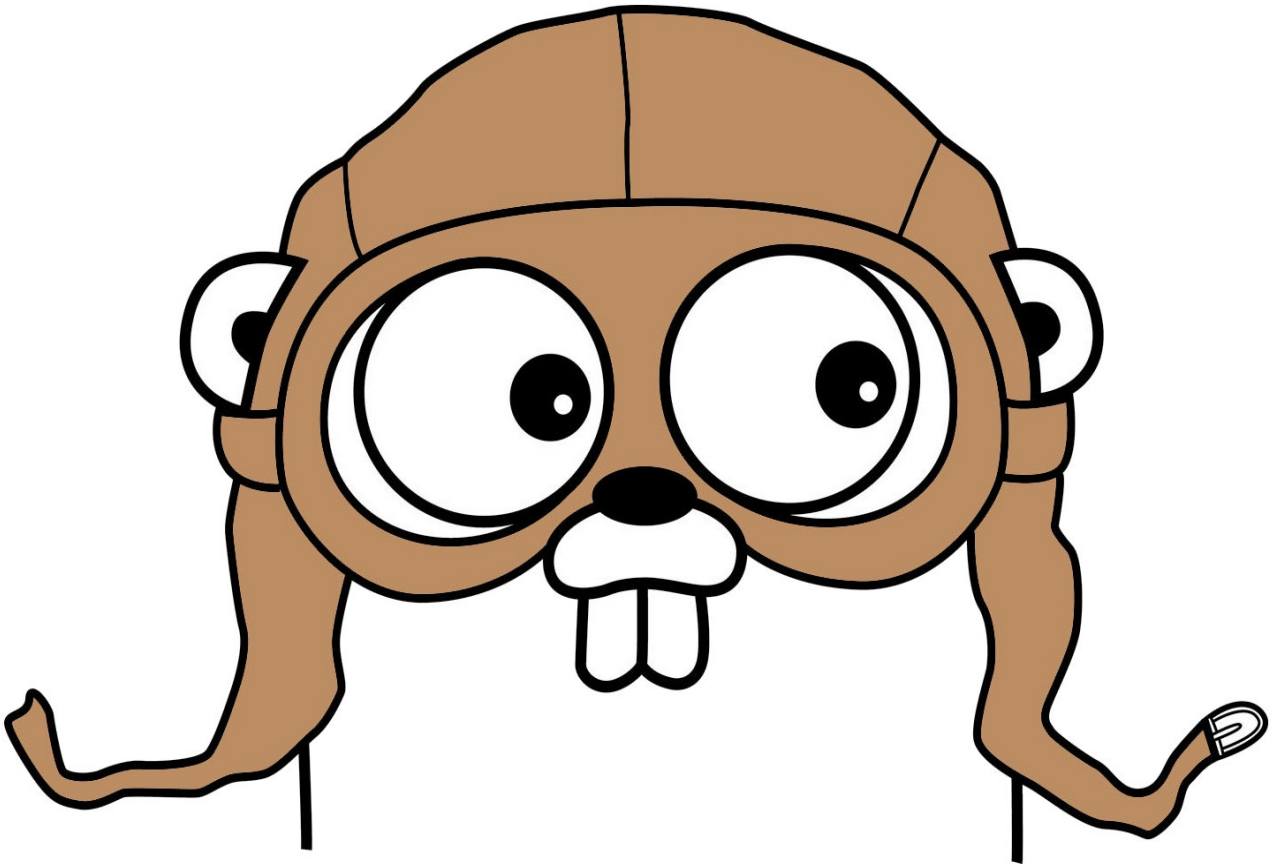


Webapps in Go

the anti textbook

Table of Contents

About the book

Following are the ways to access the book:

1. ePub
2. Mobi
3. PDF Gitbooks
4. PDF Leanpub I find the leanpub formatting a bit better than gitbooks. If you think it is worth it you can pay for it, otherwise, just slide the dial to the left and download the book for free.
5. Read online
6. Gitbooks

This book was written to teach how to develop web applications in Go.

You will create a to do list application as you go along with the book. Learning anything is faster when it is done via examples, this book strives to teach maximum concepts by examples.

This book is open source at Github Repo.

Code

There is a code section available in the Github repo for reference. The code is available for the "Introduction to Go" chapter. For all other chapters, please refer to the http://github.com/thewhitetulip/Tasks, which contains the complete application which you'll build in this book. I use the Tasks application to manage my Todo Lists, so the app will keep getting updated with new features.

Contributing

I don't profess to be a God of either Go or webdev or anything in general, and I don't claim that this is the best book for learning how to build web appplications with Go, but I do believe that good things happen when people collaborate, so pull requests are not only appreciated, but they are welcome.

I got feedback from a reddit user that maybe it is too early for me to start writing this book. Decades ago, a young student from the University of Helsinki had an endless debate with Andrew Tannenbaum on comp.minix. It was about monolithic kernels. Had the student listened to Andrew Tannenbaum, the world probably would not have had Linux. This is the whole point of open source projects, a little initiative from everyone goes a long way. I would like to thank everyone who gave their suggestions on reddit and HN.

Philosophy

- Through this book we want to teach how to develop web applications in Go. We expect the reader to know the basics of Go but we assume the reader knows **nothing** about how to write web applications.
- The book shall comprise of chapters, if the topic is huge and doesn't fit into one chapter, then we split into multiple chapters, if possible.
- Each chapter should be split into logical parts or sections with a meaningful title which'll teach the reader something.
- Every concept should be accompanied by the Go code (if there is any), for *sneak peek* type sections write the Go pseudo code, writing just the necessary parts of the code and keeping everything else.
- The code shouldn't be more than 80 characters wide because in the PDF versions of the book the code is invisible.
- *Brevity is the soul of wit*, please keep the description as small as possible. This doesn't mean we skip it, but try to explain it in as simple words as possible. in such cases do explain the concept.
- In the todo list manager which we are creating, we'll strive to implement as much functionality as possible to give a taste of practical Go programming to the reader. In cases where we re-implement stdlib stuff, we should mention it clearly.
- The main title should have one #, sections should have 2 #'s sub section should have 4 and notes should have 6 #'s (note should have a title too).
- Multi-line code should have three tabs indentation, single line of code can be indented using tabs or by backticks.

Written with love in India.

License:

Note:

1. The Go Programming Basics section has been adapted from build-web-application-with-golang by astaxie Links were updated to refer the correct aspects of the current book, titles were updated to fit into this book. Modifications to the content was done to suit to the style of the book.
2. The gopher in the cover page is taken from https://golang.org/doc/gopher/appenginegophercolor.jpg without modifications.
3. The chapter on database is adapted from https://github.com/VividCortex/go-database-sql-tutorial/ with modifications.

Links

- Next section

Installation

If you know about installation or have installed Go, you can skip to Tools.

This chapter is taken from install page verbatim, except for the changes to be made to adapt to this book's styling format.

System requirements

Go binary distributions are available for these supported operating systems and architectures. Please ensure your system meets these requirements before proceeding. If your OS or architecture is not on the list, you may be able to install from source or use gccgo instead

Operating system	Architectures	Notes
FreeBSD 8-STABLE or later	amd64	Debian GNU/kFreeBSD not supported
Linux 2.6.23 or later with glibc	amd64, 386, arm	CentOS/RHEL 5.x not supported; install from source for ARM
Mac OS X 10.7 or later	amd64	use the clang or gcc† that comes with Xcode‡
Windows XP or later	amd64, 386	use MinGW gcc†. No need for cygwin or msys

†gcc is required only if you plan to use cgo.

‡You only need to install the command line tools for Xcode. If you have already installed Xcode 4.3+, you can install it from the Components tab of the Downloads preferences panel.

Install the Go tools

If you are upgrading from an older version of Go you must first remove the existing version. Linux, Mac OS X, and FreeBSD tarballs

Download the archive and extract it into `/usr/local`, creating a Go tree in `/usr/local/go`. For example:

```
tar -C /usr/local -xzf go$VERSION.$OS-$ARCH.tar.gz
```

Choose the archive file appropriate for your installation. For instance, if you are installing Go version 1.2.1 for 64-bit x86 on Linux, the archive you want is called `go1.2.1.linux-amd64.tar.gz.`

(Typically these commands must be run as root or through sudo.)

Add `/usr/local/go/bin` to the PATH environment variable. You can do this by adding this line to your /etc/profile (for a system-wide installation) or $HOME/.profile:

```
export PATH=$PATH:/usr/local/go/bin
```

Installing to a custom location

The Go binary distributions assume they will be installed in `/usr/local/go` (or `c:\Go` under Windows), but it is possible to install the Go tools to a different location. In this case you must set the GOROOT environment variable to point to the directory in which it was installed.

For example, if you installed Go to your home directory you should add the following commands to $HOME/.profile:

```
export GOROOT=$HOME/go
```

```
export PATH=$PATH:$GOROOT/bin
```

> Note: GOROOT must be set only when installing to a custom location.

Mac OS X package installer

Download the package file, open it, and follow the prompts to install the Go tools. The package installs the Go distribution to `/usr/local/go` .

The package should put the `/usr/local/go/bin directory` in your PATH environment variable. You may need to restart any open Terminal sessions for the change to take effect.

Windows

The Go project provides two installation options for Windows users (besides installing from source): a zip archive that requires you to set some environment variables and an MSI installer that configures your installation automatically.

MSI installer

Open the MSI file and follow the prompts to install the Go tools. By default, the installer puts the Go distribution in c:\Go.

The installer should put the `c:\Go\bin` directory in your PATH environment variable. You may need to restart any open command prompts for the change to take effect.

Zip archive

Download the zip file and extract it into the directory of your choice (we suggest c:\Go).

If you chose a directory other than `c:\Go`, you must set the GOROOT environment variable to your chosen path.

Add the bin subdirectory of your Go root (for example, `c:\Go\bin`) to your PATH environment variable.

Setting environment variables under Windows

Under Windows, you may set environment variables through the "Environment Variables" button on the "Advanced" tab of the "System" control panel. Some versions of Windows provide this control panel through the "Advanced System Settings" option inside the "System" control panel.

Test your installation

Check that Go is installed correctly by setting up a workspace and building a simple program, as follows.

Create a directory to contain your workspace, `$HOME/work` for example, and set the GOPATH environment variable to point to that location.

```
$ export GOPATH=$HOME/work
```

You should put the above command in your shell startup script (`$HOME/.profile` for example) or, if you use Windows, follow the instructions above to set the `GOPATH` environment variable on your system.

Next, make the directories `src/github.com/user/hello` inside your workspace (if you use GitHub, substitute your user name for user), and inside the hello directory create a file named hello.go with the following contents:

```
package main

import "fmt"

func main() {
    fmt.Printf("hello, world\n")
}
```

Then compile it with the go tool:

```
$ go install github.com/user/hello
```

The above command will put an executable command named hello (or hello.exe) inside the bin directory of your workspace. Execute the command to see the greeting:

```
$ $GOPATH/bin/hello
```

> hello, world

If you see the "hello, world" message then your Go installation is working.

Before rushing off to write Go code please read the How to Write Go Code document, which describes some essential concepts about using the Go tools.

Uninstalling Go

To remove an existing Go installation from your system delete the go directory. This is usually `/usr/local/go` under Linux, Mac OS X, and FreeBSD or `c:\Go` under Windows.

You should also remove the Go bin directory from your PATH environment variable. Under Linux and FreeBSD you should edit `/etc/profile` or `$HOME/.profile`. If you installed Go with the Mac OS X package then you should remove the `/etc/paths.d/go` file. Windows users should read the section about setting environment variables under Windows.

Getting help

For real-time help, ask the helpful gophers in `#go-nuts` on the Freenode IRC server.

The official mailing list for discussion of the Go language is Go Nuts.

Report bugs using the Go issue tracker.

Links

-Next section

Tools

HTML: Brackets, a text editor for the web by Adobe.

Go: Any IDE of your choice which has a Go language plugin.

The toolchain

The Go programming language comes with a set of tools along with the standard installation.

gofmt

Usage:

`gofmt main.go` : Prints the formatted source code of main.go file on the console.

`gofmt -w main.go` : Writes the formatted code in the file main.go

`gofmt -w Tasks` : Runs gofmt on all the files in the folder Tasks.

`go fmt` can be used in place of `gofmt -w` .

Most IDEs can be configured to run `gofmt` on save.

It is highly recommended to run `gofmt` before committing to version control.

godoc

It extracts documentation comments on all the Go projects present in `$GOPATH/src` , and the standard library present in `$GOROOT` .

It has two interfaces:

- Web:

  Usage: `godoc -http=:6060`

  The documentation of `net/http` is present at `localhost:6060/pkg/net/http` . godoc also allows users to read the Go source code of the packages. Since Go is now implemented in Go itself, we can read the source code of Go language.

> Note: Verbose Flag
>
> Depending on how much projects are in your `$GOPATH`, it'll take time for godoc to get up and running, please use `-v` flag. Using the -v flag, we come to know when the server got up.

- Commandline:

  Usage: godoc net/http

  This will provide the documentation of net/http on the terminal, like the man command. The catch here is that we need to know the exact library name.

  The packages were comments aren't present show up as blank pages in godoc.

Documentation

The documentation covered by godoc is the documentation about the API, since only the exported functions have documentation comments in godoc. This documentation is different from the logic documentation.

go test

Go has testing support built into the language. For each code file `file.go`, the corresponding test cases should be present in a file named as `file_test.go` in the same folder. The Go compiler ignores all the `*_test.go` files while building the application.

go build

We can build our application using `go build`. It parses all the `.go` files except the `*_test.go` files in the entire folder and all sub folders along with imported libraries if any, and creates a statically linked binary. The binary name is the same as the project folder name, if we want a custom name we should use the -o flag.

Example: `go build -o tasks`

Build time

By default, `go build` builds the entire application and the depending libraries, into a static binary and later throws all away. This results in rebuilding everything every single time `go build` is executed. For caching the library builds, use `go install` first and `go build` later.

Cross compilation

Go allows cross compilation. We have to pass the OS name as linux/darwin/windows as GOOS as shown in the below commands.

```
env GOOS=darwin GOARCH=386 go build -o tasks.app
env GOOS=windows GOARCH=386 go build -o tasks.exe
```

go install

Creates a statically linked binary and places it in `$GOPATH/bin`. Also creastes a `.a` file of the library and puts it in the $GOPATH/pkg folder. In future builds this library file will be reused until the underlying code is changed.

For using tools built with Go like we use unix commands, we need to add `$GOPATH/bin` to the environment variable $PATH.

> Note: $PATH
>
> On Linux/Unix the `$PATH` environment variable is a list of directories which are known to have executables. When we call some executable from any terminal, the shell goes through all folders present in `$PATH` one at a time until it finds the executable.

In Linux/Unix, this is done using: `export PATH=$PATH:$GOPATH/bin`. This line needs to be added to either `.bashrc` or `.profile`, whichever is being used by the shell.

The profile files are present in the home folder. Do a `cd ~` and check for either of the files mentioned above.

go run

go run combines building and running the application in one command.

It generates a binary in the temp folder and executes it. The binary file isn't retained after the run.

go get

This is the package manager in Go. It internally clones the version control repository parameter passed to it, can be any local/remote git repository. It then runs `go install` on the library, making the library available in `$GOPATH/pkg`. If the repository doesn't have any buildable files then go get might complain, but that happens after the repository is cloned.

go clean

This command is for cleaning files that are generated by compilers, including the following files:

```
_obj/            // old directory of object, left by Makefiles
_test/           // old directory of test, left by Makefiles
_testmain.go     // old directory of gotest, left by Makefiles
test.out         // old directory of test, left by Makefiles
build.out        // old directory of test, left by Makefiles
*.[568ao]        // object files, left by Makefiles

DIR(.exe)        // generated by go build
DIR.test(.exe)   // generated by go test -c
MAINFILE(.exe)   // generated by go build MAINFILE.go
```

Other commands

Go provides more commands than those we've just talked about.

```
go fix // upgrade code from an old version before go1 to a new version after go1
go version // get information about your version of Go
go env // view environment variables about Go
go list // list all installed packages
go run // compile temporary files and run the application
```

For details about specific commands, `go help <command>`.

Links

-Previous section -Next section

Hello, Go

Why Go?

Go was built at Google, a company synonymous with Search and Distributed Computing, with Kubernetes. They wanted a language that was fast, worked well with automated code review and formatting and allowed a large team to write large scale software effectively, catered to the multi-core and networking era. All other major languages are at least a decade old. They were created in an era where memory was costly, where there were no massive clusters or multi-core processors.

When switching from other languages to Go, it'll be more or less frusterating to see the restrictions Go has. But as you tag along, the nusiances pay off. The language takes care of other things like formatting, and its goal is to provide a scalable approach to build the application over a long period of time.

In the C family of languages, there are two factions:

```
public static void main() {

}
```

vs

```
public static void main()
{

}
```

The same can be said of Python spaces/tabs.

> Technically speaking, this was an unintended consequence of doing away with the semi colons, since the Go compiler adds semicolons at the end of each line, we can't have the #2 definition in Go.

This might seem to be a shallow problem, but when the codebase and team size grows, then it is difficult to maintain the consistency because of different user preferences. Anyone can write code these days, few can write elegant code. Other languages had to solve this problem as an afterthought. In Go they have been built into the language. Go isn't just a language, it is an ecosystem and it caters to the entire software development cycle.

It aims to provide the efficiency of a statically-typed compiled language with the ease of programming of a dynamic language.

A list of features:

1. Unused imports/variables are compiler errors.
2. Semi-colons not needed, the compiler adds them at the line end.
3. A folder $GOPATH, as it is called contains all your Go code.
4. There is only one standard way to write Go code, use gofmt.
5. Batteries included standard library.
6. Compiled language, thus very fast.
7. Webapps can be written without a framework.
8. Has concurrency built in, just attach the word `go` before a function call to run it in it's own goroutine.
9. Supports Unicode.
10. No language change from 1.0 to 1.7

Let's start with the customary Hello World.

First Program

```
package main

import "fmt"

func main() {
    fmt.Printf("Hello, world or καληµ ρα κόσµ\n")
}
```

It prints following information.

```
Hello, world or καληµ ρα κόσµ
```

Explanation

We import the format package, `fmt` for short. Write the main function of the main package and run the code. We can access only the `Exported` objects of any library in Go. Exported means that the names should start with a capital letter. The compiler will find the `main` function of the `main` package whenever we build or execute any Go code.

We printed non ASCII characters. Go supports UTF-8 by default.

The main package

It is mandatory for each Go program to be a part of a package. The package can be main or not.

Every package except main should be a distinct folder on the `$GOPATH`. Main is a special package for which having a folder on `$GOPATH` is optional.

Building applications differs when we have a main folder and when we don't.

`$GOPATH/src/github.com/thewhitetulip/Tasks`

- main.go
- view.go

or

`$GOPATH/src/github.com/thewhitetulip/Tasks`

- main/main.go
- view.go

In scenarios like this, we need to understand two different ways of executing the application

- With the main folder:

```
[Tasks] $ go build main/main.go
[Tasks] $ ./main/main
```

This will function correctly, because we are in the Tasks directory while executing our binary, all the templates and other files are present in this folder.

- Without the main folder

```
[Tasks/main] $ go build main.go
[Tasks/main] $ ./main
```

Here, we are in the Tasks/main directory, the binary will expect all the other files in the Tasks/main directory when they are in the Tasks directory, one level up.

There can be only *one* main package & function per executable program. The main function takes no arguments passed and returns nothing.

Links

-Previous section -Next section

Variables & Data structures

Variables

We use the keyword var to define a variable. The variable type comes **after** the variable name.

```
// define a variable with name "numberOfTimes" and type "int"
// the examples are of int, but you can use any other data type.
var numberOfTimes int

// define three variables which types are "int"
var var1, var2, var3 int

// define a variable with name numberOfTimes”, type "int"
// and value "3"
var numberOfTimes int = 3

/*
Define three variables with type "int", and
initialize their values.
var1 is 1, var2 is 2, var3 is 3.
*/
var var1, var2, var3 int = 1, 2, 3
```

Variable declaration Shortcut

```
/*
Define three variables without type "type" and
without keyword "var", and initialize their values.
vname1 is v1，vname2 is v2，vname3 is v3
*/
vname1, vname2, vname3 := v1, v2, v3
```

`:=` can only be used inside functions, for defining global variables we have to stick to using var. It throws a syntax error otherwise.

Blank variable

`_` is called the blank variable and it is used to ignore a value.

Suppose we have a function divide, which takes two arguments, and will return the quotient and remainder of arg1 divided by arg2. But we only want the remainder, so we ignore the quotient using

```
_ , remainder := divide(10, 3)
```

Unused variables cause compilation errors. Compile the following code and see what happens. Sometimes, we use this notations to import libraries which we do not want to use as prefix, while working with databases we do not want to prefix the library name to each function call of the library, so we use the blank variable. We will see that in a future chapter.

```
package main

func main() {
    var i int
}
```

Constants

Constants are the values that are determined during compile time that cannot be changed during runtime. In Go, you can use number, boolean or string as types of constants.

Defining constants

```
const constantName = value
// you can assign type of constants if it's necessary
const Pi float32 = 3.1415926
```

More examples.

```
const Pi = 3.1415926
const i = 10000
const MaxThread = 10
const prefix = "astaxie_"
```

Elementary types

Boolean

The keyword `bool` is used to define a variable of the boolean type. There are two boolean values, `true` or `false`; `false` will be the default value. Unlike other languages, Go doesn't allow us to convert boolean into number.

```
// sample code
var isActive bool  // global variable
var enabled, disabled = true, false  // omit type of variables
func test() {
    var available bool  // local variable
    valid := false      // brief statement of variable
    available = true    // assign value to variable
}
```

Numerical types

Integers

Integer types include signed and unsigned integer types, `int` and `uint` respectively. They have the same length, but the specific length depends on your operating system. They use 32-bit in 32-bit operating systems, and 64-bit in 64-bit operating systems. Go also has types that have specific length including `rune`, `int8`, `int16`, `int32`, `int64`, `byte`, `uint8`, `uint16`, `uint32`, `uint64`. Note that `rune` is alias of `int32` and `byte` is alias of `uint8`.

Go doesn't allow assigning values between data types. The following operation will cause compilation errors.

```
var a int8

var b int32

c := a + b
```

Although int32 has a longer length than int8, and has the same type as int, you cannot assign values between them. (***c will be asserted as type `int` here***)

Fractions

Float types have the `float32` and `float64` types and no type called `float`. The latter one is the default type if using brief statement.

Complex numbers

Go supports complex numbers as well. `complex128` (with a 64-bit real and 64-bit imaginary part) is the default type, if you need a smaller type, there is one called `complex64` (with a 32-bit real and 32-bit imaginary part). Its form is `RE+IMi`, where `RE` is real part and `IM` is

imaginary part, the last `i` is the imaginary number. There is a example of complex number.

```
var c complex64 = 5+5i
//output: (5+5i)
fmt.Printf("Value is: %v", c)
```

String

Go uses the UTF-8 character set. Strings are represented by double quotes `""` or backticks ``` `` ```.

```
// sample code
var frenchHello string  // basic form to define string
var emptyString string = ""  // define a string with empty string
func test() {
    no, yes, maybe := "no", "yes", "maybe"  // brief statement
    japaneseHello := "Ohaiou"
    frenchHello = "Bonjour"  // basic form of assign values
}
```

String objects do not now allow value change. You will get errors when you compile the following code.

```
var s string = "hello"
s[0] = 'c'
```

For changing a string, a new string has to be created using the old string. Go doesn't allow modifications to a string variable, but you can always create a new one.

```
s := "hello"
c := []byte(s)  // convert string to []byte type
c[0] = 'c'
s2 := string(c)  // convert back to string type
fmt.Printf("%s\n", s2)
```

`+` operator cane be used to concatenate two strings.

```
s := "hello,"
m := " world"
a := s + m
fmt.Printf("%s\n", a)
```

and also.

```
s := "hello"
s = "c" + s[1:] // returns substring from index 1 till the end.
fmt.Printf("%s\n", s)
```

```` (backtick) is used to have a multiple-line string.

```
m := `hello
world`
```

` (backtick) will not escape any characters in a string.

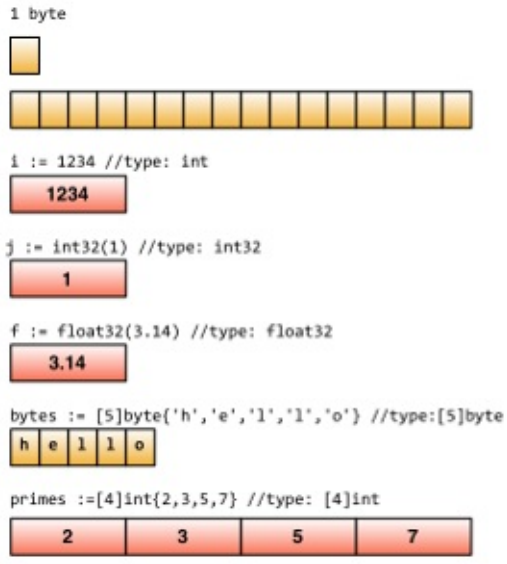
## Error types

Go doesn't have the try catch block. There is one `error` type for the purpose of dealing with errors in the package called `errors`. Go requires us to explicitly handle our errors or ignore them. Typical error handling forms a `if err != nil` ladder.

```
err := errors.New("emit macho dwarf: elf header corrupted")
if err != nil {
    fmt.Print(err)
}
```

# Underlying data structures

The following picture comes from an article about Go data structure in Russ Cox's Blog. As you can see, Go utilizes blocks of memory to store data.


```

# Important points

## Define by group

If you want to define multiple constants, variables or import packages, you can use the group form.

Basic form.

```
import "fmt"
import "os"

const i = 100
const pi = 3.1415
const prefix = "Go_"

var i int
var pi float32
var prefix string
```

Group form.

```
// if you have imports from both standard library and custom
// imports, the standard library imports go first,
// followed by others.

import(
    "fmt"
    "os"

    "github.com/thewhitetulip/Tasks/views"
)

// value of first const will be 0 unless it is assigned to iota
// If there are no assigned values for the elements except the last one,
// all constants will have the same value as the last one.

const(
    i = 100
    pi = 3.1415
    prefix = "Go_"
)

var(
    i int
    pi float32
    prefix string
)
```

## iota enumerate

Go has a keyword called `iota`, this keyword is to make `enum`, it begins with `0`, increased by `1`.

```
const(
    x = iota  // x == 0
    y = iota  // y == 1
    z = iota  // z == 2
    w  // If there is no expression after the constants name,
       // it uses the last expression,
    // so it's saying w = iota implicitly. Therefore w == 3,
    // and y and z both can omit "= iota" as well.
)

const v = iota // once iota meets keyword `const`, it resets to `0`, so v = 0.

const (
  e, f, g = iota, iota, iota // e=0,f=0,g=0 values of iota are same in one line.
)
```

## Some rules

Go is concise because it has some default behaviors.

- Any variable that begins with a capital letter means it will be exported, private otherwise.
- The same rule applies for functions and constants, no `public` or `private` keyword exist in Go.

# array, slice, map

## array

`array` is an array, we define one as follows.

```
var arr [n]type
```

in `[n]type`, `n` is the length of the array, `type` is the type of its elements. Like other languages, we use `[]` to get or set element values within arrays.

```
var arr [10]int  // an array of type [10]int
arr[0] = 42      // array is 0-based
arr[1] = 13      // assign value to element
fmt.Printf("The first element is %d\n", arr[0])
// get element value, it returns 42
fmt.Printf("The last element is %d\n", arr[9])
// it returns default value of 10th element in this array,
// which is 0 in this case.
```

Since length is a part of the array type, `[3]int` and `[4]int` are different types. We cannot change the length of arrays.

Arrays are passed as copy rather than references when passed to a function as arguments. For references, we have to use `slice`. More about slices below.

It's possible to use `:=` when you define arrays.

```
a := [3]int{1, 2, 3} // define an int array with 3 elements

b := [10]int{1, 2, 3}
// define a int array with 10 elements, of which the first three are assigned.
//The rest of them use the default value 0.

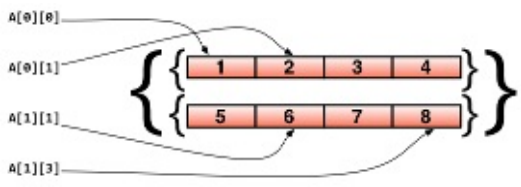
c := [...]int{4, 5, 6} // use `…` to replace the length parameter and Go will calculat
e it for you.
```

You may want to use arrays as arrays' elements. Let's see how to do this.

```
// define a two-dimensional array with 2 elements, and each element has 4 elements.
doubleArray := [2][4]int{[4]int{1, 2, 3, 4}, [4]int{5, 6, 7, 8}}

// The declaration can be written more concisely as follows.
easyArray := [2][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}}
```

Array underlying data structure.




## slice

The main disadvantage of arrays is that we need to know the size prehand. At times this isn't possible. For this, we need a "dynamic array". This is called a `slice` in Go.

`slice` is not really a `dynamic array`. It's a reference type. `slice` points to an underlying `array` whose declaration is similar to `array`, but doesn't need length.

```
// just like defining an array, but this time, we exclude the length.
var fslice []int
```

Then we define a `slice`, and initialize its data.

```
slice := []byte {'a', 'b', 'c', 'd'}
```

`slice` can redefine existing slices or arrays. `slice` uses `array[i:j]` to slice, where `i` is the start index and `j` is end index, but notice that `array[j]` will not be sliced since the length of the slice is `j - i`.

```
// define an array with 10 elements whose types are bytes
var ar = [10]byte {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}

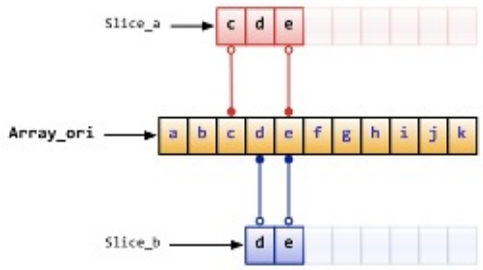
// define two slices with type []byte
var a, b []byte

// 'a' points to elements from 3rd to 5th in array ar.
a = ar[2:5]
// now 'a' has elements ar[2],ar[3] and ar[4]

// 'b' is another slice of array ar
b = ar[3:5]
// now 'b' has elements ar[3] and ar[4]
```

Notice the differences between `slice` and `array` when you define them. We use `[…]` to let Go calculate length but use `[]` to define slice only. Slices do not have length, slices point to arrays which have lengths.

Their underlying data structure.




slice has some convenient operations.

- `slice` is 0-based, `ar[:n]` equals to `ar[0:n]`
- The second index will be the length of `slice` if omitted, `ar[n:]` equals to

`ar[n:len(ar)]`.

- You can use `ar[:]` to slice whole array, reasons are explained in first two statements.

More examples:

```
// define an array
var array = [10]byte{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}
// define two slices
var aSlice, bSlice []byte

// some convenient operations
aSlice = array[:3] // equals to aSlice = array[0:3] aSlice has elements a,b,c
aSlice = array[5:] // equals to aSlice = array[5:10] aSlice has elements f,g,h,i,j
aSlice = array[:]  // equals to aSlice = array[0:10] aSlice has all elements

// slice from slice
aSlice = array[3:7]  // aSlice has elements d,e,f,g，len=4，cap=7
bSlice = aSlice[1:3] // bSlice contains aSlice[1], aSlice[2], so it has elements e,f
bSlice = aSlice[:3]  // bSlice contains aSlice[0], aSlice[1], aSlice[2], so it has d,e
,f
bSlice = aSlice[0:5] // slice could be expanded in range of cap, now bSlice contains d
,e,f,g,h
bSlice = aSlice[:]   // bSlice has same elements as aSlice does, which are d,e,f,g
```

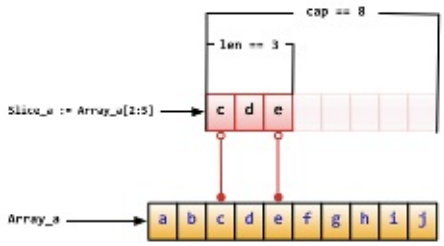
`slice` is a reference type, so any changes will affect other variables pointing to the same slice or array. For instance, in the case of `aSlice` and `bSlice` above, if you change the value of an element in `aSlice`, `bSlice` will be changed as well.

`slice` is like a struct by definition and it contains 3 parts.

- A pointer that points to where `slice` starts.
- The length of `slice`.
- Capacity, the length from start index to end index of `slice`.

```
Array_a := [10]byte{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}
Slice_a := Array_a[2:5]
```

The underlying data structure of the code above as follows.




There are some built-in functions for slice.

- `len` gets the length of `slice`.
- `cap` gets the maximum length of `slice`
- `append` appends one or more elements to `slice`, and returns `slice`.
- `copy` copies elements from one slice to the other, and returns the number of elements that were copied.

Attention: `append` will change the array that `slice` points to, and affect other slices that point to the same array.

Also, if there is not enough length for the slice ( `(cap-len) == 0` ), `append` returns a new array for this slice. When this happens, other slices pointing to the old array will not be affected.

## map

Map is a key value pair, like a dictionary in Python. Use the form `map[keyType]valueType` to define it.

The 'set' and 'get' values in `map` are similar to `slice`, however the index in `slice` can only be of type 'int' while `map` can use much more than that: for example `int`, `string`, or whatever you want. Also, they are all able to use `==` and `!=` to compare values.

```
// use string as the key type, int as the value type, and `make` initialize it.
var numbers map[string] int
// another way to define map
numbers := make(map[string]int)
numbers["one"] = 1  // assign value by key
numbers["ten"] = 10
numbers["three"] = 3

fmt.Println("The third number is: ", numbers["three"]) // get values
// It prints: The third number is: 3
```

Some notes when you use map.

- `map` is disorderly. Everytime you print `map` you will get different results. It's impossible to get values by `index`. You have to use `key`.
- `map` doesn't have a fixed length. It's a reference type just like `slice`.
- `len` works for `map` also. It returns how many `key`s that map has.
- It's quite easy to change the value through `map`. Simply use `numbers["one"]=11` to change the value of `key` one to `11`.

You can use form `key:val` to initialize map's values, and `map` has built-in methods to check if the `key` exists.

Use `delete` to delete an element in `map`.

```
// Initialize a map
rating := map[string]float32 {"C":5, "Go":4.5, "Python":4.5, "C++":2 }
// map has two return values. For the second return value, if the key doesn't
//exist，'ok' returns false. It returns true otherwise.
csharpRating, ok := rating["C#"]
if ok {
        fmt.Println("C# is in the map and its rating is ", csharpRating)
} else {
        fmt.Println("We have no rating associated with C# in the map")
}

delete(rating, "C")  // delete element with key "c"
```

As I said above, `map` is a reference type. If two `map`s point to same underlying data, any change will affect both of them.

```
m := make(map[string]string)
m["Hello"] = "Bonjour"
m1 := m
m1["Hello"] = "Salut"  // now the value of m["hello"] is Salut
```

## make, new

`make` does memory allocation for built-in models, such as `map`, `slice`, and `channel`, while `new` is for types' memory allocation.

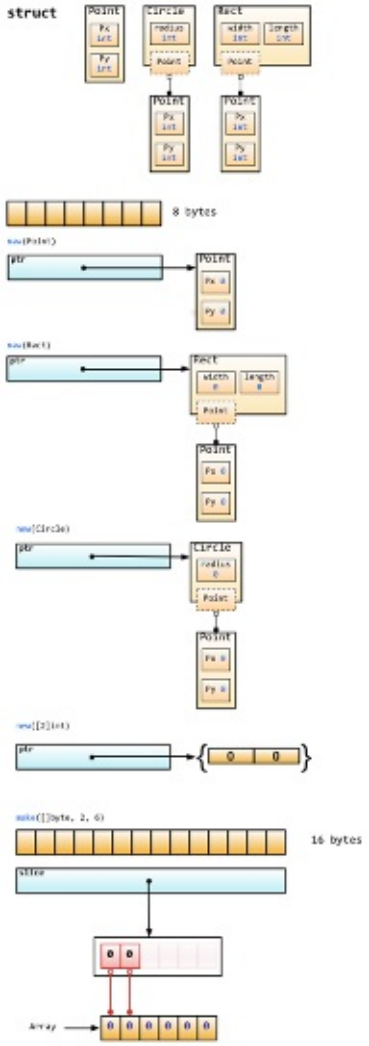
`new(T)` allocates zero-value to type `T`'s memory, returns its memory address, which is the value of type `*T`. By Go's definition, it returns a pointer which points to type `T`'s zero-value.

`new` returns pointers.

The built-in function `make(T, args)` has different purposes than `new(T)`. `make` can be used for `slice`, `map`, and `channel`, and returns a type `T` with an initial value. The reason for doing this is because the underlying data of these three types must be initialized before they point to them. For example, a `slice` contains a pointer that points to the underlying `array`, length and capacity. Before these data are initialized, `slice` is `nil`, so for `slice`, `map` and `channel`, `make` initializes their underlying data and assigns some suitable values.

`make` returns non-zero values.

The following picture shows how `new` and `make` are different.




Zero-value does not mean empty value. It's the value that variables default to in most cases. Here is a list of some zero-values.

```
int     0
int8    0
int32   0
int64   0
uint    0x0
rune    0 // the actual type of rune is int32
byte    0x0 // the actual type of byte is uint8
float32 0 // length is 4 byte
float64 0 //length is 8 byte
bool    false
string  ""
```

### Links

-Previous section -Next section

# Control statements and Functions

# Control statement

## if

`if` doesn't need parentheses in Go.

```
if x > 10 {
    //when x is greater than 10
    //program enters this block
    fmt.Println("x is greater than 10")
} else {
    //when x is smaller than 10
    //program enters this block
    fmt.Println("x is less than or equal to 10")
}
```

Go allows us to initialize and use variables in if like this:

```
// initialize x, then check if x greater than
if x := computedValue(); x > 10 {
    fmt.Println("x is greater than 10")
} else {
    fmt.Println("x is less than 10")
}

// the following code will not compile
fmt.Println(x)
```

For multiple conditions we use the else if block

```
if integer == 3 {
    fmt.Println("The integer is equal to 3")
} else if integer < 3 {
    fmt.Println("The integer is less than 3")
} else {
    fmt.Println("The integer is greater than 3")
}
```

## goto

Go has a `goto` keyword, but be careful when you use it. `goto` reroutes the control flow to a previously defined `label` within the body of same code block.

```
func myFunc() {
    i := 0
Here:   // label ends with ":"
    fmt.Println(i)
    i++
    goto Here   // jump to label "Here"
}
```

The label name is case sensitive.

## for

Go does not have while, do while. Just a `for`, which is the most powerful control logic. It can read data in loops and iterative operations, just like `while`. Like `if`, `for` doesn't need parenthesis.

```
for expression1; expression2; expression3 {
    //...
}


package main
import "fmt"

func main(){
    sum := 0;
    for index:=0; index < 10 ; index++ {
        sum += index
    }
    fmt.Println("sum is equal to ", sum)
}
// Print: sum is equal to 45
```

We can omit one or more expressions.

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

for {
    //this is an infinite loop
}
```

Using `for` like a `while`

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

## break and continue

`break` : jumps out of the loop. If you have nested loops, use `break` along with labels.

`continue` skips the current loop and starts the next one

```
for index := 10; index>0; index-- {
    if index == 5{
        break // or continue
    }
    fmt.Println(index)
}
// break prints 10、9、8、7、6
// continue prints 10、9、8、7、6、4、3、2、1
```

`for` can read data from `slice` and `map` when it is used together with `range`.

```
for k,v:=range map {
    fmt.Println("map's key:",k)
    fmt.Println("map's val:",v)
}
```

Because Go supports multi-value returns and gives compile errors when you don't use values that were defined, as discussed earlier, `_` is used to discard certain return values.

```
for _, v := range map{
    fmt.Println("map's val:", v)
}
```

## switch

Switch is an easier way to avoid long `if-else` statements.

```
switch sExpr {
case expr1:
    some instructions
case expr2:
    some other instructions
case expr3:
    some other instructions
default:
    other code
}
```

The type of `sExpr`, `expr1`, `expr2`, and `expr3` must be the same.

Conditions don't have to be constants and it checks the cases from top to bottom until it matches conditions and executes only the matching case.

If there is no statement after the keyword `switch`, then it matches `true`.

The default case is when there is no match to the switch.

```
i := 10
switch i {
case 1:
    fmt.Println("i is equal to 1")
case 2, 3, 4:
    fmt.Println("i is equal to 2, 3 or 4")
case 10:
    fmt.Println("i is equal to 10")
default:
    fmt.Println("All I know is that i is an integer")
}
```

Cases can have more than one values separated by a comma. By default switch executes only the matching case, however we can make switch execute the matching case and all cases below it using the `fallthrough` statement.

```
integer := 6
switch integer {
case 4:
    fmt.Println("integer <= 4")
    fallthrough
case 5:
    fmt.Println("integer <= 5")
    fallthrough
case 6:
    fmt.Println("integer <= 6")
    fallthrough
case 7:
    fmt.Println("integer <= 7")
    fallthrough
case 8:
    fmt.Println("integer <= 8")
    fallthrough
default:
    fmt.Println("default case")
}
```

This program prints the following information.

```
integer <= 6
integer <= 7
integer <= 8
default case
```

# Functions

`func` keyword is used to define a function.

```
func funcName(input1 type1, input2 type2) (output1 type1, output2 type2) {
    // function body
    // multi-value return
    return value1, value2
}
```

- Functions may have zero, one or more than one arguments. The argument type comes after the argument name and arguments are separated by `,`.
- Functions can return multiple values.
- There are two return values named `output1` and `output2`, you can omit their names and use their type only.
- If there is only one return value and you omitted the name, you don't need brackets for the return values.

- If the function doesn't have return values, you can omit the return parameters altogether.
- If the function has return values, you have to use the `return` statement somewhere in the body of the function.

A simple program to calculate maximum value.

```
package main
import "fmt"

// returns the greater value between a and b
func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

func main() {
    x := 3
    y := 4
    z := 5

    max_xy := max(x, y) // call function max(x, y)
    max_xz := max(x, z) // call function max(x, z)

    fmt.Printf("max(%d, %d) = %d\n", x, y, max_xy)
    fmt.Printf("max(%d, %d) = %d\n", x, z, max_xz)
    fmt.Printf("max(%d, %d) = %d\n", y, z, max(y,z)) // call function here
}
```

In a function call, if two or more arguments have the same data type, then we can put the data type only after the last argument.

`func max(a,b int, c,d string)` : this means we have four arguments, a,b: integers and c,d: string.

## Multi-value return

```
package main
import "fmt"

// return results of A + B and A * B
func SumAndProduct(A, B int) (int, int) {
return A+B, A*B
}

func main() {
    x := 3
    y := 4

    xPLUSy, xTIMESy := SumAndProduct(x, y)

    fmt.Printf("%d + %d = %d\n", x, y, xPLUSy)
    fmt.Printf("%d * %d = %d\n", x, y, xTIMESy)
}
```

SumAndProduct will return two values without names. Go allows us to have named return arguments. By using named arguments, the respective variables are returned automatically, we would just need to use `return`.

If functions are going to be used outside of current program, it is better to explicitly write return statements for the sake of readability.

```
func SumAndProduct(A, B int) (add int, multiplied int) {
    add = A+B
    multiplied = A*B
    return
}
// Since return arguments are named, the function automatically
// returns them
```

## Variadic arguments to functions

In many cases, we do not know how many arguements can be passed, in such cases, we use variadic arguments.

```
func myfunc(arg ...int) {}
```

`arg …int` tells Go that this is a function that has variable arguments. Notice that these arguments are type `int`. In the body of function, the `arg` becomes a `slice` of `int`.

```
for _, n := range arg {
    fmt.Printf("And the number is: %d\n", n)
}
```

## Pass by value and pointers

Argument are passed by value to the functions, the argument change inside the function doesn't affect the arguments used to call the function.

```
package main
import "fmt"

// simple function to add 1 to a
func add1(a int) int {
    a = a+1 // we change value of a
    return a // return new value of a
}

func main() {
    x := 3

    fmt.Println("x = ", x)  // should print "x = 3"

    x1 := add1(x)  // call add1(x)

    fmt.Println("x+1 = ", x1) // should print "x+1 = 4"
    fmt.Println("x = ", x)    // should print "x = 3"
}
```

The original value of `x` doesn't change, because we passed x as a value, so the function add1 created a copy of x. Despite having the same names, the both variables are totally independant of each other.

In cases where we want to be able to modify the argument's value, we use pass by reference using pointers.

In reality, a variable is nothing but a pointer to a location in memory. Each variable has a unique memory address. So, if we want to change the value of a variable, we must change its memory address. Therefore the function `add1` has to know the memory address of `x` in order to change its value. Here we pass `&x` to the function, and change the argument's type to the pointer type `*int`. Be aware that we pass a copy of the pointer, not copy of value.

```
package main
import "fmt"

// simple function to add 1 to a
func add1(a *int) int {
    *a = *a+1 // we changed value of a
    return *a // return new value of a
}

func main() {
    x := 3

    fmt.Println("x = ", x)  // should print "x = 3"

    x1 := add1(&x)  // call add1(&x) pass memory address of x

    fmt.Println("x+1 = ", x1) // should print "x+1 = 4"
    fmt.Println("x = ", x)    // should print "x = 4"
}
```

Advantages of pointers:

- Allows us to use more functions to operate on one variable.
- Low cost by passing memory addresses (8 bytes), copy is not an efficient way, both in terms of time and space, to pass variables.
- `string`, `slice` and `map` are reference types, so they use pointers when passing to functions by default. (Attention: If you need to change the length of `slice`, you have to pass pointers explicitly)

## defer

Defer postpones the execution of a function till the calling function has finished executing. You can have many `defer` statements in one function; they will execute in reverse order when the program reaches its end. In the case where the program opens some resource files, these files would have to be closed before the function can return with errors. Let's see some examples.

```
func ReadWrite() bool {
    file.Open("file")
// Do some work
    if failureX {
        file.Close()
        return false
    }

    if failureY {
        file.Close()
        return false
    }

    file.Close()
    return true
}
```

We saw some code being repeated several times. `defer` solves this problem very well. It doesn't only help you to write clean code but also makes your code more readable.

```
func ReadWrite() bool {
    file.Open("file")
    defer file.Close()
    if failureX {
        return false
    }
    if failureY {
        return false
    }
    return true
}
```

If there are more than one `defer`s, they will execute by reverse order. The following example will print `4 3 2 1 0`.

```
for i := 0; i < 5; i++ {
    defer fmt.Printf("%d ", i)
}
```

## Functions as values and types

Functions are also variables in Go, we can use `type` to define them. Functions that have the same signature can be seen as the same type.

```
type typeName func(input1 inputType1 , input2 inputType2 [, ...]) (result1 resultType1
 [, ...])
```

This makes Go a functional language as functions are a first class citizen.

```
package main
import "fmt"

type testInt func(int) bool // define a function type of variable

func isOdd(integer int) bool {
    if integer%2 == 0 {
        return false
    }
    return true
}

func isEven(integer int) bool {
    if integer%2 == 0 {
        return true
    }
    return false
}

// pass the function `f` as an argument to another function

func filter(slice []int, f testInt) []int {
    var result []int
    for _, value := range slice {
        if f(value) {
            result = append(result, value)
        }
    }
    return result
}

func main(){
    slice := []int {1, 2, 3, 4, 5, 7}
    fmt.Println("slice = ", slice)
    odd := filter(slice, isOdd)    // use function as values
    fmt.Println("Odd elements of slice are: ", odd)
    even := filter(slice, isEven)
    fmt.Println("Even elements of slice are: ", even)
}
```

It's very useful when we use interfaces. As you can see `testInt` is a variable that has a function as type and the returned values and arguments of `filter` are the same as those of `testInt`. Therefore, we can have complex logic in our programs, while maintaining flexibility in our code.

## Panic and Recover

Go doesn't have `try-catch` structure like Java does. Instead of throwing exceptions, Go uses `panic` and `recover` to deal with errors. However, you shouldn't use `panic` very much, although it's powerful.

`Panic` is a built-in function to break the normal flow of programs and get into panic status. When a function `F` calls `panic`, `F` will not continue executing but its `defer` functions will continue to execute. Then `F` goes back to the break point which caused the panic status. The program will not terminate until all of these functions return with panic to the first level of that `goroutine`. `panic` can be produced by calling `panic` in the program, and some errors also cause `panic` like array access out of bounds errors.

`Recover` is a built-in function to recover `goroutine`s from panic status. Calling `recover` in `defer` functions is useful because normal functions will not be executed when the program is in the panic status. It catches `panic` values if the program is in the panic status, and it gets `nil` if the program is not in panic status.

The following example shows how to use `panic`.

```
var user = os.Getenv("USER")

func init() {
    if user == "" {
        panic("no value for $USER")
    }
}
```

The following example shows how to check `panic`.

```
func throwsPanic(f func()) (b bool) {
    defer func() {
        if x := recover(); x != nil {
            b = true
        }
    }()
    f() // if f causes panic, it will recover
    return
}
```

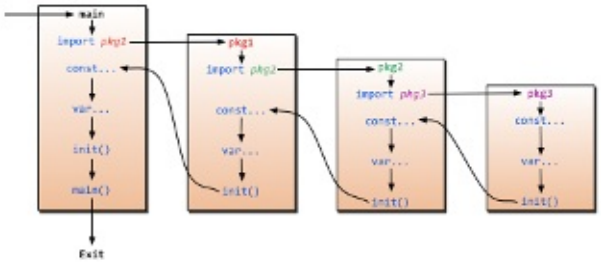
## main and init functions

Go has two retentions which are called `main` and `init`, where `init` can be used in all packages and `main` can only be used in the `main` package. These two functions are not able to have arguments or return values. Even though we can write many `init` functions in

one package, I strongly recommend writing only one `init` function for each package.

Go programs will call `init()` and `main()` automatically, so you don't need to call them by yourself. For every package, the `init` function is optional, but `package main` has one and only one `main` function.

Programs initialize and begin execution from the `main` package. If the `main` package imports other packages, they will be imported in the compile time. If one package is imported many times, it will be only compiled once. After importing packages, programs will initialize the constants and variables within the imported packages, then execute the `init` function if it exists, and so on. After all the other packages are initialized, programs will initialize constants and variables in the `main` package, then execute the `init` function inside the package if it exists. The following figure shows the process.




## import

`import` is very often used in Go programs.

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

Methods of `fmt` are called as follows.

```
fmt.Println("hello world")
```

`fmt` is from Go standard library, it is located within $GOROOT/pkg. Go supports third-party packages in two ways.

1. Relative path import "./model" // load package in the same directory, I don't recommend this way.
2. Absolute path import "shorturl/model" // load package in path "$GOPATH/pkg/shorturl/model"

There are some special operators when we import packages, and beginners are always confused by these operators.

### Dot operator

Sometimes we see people use following way to import packages.

```
import(
    . "fmt"
)

The dot operator means you can omit the package name when you call functions inside of
 that package. Now `fmt.Printf("Hello world")` becomes to `Printf("Hello world")`.
```

### Alias operation

```
It changes the name of the package that we imported when we call functions that belong
 to that package.

    import(
        f "fmt"
    )

Now `fmt.Printf("Hello world")` becomes to `f.Printf("Hello world")`.
```

### `_` operator

```
This is the operator that is difficult to understand without someone explaining it to
you.

    import (
        "database/sql"
        _ "github.com/ziutek/mymysql/godrv"
    )

The `_` operator actually means we just want to import that package and execute its `i
nit` function, and we are not sure if want to use the functions belonging to that pack
age.
```

### Links

-Previous section -Next section

# Object-oriented

Object oriented languages allow programmers to declare a function insdie the class definition. Go doesn't allow us to do that, we have to declare a method on a struct via a special syntax.

## methods

We defined a "rectangle" struct and we want to calculate its area. Normally, we would create a function, pass the struct's instance and calculate the area.

```
package main
import "fmt"

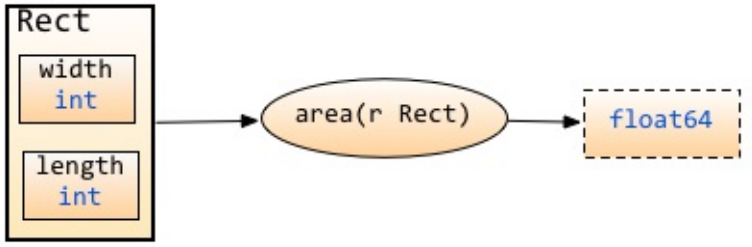
type Rectangle struct {
    width, height float64
}

func area(r Rectangle) float64 {
    return r.width*r.height
}

func main() {
    r1 := Rectangle{12, 2}
    r2 := Rectangle{9, 4}
    fmt.Println("Area of r1 is: ", area(r1))
    fmt.Println("Area of r2 is: ", area(r2))
}
```

The above example calculates a rectangle's area. The function and struct are two independent things as you may notice.

The problem arises when we want to generalize things, suppose we want to calculate area of a square, we need to define yet another function called `area`, but we can't have two functions of the same name in a file. Also `area` isn't a property of the struct `Rectangle`.



A `method` is affiliated with the type. It has the same syntax as functions do except for an additional parameter after the `func` keyword called the `receiver`, which is the main body of that method.

Using the same example, `Rectangle.area()` belongs directly to rectangle, instead of as a peripheral function. More specifically, `length`, `width` and `area()` all belong to rectangle.

As Rob Pike said.

> "A method is a function with an implicit first argument, called a receiver."

Syntax of method.

```
func (r ReceiverType) funcName(parameters) (results)
```

Let's change our example using `method` instead.

file: `code/ObjectOriented/area/area.go`

```
package main
import (
    "fmt"
    "math"
)

type Rectangle struct {
    width, height float64
}

type Circle struct {
    radius float64
}

func (r Rectangle) area() float64 {
    return r.width*r.height
}

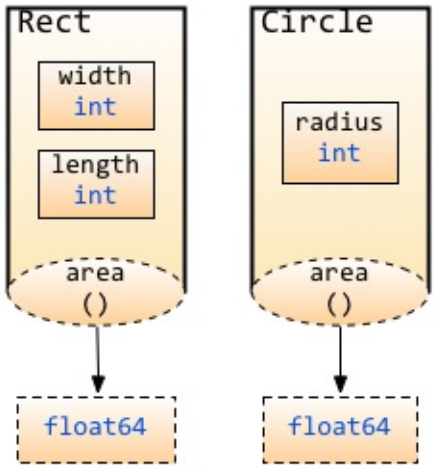
func (c Circle) area() float64 {
    return c.radius * c.radius * math.Pi
}

func main() {
    r1 := Rectangle{12, 2}
    r2 := Rectangle{9, 4}
    c1 := Circle{10}
    c2 := Circle{25}

    fmt.Println("Area of r1 is: ", r1.area())
    fmt.Println("Area of r2 is: ", r2.area())
    fmt.Println("Area of c1 is: ", c1.area())
    fmt.Println("Area of c2 is: ", c2.area())
}
```

Notes for using methods.

- If the name of methods are the same but they don't share the same receivers, they are not the same.
- Methods are able to access fields within receivers.
- Use `.` to call a method in the struct, the same way fields are called.




In the example above, the area() methods belong to both Rectangle and Circle respectively, so the receivers are Rectangle and Circle.

One thing that's worth noting is that the method with a dotted line means the receiver is passed by value, not by reference. The difference between them is that a method can change its receiver's values when the receiver is passed by reference, and it gets a copy of the receiver when the receiver is passed by value.

Any type can be the receiver of a method.

### Custom data types

Use the following format to define a custom type.

```
type typeName typeLiteral
```

Examples of customized types:

```
type ages int

type money float32

type months map[string]int

m := months {
    "January":31,
    "February":28,
    ...
    "December":31,
}
```

Similar to `typedef` in C, we use `ages` to substitute `int` in the above example.

You can use as many methods in custom types as you want.

file: `code/ObjectOriented/box/box.go`

```
package main
import "fmt"

const(
    WHITE = iota
    BLACK
    BLUE
    RED
    YELLOW
)

type Color byte

type Box struct {
    width, height, depth float64
    color Color
}

type BoxList []Box //a slice of boxes

func (b Box) Volume() float64 {
    return b.width * b.height * b.depth
}

func (b *Box) SetColor(c Color) {
    b.color = c
}

func (bl BoxList) BiggestsColor() Color {
    v := 0.00
    k := Color(WHITE)
    for _, b := range bl {
        if b.Volume() > v {
            v = b.Volume()
            k = b.color
        }
    }
    return k
}

func (bl BoxList) PaintItBlack() {
    for i, _ := range bl {
        bl[i].SetColor(BLACK)
    }
}

func (c Color) String() string {
    strings := []string {"WHITE", "BLACK", "BLUE", "RED", "YELLOW"}
    return strings[c]
}
```

```
func main() {
    boxes := BoxList {
        Box{4, 4, 4, RED},
        Box{10, 10, 1, YELLOW},
        Box{1, 1, 20, BLACK},
        Box{10, 10, 1, BLUE},
        Box{10, 30, 1, WHITE},
        Box{20, 20, 20, YELLOW},
    }

    fmt.Printf("We have %d boxes in our set\n", len(boxes))
    fmt.Println("The volume of the first one is", boxes[0].Volume(), "cm³")
    fmt.Println("The color of the last one is",boxes[len(boxes)-1].color.String())
    fmt.Println("The biggest one is", boxes.BiggestsColor().String())

    fmt.Println("Let's paint them all black")
    boxes.PaintItBlack()
    fmt.Println("The color of the second one is", boxes[1].color.String())

    fmt.Println("Obviously, now, the biggest one is", boxes.BiggestsColor().String())
}
```

We define some constants and customized types.

- Use `Color` as alias of `byte`.
- Define a struct `Box` which has fields height, width, length and color.
- Define a struct `BoxList` which has `Box` as its field.

Then we defined some methods for our customized types.

- Volume() uses Box as its receiver and returns the volume of Box.
- SetColor(c Color) changes Box's color.
- BiggestsColor() returns the color which has the biggest volume.
- PaintItBlack() sets color for all Box in BoxList to black.
- String() use Color as its receiver, returns the string format of color name.

Is it much clearer when we use words to describe our requirements? We often write our requirements before we start coding.

## Use pointer as receiver

Let's take a look at `SetColor` method. Its receiver is a pointer of Box. Yes, you can use `*Box` as a receiver. Why do we use a pointer here? Because we want to change Box's color in this method. Thus, if we don't use a pointer, it will only change the value inside a copy of Box.

If we see that a receiver is the first argument of a method, it's not hard to understand how it works.

You might be asking why we aren't using `(*b).Color=c` instead of `b.Color=c` in the SetColor() method. Either one is OK here because Go knows how to interpret the assignment. Do you think Go is more fascinating now?

You may also be asking whether we should use `(&bl[i]).SetColor(BLACK)` in `PaintItBlack` because we pass a pointer to `SetColor` . Again, either one is OK because Go knows how to interpret it!

## Inheritance of method

We learned about inheritance of fields in the last section. Similarly, we also have method inheritance in Go. If an anonymous field has methods, then the struct that contains the field will have all the methods from it as well.

file: `code/ObjectOriented/employee/employee.go`

```
package main
import "fmt"

type Human struct {
    name string
    age int
    phone string
}

type Student struct {
    Human // anonymous field
    school string
}

type Employee struct {
    Human
    company string
}

// define a method in Human
func (h *Human) SayHi() {
    fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)
}

func main() {
    mark := Student{Human{"Mark", 25, "222-222-YYYY"}, "MIT"}
    sam := Employee{Human{"Sam", 45, "111-888-XXXX"}, "Golang Inc"}

    mark.SayHi()
    sam.SayHi()
}
```

## Method overload

If we want Employee to have its own method `SayHi`, we can define a method that has the same name in Employee, and it will hide `SayHi` in Human when we call it.

file: `code/ObjectOriented/human/human.go`

```
package main
import "fmt"

type Human struct {
    name string
    age int
    phone string
}

type Student struct {
    Human
    school string
}

type Employee struct {
    Human
    company string
}

func (h *Human) SayHi() {
    fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)
}

func (e *Employee) SayHi() {
    fmt.Printf("Hi, I am %s, I work at %s. Call me on %s\n", e.name,
        e.company, e.phone) //Yes you can split into 2 lines here.
}

func main() {
    mark := Student{Human{"Mark", 25, "222-222-YYYY"}, "MIT"}
    sam := Employee{Human{"Sam", 45, "111-888-XXXX"}, "Golang Inc"}

    mark.SayHi()
    sam.SayHi()
}
```

Methods use rule of capital letter to decide whether public or private as well.

### Links

-Previous section -Next section

# Interface

# Interface

One of the subtlest design features in Go are interfaces. After reading this section, you will likely be impressed by their implementation.

## What is an interface?

In short, an interface is a set of methods that we use to define a set of actions.

Like the examples in previous sections, both Student and Employee can `SayHi()`, but they don't do the same thing.

Let's do some more work. We'll add one more method `Sing()` to them, along with the `BorrowMoney()` method to Student and the `SpendSalary()` method to Employee.

Now, Student has three methods called `SayHi()`, `Sing()` and `BorrowMoney()`, and Employee has `SayHi()`, `Sing()` and `SpendSalary()`.

This combination of methods is called an interface and is implemented by both Student and Employee. So, Student and Employee implement the interface: `SayHi()` and `Sing()`. At the same time, Employee doesn't implement the interface: `SayHi()`, `Sing()`, `BorrowMoney()`, and Student doesn't implement the interface: `SayHi()`, `Sing()`, `SpendSalary()`. This is because Employee doesn't have the method `BorrowMoney()` and Student doesn't have the method `SpendSalary()`.

## Types of Interface

An interface defines a set of methods, so if a type implements all the methods we say that it implements the interface.

```
type Human struct {
    name  string
    age   int
    phone string
}

type Student struct {
    Human
    school string
    loan   float32
```

```
}

type Employee struct {
    Human
    company string
    money   float32
}

func (h *Human) SayHi() {
    fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)
}

func (h *Human) Sing(lyrics string) {
    fmt.Println("La la, la la la, la la la la la...", lyrics)
}

func (h *Human) Guzzle(beerStein string) {
    fmt.Println("Guzzle Guzzle Guzzle...", beerStein)
}

// Employee overloads Sayhi
func (e *Employee) SayHi() {
    fmt.Printf("Hi, I am %s, I work at %s. Call me on %s\n", e.name,
        e.company, e.phone) //Yes you can split into 2 lines here.
}

func (s *Student) BorrowMoney(amount float32) {
    s.loan += amount // (again and again and...)
}

func (e *Employee) SpendSalary(amount float32) {
    e.money -= amount // More vodka please!!! Get me through the day!
}

// define interface
type Men interface {
    SayHi()
    Sing(lyrics string)
    Guzzle(beerStein string)
}

type YoungChap interface {
    SayHi()
    Sing(song string)
    BorrowMoney(amount float32)
}

type ElderlyGent interface {
    SayHi()
    Sing(song string)
    SpendSalary(amount float32)
}
```

We know that an interface can be implemented by any type, and one type can implement many interfaces simultaneously.

Note that any type implements the empty interface `interface{}` because it doesn't have any methods and all types have zero methods by default.

## Value of interface

So what kind of values can be put in the interface? If we define a variable as a type interface, any type that implements the interface can assigned to this variable.

Like the above example, if we define a variable "m" as interface Men, then any one of Student, Human or Employee can be assigned to "m". So we could have a slice of Men, and any type that implements interface Men can assign to this slice. Be aware however that the slice of interface doesn't have the same behavior as a slice of other types.

file: `code/Interface/InterfaceValue/value.go`

```
package main

import "fmt"

type Human struct {
    name  string
    age   int
    phone string
}

type Student struct {
    Human
    school string
    loan   float32
}

type Employee struct {
    Human
    company string
    money   float32
}

func (h Human) SayHi() {
    fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)
}

func (h Human) Sing(lyrics string) {
    fmt.Println("La la la la...", lyrics)
}

func (e Employee) SayHi() {
```

```
    fmt.Printf("Hi, I am %s, I work at %s. Call me on %s\n", e.name,
        e.company, e.phone) //Yes you can split into 2 lines here.
}

// Interface Men implemented by Human, Student and Employee
type Men interface {
    SayHi()
    Sing(lyrics string)
}

func main() {
    mike := Student{Human{"Mike", 25, "222-222-XXX"}, "MIT", 0.00}
    paul := Student{Human{"Paul", 26, "111-222-XXX"}, "Harvard", 100}
    sam := Employee{Human{"Sam", 36, "444-222-XXX"}, "Golang Inc.", 1000}
    tom := Employee{Human{"Sam", 36, "444-222-XXX"}, "Things Ltd.", 5000}

    // define interface i
    var i Men

    //i can store Student
    i = mike
    fmt.Println("This is Mike, a Student:")
    i.SayHi()
    i.Sing("November rain")

    //i can store Employee
    i = tom
    fmt.Println("This is Tom, an Employee:")
    i.SayHi()
    i.Sing("Born to be wild")

    // slice of Men
    fmt.Println("Let's use a slice of Men and see what happens")
    x := make([]Men, 3)
    // these three elements are different types but they all implemented interface Men
    x[0], x[1], x[2] = paul, sam, mike

    for _, value := range x {
        value.SayHi()
    }
}
```

An interface is a set of abstract methods, and can be implemented by non-interface types. It cannot therefore implement itself.

## Empty interface

An empty interface is an interface that doesn't contain any methods, so all types implement an empty interface. This fact is very useful when we want to store all types at some point, and is similar to void* in C.

```
// define a as empty interface
var a interface{}
var i int = 5
s := "Hello world"
// a can store value of any type
a = i
a = s
```

If a function uses an empty interface as its argument type, it can accept any type; if a function uses empty interface as its return value type, it can return any type.

## Method arguments of an interface

Any variable can be used in an interface. So how can we use this feature to pass any type of variable to a function?

For example we use fmt.Println a lot, but have you ever noticed that it can accept any type of argument? Looking at the open source code of fmt, we see the following definition.

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

This means any type that implements interface Stringer can be passed to fmt.Println as an argument. Let's prove it.

file: `code/Interface/Stringer/stringer.go`

```
package main

import (
    "fmt"
    "strconv"
)

type Human struct {
    name  string
    age   int
    phone string
}

// Human implemented fmt.Stringer
func (h Human) String() string {
    return "Name:" + h.name + ", Age:" + strconv.Itoa(h.age) + " years, Contact:" + h.
phone
}

func main() {
    Bob := Human{"Bob", 39, "000-7777-XXX"}
    fmt.Println("This Human is : ", Bob)
}
```

Looking back to the example of Box, you will find that Color implements interface Stringer as well, so we are able to customize the print format. If we don't implement this interface, fmt.Println prints the type with its default format.

```
fmt.Println("The biggest one is", boxes.BiggestsColor().String())
fmt.Println("The biggest one is", boxes.BiggestsColor())
```

Attention: If the type implemented the interface `error`, fmt will call `error()`, so you don't have to implement Stringer at this point.

## Type of variable in an interface

If a variable is the type that implements an interface, we know that any other type that implements the same interface can be assigned to this variable. The question is how can we know the specific type stored in the interface. There are two ways which I will show you.

- Assertion of Comma-ok pattern

Go has the syntax `value, ok := element.(T)`. This checks to see if the variable is the type that we expect, where "value" is the value of the variable, "ok" is a variable of boolean type, "element" is the interface variable and the T is the type of assertion.

If the element is the type that we expect, ok will be true, false otherwise.

Let's use an example to see more clearly.

file: `code/Interface/Person/person.go`

```
package main

import (
    "fmt"
    "strconv"
)

type Element interface{}
type List []Element

type Person struct {
    name string
    age  int
}

func (p Person) String() string {
    return "(name: " + p.name + " - age:     " + strconv.Itoa(p.age) + " years)"
}

func main() {
    list := make(List, 3)
    list[0] = 1       // an int
    list[1] = "Hello" // a string
    list[2] = Person{"Dennis", 70}

    for index, element := range list {
        if value, ok := element.(int); ok {
            fmt.Printf("list[%d] is an int and its value is %d\n", index, value)
        } else if value, ok := element.(string); ok {
            fmt.Printf("list[%d] is a string and its value is %s\n", index, value)
        } else if value, ok := element.(Person); ok {
            fmt.Printf("list[%d] is a Person and its value is %s\n", index, value)
        } else {
            fmt.Printf("list[%d] is of a different type\n", index)
        }
    }
}
```

It's quite easy to use this pattern, but if we have many types to test, we'd better use `switch`.

- switch test

Let's use `switch` to rewrite the above example.

file: `code/Interface/switch/switch.go`

```
package main

import (
    "fmt"
    "strconv"
)

type Element interface{}
type List []Element

type Person struct {
    name string
    age  int
}

func (p Person) String() string {
    return "(name: " + p.name + " - age: " + strconv.Itoa(p.age) + " years)"
}

func main() {
    list := make(List, 3)
    list[0] = 1       //an int
    list[1] = "Hello" //a string
    list[2] = Person{"Dennis", 70}

    for index, element := range list {
        switch value := element.(type) {
        case int:
            fmt.Printf("list[%d] is an int and its value is %d\n", index, value)
        case string:
            fmt.Printf("list[%d] is a string and its value is %s\n", index, value)
        case Person:
            fmt.Printf("list[%d] is a Person and its value is %s\n", index, value)
        default:
            fmt.Println("list[%d] is of a different type", index)
        }
    }
}
```

One thing you should remember is that `element.(type)` cannot be used outside of the `switch` body, which means in that case you have to use the `comma-ok` pattern .

## Embedded interfaces

The most beautiful thing is that Go has a lot of built-in logic syntax, such as anonymous fields in struct. Not suprisingly, we can use interfaces as anonymous fields as well, but we call them `Embedded interfaces` . Here, we follow the same rules as anonymous fields. More

specifically, if an interface has another interface embedded within it, it will behave as if it has all the methods that the embedded interface has.

We can see that the source file in `container/heap` has the following definition:

```
type Interface interface {
        sort.Interface // embedded sort.Interface
        Push(x interface{}) //a Push method to push elements into the heap
        Pop() interface{} //a Pop method that pops elements from the heap
}
```

We see that `sort.Interface` is an embedded interface, so the above Interface has the three methods contained within the `sort.Interface` implicitly.

```
type Interface interface {
        // Len is the number of elements in the collection.
        Len() int
        // Less returns whether the element with index i should sort
        // before the element with index j.
        Less(i, j int) bool
        // Swap swaps the elements with indexes i and j.
        Swap(i, j int)
}
```

Another example is the `io.ReadWriter` in package `io`.

```
// io.ReadWriter
type ReadWriter interface {
        Reader
        Writer
}
```

## Reflection

Reflection in Go is used for determining information at runtime. We use the `reflect` package, and this official article explains how reflect works in Go.

There are three steps involved when using reflect. First, we need to convert an interface to reflect types (reflect.Type or reflect.Value, this depends on the situation).

```
t := reflect.TypeOf(i)    // get meta-data in type i, and use t to get all elements
v := reflect.ValueOf(i)   // get actual value in type i, and use v to change its value
```

After that, we can convert the reflected types to get the values that we need.

```
var x float64 = 3.4
v := reflect.ValueOf(x)
fmt.Println("type:", v.Type())
fmt.Println("kind is float64:", v.Kind() == reflect.Float64)
fmt.Println("value:", v.Float())
```

Finally, if we want to change the values of the reflected types, we need to make it modifiable. As discussed earlier, there is a difference between pass by value and pass by reference. The following code will not compile.

```
var x float64 = 3.4
v := reflect.ValueOf(x)
v.SetFloat(7.1)
```

Instead, we must use the following code to change the values from reflect types.

```
var x float64 = 3.4
p := reflect.ValueOf(&x)
v := p.Elem()
v.SetFloat(7.1)
```

**Links**

-Previous section -Next section

# Concurrency

## goroutine

goroutines and concurrency are built into the core design of Go. They're similar to threads but work differently. More than a dozen goroutines maybe only have 5 or 6 underlying threads. Go also gives you full support to sharing memory in your goroutines. One goroutine usually uses 4~5 KB of stack memory. Therefore, it's not hard to run thousands of goroutines on a single computer. A goroutine is more lightweight, more efficient and more convenient than system threads.

goroutines run on the thread manager at runtime in Go. We use the `go` keyword to create a new goroutine, which is a function at the underlying level ( ***main() is a goroutine*** ).

```
go hello(a, b, c)
```

Let's see an example.

```
package main

import (
    "fmt"
    "runtime"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        runtime.Gosched()
        fmt.Println(s)
    }
}

func main() {
    go say("world") // create a new goroutine
    say("hello") // current goroutine
}
```

Output：

```
hello
world
hello
world
hello
world
hello
world
hello
```

We see that it's very easy to use concurrency in Go by using the keyword `go`. In the above example, these two goroutines share some memory, but we would better off following the design recipe: Don't use shared data to communicate, use communication to share data.

runtime.Gosched() means let the CPU execute other goroutines, and come back at some point.

The scheduler only uses one thread to run all goroutines, which means it only implements concurrency. If you want to use more CPU cores in order to take advantage of parallel processing, you have to call runtime.GOMAXPROCS(n) to set the number of cores you want to use. If `n<1`, it changes nothing. This function may be removed in the future, see more details about parallel processing and concurrency in this article.

## channels

goroutines run in the same memory address space, so you have to maintain synchronization when you want to access shared memory. How do you communicate between different goroutines? Go uses a very good communication mechanism called `channel`. `channel` is like a two-way pipeline in Unix shells: use `channel` to send or receive data. The only data type that can be used in channels is the type `channel` and the keyword `chan`. Be aware that you have to use `make` to create a new `channel`.

```
ci := make(chan int)
cs := make(chan string)
cf := make(chan interface{})
```

channel uses the operator `<-` to send or receive data.

```
ch <- v    // send v to channel ch.
v := <-ch  // receive data from ch, and assign to v
```

Let's see more examples.

```
package main

import "fmt"

func sum(a []int, c chan int) {
    total := 0
    for _, v := range a {
    total += v
    }
    c <- total  // send total to c
}

func main() {
    a := []int{7, 2, 8, -9, 4, 0}

    c := make(chan int)
    go sum(a[:len(a)/2], c)
    go sum(a[len(a)/2:], c)
    x, y := <-c, <-c  // receive from c

    fmt.Println(x, y, x + y)
}
```

Sending and receiving data in channels blocks by default, so it's much easier to use synchronous goroutines. What I mean by block is that a goroutine will not continue when receiving data from an empty channel, i.e ( `value := <-ch` ), until other goroutines send data to this channel. On the other hand, the goroutine will not continue until the data it sends to a channel, i.e ( `ch<-5` ), is received.

## Buffered channels

I introduced non-buffered channels above. Go also has buffered channels that can store more than a single element. For example, `ch := make(chan bool, 4)` , here we create a channel that can store 4 boolean elements. So in this channel, we are able to send 4 elements into it without blocking, but the goroutine will be blocked when you try to send a fifth element and no goroutine receives it.

```
ch := make(chan type, n)

n == 0 ! non-buffer（block）
n > 0 ! buffer（non-block until n elements in the channel）
```

You can try the following code on your computer and change some values.

```
package main

import "fmt"

func main() {
    c := make(chan int, 2)  // change 2 to 1 will have runtime error, but 3 is fine
    c <- 1
    c <- 2
    fmt.Println(<-c)
    fmt.Println(<-c)
}
```

## Range and Close

We can use range to operate on buffer channels as in slice and map.

```
package main

import (
    "fmt"
)

func fibonacci(n int, c chan int) {
    x, y := 1, 1
    for i := 0; i < n; i++ {
        c <- x
        x, y = y, x + y
    }
    close(c)
}

func main() {
    c := make(chan int, 10)
    go fibonacci(cap(c), c)
    for i := range c {
        fmt.Println(i)
    }
}
```

`for i := range c` will not stop reading data from channel until the channel is closed. We use the keyword `close` to close the channel in above example. It's impossible to send or receive data on a closed channel; you can use `v, ok := <-ch` to test if a channel is closed. If `ok` returns false, it means the there is no data in that channel and it was closed.

Remember to always close channels in producers and not in consumers, or it's very easy to get into panic status.

Another thing you need to remember is that channels are not like files. You don't have to close them frequently unless you are sure the channel is completely useless, or you want to exit range loops.

## Select

In the above examples, we only use one channel, but how can we deal with more than one channel? Go has a keyword called `select` to listen to many channels.

`select` is blocking by default and it continues to execute only when one of channels has data to send or receive. If several channels are ready to use at the same time, select chooses which to execute randomly.

```
package main

import "fmt"

func fibonacci(c, quit chan int) {
    x, y := 1, 1
    for {
        select {
        case c <- x:
            x, y = y, x + y
        case <-quit:
        fmt.Println("quit")
            return
        }
    }
}

func main() {
    c := make(chan int)
    quit := make(chan int)
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(<-c)
        }
        quit <- 0
    }()
    fibonacci(c, quit)
}
```

`select` has a `default` case as well, just like `switch`. When all the channels are not ready for use, it executes the default case (it doesn't wait for the channel anymore).

```
select {
case i := <-c:
    // use i
default:
    // executes here when c is blocked
}
```

## Timeout

Sometimes a goroutine becomes blocked. How can we avoid this to prevent the whole program from blocking? It's simple, we can set a timeout in the select.

```
func main() {
    c := make(chan int)
    o := make(chan bool)
    go func() {
        for {
            select {
                case v := <- c:
                       println(v)
                case <- time.After(5 * time.Second):
                    println("timeout")
                    o <- true
                    break
            }
        }
    }()
    <- o
}
```

## Runtime goroutine

The package `runtime` has some functions for dealing with goroutines.

- `runtime.Goexit()`

  Exits the current goroutine, but defered functions will be executed as usual.

- `runtime.Gosched()`

  Lets the scheduler execute other goroutines and comes back at some point.

- `runtime.NumCPU() int`

  Returns the number of CPU cores

- `runtime.NumGoroutine() int`

Returns the number of goroutines

- `runtime.GOMAXPROCS(n int) int`

  Sets how many CPU cores you want to use

**Links**

-Previous section -Next section

# Managing the Workspace

## `$GOPATH` and `$GOROOT`

`$GOROOT` : This is an environment variable which stores the path where your Go installation is present if you have customized it. `$GOPATH` : The Go universe for your machine. The idea is that all your Go code should reside in a directory tree so the code isn't lying around in random places.

My `$GOPATH` is `/usr/home/suraj/Go` , it has the following structure

- pkg: The libraries which'll be imported in our code, there is a `.a` file created at the respective path.
- bin: The binary file of our project will be installed here on calling `go install` in the project folder
- src: Will hold the actual code, if you have a github account, then you can create a folder tree inside like

go

- src
  - github.com
    - thewhitetulip
      - wshare
      - picsort
  - golang.net
  - sourcegraph.com
- bin (binaries)
  - wshare
  - picsort
- pkg

Go takes this unique approach so that all the Go code is well organized and not thrown in a haphazard manner. This makes it easy to locate code for humans and software alike.

# Packages

Packages are one or more file(s) which contains Go source code. They can be named anything. Each package except the main needs to be a in a distinct folder on the `GOPATH` . Each Go program starts with the package declaration. As a convention, package names are

short and self documenting. The Format package is called `fmt` since it is short and concise that way.

There can be any number of files in a package directory, but only *one* file with the *main function*. While building the code, the compiler starts with the main function in the main package.

## Package naming

```
import "views"
```

When the Go compiler executes this statement, it'll try to find the library called "views" in $GOROOT/views or $GOPATH/src/views. It'll complain if it doesn't find the library in either of the two paths.

The actual package name is just "views", but for evaluating it, it's the absolute path of the package from the `$GOPATH/src` directory.

For the Tasks app, it resides in `$GOPATH/src/github/thewhitetulip/Tasks`, thus, my packages will lie within the Tasks folder. This makes it easy for distributing the libraries.

> Note: Bad Workaround.
>
> There is a workaround for this by treating `$GOPATH/src` itself as your entire project and having libraries directly inside it. This is not the Go way of doing things. Please do not do it.

# Internal deployment

We'll follow the standard practice and put our code in `$GOPATH/src/github.com/thewhitetulip/Tasks` folder. While testing our app, we need a deployment version. I used `Tasks` while building Tasks, I have made a folder `~/Tasks` and here I keep the deployment version of Tasks, which contains the binary version and the static files. Every new build in the `src` gets pushed in this folder.

## Running a server

Webapps are deployed on IP addresses & ports. It is possible to run multiple websites on a single machine on different ports. There are an awfully large number of ports on each computer. Typically we choose a large port number like 8000 so it doesn't affect some other applications. Certain ports are restricted and need sudo access, like 80.

The IP address we bind the server to can be a public or private. Public means using ":8080" or "0.0.0.0:8080". Private means "127.0.0.1:8080"

- Public IP means that any computer in the same network can access the web app.
- Private means only your computer will be able to access it.

```
//Public
log.Fatal(http.ListenAndServe(":8080", nil))
//Private
log.Fatal(http.ListenAndServe("127.0.0.1:8080", nil))
```

> Note: 127.0.0.1
>
> `127.0.0.1`, called localhost, is a special IP address which is given to every machine to refer to itself. It is called as loopback address. It doesn't matter if you are connected to the network or not, your machine will always have this IP address, so if you are want privacy use `127.0.0.1`.

If you do want your web application to be accessible via the network, use `0.0.0.0:8080` or just `:8080`, when you give just the port number, the Go language assumes the IP address the machine is '0.0.0.0' and **not** `127.0.0.1`.

## MVC Pattern

When I started learning building web apps in Django, I first read about the MVC pattern, the models, the views and what not. But that knowledge was to be *assumed* by me and I had no idea why that decision was made, along with that Django works like magic, one has to adapt to it's quirks. Hence while programming in Go, it'll be immensely benefitial to grow up to the MVC pattern by starting out small. The first app should entirely reside in the main.go file, then slowly, as the app grows, it needs to be structured well, so the all the handlers go into a views package, templates go in a templates folder, database related entities go in another package.

Learning by doing and experimenting is the best way. Challenge the status quo.

#### Links

-Previous section -Next section

# Web Programming Basics

Web servers are programs which get a request for a URL and they respond back with it's contents. The request-response is in the HTTP protocol. The Go language has http support in it's standard library as `net/http`.

HTTP was built for transferring plain text, later it allowed multimedia content too. The HTTP2 protocol, which is the successor of HTTP1, is binary.

When we type www.github.com on our browser's address bar, the following things happen:

1. Browser adds either `https://` or `http://` and a trailing forward slash.
2. Our request becomes HTTP GET / github.com.
3. Browser sends out the HTTP GET request to the IP address of the github.com. (Called DNS resolution)
4. Github's servers will process the request and send back a response to our IP address.
5. Our browser will render the page sent in the HTTP Response.

The HTTP response contains a status code.

The status code informs the client of the status of the HTTP server's response.

Following are the status codes defined in HTTP/1.1:

- 1xx Informational
- 2xx Success
- 3xx Redirection
- 4xx Client Error
- 5xx Server Error

For those who are writing web applications, this might be a big deal, to write servers. But it isn't. At the bottom of it, a server is nothing different from our normal programs, it takes an input, does something about it in the database mostly and responds back with a response. The response might be empty, depending on if the request was valid or invalid.

The browser just abstracts the users from sending and receiving the HTTP request/responses. We can totally do what the browser does manually, build the HTTP request and send it to the server. Receive the response and render it into HTML. There are always two sides to any web app, the browser side and the API side. Browser access is done via the browser and API access is done programatically. Both the ways use HTTP for request-response, but using the API, we can automate a lot of things.

# HTTP Methods

HTTP request-response is done via HTTP methods.

List of HTTP methods: GET, POST, DELETE, PUT.

GET : Used to retrieve the URL, `GET /` will get the home page. POST: Used to create data stored on the URL. PUT: Used to update data on the URL. DELETE: Used to delete data in the URL.

```
// Create a new category.
// POST /categories

// Update an existing category.
// PUT /categories/12

// View the details of a category.
// GET /categories/12

// Delete an existing category.
// DELETE /categories/12
```

Think of categories as a document, POST to create it, GET to fetch it, PUT to update it, and DELETE to delete it. For deleting a task, rather than sending GET `/delete/1234`, we should send a DELETE `/tasks/1234`.

The difference isn't just in the symantics, absolutely anything can send a GET request. GET requires no validation, nothing. If I write a Go program to send GET /delete/ where id goes from 0 till a very large number, I can delete all the tasks in our application. Of course, we can also send a DELETE request a million times. GET is the default way to 'get' data from the server, it should not be used as a way of doing things.

## GET vs POST

Apart from their functional differences, GET and POST differ in security perspectives. Both are insecure.

GET transfers data via the URL. POST sends data in the request's body or payload, but that isn't hidden or encrypted by default, but it isn't visible on the URL, it is easily accessible to anyone who knows how to read a HTTP request.

Security is something you build your application around. There isn't much difference between GET and POST when we consider security, both transfer data in plain text GET is just relatively a little less secure since URLs are logged by a proxy server/firewall/browser history and that GET requests can be done by the browser on behalf of the user without

confirmation. Bots are common over the internet, bots can visit randomly to every link present in your application, but they don't send random data to any Form you have, or if so, very few bots can do that.

For protecting data of the webapp, one has to stick to using HTTPS and sanitize any data that comes from the user.

**Example**

A blog consists of a collection of posts, a post has tags, is written by some author, at some time and has some primary key to uniquely identify it in our database and it has a slug which means the URL.

This is the era of semantic web, thus the new beautiful URLs like, `surajblog.com/posts/welcome-the-new-year`, the slug is the `welcome-the-new-year`.

When the server gets a HTTP GET request of `/posts/welcome-the-new-year`, it'll search for URL handlers starting with the list of URL handlers we have given, then it'll find the closest match, in our case it'll be `/post/`, then it'll call the handler of this URL.

Our `/` root URL should be at the very bottom of our list. Because while executing, checks are done from top to bottom.

```
//sample handler definition
http.HandleFunc("/post/", ShowPostBySlug)
http.HandleFunc("/", ShowAllPosts)
```

Handler talk to the database, fetch the data and render templates which show up as HTML pages in our browser.

# What is a template?

Templates are a way to present data to the user. The server populated the templates and sends the HTML page back to the browser. For a blog, it doesn't make sense to use a separate html page for each post. This is why there is a post template and the server will get all the details like content, title, date published and populate the post template and return it back to the browser.

A web application is basically a way of representing data stored in the database to the end user using HTTP.

Writing a web application:

1. Fix the database structure.

2. Understand how data flows and decide the URLs.
3. Write templates to corresponding to each URL set.
4. Write functions in Go to handle each URL pattern, called `handlers`.
5. Handlers fetch data from the database and populate data in the templates.

#### Not abusing templates

The logic behind creating templates was to not to repeat HTML code. Templates support variables which our handlers are going to use. The standard way is to handle the business login inside the handler and use templates just for rendering the data. Templates are to be used only for the presentation logic *not* the business logic. It becomes difficult to maintain applications which have business logic inside the template. It should never be done and is a very bad programming practice.

## Example:

We are going to build a todo list manager in this book. It support multiple users.

Wrong way: Fetch all tasks in the template and only show those of the current user. i.e. filter the tasks in the template Correct way: Fetch only the tasks belonging to the current user. i.e. filter the tasks in the handler.

Functionality of our *EditTask* URL which is `/edit/<id>`.

file `views/addViews.go`

```
//EditTaskFunc is our handler which will handle the /edit/<id> URL
func EditTaskFunc(w http.ResponseWriter, r *http.Request) {
        //Code
        task := db.GetTaskByID(id)
        editTemplate.Execute(w, task)
        //Code
}
```

file `db/tasks.go`

```
func GetTaskByID(id int) types.Context {
        //Code to fetch tasks of the current user
        context := types.Context{Tasks: tasks}
        return context
}
```

The EditTaskFunc talks to the database with the GetTaskByID function and fetches the tasks for the current user and populates the editTemplate.

Thus we can split an application into views, database, templates and controller(main package).

# Static Files

Static files are the CSS/JS/Images which we load into our html templates.

The URL which responds to static files will be `/static/` .

Execution:

1. We get a request like `/static/<filepath>`
2. We go to the public directory of our application and look for
3. If we get a file of that path then we serve the file, othewise send a 404 error.

The `public` folder contains all your static files. We will have a templates folder on the same folder where the public is present.

The reason templates is a separate folder is that it is a separate entity and shouldn't be publicly available using the /static/ URL.

```
public
|   |-- static
|   |   |-- css
|   |   |   `-- styles.css
                ..and more
|   |   `-- js
|   |       |-- bootstrap.min.js
|   |       .... and more
templates
|   |-- completed.html
|   |   ...and more
```

**Note Output**

The above output is of the `tree` program.

**Links**

-Previous section -Next section

# Basic web application

Make a folder in your `$GOPATH/src/github.com/<yourname>/Tasks` substitute your username in lieu of `<yourname>`.

Create a file main.go

file `main.go`

```
package main

import (
    "log"
    "net/http"
)

func main() {
    PORT := "127.0.0.1:8080"
    log.Fatal(http.ListenAndServe(PORT, nil))
}
```

We import the http package in our application with `import net/http`.

Now go to your terminal and type

```
[Tasks] $ go run main.go
```

You will notice that the program doesn't print anything because we told it only to listen on the port. If we want to user to know that we are running a server on that port, we should print a message saying so, as shown in the below example.

code example file: `3.1basicServer.go`

```
package main

import (
    "log"
    "net/http"
)

func main() {
    PORT := ":8080"
    log.Print("Running server on "+ PORT)
    log.Fatal(http.ListenAndServe(PORT, nil))
}
```

Whenever we run this, we'll get the below output

```
2016/01/01 22:00:36 Running server on :8080
```

Open `localhost:8080` in a browser and you'll get the message "404 page not found"

We have just started a server to listen on the port 8080. We haven't *handled* the URL, i.e. we do not have a mechanism to respond to the GET / request which the browser is going to send when we hit the localhost:8080 URL.

# Handling URLs

```
http.HandleFunc("/complete/", ShowCompleteTasksFunc)

//ShowCompleteTasksFunc is used to populate the "/completed/" URL
func ShowCompleteTasksFunc(w http.ResponseWriter, r *http.Request) {
}
```

We use HandleFunc in the `net/http` package to handle URLs. The first argument is the URL to be handled and the second parameter is the function. We can define functions in the second parameter, but you are advised against it.

The handler function requires two arguments, a ResponseWriter object and a Request object. We are going to write to the ResponseWriter depending on what we get in the Request object.

## Handler Example

We want to write the URL that the user is visiting on our webapp. The URL can be found in the request object, `r.URL.Path`.

```
    package main

    import (
        "log"
        "net/http"
    )

    func main() {
        PORT := ":8080"
        log.Print("Running server on "+ PORT)
        http.HandleFunc("/",CompleteTaskFunc)
        log.Fatal(http.ListenAndServe(PORT, nil))
    }

    func CompleteTaskFunc(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte(r.URL.Path))
    }
```

## Parameterized routing

When we get the URL /tasks/124, we want to get the task number 124, we do the following

```
//GetTaskFunc is used to delete a task,
func GetTaskFunc(w http.ResponseWriter, r *http.Request) {
    if r.Method == "GET" {
        id := r.URL.Path[len("/tasks/"):]
        w.Write([]byte("Get the task "+id))
}
```

We take a sub string of the URL and remove the `/delete/` part and we have the ID of the task.

This example makes use of the slicing concept. It is simple, if `Path` is our string variable then Path[1:] is the substring which includes everything from the first character, index being zero.

#### Note: Parameterized routing

Ideally we should not be checking the URL path inside our view, we are supposed to use a router. For real projects, you should use a router. Here we are learning, so we won't be using a router.

# Serving static files

```
    http.Handle("/static/", http.FileServer(http.Dir("public")))
```

For serving static files, we use the `FileServer` method of the http package. It takes a folder as an argument. Make sure you give only the public folder path in the argument.

# Homework

- Read the documentation of `net/http` & `log` and get to know of the methods/functions in the packages.
- Find out how many alternatives are there to ListenAndServe.
- Write a handler to serve static files, if file is available on that path then it should serve the file, otherwise it must return an HTTP404 error.
- Write a command line application to share files/folder over HTTP, the syntax should be like this `./wshare -f file.pdf` file on link = 192.168.2.1:8080. An advanced version of this can be a dropbox clone, where we have a UI to share files. If you build this, do email me!

## Footnotes

Learn how to read documentation, it saves a lot of time.

#### Links

-Previous section -Next section

# Designing our web app

The design phase is the most important in any software project as one small mistake in designing costs the project serverly. It becomes very costly to fix that later. In most projects, there are a few solutions applied to mistakes in design, they are just glued together until it becomes impossible to go forward. The term is called *technical debt*.

Our webapp is going to be a todo list manager for multiple users. It'll have the following features:

```
    1. Ability to add/delete/modify task.
    2. Ability to mark tasks as complete/incomplete.
    3. Display the tasks as a one column list.
    4. Ability to switch modes between pending/completed/deleted tasks.
    5. Display notifications like note added/deleted/archived.
    6. Search for text and highlight the text in the search results page.
    1. Login/Sign up.
```

# The Design

Translating our design to API, we get the following.

```
/add/           POST = add new task
/               GET     = show pending tasks
/complete/      GET  = show completed tasks
/deleted/       GET  = show deleted tasks
/edit/<id>      POST = edit post
/edit/<id>      GET  = show the edit page
/trash/<id>     POST = trash post to recycle bin
/delete/<id>    POST = permanently delete post
/complete/<id>  POST = mark post as complete
/login/         POST = do the login
/login/         GET  = show login page
/logout/        POST = log the user out
/restore/<id>   POST = restore that task
/update/<id>    POST = update task
/change/        GET  = will allow changing password
/register/      GET  = show the register page
/register/      POST = will add entries into database
```

file `~/main/main.go`

```
package main
import (
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/complete/", CompleteTaskFunc)
    http.HandleFunc("/delete/", DeleteTaskFunc)
    http.HandleFunc("/deleted/", ShowTrashTaskFunc)
    http.HandleFunc("/trash/", TrashTaskFunc)
    http.HandleFunc("/edit/", EditTaskFunc)
    http.HandleFunc("/completed/", ShowCompleteTasksFunc)
    http.HandleFunc("/restore/", RestoreTaskFunc)
    http.HandleFunc("/add/", AddTaskFunc)
    http.HandleFunc("/update/", UpdateTaskFunc)
    http.HandleFunc("/search/", SearchTaskFunc)
    http.HandleFunc("/login", GetLogin)
    http.HandleFunc("/register", PostRegister)
    http.HandleFunc("/admin", HandleAdmin)
    http.HandleFunc("/add_user", PostAddUser)
    http.HandleFunc("/change", PostChange)
    http.HandleFunc("/logout", HandleLogout)
    http.HandleFunc("/", ShowAllTasksFunc)

    http.Handle("/static/", http.FileServer(http.Dir("public")))
    log.Print("running on port 8080")
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Create all functions we mentioned above and make the necessary changes as per one definition that we show below in this file.

```
func ShowAllTasksFunc(w http.ResponseWriter, r *http.Request) {
    var message string
    if r.Method == "GET" {
        message := "all pending tasks GET"
    } else {
        message := "all pending tasks POST"
    }
    w.Write([]byte(message))
}
```

After you create these functions run the server as below,

```
[Tasks] $ go build
[Tasks] $ ./Tasks
```

In your browser type `localhost:8080` and type all these URLs and see what message you get.

# Homework

Check the documentation for `http.ResponseWriter` and `http.Request` objects and get to know all the variables/functions/constants for http package and these two which we mentioned.

**Links**

-Previous section -Next section

# Using databases in Go

Go doesn't provide out of the box support for any database, but it provides an interface, which can be used by database library creators to keep all the database libraries compatible with each other.

We will use sqlite for this book.

# Creating and configuring database

Before we can build the front end, we need the backend ready. Below is the DDL and the DML which will fill our database with dummy data.

Use the following insert statements to enter data in our table, so we'll begin reading data in our ShowAllTasks function which we wrote in the previous chapter

```
  --user

PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE user (
    id integer primary key autoincrement,
    username varchar(100),
    password varchar(1000),
    email varchar(100)
);
INSERT INTO "user" VALUES(1,'suraj','suraj','sapatil@live.com');


--category
CREATE TABLE category(
    id integer primary key autoincrement,
    name varchar(1000) not null,
    user_id references user(id)
);

INSERT INTO "category" VALUES(1,'TaskApp',1);

--status
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE status (
    id integer primary key autoincrement,
    status varchar(50) not null
);
```

```
INSERT INTO "status" VALUES(1,'COMPLETE');
INSERT INTO "status" VALUES(2,'PENDING');
INSERT INTO "status" VALUES(3,'DELETED');
COMMIT;

--task

PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE task (
    id integer primary key autoincrement,
    title varchar(100),
    content text,
    created_date timestamp,
    last_modified_at timestamp,
    finish_date timestamp,
    priority integer,
    cat_id references category(id),
    task_status_id references status(id),
    due_date timestamp,
    user_id references user(id),
    hide int
);

INSERT INTO "task" VALUES(1,'Publish on github','Publish the source of tasks and picso
rt on github','2015-11-12 15:30:59','2015-11-21 14:19:22','2015-11-17 17:02:18',3,1,1,
NULL,1,0);
INSERT INTO "task" VALUES(4,'gofmtall','The idea is to run gofmt -w file.go on every g
o file in the listing, *Edit turns out this is is difficult to do in golang **Edit bar
ely 3 line bash script. ','2015-11-12 16:58:31','2015-11-14 10:42:14','2015-11-13 13:1
6:48',3,1,1,NULL,1,0);

CREATE TABLE comments(id integer primary key autoincrement, content ntext, taskID refe
rences task(id), created datetime, user_id references user(id));

CREATE TABLE files(name varchar(1000) not null, autoName varchar(255) not null, user_i
d references user(id), created_date timestamp);
```

# Installing sqlite driver

We'll use the go-sqlite3 driver created by mattn. The reason being it implements the `database/sql` interface. The advantage of using a database library which uses the database/sql interface is that, the libraries are swappable by other libraries which implement the interface.

Type this in your terminal:

```
go get -u "github.com/mattn/go-sqlite3"
```

# Accessing the database

To access databases in Go, you use a sql.DB. You use this type to create statements and transactions, execute queries, and fetch results.

The first thing you should know is that a sql.DB isn't a database connection. It also doesn't map to any particular database software's notion of a "database" or "schema." It's an abstraction of the interface and existence of a database, which might be as varied as a local file, accessed through a network connection, or in-memory and in-process.

The sql.DB performs some important tasks for you behind the scenes:

1. It opens and closes connections to the actual underlying database, via the driver.
2. It manages a pool of connections as needed, which may be a variety of things as mentioned.

The sql.DB abstraction is designed to keep you from worrying about how to manage concurrent access to the underlying datastore. A connection is marked in-use when you use it to perform a task, and then returned to the available pool when it's not in use anymore. One consequence of this is that if you fail to release connections back to the pool, you can cause db.SQL to open a lot of connections, potentially running out of resources (too many connections, too many open file handles, lack of available network ports, etc). We'll discuss more about this later.

After creating a sql.DB, you can use it to query the database that it represents, as well as creating statements and transactions.

# Importing driver

To use database/sql you'll need the package itself, as well as a driver for the specific database you want to use.

You generally shouldn't use driver packages directly, although some drivers encourage you to do so. (In our opinion, it's usually a bad idea.) Instead, your code should only refer to types defined in database/sql, if possible. This helps avoid making your code dependent on the driver, so that you can change the underlying driver (and thus the database you're accessing) with minimal code changes. It also forces you to use the Go idioms instead of ad-hoc idioms that a particular driver author may have provided.

In this documentation, we'll use the excellent MySQL drivers from @julienschmidt and @arnehormann for examples.

Add the following to the top of your Go source file:

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

Notice that we're loading the driver anonymously, aliasing its package qualifier to _ so none of its exported names are visible to our code. Under the hood, the driver registers itself as being available to the database/sql package, but in general nothing else happens.

Now you're ready to access a database.

Every database has a connection mechanism, file for sqlite and IP address for MySQL/Postgres.

# Retrieving Result Sets

There are several idiomatic operations to retrieve results from the datastore.

1. Execute a query that returns rows.
2. Prepare a statement for repeated use, execute it multiple times, and destroy it.
3. Execute a statement in a once-off fashion, without preparing it for repeated use.
4. Execute a query that returns a single row. There is a shortcut for this special case.

Go's database/sql function names are significant. If a function name includes `Query`, it is designed to **ask a question of the database**, and will return a set of rows, even if it's empty. Statements that don't return rows should not use Query functions; they should use Exec().

# Fetching Data from the Database

Let's take a look at an example of how to query the database, working with results. We'll query the users table for a user whose id is 1, and print out the user's id and name. We will assign results to variables, a row at a time, with rows.Scan().

```
getTaskSQL = "select id, title, content, created_date from task
    where finish_date is null and is_deleted='N' order by created_date asc"

rows, err := database.Query(getTaskSQL)

if err != nil {
    log.Println(err)
}

defer rows.Close()
for rows.Next() {
    err := rows.Scan(&TaskID, &TaskTitle, &TaskContent, &TaskCreated)
    TaskContent = strings.Replace(TaskContent, "\n", "<br>", -1)
    if err != nil {
        log.Println(err)
    }
    fmt.Println(TaskID, TaskTitle, TaskContent, TaskCreated)
}

taskSQL := "delete from task"
tx := database.begin()
_, err = tx.Stmt(SQL).Exec(args...)
if err != nil {
    tx.Rollback()
} else {
    tx.Commit()
}
```

### Defer keyword

We use defer inside a function call.

```
package main
import (
    "fmt"
    "io/ioutil"
    "os"
)

func main() {
    file, err := os.Open('file.dat')
    if err != nil {
        fmt.Println("File doesn't exist or you don't have
        read permission")
    }

    defer file.Close()
    inputReader := bufio.NewReader(file)
    //do something about inputReader
}
```

The defer statement puts the function call at the bottom of the call stack, so whenever the function returns, defer is triggered. One has to be careful with using defer, it can cause difficult to find bugs.

file `~/main/main.go`

Find and fix the bug:

```
package main
import (
    _ "github.com/mattn/go-sqlite3"
    "fmt"
)

var database   *sql.DB

func init() {
    defer database.Close()
    database, err = sql.Open("sqlite3", "./tasks.db")
    if err != nil {
        fmt.Println(err)
    }
}
//intentional bug exists, fix it
func main() {
    getTaskSQL = "select id, title, content, created_date from task
        where finish_date is null and is_deleted='N' order by created_date asc"

    rows, err := database.Query(getTaskSQL)
    if err != nil {
        fmt.Println(err)
    }
    defer rows.Close()
    for rows.Next() {
        err := rows.Scan(&TaskID, &TaskTitle, &TaskContent, &TaskCreated)
        TaskContent = strings.Replace(TaskContent, "\n", "<br>", -1)
        if err != nil {
            fmt.Println(err)
        }
        fmt.Println(TaskID, TaskTitle, TaskContent, TaskCreated)
    }
    err = rows.Err()
    if err != nil {
        log.Fatal(err)
    }
}
```

Always defer `rows.Close()`, to free the database connection in the pool. So long as `rows` contains the result set, the database connection is in use and not available in the connection pool.

When the `rows.Next()` function returns EOF (End of File), which means that it has reached the end of records, it'll call `rows.Close()` for you, `Close()` can be called multiple times without side effects.

Here's what's happening in the above code:

1. We're using db.Query() to send the query to the database. We check the error, as usual.
2. We defer rows.Close(). This is very important.
3. We iterate over the rows with rows.Next().
4. We read the columns in each row into variables with rows.Scan().
5. We check for errors after we're done iterating over the rows.

This is pretty much the only way to do it in Go. You can't get a row as a map, for example. That's because everything is strongly typed. You need to create variables of the correct type and pass pointers to them, as shown.

A couple parts of this are easy to get wrong, and can have bad consequences.

- You should always check for an error at the end of the for rows.Next() loop. If there's an error during the loop, you need to know about it. Don't just assume that the loop iterates until you've processed all the rows.
- Second, as long as there's an open result set (represented by rows), the underlying connection is busy and can't be used for any other query. That means it's not available in the connection pool. If you iterate over all of the rows with rows.Next(), eventually you'll read the last row, and rows.Next() will encounter an internal EOF error and call rows.Close() for you. But if for some reason you exit that loop – an early return, or so on – then the rows doesn't get closed, and the connection remains open. (It is auto-closed if rows.Next() returns false due to an error, though). This is an easy way to run out of resources.
- rows.Close() is a harmless no-op if it's already closed, so you can call it multiple times. Notice, however, that we check the error first, and only call rows.Close() if there isn't an error, in order to avoid a runtime panic.
- You should always defer rows.Close(), even if you also call rows.Close() explicitly at the end of the loop, which isn't a bad idea.
- Don't defer within a loop. A deferred statement doesn't get executed until the function exits, so a long-running function shouldn't use it. If you do, you will slowly accumulate memory. If you are repeatedly querying and consuming result sets within a loop, you should explicitly call rows.Close() when you're done with each result, and not use defer.

# How Scan() Works

When you iterate over rows and scan them into destination variables, Go performs data type conversions work for you, behind the scenes. It is based on the type of the destination variable. Being aware of this can clean up your code and help avoid repetitive work.

For example, suppose you select some rows from a table that is defined with string columns, such as VARCHAR(45) or similar. You happen to know, however, that the table always contains numbers. If you pass a pointer to a string, Go will copy the bytes into the string. Now you can use strconv.ParseInt() or similar to convert the value to a number. You'll have to check for errors in the SQL operations, as well as errors parsing the integer. This is messy and tedious.

Or, you can just pass Scan() a pointer to an integer. Go will detect that and call strconv.ParseInt() for you. If there's an error in conversion, the call to Scan() will return it. Your code is neater and smaller now. This is the recommended way to use database/sql.

# Preparing Queries

You should, in general, always prepare queries to be used multiple times. The result of preparing the query is a prepared statement, which can have placeholders (a.k.a. bind values) for parameters that you'll provide when you execute the statement. This is much better than concatenating strings, for all the usual reasons (avoiding SQL injection attacks, for example).

In MySQL, the parameter placeholder is ?, and in PostgreSQL it is $N, where N is a number. SQLite accepts either of these. In Oracle placeholders begin with a colon and are named, like :param1. We'll use ? because we're using MySQL as our example.

```
stmt, err := db.Prepare("select id, title, content, created_date from task
        where finish_date is null and is_deleted='N' and task.user=?")
if err != nil {
    log.Fatal(err)
}
defer stmt.Close()
rows, err := stmt.Query(1)
if err != nil {
    log.Fatal(err)
}
defer rows.Close()
for rows.Next() {
    // ...
}
if err = rows.Err(); err != nil {
    log.Fatal(err)
}
```

Under the hood, db.Query() actually prepares, executes, and closes a prepared statement. That's three round-trips to the database. If you're not careful, you can triple the number of database interactions your application makes! Some drivers can avoid this in specific cases, but not all drivers do. See prepared statements for more.

# Single-Row Queries

If a query returns at most one row, you can use a shortcut around some of the lengthy boilerplate code:

```
var name string
query:="select taskDescription from task where id = ?"
err = db.QueryRow(query, 1).Scan(&taskDescription)
if err != nil {
    log.Fatal(err)
}
fmt.Println(name)
```

Errors from the query are deferred until Scan() is called, and then are returned from that. You can also call QueryRow() on a prepared statement:

```
query :="select taskDescription from task where id = ?"
stmt, err := db.Prepare(query, 1).Scan(&taskDescription)

if err != nil {
    log.Fatal(err)
}

var taskDescription string
err = stmt.QueryRow(1).Scan(&taskDescription)

if err != nil {
    log.Fatal(err)
}

fmt.Println(taskDescription)
```

# Modifying Data and Using Transactions

Now we're ready to see how to modify data and work with transactions. The distinction might seem artificial if you're used to programming languages that use a "statement" object for fetching rows as well as updating data, but in Go, there's an important reason for the difference.

# Statements that Modify Data

Use Exec(), preferably with a prepared statement, to accomplish an INSERT, UPDATE, DELETE, or other statement that doesn't return rows. The following example shows how to insert a row and inspect metadata about the operation:

```
stmt, err := db.Prepare("INSERT INTO users(username, password, email) VALUES(?,?,?)")
if err != nil {
    log.Fatal(err)
}
res, err := stmt.Exec("Sherlock", "notaSmartPassword", "sherlock@startr.com")
if err != nil {
    log.Fatal(err)
}
lastId, err := res.LastInsertId()
if err != nil {
    log.Fatal(err)
}
rowCnt, err := res.RowsAffected()
if err != nil {
    log.Fatal(err)
}
log.Printf("ID = %d, affected = %d\n", lastId, rowCnt)
```

Executing the statement produces a sql.Result that gives access to statement metadata: the last inserted ID and the number of rows affected.

What if you don't care about the result? What if you just want to execute a statement and check if there were any errors, but ignore the result? Wouldn't the following two statements do the same thing?

```
_, err := db.Exec("DELETE FROM users")  // OK
_, err := db.Query("DELETE FROM users") // BAD
```

The answer is no. They do not do the same thing, and you should never use Query() like this. The Query() will return a sql.Rows, which reserves a database connection until the sql.Rows is closed. Since there might be unread data (e.g. more data rows), the connection can not be used. In the example above, the connection will never be released again. The garbage collector will eventually close the underlying net.Conn for you, but this might take a long time. Moreover the database/sql package keeps tracking the connection in its pool, hoping that you release it at some point, so that the connection can be used again. This anti-pattern is therefore a good way to run out of resources (too many connections, for example).

# Working with Transactions

In Go, a transaction is essentially an object that reserves a connection to the datastore. It lets you do all of the operations we've seen thus far, but guarantees that they'll be executed on the same connection.

You begin a transaction with a call to db.Begin(), and close it with a Commit() or Rollback() method on the resulting Tx variable. Under the covers, the Tx gets a connection from the pool, and reserves it for use only with that transaction. The methods on the Tx map one-for-one to methods you can call on the database itself, such as Query() and so forth.

Prepared statements that are created in a transaction are bound exclusively to that transaction. See prepared statements for more.

You should not mingle the use of transaction-related functions such as Begin() and Commit() with SQL statements such as BEGIN and COMMIT in your SQL code. Bad things might result:

1. The Tx objects could remain open, reserving a connection from the pool and not returning it.
2. The state of the database could get out of sync with the state of the Go variables representing it.
3. You could believe you're executing queries on a single connection, inside of a transaction, when in reality Go has created several connections for you invisibly and some statements aren't part of the transaction.

While you are working inside a transaction you should be careful not to make calls to the Db variable. Make all of your calls to the Tx variable that you created with db.Begin(). The Db is not in a transaction, only the Tx is. If you make further calls to db.Exec() or similar, those will happen outside the scope of your transaction, on other connections.

If you need to work with multiple statements that modify connection state, you need a Tx even if you don't want a transaction per se. For example:

1. Creating temporary tables, which are only visible to one connection.
2. Setting variables, such as MySQL's SET @var := somevalue syntax.
3. Changing connection options, such as character sets or timeouts.

If you need to do any of these things, you need to bind your activity to a single connection, and the only way to do that in Go is to use a Tx.

Below lies an example of using transaction

file `db/db.go`

```
//RestoreTask is used to restore tasks from the Trash
func RestoreTask(id int) error {
    query := "update task set is_deleted='N',last_modified_at=datetime() where id=?"
    restoreSQL, err := database.Prepare(query)
    if err != nil {
        fmt.Println(err)
    }
    tx, err := database.Begin()
    if err != nil {
        fmt.Println(err)
    }
    _, err = tx.Stmt(restoreSQL).Exec(id)
    if err != nil {
        fmt.Println("doing rollback")
        tx.Rollback()
    } else {
        tx.Commit()
    }
    return err
}
```

# Using Prepared Statements

Prepared statements have all the usual benefits in Go: security, efficiency, convenience. But the way they're implemented is a little different from what you might be used to, especially with regards to how they interact with some of the internals of `database/sql`.

# Prepared Statements And Connections

At the database level, a prepared statement is bound to a single database connection. The typical flow is that the client sends a SQL statement with placeholders to the server for preparation, the server responds with a statement ID, and then the client executes the statement by sending its ID and parameters.

In Go, however, connections are not exposed directly to the user of the `database/sql` package. You don't prepare a statement on a connection. You prepare it on a `DB` or a `Tx`. And `database/sql` has some convenience behaviors such as automatic retries. For these reasons, the underlying association between prepared statements and connections, which exists at the driver level, is hidden from your code.

Here's how it works:

1. When you prepare a statement, it's prepared on a connection in the pool.

2. The `Stmt` object remembers which connection was used.
3. When you execute the `Stmt`, it tries to use the connection. If it's not available because it's closed or busy doing something else, it gets another connection from the pool *and re-prepares the statement with the database on another connection.*

Because statements will be re-prepared as needed when their original connection is busy, it's possible for high-concurrency usage of the database, which may keep a lot of connections busy, to create a large number of prepared statements. This can result in apparent leaks of statements, statements being prepared and re-prepared more often than you think, and even running into server-side limits on the number of statements.

# Avoiding Prepared Statements

Go creates prepared statements for you under the covers. A simple `db.Query(sql, param1, param2)`, for example, works by preparing the sql, then executing it with the parameters and finally closing the statement.

Sometimes a prepared statement is not what you want, however. There might be several reasons for this:

1. The database doesn't support prepared statements. When using the MySQL driver, for example, you can connect to MemSQL and Sphinx, because they support the MySQL wire protocol. But they don't support the "binary" protocol that includes prepared statements, so they can fail in confusing ways.
2. The statements aren't reused enough to make them worthwhile, and security issues are handled in other ways, so performance overhead is undesired. An example of this can be seen at the VividCortex blog.

If you don't want to use a prepared statement, you need to use `fmt.Sprint()` or similar to assemble the SQL, and pass this as the only argument to `db.Query()` or `db.QueryRow()`. And your driver needs to support plaintext query execution, which is added in Go 1.1 via the `Execer` and `Queryer` interfaces, documented here.

# Prepared Statements in Transactions

Prepared statements that are created in a `Tx` are bound exclusively to it, so the earlier cautions about repreparing do not apply. When you operate on a `Tx` object, your actions map directly to the one and only one connection underlying it.

This also means that prepared statements created inside a `Tx` can't be used separately from it. Likewise, prepared statements created on a `DB` can't be used within a transaction, because they will be bound to a different connection.

To use a prepared statement prepared outside the transaction in a `Tx`, you can use `Tx.Stmt()`, which will create a new transaction-specific statement from the one prepared outside the transaction. It does this by taking an existing prepared statement, setting the connection to that of the transaction and repreparing all statements every time they are executed. This behavior and its implementation are undesirable and there's even a TODO in the `database/sql` source code to improve it; we advise against using this.

Caution must be exercised when working with prepared statements in transactions. Consider the following example:

```
tx, err := db.Begin()
if err != nil {
    log.Fatal(err)
}
defer tx.Rollback()
stmt, err := tx.Prepare("INSERT INTO foo VALUES (?)")
if err != nil {
    log.Fatal(err)
}
defer stmt.Close() // danger!
for i := 0; i < 10; i++ {
    _, err = stmt.Exec(i)
    if err != nil {
        log.Fatal(err)
    }
}
err = tx.Commit()
if err != nil {
    log.Fatal(err)
}
// stmt.Close() runs here!
```

Before Go 1.4 closing a `*sql.Tx` released the connection associated with it back into the pool, but the deferred call to Close on the prepared statement was executed **after** that has happened, which could lead to concurrent access to the underlying connection, rendering the connection state inconsistent. If you use Go 1.4 or older, you should make sure the statement is always closed before the transaction is committed or rolled back. This issue was fixed in Go 1.4 by CR 131650043.

# Parameter Placeholder Syntax

The syntax for placeholder parameters in prepared statements is database-specific. For example, comparing MySQL, PostgreSQL, and Oracle:

```
MySQL               PostgreSQL            Oracle
=====               ==========            ======
WHERE col = ?       WHERE col = $1        WHERE col = :col
VALUES(?, ?, ?)     VALUES($1, $2, $3)    VALUES(:val1, :val2, :val3)
```

# Handling Errors

Almost all operations with `database/sql` types return an error as the last value. You should always check these errors, never ignore them.

There are a few places where error behavior is special-case, or there's something additional you might need to know.

# Errors From Iterating Resultsets

Consider the following code:

```
for rows.Next() {
    // ...
}
if err = rows.Err(); err != nil {
    // handle the error here
}
```

The error from `rows.Err()` could be the result of a variety of errors in the `rows.Next()` loop. The loop might exit for some reason other than finishing the loop normally, so you always need to check whether the loop terminated normally or not. An abnormal termination automatically calls `rows.Close()`, although it's harmless to call it multiple times.

# Errors From Closing Resultsets

You should always explicitly close a `sql.Rows` if you exit the loop prematurely, as previously mentioned. It's auto-closed if the loop exits normally or through an error, but you might mistakenly do this:

```
for rows.Next() {
    // ...
    break; // whoops, rows is not closed! memory leak...
}
// do the usual "if err = rows.Err()" [omitted here]...
// it's always safe to [re?]close here:
if err = rows.Close(); err != nil {
    // but what should we do if there's an error?
    log.Println(err)
}
```

The error returned by `rows.Close()` is the only exception to the general rule that it's best to capture and check for errors in all database operations. If `rows.Close()` returns an error, it's unclear what you should do. Logging the error message or panicing might be the only sensible thing, and if that's not sensible, then perhaps you should just ignore the error.

# Errors From QueryRow()

Consider the following code to fetch a single row:

```
var name string
err = db.QueryRow("select name from users where id = ?", 1).Scan(&name)
if err != nil {
    log.Fatal(err)
}
fmt.Println(name)
```

What if there was no user with `id = 1`? Then there would be no row in the result, and `.Scan()` would not scan a value into `name`. What happens then?

Go defines a special error constant, called `sql.ErrNoRows`, which is returned from `QueryRow()` when the result is empty. This needs to be handled as a special case in most circumstances. An empty result is often not considered an error by application code, and if you don't check whether an error is this special constant, you'll cause application-code errors you didn't expect.

Errors from the query are deferred until `Scan()` is called, and then are returned from that. The above code is better written like this instead:

```
var name string
err = db.QueryRow("select name from users where id = ?", 1).Scan(&name)
if err != nil {
    if err == sql.ErrNoRows {
        // there were no rows, but otherwise no error occurred
    } else {
        log.Fatal(err)
    }
}
fmt.Println(name)
```

One might ask why an empty result set is considered an error. There's nothing erroneous about an empty set. The reason is that the `QueryRow()` method needs to use this special-case in order to let the caller distinguish whether `QueryRow()` in fact found a row; without it, `Scan()` wouldn't do anything and you might not realize that your variable didn't get any value from the database after all.

You should only run into this error when you're using `QueryRow()`. If you encounter this error elsewhere, you're doing something wrong.

# Identifying Specific Database Errors

It can be tempting to write code like the following:

```
rows, err := db.Query("SELECT someval FROM sometable")
// err contains:
// ERROR 1045 (28000): Access denied for user 'foo'@'::1' (using password: NO)
if strings.Contains(err.Error(), "Access denied") {
    // Handle the permission-denied error
}
```

This is not the best way to do it, though. For example, the string value might vary depending on what language the server uses to send error messages. It's much better to compare error numbers to identify what a specific error is.

The mechanism to do this varies between drivers, however, because this isn't part of `database/sql` itself. In the MySQL driver that this tutorial focuses on, you could write the following code:

```
if driverErr, ok := err.(*mysql.MySQLError); ok { // Now the error number is accessibl
e directly
    if driverErr.Number == 1045 {
        // Handle the permission-denied error
    }
}
```

Again, the `MySQLError` type here is provided by this specific driver, and the `.Number` field may differ between drivers. The value of the number, however, is taken from MySQL's error message, and is therefore database specific, not driver specific.

This code is still ugly. Comparing to 1045, a magic number, is a code smell. Some drivers (though not the MySQL one, for reasons that are off-topic here) provide a list of error identifiers. The Postgres `pq` driver does, for example, in error.go. And there's an external package of MySQL error numbers maintained by VividCortex. Using such a list, the above code is better written thus:

```
if driverErr, ok := err.(*mysql.MySQLError); ok {
    if driverErr.Number == mysqlerr.ER_ACCESS_DENIED_ERROR {
        // Handle the permission-denied error
    }
}
```

# Handling Connection Errors

What if your connection to the database is dropped, killed, or has an error?

You don't need to implement any logic to retry failed statements when this happens. As part of the connection pooling in `database/sql`, handling failed connections is built-in. If you execute a query or other statement and the underlying connection has a failure, Go will reopen a new connection (or just get another from the connection pool) and retry, up to 10 times.

There can be some unintended consequences, however. Some types of errors may be retried when other error conditions happen. This might also be driver-specific. One example that has occurred with the MySQL driver is that using `KILL` to cancel an undesired statement (such as a long-running query) results in the statement being retried up to 10 times.

# Working with NULLs

Nullable columns are annoying and lead to a lot of ugly code. If you can, avoid them. If not, then you'll need to use special types from the `database/sql` package to handle them, or define your own.

There are types for nullable booleans, strings, integers, and floats. Here's how you use them:

```
for rows.Next() {
    var s sql.NullString
    err := rows.Scan(&s)
    // check err
    if s.Valid {
       // use s.String
    } else {
       // NULL value
    }
}
```

Limitations of the nullable types, and reasons to avoid nullable columns in case you need more convincing:

1. There's no `sql.NullUint64` or `sql.NullYourFavoriteType`. You'd need to define your own for this.
2. Nullability can be tricky, and not future-proof. If you think something won't be null, but you're wrong, your program will crash, perhaps rarely enough that you won't catch errors before you ship them.
3. One of the nice things about Go is having a useful default zero-value for every variable. This isn't the way nullable things work.

If you need to define your own types to handle NULLs, you can copy the design of `sql.NullString` to achieve that.

If you can't avoid having NULL values in your database, there is another work around that most database systems support, namely `COALESCE()`. Something like the following might be something that you can use, without introducing a myriad of `sql.Null*` types.

```
rows, err := db.Query(`
    SELECT
        name,
        COALESCE(other_field, '') as other_field
    WHERE id = ?
`, 42)

for rows.Next() {
    err := rows.Scan(&name, &otherField)
    // ..
    // If `other_field` was NULL, `otherField` is now an empty string. This works with
 other data types as well.
}
```

# Working with Unknown Columns

The `Scan()` function requires you to pass exactly the right number of destination variables. What if you don't know what the query will return?

If you don't know how many columns the query will return, you can use `Columns()` to find a list of column names. You can examine the length of this list to see how many columns there are, and you can pass a slice into `Scan()` with the correct number of values. For example, some forks of MySQL return different columns for the `SHOW PROCESSLIST` command, so you have to be prepared for that or you'll cause an error. Here's one way to do it; there are others:

```
cols, err := rows.Columns()
if err != nil {
    // handle the error
} else {
    dest := []interface{}{ // Standard MySQL columns
        new(uint64), // id
        new(string), // host
        new(string), // user
        new(string), // db
        new(string), // command
        new(uint32), // time
        new(string), // state
        new(string), // info
    }
    if len(cols) == 11 {
        // Percona Server
    } else if len(cols) > 8 {
        // Handle this case
    }
    err = rows.Scan(dest...)
    // Work with the values in dest
}
```

If you don't know the columns or their types, you should use `sql.RawBytes`.

```
cols, err := rows.Columns() // Remember to check err afterwards
vals := make([]interface{}, len(cols))
for i, _ := range cols {
    vals[i] = new(sql.RawBytes)
}
for rows.Next() {
    err = rows.Scan(vals...)
    // Now you can check each element of vals for nil-ness,
    // and you can use type introspection and type assertions
    // to fetch the column into a typed variable.
}
```

# The connection pool

There is a basic connection pool in the database/sql package. There isn’t a lot of ability to control or inspect it, but here are some things you might find useful to know:

1. Connection pooling means that executing two consecutive statements on a single database might open two connections and execute them separately. It is fairly common for programmers to be confused as to why their code misbehaves. For example, LOCK TABLES followed by an INSERT can block because the INSERT is on a connection that

does not hold the table lock.
2. Connections are created when needed and there isn’t a free connection in the pool.
3. By default, there’s no limit on the number of connections. If you try to do a lot of things at once, you can create an arbitrary number of connections. This can cause the database to return an error such as “too many connections.”
4. In Go 1.1 or newer, you can use db.SetMaxIdleConns(N) to limit the number of idle connections in the pool. This doesn’t limit the pool size, though.
5. In Go 1.2.1 or newer, you can use db.SetMaxOpenConns(N) to limit the number of total open connections to the database. Unfortunately, a deadlock bug (fix) prevents db.SetMaxOpenConns(N) from safely being used in 1.2.
6. Connections are recycled rather fast. Setting a high number of idle connections with db.SetMaxIdleConns(N) can reduce this churn, and help keep connections around for reuse.
7. Keeping a connection idle for a long time can cause problems (like in this issue with MySQL on Microsoft Azure). Try db.SetMaxIdleConns(0) if you get connection timeouts because a connection is idle for too long.

# Surprises, Antipatterns and Limitations

Although `database/sql` is simple once you're accustomed to it, you might be surprised by the subtlety of use cases it supports. This is common to Go's core libraries.

## Resource Exhaustion

As mentioned throughout this site, if you don't use `database/sql` as intended, you can certainly cause trouble for yourself, usually by consuming some resources or preventing them from being reused effectively:

- Opening and closing databases can cause exhaustion of resources.
- Failing to read all rows or use `rows.Close()` reserves connections from the pool.
- Using `Query()` for a statement that doesn't return rows will reserve a connection from the pool.
- Failing to be aware of how prepared statements work can lead to a lot of extra database activity.

## Large uint64 Values

Here's a surprising error. You can't pass big unsigned integers as parameters to statements if their high bit is set:

_, err := db.Exec("INSERT INTO users(id) VALUES", math.MaxUint64) // Error

This will throw an error. Be careful if you use `uint64` values, as they may start out small and work without error, but increment over time and start throwing errors.

# Connection State Mismatch

Some things can change connection state, and that can cause problems for two reasons:

1. Some connection state, such as whether you're in a transaction, should be handled through the Go types instead.
2. You might be assuming that your queries run on a single connection when they don't.

For example, setting the current database with a `USE` statement is a typical thing for many people to do. But in Go, it will affect only the connection that you run it in. Unless you are in a transaction, other statements that you think are executed on that connection may actually run on different connections gotten from the pool, so they won't see the effects of such changes.

Additionally, after you've changed the connection, it'll return to the pool and potentially pollute the state for some other code. This is one of the reasons why you should never issue BEGIN or COMMIT statements as SQL commands directly, too.

# Database-Specific Syntax

The `database/sql` API provides an abstraction of a row-oriented database, but specific databases and drivers can differ in behavior and/or syntax, such as prepared statement placeholders.

# Multiple Result Sets

The Go driver doesn't support multiple result sets from a single query in any way, and there doesn't seem to be any plan to do that, although there is a feature request for supporting bulk operations such as bulk copy.

This means, among other things, that a stored procedure that returns multiple result sets will not work correctly.

# Invoking Stored Procedures

Invoking stored procedures is driver-specific, but in the MySQL driver it can't be done at present. It might seem that you'd be able to call a simple procedure that returns a single result set, by executing something like this:

```
err := db.QueryRow("CALL mydb.myprocedure").Scan(&result) // Error
```

In fact, this won't work. You'll get the following error: *Error 1312: PROCEDURE mydb.myprocedure can't return a result set in the given context*. This is because MySQL expects the connection to be set into multi-statement mode, even for a single result, and the driver doesn't currently do that (though see this issue).

# Multiple Statement Support

The `database/sql` doesn't explicitly have multiple statement support, which means that the behavior of this is backend dependent:

```
_, err := db.Exec("DELETE FROM tbl1; DELETE FROM tbl2") // Error/unpredictable result
```

The server is allowed to interpret this however it wants, which can include returning an error, executing only the first statement, or executing both.

Similarly, there is no way to batch statements in a transaction. Each statement in a transaction must be executed serially, and the resources in the results, such as a Row or Rows, must be scanned or closed so the underlying connection is free for the next statement to use. This differs from the usual behavior when you're not working with a transaction. In that scenario, it is perfectly possible to execute a query, loop over the rows, and within the loop make a query to the database (which will happen on a new connection):

```
rows, err := db.Query("select * from tbl1") // Uses connection 1
for rows.Next() {
    err = rows.Scan(&myvariable)
    // The following line will NOT use connection 1, which is already in-use
    db.Query("select * from tbl2 where id = ?", myvariable)
}
```

But transactions are bound to just one connection, so this isn't possible with a transaction:

```
tx, err := db.Begin()
rows, err := tx.Query("select * from tbl1") // Uses tx's connection
for rows.Next() {
    err = rows.Scan(&myvariable)
    // ERROR! tx's connection is already busy!
    tx.Query("select * from tbl2 where id = ?", myvariable)
}
```

Go doesn't stop you from trying, though. For that reason, you may wind up with a corrupted connection if you attempt to perform another statement before the first has released its resources and cleaned up after itself. This also means that each statement in a transaction results in a separate set of network round-trips to the database.

# Database Encapsulation

We encapsulate our db object inside a struct. We also encapsulate the database actions as shown below

```
var database Database

//Database encapsulates database
type Database struct {
    db *sql.DB
}

func (db Database) begin() (tx *sql.Tx) {
    tx, err := db.db.Begin()
    if err != nil {
        log.Println(err)
        return nil
    }
    return tx
}

func (db Database) prepare(q string) (stmt *sql.Stmt) {
    stmt, err := db.db.Prepare(q)
    if err != nil {
        log.Println(err)
        return nil
    }
    return stmt
}

func (db Database) query(q string,
    args ...interface{}) (rows *sql.Rows) {
      rows, err := db.db.Query(q, args...)
      if err != nil {
```

```
            log.Println(err)
            return nil
        }
        return rows
}

func init() {
    database.db, err =
       sql.Open("sqlite3", "./newtask.db")
    if err != nil {
        log.Fatal(err)
    }
}

//Close database connection
func Close() {
    database.db.Close()
}

//taskQuery encapsulates Exec()
func taskQuery(sql string, args ...interface{}) error {
    SQL := database.prepare(sql)
    tx := database.begin()
    _, err = tx.Stmt(SQL).Exec(args...)
    if err != nil {
        log.Println("taskQuery: ", err)
        tx.Rollback()
    } else {
        tx.Commit()
    }
    return err
}
```

> Note: init()
>
> The init function is the first function to run when the package is imported or executed. This is why we do the initialization in it.

**The fault in our code:**

Fixing the intentional bug in the above code:

```
func init() {
        defer database.Close()
        database, err = sql.Open("sqlite3", "./tasks.db")
        if err != nil {
            fmt.Println(err)
        }
    }
```

### Homework

See the `/code/chapter-4/4.5database` in our code repository and modify the file to insert data from the `4.3formsupload` folder. We have two working code set, one of printing form values on the console and one of fetching db values and rendering a template. What you have to do is based on this chapter, write methods to insert values from the form to the database.

### Links

-Previous section -Next section

# An Example

Expected output:

open your `task.db` file in sqlite3 like this

```
[Tasks] $ sqlite3 task.db
sqlite> select title from task limit 1;
Publish on github
```

Now this output should match with the one we see at localhost:8080

After running the file, go to `localhost:8080` and `localhost:8080/add`

file `~/main/main.go`

```
package main

import (
    "database/sql"
    "fmt"
    "log"
    "net/http"
    "time"

    _ "github.com/mattn/go-sqlite3"
)

var database *sql.DB
var err error

//Task is the struct used to identify tasks
type Task struct {
    Id      int
    Title   string
    Content string
    Created string
}

//Context is the struct passed to templates
type Context struct {
    Tasks      []Task
    Navigation string
    Search     string
    Message    string
}
```

```
func init() {
    database, err = sql.Open("sqlite3", "./tasks.db")
    if err != nil {
        fmt.Println(err)
    }
}

func main() {
    http.HandleFunc("/", ShowAllTasksFunc)
    http.HandleFunc("/add/", AddTaskFunc)
    fmt.Println("running on 8080")
    log.Fatal(http.ListenAndServe(":8080", nil))
}

//ShowAllTasksFunc is used to handle the "/" URL which is the default one
func ShowAllTasksFunc(w http.ResponseWriter, r *http.Request) {
    if r.Method == "GET" {
        context := GetTasks() //true when you want non deleted notes
        w.Write([]byte(context.Tasks[0].Title))
    } else {
        http.Redirect(w, r, "/", http.StatusFound)
    }
}

func GetTasks() Context {
    var task []Task
    var context Context
    var TaskID int
    var TaskTitle string
    var TaskContent string
    var TaskCreated time.Time
    var getTasksql string

    getTasksql = "select id, title, content, created_date from task;"

    rows, err := database.Query(getTasksql)
    if err != nil {
        fmt.Println(err)
    }
    defer rows.Close()
    for rows.Next() {
        err := rows.Scan(&TaskID, &TaskTitle, &TaskContent, &TaskCreated)
        if err != nil {
            fmt.Println(err)
        }
        TaskCreated = TaskCreated.Local()
        a := Task{Id: TaskID, Title: TaskTitle, Content: TaskContent,
            Created: TaskCreated.Format(time.UnixDate)[0:20]}
        task = append(task, a)
    }
    context = Context{Tasks: task}
    return context
}
```

```
//AddTaskFunc is used to handle the addition of new task, "/add" URL
func AddTaskFunc(w http.ResponseWriter, r *http.Request) {
    title := "random title"
    content := "random content"
    truth := AddTask(title, content)
    if truth != nil {
        log.Fatal("Error adding task")
    }
    w.Write([]byte("Added task"))
}

//AddTask is used to add the task in the database
func AddTask(title, content string) error {
    query:="insert into task(title, content, created_date, last_modified_at)\
    values(?,?,datetime(), datetime())"
    restoreSQL, err := database.Prepare(query)
    if err != nil {
        fmt.Println(err)
    }
    tx, err := database.Begin()
    _, err = tx.Stmt(restoreSQL).Exec(title, content)
    if err != nil {
        fmt.Println(err)
        tx.Rollback()
    } else {
        log.Print("insert successful")
        tx.Commit()
    }
    return err
}
```

# Homework

The homework is to split the code into packages and get it to work, the type definition goes into the `types/types.go` file, the handler definition goes into the `views/views.go`, the database read and write methods go into the `db/db.go`. Make sure that after you refactor the code, that the code runs.

### Links

-Previous section -Next section

# Working with Forms

HTML forms are used to get data from the user. Forms can use both the GET and POST methods for transferring data to the server, but it is recommended to use HTTP POST method just because it doesn't highlight data in the URL and because of the many things we discussed in the chapter Web Programming Basics.

Forms are rendered by templating, which we'll see in a later chapter. As of now we want to understand how to get data from the end user by using forms.

There are two parts of working with forms, the HTML part and the Go part. The HTML page gets the data and sends it to the server as a POST `/GET` and the Go part will parse the form to do some task like letting the user to login or inserting data in the database.

Each form element has a *name* which is referenced in the server side part of the form, below we have the file upload and a drop down list. Both of which have a unique name.

```
<form action="/add/" method="POST">
    <div class="form-group">
        <input type="text" name="title" class="form-control" id="add-note-title" place
holder="Title"
        style="border:none;border-bottom:1px solid gray; box-shadow:none;">
    </div>
    <div class="form-group">
        <textarea class="form-control" name="content" id="add-note-content" placeholde
r="Content"
        rows="10" style="border:none;border-bottom:1px solid gray; box-shadow:none;"><
/textarea>
    File: <input    type="file"    name="uploadfile"    /> <br>
    Priority: <select name="priority">
        <option>---</option>
        <option value="3">High</option>
        <option value="2">Medium</option>
        <option value="1">Low</option>
        </select>
    </div>
    </div>
    <div class="modal-footer">
        <button type="button" class="btn btn-default" data-dismiss="modal">Close</butt
on>
        <input type="submit" text="submit" class="btn btn-default" />
    </div>
</form>
```

When we are populating this HTML page, we can also populate it dynamically, without getting bogged down by syntax, ignore templating for the while We have a variable called Navigation and one called Categories, we loop through Categories and if the Navigation is equal to that category then the checked value is true.

```
Category:
    <select name="category" class="dropdown">
        <option>---</option>
        {{$navigation := .Navigation}} {{$categories := .Categories}}
        {{range $cat := $categories}}
        <option value="{{$cat.Name}}" {{if eq $cat.Name $navigation }} checked="checke
d" {{end}}> {{$cat.Name}} </option>
        {{end}}
    </select>
```

Here we have used a drop down box, you can use radio buttons like below

```
<input type="radio" name="gender" value="1">Female
<input type="radio" name="gender" value="2">Male
<input type="radio" name="gender" value="3">Other
```

Or checkboxes

```
<input type="checkbox" name="gender" value="female">Female
<input type="checkbox" name="gender" value="male">Male
<input type="checkbox" name="gender" value="other">Other
```

We get the value of a drop down box/radio button/checkbox on the server side by using the **name** field like below:

```
value := r.Form.Get("gender")
value := r.FormValue("gender")
```

As we saw earlier, our webservers basically take a HTTP Request object and return an HTTP Response Object, below is an example of a sample HTTP Request object.

The host is the IP address sending the req, User Agent: fingerprint of the machine, the Accept- fields define various parts like the language, encoding Referer is which IP made the call, Cookie is the value of the cookie stored on the system and Connection is the type of connection.

In this request snapshot, we also have a file which we upload, for file upload, the content type is multipart/form-data

### Request Header

```
Host: 127.0.0.1:8081
User-Agent: ...
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://127.0.0.1:8081/
Cookie: csrftoken=abcd
Connection: keep-alive
```

### Request Body

```
Content-Type: multipart/form-data;
boundary=--------------------------629926480231270473150794805 3
Content-Length: 15031

----------------------------629926480231270473150794805 3
Content-Disposition: form-data; name="title"

working with forms
----------------------------629926480231270473150794805 3
Content-Disposition: form-data; name="CSRFToken"

abcd
----------------------------629926480231270473150794805 3
Content-Disposition: form-data; name="content"

finish the chapter working with forms
----------------------------629926480231270473150794805 3
Content-Disposition: form-data; name="uploadfile"; filename="2.4workingwithform.md"
Content-Type: text/x-markdown

--file content--
----------------------------629926480231270473150794805 3
Content-Disposition: form-data; name="priority"

3
----------------------------629926480231270473150794805 3--
```

If you had wondered how Google's home page shows a pop up when you visit google.com on IE of Firefox, it checks your User-Agent. The thing with HTTP request is that they can be modified to any extent, the Chrome developer tools gives you quite sophisticated tools to modify your requests, even User Agent spoofing is a default feature avaiable, this feature is available so we can test our webapps in one window simulating many internet browsers at one go.

The basic part of working with forms is to identify which user that particular form belongs to, there are ways to attain that, we can either have a stateful or a stateless web server.

A stateless server doesn't store sessions, it requires an authentication key for each request while a stateful server stores sessions. For storing sessions a cookie is used, which is a file which is stored in the private memory of the web browser which we use. Only the website which created the cookie can access the cookie, no third party websites can access the cookies, but the OS user can read/edit/delete cookies using the web browser.

# CSRF

CSRF stands for Cross Request Site Forgery. Any website can send a POST request to your web server, who sent the request can be found in the `Referer` field of your HTTP response. It is a form of confused deputy attack in which the deputy is your web browser. A malicious user doesn't have direct access to your website, so it makes use of your browser to send a malicious request. Typically cookies enable your browser to authenticate itself to a webserver, so what these malicious websites do is, they send in a HTTP request on behalf of your browser.

We can thwart this attack by restricting the referer to your own domain, but it is quite easy to manipulate the misspelt referer field of a HTTP request.

Another way is to use tokens. While rendering our form, we send in a hidden field with crypto generated string of 256 characters, so when we process the POST request, we first check if the token is valid or not and then decide if the data came from a genuine source or from a malicious source. It doesn't have to be malicious actually, even if a legitimate user tried to trick your webserver into accepting data, we shouldn't entertain it.

To check the csrf token, we serve the token to the form and store it in a cookie, when we get the POST request, we check if both are equal or not. This is because a malicious person might trick a user to click on a form but they can't set cookies for your webapplication.

A point to note here is that *never trust user data*. Always clean/sanitize data which you get from the end user.

### Javascript

If you are serious about web development, you ought to learn Javascript in detail. While building a web app, there will be times when you would want to improve the UI of your application, which would mean a change in the html page. Using JS is inevitable while building beautiful webapps, while adding some new html feature, open the "web inspector" present in the developer tools and dynamically add the html code. The web inspector allows us to manipulate the CSS and HTML part. Now open the javascript console, that enables

you to test the JS feature which you are willing to add. For instance, in the tasks application, there was no provision to expand/contract the size of the task, so I added a button in the web inspector,

```
        <button class="toggle"></button>
```

In the JS console, to toggle the visibility of my `.noteContent` field, I did this:

```
        $('.toggle').next().toggle()
```

This proved that it works, so now go to your template and actually add the code. Make sure the html is correct because while running, the html files are parsed once, so for any html change, you have to run the web app for each HTML change. So once the html is set up, if you change the JS/CSS then you just have to refresh the page, because the html page gets the JS/CSS each time the page is loaded.

As we saw in the above paragraph, for preventing CSRF, we need to generate a *token* and send as a hidden field in the form and store it in a cookie, when we get the POST request from the

## Forms in Go

In the below function we set the cookie, we first generate a CSRF token, which'll be unique for each HTTP request which we get and store it in a cookie.

```
//ShowAllTasksFunc is used to handle the "/" URL which is the default ons
func ShowAllTasksFunc(w http.ResponseWriter, r *http.Request) {
    if r.Method == "GET" {
        context := db.GetTasks("pending") //true when you want non deleted notes
        if message != "" {
            context.Message = message
        }
        context.CSRFToken = "abcd"
        message = ""
        expiration := time.Now().Add(365 * 24 * time.Hour)
        cookie    :=    http.Cookie{Name: "csrftoken",Value:"abcd",Expires:expiration}
        http.SetCookie(w, &cookie)
        homeTemplate.Execute(w, context)
    } else {
        message = "Method not allowed"
        http.Redirect(w, r, "/", http.StatusFound)
    }
}
```

The below handler handles the POST request sent by our form, it fetches the value of the csrftoken cookie and gets the value of the hidden `CSRFToken` field of the add task form. If the value of the cookie is equal to the value fetched by the form, then we allow it to go to the database.

The call to `ParseForm` will parse the contents of the form into Gettable fields which we can fetch using the Get function. This call is compulsory.

```
//AddTaskFunc is used to handle the addition of new task, "/add" URL
func AddTaskFunc(w http.ResponseWriter, r *http.Request) {
    if r.Method == "POST" {
        r.ParseForm()
        file, handler, err := r.FormFile("uploadfile")
        if err != nil {
            log.Println(err)
        }

        taskPriority, priorityErr := strconv.Atoi(r.FormValue("priority"))
        if priorityErr != nil {
            log.Print("unable to convert priority to integer")
        }
        priorityList := []int{1, 2, 3}
        for _, priority := range priorityList {
            if taskPriority != priority {
                log.Println("incorrect priority sent")
                //might want to log as security incident
                taskPriority=1 //this defaults the priority to low
            }
        }
        title := template.HTMLEscapeString(r.Form.Get("title"))
        content := template.HTMLEscapeString(r.Form.Get("content"))
        formToken := template.HTMLEscapeString(r.Form.Get("CSRFToken"))

        cookie, _ := r.Cookie("csrftoken")
        if formToken == cookie.Value {
            if handler != nil {
                r.ParseMultipartForm(32 << 20) //defined maximum size of file
                defer file.Close()
                f, err := os.OpenFile("./files/"+handler.Filename, os.O_WRONLY|os.O_CR
EATE, 0666)
                if err != nil {
                    log.Println(err)
                    return
                }
                defer f.Close()
                io.Copy(f, file)
                filelink :=
                "<br> <a href=/files/" + handler.Filename + ">" + handler.Filename + "
</a>"
                content = content + filelink
            }
```

```
            truth := db.AddTask(title, content, taskPriority)
            if truth != nil {
                message = "Error adding task"
                log.Println("error adding task to db")
            } else {
                message = "Task added"
                log.Println("added task to db")
            }
            http.Redirect(w, r, "/", http.StatusFound)
        } else {
            log.Fatal("CSRF mismatch")
        }

    } else {
        message = "Method not allowed"
        http.Redirect(w, r, "/", http.StatusFound)
    }
}
```

> Note Cookies
>
> Cookie is a way to store data on the browser, HTTP is a stateless protocol, it wasn't built for sessions, basically the Internet itself wasn't built considering security in mind since initially it was just a way to share documents online, hence HTTP is stateless, when the web server receives requests, it can't distinguish between two consequitive requests, hence the concept of cookies were added, thus while starting a session, we generate a session ID and store it on the database in memory or on the database and we store the same ID on a cookie on the web browser and we validate both of them to authenticate them. We have to note that, if we set an expiry date for a cookie, then it is stored on the filesystem otherwise it is stored in memory in the browser. In the *incognito* mode, this is the case, all the cookies are stored in memory and not in the filesystem.

### HTTP Request

```
Host: localhost:8080
User-Agent: .....
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://localhost:8080/
Cookie: csrftoken=abcd
Connection: keep-alive
Cache-Control: max-age=0
```

The browser, while sending a response appends all the cookie data stored in its memory or file along with the other aspects of the HTTP request so we can access the cookie as `r.Cookie`, it'll contain *every* cookie for that particular *domain*, we'd then loop through it to fetch the data which we want.

This is important for security reasons, suppose I set a cookie on my `imaginedomain.com` and if it contains a csrf token and if that cookie is accessible to some other webapp then it is horrible since they can masquerade as any legitmate user. But this ain't possible, since a website can only access the cookie stored for its own domain, plus we have the feature to set HTTP only value of a cookie as true, which says that even javascript can't access the cookies.

HTTP Response

```
Content-Type: text/html; charset=utf-8
Date: Tue, 12 Jan 2016 16:43:53 GMT
Set-Cookie: csrftoken=abcd; Expires=Wed, 11 Jan 2017 16:43:53 GMT
Transfer-Encoding: chunked
```

When we set cookies, we write then to the HTTP response which we send to the client and the browser reads the Cookie information and stores them in the memory or the filesystem depending on the `Expires` field of the response.

From the go documentation:

```
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
}
```

# Input Validation

The basic aspect of web applications is that no data can be trusted, even if the user isn't malicious herself, there are many ways to trick the browser into sending HTTP requests and fooling the web server to respond to what seems like a legitimate request. Hence we have to verify everything that comes from the user.

One might argue here that we do all sorts of validation using javascript, but there are ways to evade JS validations, the simplest way is to disable JS and more sophisticated ways are to manipulate the HTTP request before the browser sends it, it is literally trivial when we use just the web developer tools that these days browsers provide.

```
if formToken == cookie.Value and title != nil and content!=nil
```

The title and content of the task is mandatory, hence we added the not nil part.

We do input validation when we don't want junk data to go into our database, suppose the user hit the submit button twice, or some other scenario. But we also have to consider the case where the user wants to run some script on our website, which is dangerous, so we use the `template.HTMLEscapeString` method to escape what might run in memory of the current browser session. Even the data which comes from your drop down list should be validated.

Everything in a web form is sent via a Request as we saw in the above example, we use a drop down list when we have are expecting a particular set of inputs but for someone who knows what firefox dev tools are, can easily modify anything in the request, for example we have the priority as drop down list we might think that since we have only three entries in the drop down list, should we keep a check in our view handler to not accept anything but the three values which are present in our template. We ought to keep a check like below:

```
priorityList := []int{1, 2, 3}
for _, priority := range priorityList {
    if taskPriority != priority {
        log.Println("incorrect priority sent")
        //might want to log as security incident
    }
}
```

This is the priority field of our request

```
    Content-Disposition: form-data; name="priority"
        3
```

All a malicious user has to do is change this value and resend the request, they can literally insert any value here, just think what if they send `rm -fr *.*` and accidentally enough this command is executed on our server. This also brings in another aspect of security, *never* run your machine in root mode, always keep the root mode for admin tasks and use a non root mode. Even if that isn't the case, a less dangerous example will be sending a huge number with the request, assuming that we have used integer as our variable, the program might crash if it has to handle a number beyond its storage capacity. This might be termed as a denial of service attack.

**Links**

-Previous section -Next section

# Uploading files

Uploading files is the next step in form processing, in case of files, we send the entire file data in the HTTP header, so we have to set the form encoding to `enctype="multipart/form-data"` . This will inform our server that we are going to get a file from the form along with the rest of the fields, if any.

This means we can get either either file(s) and data or just file(s) or just data and no file(s).

At the first line of our HTTP handler, we have to write this line, if this line is not present in the first line then it gives unexpected results

```
        file, handler, err := r.FormFile("uploadfile")

        if handler != nil {
                r.ParseMultipartForm(32 << 20) //defined maximum size of file
                defer file.Close()
                f, err := os.OpenFile("./files/"+handler.Filename, os.O_WRONLY|os.O_CR
EATE, 0666)
                if err != nil {
                    log.Println(err)
                    return
                }
                defer f.Close()
                io.Copy(f, file)
                filelink := "<br> <a href=./files/"+handler.Filename+">"+ handler.File
name+"</a>"
                content =  content + filelink
        }
```

We first provide the maximum size of the file which is 32 ^ 20, which is gigantic for our webapp, not everyone has the Internet infrastructure to upload that big a file, but we want to be flexible.

We basically get a file from the form request, in the form handler we open another file with the same/different name and then read the file form the request and write it on the server. We need to handle the scene where we name the file differently so we'd need to store the old file name -> new file name relation somewhere, it can be a database table.

The file name should be scrubbed, since the user can give any malicious name to damage our application.

We now want to randomize the file name of the files which the users upload. In your `AddTaskFunc` add the following lines

```
    if handler != nil {
            r.ParseMultipartForm(32 << 20) //defined maximum size of file
            defer file.Close()
            randomFileName := md5.New()
            io.WriteString(randomFileName, strconv.FormatInt(time.Now().Unix(), 10))
            io.WriteString(randomFileName, handler.Filename)
            token := fmt.Sprintf("%x", randomFileName.Sum(nil))
            f, err := os.OpenFile("./files/"+token, os.O_WRONLY|os.O_CREATE, 0666)
            if err != nil {
                log.Println(err)
                return
            }
            defer f.Close()
            io.Copy(f, file)

            filelink := "<br> <a href=/files/" + token + ">" + handler.Filename + "</a
>"
            content = content + filelink

            fileTruth := db.AddFile(handler.Filename, token)
            if fileTruth != nil {
                message = "Error adding filename in db"
                log.Println("error adding task to db")
            }
        }
```

file `~/Tasks/db/db.go`

```
    // AddFile is used to add the md5 of a file name which is uploaded to our applicat
ion
    // this will enable us to randomize the URL without worrying about the file names
    func AddFile(fileName, token string) error {
        SQL, err := database.Prepare("insert into files values(?,?)")
        if err != nil {
            log.Println(err)
        }
        tx, err := database.Begin()

        if err != nil {
            log.Println(err)
        }
        _, err = tx.Stmt(SQL).Exec(fileName, token)
        if err != nil {
            log.Println(err)
            tx.Rollback()
        } else {
            log.Println(tx.Commit())
        }
        return err
    }
```

table structure

```
CREATE TABLE files(name varchar(1000) not null, autoName varchar(255) not null);
```

These block of code do the following things:

1. Create a version of name for each uploaded file
2. Insert it in database
3. Reference the file as `/files/<randomName>`
4. File is now referenced by our name rather than the user supplied name

The next part to do is registering the `/files/` handler.

file: `~/Tasks/views/views.go`

```
    // UploadedFileHandler is used to handle the uploaded file related requests
    func UploadedFileHandler(w http.ResponseWriter, r *http.Request) {
        if r.Method == "GET" {
            log.Println("into the handler")
            token := r.URL.Path[len("/files/"):]

            //file, err := db.GetFileName(token)
            //if err != nil {
            log.Println("serving file ./files/" + token)
            http.ServeFile(w, r, "./files/"+token)
            //}
        }
    }
```

**Links**

-Previous section -Next section

# Templates

package: `text/template`

In the first chapter we had a cursory glance over the concept of templates. This chapter is dedicated entirely to templates. As we said previously, a web application responds to certain URLs and gives an html page to the browser which the browser then interprets and shows to the end user. This html page which is sent to the browser is what is called templates in the backend for we have a template which stores some variables, and in real time data is provided into the template which makes it a complete html page.

Let's take a practical example. Suppose we are building a micro blogging site. We would start with creating the front end in html. Our microblogging site will show `Hi User` on the right corner.

In our static html we write this `<p>Hi User</p>`

But if we serve this page on our webserver it'll not change anything, it'll show `Hi User`, the name of the user won't come magically, we have to put it into the page somehow, here we use a variable so in Go we'd approach this by using a variable named `{{.Name}}` so our html now will be `<p>Hi {{.Name}}</p>`. The `.` is mandatory.

Now, this is the logic we apply to all our html pages, keeping such `{{}}` variable expansion parameters and serving the value of the parameter while executing the template. If you are wondering how we'd do that, this is the syntax

```
    homeTemplate.Execute(w, context)

    //Context is the struct passed to templates
    type Context struct {
        Tasks      []Task
        Name       string
        Search     string
        Message    string
    }
```

These are the three parts of using templates, first you need to create types like we have created the `Context` type, then we need to read the template, then we need to use the components of that type in our template file. So what remains now is passing an object of that type during template execution.

Every template requires the context object because that is what defines the data to be populated in the template.

Reading template:

```
templates, err = template.Must(template.ParseFiles(allFiles...))
```

**template.Must**

Must is a helper that wraps a call to a function returning (*Template, error) and panics if the error is non-nil where allFiles is populated as below:

```
var allFiles []string
templatesDir := "./public/templates/"
files, err := ioutil.ReadDir(templatesDir)
if err != nil {
    fmt.Println("Error reading template dir")
}
for _, file := range files {
    filename := file.Name()
    if strings.HasSuffix(filename, ".html") {
        allFiles = append(allFiles, templatesDir+filename)
    }
}
```

For the sake of demonstration of how to parse multiple files we have used the `ParseFiles` method to parse all the `.html` files, you can use the `ParseGlob` method which is available in the standard library.

```
`template.Must(template.ParseGlob(templatesDir + "*.html"))`
```

The definition of ParseGlob is:

```
`func ParseGlob(pattern string) (*Template, error)`
```

We have to specify the Pattern for the ParseGlob function, but we have passed the path and the pattern because just passing the pattern is useless, we need the path where the pattern will be applied to find the list of all files meeting the criteria.

> Note

1. `...` operator : allFiles is a string slice and allFiles... passes the function a parameter as a string.
2. ParseFiles performance:

There is one point to note here about performance in parsing files, typically a template file won't change until there is some major change to the codebase so we should only parse the files **once**, rather than keep this code in each view handler and doing a

```
template.ParseFiles("home.html")
template.Execute(w, context)
```

This block of code will unnecessarily read the html page each time while serving the response of a request, which means if ten people are using our blogging site then for each page they visit the code will read the html page, but there is no need to do things this way, we can read the html files once at the start and then use the `Lookup` method of the template class,

```
homeTemplate = templates.Lookup("home.html")
homeTemplate.Execute(w, context)
```

# Sub templating

We learnt how to pass data to templates and display it in the html page, it so happens that a lot of code is used in all templates suppose the navigation bar or the header, then we need not write the same chunk everywhere, we can create a template to store that, and use sub templating `{{template "_head.html" .}}`

Here, we have identified a chunk of code which we want to replicate and put it in `_head.html`. Then we put the above statement in each template file where we wish to have our header. This way templates become a lot smaller and don't contain replicated code everywhere. Do note that the `.` before the first `}` is intentional and it means that all the variables which were passed to the current template are passed to the sub template.

The sub templates which we create depends on our requirement, but the basic point behind it is that if we are going to repeat a block of HTML code then we should form it as a template.

The main point to note over here is that when we are going to use our templates or sub templates, all those html files need to be **parsed**. In templating we have a variable which stores *all* templates even if we aren't going to refer to that directly in our code using the `Lookup` method on the template variable. The lookup method takes the name of the template. When the `{{template _head.html .}}` is evaluated, it goes to our template variable and tries to find out the template parsed with the exact name, if it is not present then it doesn't complain by default, we should use `Must` method if we want it to complain.

# Example

file `views/views.go`

```
package views

        import (
            "io/ioutil"
            "net/http"
            "os"
            "strconv"
            "strings"
            "text/template"
        )

        var (
            homeTemplate      *template.Template
            deletedTemplate   *template.Template
            completedTemplate *template.Template
            loginTemplate      *template.Template
            editTemplate      *template.Template
            searchTemplate    *template.Template
            templates         *template.Template
            message           string
            //message will store the message to be shown as notification
            err               error
        )

        //PopulateTemplates is used to parse all templates present in
        //the templates folder
        func PopulateTemplates() {
            var allFiles []string
            templatesDir := "./public/templates/"
            files, err := ioutil.ReadDir(templatesDir)
            if err != nil {
                fmt.Println("Error reading template dir")
            }
            for _, file := range files {
                filename := file.Name()
                if strings.HasSuffix(filename, ".html") {
                    allFiles = append(allFiles, templatesDir+filename)
                }
            }

            if err != nil {
                fmt.Println(err)
                os.Exit(1)
            }
            templates, err = template.Must(template.ParseFiles(allFiles...))
            // templates, err := template.Must(template.ParseGlob(templatesDir+".html"
))
            if err != nil {
```

```
                fmt.Println(err)
                os.Exit(1)
            }
            homeTemplate = templates.Lookup("home.html")
            deletedTemplate = templates.Lookup("deleted.html")

            editTemplate = templates.Lookup("edit.html")
            searchTemplate = templates.Lookup("search.html")
            completedTemplate = templates.Lookup("completed.html")
            loginTemplate = templates.Lookup("login.html")

        }

        //ShowAllTasksFunc is used to handle the "/" URL
        //TODO add http404 error
        func ShowAllTasksFunc(w http.ResponseWriter, r *http.Request) {
            if r.Method == "GET" {
                context := db.GetTasks("pending")
                //true when you want non deleted notes
                if message != "" {
                    context.Message = message
                }
                homeTemplate.Execute(w, context)
                message = ""
            } else {
                message = "Method not allowed"
                http.Redirect(w, r, "/", http.StatusFound)
            }
        }
```

## Looping through arrays

```
<div class="timeline">
    {{ if .Tasks}}
        {{range .Tasks}}
            <div class="note">
            <p class="noteHeading">{{.Title}}</p>
            <hr>
            <p class="noteContent">{{.Content}}</p>
            </div>
        {{end}}
    {{else}}
        <div class="note">
            <p class="noteHeading">No Tasks here</p>
            <p class="notefooter">Create new task<button> here </button> </p>
        </div>
    {{end}}
</div>
```

The `{{ if .Tasks}}` block checks if the array is empty or not, if it is not then it'll go to the `{{ .range .Tasks }}` which will loop through the array, then the `{{.Title}}` will access the title and `{{.Content}}` will access the `Content` of that particular Task instance and we'll see all the tasks as a list.

## Template variables

We have this scenario, we have a range of categories in the navigation drawer and if we visit the `/category/study`, then our category name should be highlighted in the navigation drawer. We do this by storing the value of the .Navigation field - which tells if it is a Edit page/Trash page/Category page

```
{{$nav := .Navigation}}
{{ range $index, $cat := .Categories }}
    <li class="sidebar-item">
        <a href="/category/{{$cat.Name}}" {{ if eq $cat.Name $nav }} class="active" {{
end}}>
            <span class="nav-item"> {{$cat.Name}}</span> <span class="badge pull-right
">{{$cat.Count}}</span></a>
    </li>
{{end}}
```

#### #Creating variables

`{{$url := ""}}` will create a blank variable, one has to not that all variables are practically strings once they are rendered. `{{$url := .Navigation}}` will create a new variable and initialize it with the value of `.Navigation`

For understanding why template variables are required, we need to go into the above block of code, when we are using the `range` operator, we are parsing the array and the range block gets the elements of the block by default.

My Context type is

```
type Context struct {
    Tasks      []Task
    Navigation string
    Search     string
    Message    string
    CSRFToken  string
    Categories []CategoryCount
    Referer    string
}
```

Hence when we do a `{{range .Categories}}` we will be accessing each element as `{{.}}` per loop. If we use any other valid variable here, like the .Navigation, then the block tries to find the .Navigation *inside* .Categories, which obviously isn't present.

Now we need to make the page aware of which category it is showing, should the user go to the `/categories/` page.

The logic behind making that page category aware is that we create a CSS class to mark that particular category as *active*, but for that, we'd need to access the Category name and Navigation *within* the `range .Categories` block, thus we create two variables, one to store the category name from the `.Navigation` variable and use the if statement like

```
{{if eq $cat $nav}} class="active" {{end}}
```

This will mark only that particular category as active and not all of them.

In templating logic the operator is first and then the two operands.

eq: equal, le: less than equal, ge: greater than equal

You can also use if else clause like below:

```
{{$url := ""}}
<div class="navbar-header">
    {{if .Search}} <a class="navbar-brand"> Results for: {{.Search}}</a> {{else}} {{ i
f eq .Navigation "pending"}}
    {{ $url:="" }} {{else if eq .Navigation "completed"}} {{ $url := "" }} {{else if e
q .Navigation "deleted"}}
    {{$url := ""}} {{else if eq .Navigation "edit"}} {{$url := ""}} {{else}} {{$url :=
 "/category"}}{{end}}

    <p class="navbar-brand" href="{{$url}}/{{.Navigation}}">
        {{if eq .Navigation "pending"}} Pending {{ else if eq .Navigation "completed"}
}Completed {{ else if eq .Navigation "deleted"}}Deleted
        {{ else if eq .Navigation "edit"}} Edit {{else }} {{.Navigation}} {{end}} {{en
d}}
    </p>
```

Here we had some complicated stuff, if our page is a search one, we had to show `Results for : <query>` , pending, deleted,edit, completed for respective and /category/ if we are in the category. So we defined an empty URL and assigned the URL values according to the complicated if else structure.

# Homework

1. Take the html pages from http://github.com/thewhitetulip/omninotesweb and modify them to suit our purposes We would need to create one template each for the ones we mentioned in the above variable declaration, use templating as far as possible and later check your results with http://github.com/thewhitetulip/Tasks, please do the exercise on your own first and then only check the Tasks repository.
2. Implement a search interface. Take a query as input, search tasks for that query and return an html page with the query highlighted in the resulting page.

**Links**

-Previous section -Next section

# Authentication

Authentication is used to verify if the users have access to that particular part of your web application. For understanding how to implement authentication we need to understand what happens behind the scenes of a browser. Suppose we run a bank webapplication. We want only our legitimate users to access our webapplication. We set up a login page and provide our users with their username and password which they can use to validate their claim to our webapp.

When we submit the login form, the browser takes our username, password and sends a POST request to the webserver, which again responds with a HTTP redirect response and we are returned to our bank dashboard.

The HTTP protocol is stateless, which means every request is unique. There is no way for identifying automatically if a request is related to another request. This brings about the problem of authentication, how then can we validate if the users have access to our webapp?

We can send the username along with each HTTP request, either in the URL via a GET request or in the POST request. But this is inefficient since for each request, the webserver would need to hit the database to validate the username, also this would mean weak security since if I know your username, I can impersonate you pretty easily and the webserver is helpless to identify this impersonation.

To solve this problems Sessions were invented, sessions need to use cookies on the browser to function. The basic idea is that the server generates a sessionID and stores it in a cookie. With subsequent requests, the browser will send the sessionID along with the request, the webserver will then come to know from that sessionID if the request is a fake one or not. Also we get to know who the user is from that.

# Cookies

Cookies, as we saw in a previous chapter can be used to store a key,value pair. We used a cookie to store the CSRF token, the cookie had the name as CSRF and value as the token.

Please don't confuse sessions with cookies, because sessions aren't a key,value pair. Sessions are a way of working with cookies on the server side. There is a gap of the entire Internet between sessions and cookies.

Cookies are stored in our browsers, for security reasons we need to enable the "isHTTPOnly" field of our cookies, so only our webapplication can read the cookie. Otherwise anyone javascript application can easily read our cookie defeating its purpose, we might as well not keep an authentication mechanism for our webapp.

From the go documentation

```
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
}
```

The `domain` of our cookie enables a restricted access to our cookie. A visitor goes to our fictional bank website, sbank.com and enters a username and password, a cookie is stored in the browser which only the sbank.com domain can access since we are security aware and we have set the HttpOnly field to true while setting the cookie. This means some malicious website which is set up by an attacker to intentionally target our bank isn't able to access the cookie using javascript.

One has to remember that cookie is nothing but a file stored in a user's browser, if can be accessed over HTTP by our webserver, a client browser does allow access to it through browser settings or custom javascript. The browser, after all is a platform, and we have APIs to that platform.

# Sessions

A session is a series of actions performed between you and the webapp you are acting, enabled by the browser and the Internet you are using.

While generating a new session, we need to check if a sessions is already active, if so we return the same session ID rather than create a new one, if not, we generate a new session ID. Session IDs need to be sufficiently random. Of course we can't generate something totally random, but we have to ensure to generate something that nobody else can replicate, unless they have access to our private key which we use to generate our random number.

## Session handling using gorilla/sessions

Till now we never used any third party library or a framework in this book, this is for the first time that we are doing so, we better use libraries for handling sessions, since security is the primary aspect of any web application it is wiser to use existing (and good) packages to manage security.

Path: `~/Tasks/sessions/sessions.go`

```
package sessions

import (
    "net/http"

    "github.com/gorilla/sessions"
)

//Store the cookie store which is going to store session data in the cookie
var Store = sessions.NewCookieStore([]byte("secret-password"))

//IsLoggedIn will check if the user has an active session and return True
func IsLoggedIn(r *http.Request) bool {
    session, _ := Store.Get(r, "session")
    if session.Values["loggedin"] == "true" {
        return true
    }
    return false
}
```

This is the sessions package which we will use in our application.

We create a CookieStore which stores the sessions information under the "sessions" in the browser. We get the session ID stored under the session cookie and store it the `session` variable. When it comes to using this function, we have the `AddCommentFunc` view below which is going to handle the commenting feature of our application, it'll first check if the user has an active session and if so, it'll handle the POST request to add a comment, if not, it'll redirect the user to the login page.

Path: `~/Tasks/Views/addViews.go`

```
//AddCommentFunc will be used
func AddCommentFunc(w http.ResponseWriter, r *http.Request) {
    if sessions.IsLoggedIn(r) {
        if r.Method == "POST" {
            r.ParseForm()
            text := r.Form.Get("commentText")
            id := r.Form.Get("taskID")

            idInt, err := strconv.Atoi(id)

            if (err != nil) || (text == "") {
                log.Println("unable to convert into integer")
                message = "Error adding comment"
            } else {
                err = db.AddComments(idInt, text)

                if err != nil {
                    log.Println("unable to insert into db")
                    message = "Comment not added"
                } else {
                    message = "Comment added"
                }
            }

            http.Redirect(w, r, "/", http.StatusFound)

        }
    } else {
        http.Redirect(w, r, "/login", 302)
    }
}
```

The below file contains the code to login and logout of our application, we basically are going to set the "loggedin" property of our cookiestore.

Path: `~/Tasks/Views/sessionViews.go`

```
package views

import (
    "net/http"

    "github.com/thewhitetulip/Tasks/sessions"
)

//LogoutFunc Implements the logout functionality.
//WIll delete the session information from the cookie store
func LogoutFunc(w http.ResponseWriter, r *http.Request) {
    session, err := sessions.Store.Get(r, "session")
    if err == nil { //If there is no error, then remove session
        if session.Values["loggedin"] != "false" {
            session.Values["loggedin"] = "false"
            session.Save(r, w)
        }
    }
    http.Redirect(w, r, "/login", 302)
    //redirect to login irrespective of error or not
}

//LoginFunc implements the login functionality, will
//add a cookie to the cookie store for managing authentication
func LoginFunc(w http.ResponseWriter, r *http.Request) {
    session, err := sessions.Store.Get(r, "session")

    if err != nil {
        loginTemplate.Execute(w, nil)
        // in case of error during
        // fetching session info, execute login template
    } else {
        isLoggedIn := session.Values["loggedin"]
        if isLoggedIn != "true" {
            if r.Method == "POST" {
                if r.FormValue("password") == "secret"
                   && r.FormValue("username") == "user" {
                    session.Values["loggedin"] = "true"
                    session.Save(r, w)
                    http.Redirect(w, r, "/", 302)
                    return
                }
            } else if r.Method == "GET" {
                loginTemplate.Execute(w, nil)
            }
        } else {
            http.Redirect(w, r, "/", 302)
        }
    }
}
```

There is a better way of handling sessions using middleware, we'll introduce that concept in the next chapter.

# Users

## Signing users up

The above example just hardcodes the username and password, of course we'd want people to sign up to our service. We create a user table.

```
CREATE TABLE user (
    id integer primary key autoincrement,
    username varchar(100),
    password varchar(1000),
    email varchar(100)
);
```

For the sake of simplicity, it'll only contain ID, username, password and email.

There are two parts here, first one where we allow users to sign up, and another part where we replace the hard coded username and password in our login logic.

file: `~/Tasks/main.go`

```
http.HandleFunc("/signup/", views.SignUpFunc)
```

file: `~/Tasks/views/sessionViews.go`

```
//SignUpFunc will enable new users to sign up to our service
func SignUpFunc(w http.ResponseWriter, r *http.Request) {
    if r.Method == "POST" {
        r.ParseForm()

        username := r.Form.Get("username")
        password := r.Form.Get("password")
        email := r.Form.Get("email")

        log.Println(username, password, email)

        err := db.CreateUser(username, password, email)
        if err != nil {
            http.Error(w, "Unable to sign user up", http.StatusInternalServerError)
        } else {
            http.Redirect(w, r, "/login/", 302)
        }
    }
}
```

file: `~/Tasks/db/user.go`

```
//CreateUser will create a new user, take as input the parameters and
//insert it into database
func CreateUser(username, password, email string) error {
    err := taskQuery("insert into user(username, password, email) values(?,?,?)", user
name, password, email)
    return err
}
```

We saw TaskQuery in our chapter on DB, it is a simple wrapper around db.Exec().

> Note: In a real web app, you'd want to encrypt the password and not store it in plain text, this is a dummy app which'll never see the light of the day so I am keeping it plaintext.

## Login

file `~/Tasks/views/sessionViews.go`

```
//LoginFunc implements the login functionality, will add a cookie to the cookie store
for managing authentication
func LoginFunc(w http.ResponseWriter, r *http.Request) {
    session, err := sessions.Store.Get(r, "session")

    if err != nil {
        log.Println("error identifying session")
        loginTemplate.Execute(w, nil)
        return
    }

    switch r.Method {
    case "GET":
        loginTemplate.Execute(w, nil)
    case "POST":
        log.Print("Inside POST")
        r.ParseForm()
        username := r.Form.Get("username")
        password := r.Form.Get("password")

        if (username != "" && password != "") && db.ValidUser(username, password) {
            session.Values["loggedin"] = "true"
            session.Values["username"] = username
            session.Save(r, w)
            log.Print("user ", username, " is authenticated")
            http.Redirect(w, r, "/", 302)
            return
        }
        log.Print("Invalid user " + username)
        loginTemplate.Execute(w, nil)
    }
}
```

file `~/Tasks/db/user.go`

```
//ValidUser will check if the user exists in db and if exists if the username password
//combination is valid
func ValidUser(username, password string) bool {
    var passwordFromDB string
    userSQL := "select password from user where username=?"
    log.Print("validating user ", username)
    rows := database.query(userSQL, username)

    defer rows.Close()
    if rows.Next() {
        err := rows.Scan(&passwordFromDB)
        if err != nil {
            return false
        }
    }
    //If the password matches, return true
    if password == passwordFromDB {
        return true
    }
    //by default return false
    return false
}
```

> Note: Since we are using gorilla/sessions, we just have to plugin the functionality that the package provides and don't have to bother about the actual implementation of sessions handling, if you are interested then by all means go ahead and checkout the source code of gorilla/sessions!

We keep resetting the password as an exercise, formulate a mechanism to resetting the password, requires us to create a user table and store some profile information, either email-ID from where we'll send a security code or by doing something else! Brainstorm!

### Links

-Previous section -Next section

# Files

JSON and XML are two of the most common ways to transmit data between web applications. We'll use JSON for our configuration file.

For our webapplication we have a set of configuration values like the server port where our application will run. Suppose you are developing your application in $GOPATH and also using it somewhere else, then you can't run them in parallel because both sources use the same port number. Naturally we want a way to parameterize that port number. The parameter or configuration value list may contain more things like database connection information. As of now we will use a `config.json` file and read the serverPort variable and bind our server on that port.

Our configuration file uses a fixed structure, hence it is simple enough to UnMarshal it to a struct type, we can use some advance concepts to accomodate unstructured JSON files, because that is the whole point of JSON, we can have data in an unstructured format.

NoSQL has been famous lately, they are basically JSON document stores. We have projects like `boltdb` which store data in a key value pair, ultimately in flat files or in memory.

file: `$GOPATH/src/github.com/thewhitetulip/Tasks/config/config.go`

```
package config

import (
    "encoding/json"
    "io/ioutil"
    "log"
)

// Stores the main configuration for the application
// define a struct object containing
type Configuration struct {
    ServerPort string
}

var err error
var config Configuration

// ReadConfig will read the config.json file to read the parameters
// which will be passed in the config object
func ReadConfig(fileName string) Configuration {
    configFile, err := ioutil.ReadFile(fileName)
    if err != nil {
        log.Fatal("Unable to read log file")
    }
    //log.Print(configFile)
    err = json.Unmarshal(configFile, &config)
    if err != nil {
        log.Print(err)
    }
    return config
}
```

file: $GOPATH/src/github.com/thewhitetulip/Tasks/config.json

```
{
    "ServerPort": ":8081"
}
```

file: `$GOPATH/src/github.com/thewhitetulip/Tasks/main.go`

```
values := config.ReadConfig("config.json")

// values is the object now, we can use the
// below statement to access the port name

values.ServerPort
```

We use the `json.Unmarshal` to read the JSON file into our structure object. This is a very simple and basic example of parsing JSON files, you can have nested structures of many levels inside the main config object, but that is the features of Go, so long as it can be represented as a JSON document you can use the `Unmarshal` method to translate the file into an object which you can use in your program.

### Homework

- Alter the config.json file to take the name of the sqlite database as a configuration parameter.
- Read about the JSON library in godoc

### Links

-Previous section -Next section

# Routing

Till now we used routing directly inside of our handlers. For a large application though, it'd be better to have a router in place. We can either use a third party one with countless features or the standard mux.

As our application matures, routing plays a big role into it. We intentionally avoided routing till now because as a web developer, we must understand what happens in the background of our application. Web frameworks allow us to build applications quickly, but the downside of them is that they provide us with a *framework* as the name suggests, which means you are totally restricted by the API which the framework will provide. Thus we need to know how to implement bare bone stuff, so in future we might want to modify the framework we need, or rather create our own one.

First of all we'd need to install the httprouter, do a `go get -u github.com/julienschmidt/httprouter`

## From the documentation

Package httprouter is a trie based high performance HTTP request router.

```
package main

import (
    "fmt"
    "github.com/julienschmidt/httprouter"
    "net/http"
    "log"
)

func Index(w http.ResponseWriter, r *http.Request, _ httprouter.Params) {
    fmt.Fprint(w, "Welcome!\n")
}

func Hello(w http.ResponseWriter, r *http.Request, ps httprouter.Params) {
    fmt.Fprintf(w, "hello, %s!\n", ps.ByName("name"))
}

func main() {
    router := httprouter.New()  // creates a new router
    router.GET("/", Index)      // will direct the GET / request to the Index function
    router.GET("/hello/:name", Hello) // will redirect the GET /name to Hello, stores
the name of the parameter
                                      // in the a variable of httprouter.Params

    log.Fatal(http.ListenAndServe(":8080", router))
}
```

httprouter uses a custom `Http.HandleFunc` method to accomodate parameterized routing.

Here, we can route our requests depending on the HTTP method through which it is used, and we can handle parameterized routing for free.

To handle that scenario we were initially using the `r.URL.Path` variable and then extracting the parameters, it is just the solution of finding the variable parameter of the URL, there is a wide area where httprouter is awesome, there is no need to handle the / for each requests.

In web applications sometimes having a forward slash is very critical, the URL index is totally different from /index and is marginally different from `/index/` httprouter takes care of the trailing slashes. The Go default MUX requires you to handle the routes in the descending order, meaning the most generic URL the "/" should be at the bottom and the least generic should be at the top, as it goes sequentially up and down matching the URLs, httprouter provides a wide variety of advantages when it comes to routing.

This is because our application should separate routing from the handler logic, we were mixing it just to get a feel for how to program in Go, but eventually when the app grows, it is tiresome and non maintainable for doing the `if r.Method==POST` check in each handler,

rather than that we can have 3 different handlers one for when request comes via AJAX, one for normal GET and one for normal POST. This way we do not have one function handler checking the type of request and then writing three separate logic inside one function.

But that doesn't mean we *have* to use httprouter, if in an app there aren't much sophisticated routing required then we can use the default Mux since it is good enough, but if we have complicated routing then httprouter is the best.

#### Homework

Read the httprouter documentation and source code to get a deeper understanding of routers, since routers are an integral part of any web application. Then rewrite our application using the httprouter.

### Links

-Previous section -Next section

# Middlewares

Middleware is really anything that extends your webapp in a modular way. Most common examples are probably parsing the request parameters/body and storing them in an easily-accessible format so you don't have to do it in every single handler, or session handling as we mentioned in the previous chapter. Other examples could be throttling or IP filtering, which would also happen before you start building your response, or compression, which would happen after you've built your response.

```
//RequiresLogin is a middleware which will be used for each
//httpHandler to check if there is any active session
func RequiresLogin(handler func(w http.ResponseWriter, r *http.Request))
    func(w http.ResponseWriter, r *http.Request) {
    return func(w http.ResponseWriter, r *http.Request) {
        if !sessions.IsLoggedIn(r) {
            http.Redirect(w, r, "/login/", 302)
            return
        }
        handler(w, r)
    }
}
```

The above function counts as middleware - it doesn't know anything about your app except how you handle sessions. If you're not logged in, it redirects, otherwise it doesn't do anything and passes along to the next handler. That next handler might be where you actually build your response, or it could be another middleware component that does something else first.

To know someone's logged in, yes, you want to create a session identifier and store that somewhere on the server side (in memory or a database) and also set it in the user's cookies. Your session IDs should be sufficiently random and long that they couldn't be easily guessed. I think a common way of satisfying that is creating a UUID and then base64 encode that. Or, you could just generate a bunch of random bytes.

To know which user is logged in, the session ID should be the key that maps to a user ID. So, you'd make a map of Session ID => User ID, or something similar in your database.

Then, before every request, you'd

1. Check user's cookies for Session ID. If none, user is not logged in.
2. Check your store for user's Session ID. If it's not found, then it's invalid - user is not logged in.
3. If you found it, use it to look up the user's ID. User is now logged in.

4. Now you've got the user ID and can use it as a filter when querying your DB if you only want to show that user's tasks.

# Example

Without middleware:

```
//IsLoggedIn will check if the user has an active session and return True
func IsLoggedIn(r *http.Request) bool {
    session, _ := Store.Get(r, "session")
    if session.Values["loggedin"] == "true" {
        return true
    }
    return false
}


//SearchTaskFunc is used to handle the /search/ url, handles the search function
func SearchTaskFunc(w http.ResponseWriter, r *http.Request) {
    if sessions.IsLoggedin(r) {
        if r.Method == "POST" {
            r.ParseForm()
            query := r.Form.Get("query")
            context := db.SearchTask(query)
            categories := db.GetCategories()
            context.Categories = categories
            searchTemplate.Execute(w, context)
        }
    } else {
        http.Redirect(w, r, "/login/", 302)
    }
```

With Middleware:

```
http.HandleFunc("/", views.RequiresLogin(views.ShowAllTasksFunc))

//SearchTaskFunc is used to handle the /search/ url,
//handles the search function
func SearchTaskFunc(w http.ResponseWriter, r *http.Request) {
    if r.Method == "POST" {
        r.ParseForm()
        query := r.Form.Get("query")
        context := db.SearchTask(query)
        categories := db.GetCategories()
        context.Categories = categories
        searchTemplate.Execute(w, context)
     }
}
```

This way, we do not have to repeat the `if sessions.IsLoggedin()` block in each of our view which requires authentication. In this example we have used it for session handling, but it can be used for any purpose which requires some kind of pre handling of any view.

**Links**

-Previous section -Next section

# Building an API

API stands for Application Programming Interface, it is just an interface to the webapp. When we use a browser to access a web application, we interact in HTTP and get back HTML pages, which the browser will render for us. Let's say we want to interact with our web app to get some data out of it using a host programming language like Go or Python. We'd have to maintain cookies in Python, or write a python module as an extension of a browser to handle this scenario.

There is a simple way out of this, we equip our web application itself to interact with any host language which can talk in HTTP. This way developers can easily get access to our system, using valid credentials, of course.

Browser:

1. We send the username,password and get a cookie stored on our machine.
2. We use the token in the cookie until it is valid to send HTTP requests.
3. The browser is responsible for rendering the HTML pages sent by the server.

API:

1. We send the username, password and get a token.
2. We send this token in each of our request to the server

Typically we send the token in a custom HTTP header called token.

When we use a browser, the server stores our information as a session, when we send it a request, it is aware of our session. A web app typically uses cookies to store the session ID, which is used to identify the user. Such a server is called a stateful server.

When we write APIs, they are stateless servers, they do not store sessions information anywhere on the server. To it, each request is unique. Which is why, we need to pass along the authentication token in each request.

> Note: Don't mess around with tokens
>
> There are apps where "single sign in" feature is available, the user has to log in only once and they are logged in forever, this is very dangerous. Because if a malicious person gets their hands on the security token, they can send malicious requests for data which look genuine and are impossible to clasify as malicious. Don't do this, always have some expiration time for security tokens, depends on your application really, two hours, six hours, but never infinite hours.

# JWT

Javascript Web Tokens is a standard for generating tokens. We will use the jwt-go library.

Lets start by defining our routes

```
http.HandleFunc("/api/get-task/", views.GetTasksFuncAPI)
http.HandleFunc("/api/get-deleted-task/", views.GetDeletedTaskFuncAPI)
http.HandleFunc("/api/add-task/", views.AddTaskFuncAPI)
http.HandleFunc("/api/update-task/", views.UpdateTaskFuncAPI)
http.HandleFunc("/api/delete-task/", views.DeleteTaskFuncAPI)

http.HandleFunc("/api/get-token/", views.GetTokenHandler)
http.HandleFunc("/api/get-category/", views.GetCategoryFuncAPI)
http.HandleFunc("/api/add-category/", views.AddCategoryFuncAPI)
http.HandleFunc("/api/update-category/", views.UpdateCategoryFuncAPI)
http.HandleFunc("/api/delete-category/", views.DeleteCategoryFuncAPI)
```

file: `$GOPATH/src/github.com/thewhitetulip/Tasks/main.go`

We will have the same URLs for the API, but it'll start with `/api/`

Our logic is that we will send the username and password in a POST request to `/api/get-token/` that will return the token for us.

We are using `jwt-go` version 3. We have to define a custom claims struct which we will use in token generation and token verification. It'll contain the StandardClaims class and whichever extra fields we require.

```
type MyCustomClaims struct {
    Username string `json:"username"`
    jwt.StandardClaims
}
```

file: `$GOPATH/src/github.com/thewhitetulip/Tasks/views/api.go`

```
import "github.com/dgrijalva/jwt-go"
var mySigningKey = []byte("secret")


//GetTokenHandler will get a token for the username and password
func GetTokenHandler(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        w.Write([]byte("Method not allowed"))
        return
    }

    r.ParseForm()
    username := r.Form.Get("username")
    password := r.Form.Get("password")
    log.Println(username, " ", password)
    if username == "" || password == "" {
        w.Write([]byte("Invalid Username or password"))
        return
    }
    if db.ValidUser(username, password) {
        /* Set token claims */

        // Create the Claims
        claims := MyCustomClaims{
            username,
            jwt.StandardClaims{
                ExpiresAt: time.Now().Add(time.Hour * 5).Unix(),
            },
        }

        token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)

        /* Sign the token with our secret */
        tokenString, err := token.SignedString(mySigningKey)
        if err != nil {
            log.Println("Something went wrong with signing token")
            w.Write([]byte("Authentication failed"))
            return
        }

        /* Finally, write the token to the browser window */
        w.Write([]byte(tokenString))
    } else {
        w.Write([]byte("Authentication failed"))
    }
}
```

We need to send a POST request to /api/get-token/ with the username and password in the Form data.

If the HTTP method is anything other than POST, we throw an error, if the username and password is wrong we throw an error, otherwise, we generate a token and send it to the client.

The client will now use this token for future requests, the ValidateToken function will validate the token.

```
//ValidateToken will validate the token
func ValidateToken(myToken string) (bool, string) {
    token, err := jwt.ParseWithClaims(myToken, &MyCustomClaims{}, func(token *jwt.Toke
n) (interface{}, error) {
        return []byte(mySigningKey), nil
    })

    if err != nil {
        return false, ""
    }

    claims := token.Claims.(*MyCustomClaims)
    return token.Valid, claims.Username
}
```

We'll call the Parse method on the token which we receive as a parameter in the function call. The token.Valid field is a boolen variable which is true if the token is valid and false otherwise.

# Making an API call

Making an API call is analogous to our normal view.

```
//GetCategoryFuncAPI will return the categories for the user
//depends on the ID that we get, if we get all, then return all
//categories of the user
func GetCategoryFuncAPI(w http.ResponseWriter, r *http.Request) {
    if r.Method == "GET" {
        var err error
        var message string
        var status types.Status
        //get the custom HTTP header called Token
        token := r.Header["Token"][0]

        w.Header().Set("Content-Type", "application/json; charset=UTF-8")

        IsTokenValid, username := ValidateToken(token)
        //When the token is not valid show the
        //default error JSON document
        if !IsTokenValid {
            status = types.Status
            {
                StatusCode: http.StatusInternalServerError,
                Message: message
            }
            w.WriteHeader(http.StatusInternalServerError)
            //the following statement will write the JSON document to
            //the HTTP ResponseWriter object.
            err = json.NewEncoder(w).Encode(status)

            if err != nil {
                panic(err)
            }
            return
        }

        log.Println("token is valid " + username + " is logged in")
        categories := db.GetCategories(username)
        w.Header().Set("Content-Type", "application/json; charset=UTF-8")

        w.WriteHeader(http.StatusOK)

        err = json.NewEncoder(w).Encode(categories)
        if err != nil {
            panic(err)
        }
    }

}
```

During an API call, we send data in JSON format, for that, we need to set our content-type as application/json, by doing this, even a web browser will detect that it is getting a JSON document. When we need to write a JSON document to the response writer object, we use

the `json.NewEncoder(w).Encode(categories)` method, where categories is our JSON document.

# Formatting a JSON document

This, below, is our Tasks struct, which will be populated as a JSON document when we run our server. As you might know, we can't use Capital letter as the first letter in a JSON title, by convention they should all be small letters. Go has a special way of letting us do that. The example is below, when we write `json:"id"`, we are telling Go that the name of this field in a JSON rendering should be id and not Id. There is another special syntax called omitempty, in some JSON documents you might want some field to not be displayed. It so happens that there are fields which you would want to disappear when their values aren't present, they may not be important or it'd be too clunky to have them as NULL in all JSON documents.

```
type Task struct {
    Id        int       `json:"id"`
    Title     string    `json:"title"`
    Content   string    `json:"content"`
    Created   string    `json:"created"`
    Priority  string    `json:"priority"`
    Category  string    `json:"category"`
    Referer   string    `json:"referer,omitempty"`
    Comments  []Comment `json:"comments,omitempty"`
    IsOverdue bool      `json:"isoverdue, omitempty"`
}
```

# Testing API

We'll use Firefox and RestClient extension to test our API. RestClient allows us to send various requests to our API server, if you are on Chrome, POSTman is the best alternative.

For RestClient to send Form data, set a custom header Name: Content-Type Value: application/x-www-form-urlencoded

Otherwise you'll be sending blank POST requests all the time. The server needs to understand the content type of the data it is getting form the client.

To send the actual form data, example: we have three fields, username, password and name. Then we write it in the body section like this:

username=thewhitetulip&password=password

Also set a custom HTTP header by the Name: token Value: the token which you get in /api/get-token/ call

# Writing an client

APIs are built so that we can write applications on top of our service. APIs allow us to write clients that access our service apart from the browser.

```
package main

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

func main() {
resp, err := http.Get("http://127.0.0.1:8081/")
if err != nil {
    fmt.Println(err)
}

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

When we use the service via a browser, the browser is essentially making the GET and POST calls on our behalf. When we have to write an app to access our service, we have to write those methods in the app.

For executing the above code, we need to ensure that our Tasks application is running on port 8081. When we execute, it must print the login html page on the terminal.

This might be a great example to get started with using HTTP methods from Go, but for our application we require to send POST reqests with content type as form and send the form data as request body.

# Getting the token

We use the PostForm method which will send a Form via a POST method, effectively saving the trouble of setting the content type field for us. If you run the below code, it will print the authentication token on the terminal.

```
import "net/http"
import "net/url"
import "fmt"
import "io/ioutil"

func main() {
    usernamePwd := url.Values{}
    usernamePwd.Set("username", "suraj")
    usernamePwd.Set("password", "suraj")

    resp, err := http.PostForm("http://127.0.0.1:8081/api/get-token/", usernamePwd)
    if err != nil {
        fmt.Println(err)
    }

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

The next steps are to store this token and get the task list.

# Error handling

We should throw a user friendly error message if the Tasks application isn't running in the background. And then exit the client.

# Example

The below code will get a new token and print all the tasks for that particular user.

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "net/url"
```

```
    "os"
)

func main() {
    baseURL := "http://127.0.0.1:8081/"

    usernamePwd := url.Values{}
    usernamePwd.Set("username", "suraj")
    usernamePwd.Set("password", "suraj")

    resp, err := http.PostForm(baseURL+"api/get-token/", usernamePwd)
    if err != nil {
        fmt.Println("Is the server running?")
        os.Exit(1)
    } else {
        fmt.Println("response received")
    }

    defer resp.Body.Close()
    body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        fmt.Println("Error reading body")
    } else {
        fmt.Println("Token received")
    }
    token := string(body)

    client := &http.Client{}
    req, err := http.NewRequest("GET", baseURL+"api/get-task/", nil)

    if err != nil {
        fmt.Println("Unable to form a GET /api/get-task/")
    }

    req.Header.Add("Token", token)
    resp, err = client.Do(req)

    if (err != nil) || (resp.StatusCode != 200) {
        fmt.Println("Something went wrong in the getting a response")
    }

    defer resp.Body.Close()
    body, err = ioutil.ReadAll(resp.Body)
    fmt.Println(string(body))

}
```

# Advanced Usage

We can write a complete command line client for tasks in Go as we saw in this chapter, it would mean implementing a lot of command line flags to get what the user wants us to get and we would want to store the key in a text file to cache it.

## Homework

- Read the unit testing chapter and write a unit test to test the API client rather than using the firefox addon.
- Build a REST api client to Tasks and add/delete notes via the command line. Note that you'll have to modify how things look & not generate HTML. Render just the markdown text.

#### Links

-Previous section -Next section

# Unit Testing

Testing is done to ensure that our application behaves in a predictable manner. Writing Unit tests is the core responsibility of a developer and it is a part of development. While manual testing can be done, to ensure that our app works well, the need for unit testing will arise when we add features to an already built application and deploy it, only to find out that it broke some earlier functionality.

In one line testing is about ensuring our app doesn't break if a user gives a thoughtful but random input. We think of all such edge cases while testing the app and ensure that our app works fine.

A practical example: you wrote your first commandline Vi calculator and you are about to call a VC firm on your path to becoming filthy rich, but your team member finds out that if you divide 4 by 0, your app crashes.

"Of course nobody is going to divide by 0, don't they know math?", but they do

Go has a package called "testing" in the standard library.

While testing webapps, there are two aspects:

1. Testing the look and feel.
2. Testing if the pages displayed are with the valid statusCode.
3. Testing if the app doesn't break when new functionality is added.
4. The data IO works correctly.

The look and feel is a part of UI/UX and that can be done manually. Doesn't require us to write a single line of Go.

## Validation the Status Code for edge cases.

A real life scenario would be, what if a user who has a basic idea of HTTP decides to send you a OPTIONS /add/ request where you only accept either a GET or a POST.

Example: Testing the AddComment function

```
//AddCommentFunc will be used
func AddCommentFunc(w http.ResponseWriter, r *http.Request) {
    if r.Method == "POST" {
        r.ParseForm()
        text := r.Form.Get("commentText")
        id := r.Form.Get("taskID")

        idInt, err := strconv.Atoi(id)

        if (err != nil) || (text == "") {
            log.Println("unable to convert into integer")
            message = "Error adding comment"
        } else {
            username := sessions.GetCurrentUserName(r)
            err = db.AddComments(username, idInt, text)

            if err != nil {
                log.Println("unable to insert into db")
                message = "Comment not added"
            } else {
                message = "Comment added"
            }
        }

        http.Redirect(w, r, "/", http.StatusFound)

    }
}
```

This function will do the following: If the method is POST then take the comment content, extract the post ID and add the comment to the database.

We will test what happens when the user will send a HTTP GET request for the add comment function, which we do not support.

The test case function is:

```
func TestAddCommentWithWrongMethod(t *testing.T) {
    ts := httptest.NewServer(http.HandlerFunc(AddCommentFunc))
    defer ts.Close()
    req, err := http.NewRequest("GET", ts.URL, nil)
    if err != nil {
        t.Errorf("Error occured while constructing request: %s", err)
    }

    w := httptest.NewRecorder()
    AddCommentFunc(w, req)
    if w.Code != http.StatusBadRequest {
        t.Errorf("Actual status: (%d); Expected status:(%d)", w.Code, http.StatusBadRe
quest)
    }
}
```

`func NewServer(handler http.Handler) *Server`

NewServer starts and returns a new Server. The caller should call Close when finished, to shut it down. For testing, we do not need to run a server, we'll start a temporary server using `NewServer` every time that we need it.

Afer that, we have to send a HTTP request, which will be evaluated and a HTTP response will be sent by our handler.

In a regular HTTP application, we send a HTTP request with a URL to the server and the server takes the HTTP Request and generates a HTTP Response. While testing it, we have to mock the request and response. `NewRequest()` will create a request, and `httptest.NewRecorder()` will return an initialized ResponseRecorder.

`func NewRecorder() *ResponseRecorder`

```
[views]  go test . -v -run AddCommentWith
=== RUN   TestAddCommentWithWrongMethod
--- FAIL: TestAddCommentWithWrongMethod (0.00s)
           addViews_test.go:100: Actual status: (200); Expected status:(400)
FAIL
exit status 1
FAIL       github.com/thewhitetulip/Tasks/views       0.084s
```

We can see here that the actual status is 200, which is Okay. But this isn't what we want, for invalid status request, we should send a Bad Request.

We change the first line and add this, modify the rest of the function accordingly:

```
if r.Method != "POST" {
    log.Println(err)
    http.Redirect(w, r, "/", http.StatusBadRequest)
    return
}
```

If the method is anything but a POST, we'll send a error message.

```
[views]  go test . -v -run AddCommentWith
=== RUN   TestAddCommentWithWrongMethod
2016/08/27 14:19:33 <nil>
--- PASS: TestAddCommentWithWrongMethod (0.00s)
PASS
ok         github.com/thewhitetulip/Tasks/views       0.071s
```

### Incremental testing

An application requires suit of test cases. A suit is going to have a lot of test cases. Suppose you already have a hundred test cases and you add `TestValueAdd` test case as a part of a new feature. While developing that feature. While you are testing if the `TestValueAdd` feature works correctly, you want to run only that one test case.

In such cases, you can run it as `go test -run ValueAdd` .

`-run` supports arguments as a regular expression.

After we are satisfied with this testcase, we have to run the entire test suit for integration testing.

### Homework

Just as we tested the handler for wrong method, try running at able driven test for AddTask handler. You'll have to set up a dummy sqlite database, create few edge cases for inserting tasks, run the AddTask method and if the number of tasks isn't what you expect in the sqlite, then the test fails.

**Links**

-Previous section -Next section

# Version Control Basics

> If you know the basics of git, please feel free to skip this chapter. This is for only those who do not know git.

Version control systems are used to keep track of the code changes being done. They can be used in any place which requires to keep track of changes, this book is written in markdown, using git to store tracking information and it is uploaded on github.

We will be using `git` in this chapter, the concepts are same for other version control systems, just the syntaxes might differ.

# Using git

## Init

We call each project a repository. We first need to initialize a git repository.

```
[Tasks] git init
```

This command will create an empty git repository in the current folder you are in. Ideally, this is the root folder of your project, unless your requirement is different.

```
[Tasks] vi main.go
```

## Stage changes

Create an empty file and write something into it.

```
[Tasks] git add main.go
```

Add the file to the git repo, so git knows that it should track it.

We can stage as many files we want, in this step, but the recommended things is to stage and commit one change at a time. Suppose I add a new feature of notifications, I'll make only those changes and commit all the files related **only** to that change. Because, in future if

we want to rollback to the original repo, we'd rollback to the previous commit. As a consequence to this usage, we have an awful number of commits to the repo, but that's fine, since it doesn't matter. Better software managment is what matters.

When we add files to the git repo, the file is *staged*.

## Commit

```
[Tasks] git commit -m "Added initial version of main.go"
```

When we `commit`, we tell git to `finalize` the changes, it creates a unique hash to point to this time i.e. the files which we changed. Commit messages should be short and meaningful.

## Checkout

```
[Tasks] git checkout main.go
```

This unstages all the changes made to this file before the last commit. Use this with caution because it'll remove all the file changes.

## Log

```
[Tasks] git log
```

This prints a log of all commits made to the repo.

## Diff

```
[Tasks] git diff
[Tasks] git diff <commit hash>
```

diff will show the changes made to the file as compared to the file's contents of the last commit.

# Branching

While working on a team project, we are using the same codebase (on the server) with local repos on our individual machines. Adding a new feature to the `master` branch is a bad way of writing code, because if our change fails and we push the changes to the remote server, we make the server's version fail as well. This is why, branching is a good way to write code.

1. Create a new branch per feature.
2. Make changes to the branch.
3. Submit a pull request.
4. Merge the branch only if everything works fine.

By default the branch is master.

```
[Tasks] git branch new-feature
```

Creates a new branch.

```
[Tasks] git branch
* master
  new-feature
```

The branch in use is still the master.

```
[Tasks] git checkout new-feature
```

Now the branch being used is the new-feature.

Going forwards, all the changes would be made to the new-feature repository.

# Remote

```
[Tasks] git remote add origin https://github.com/thewhitetulip/Tasks

[Tasks] git remote -v
origin  https://github.com/thewhitetulip/Tasks (fetch)
origin  https://github.com/thewhitetulip/Tasks (push)
```

## Push and Pull

If you want to push your changes to the remote server, do a

```
[Tasks] git push origin master
```

`Origin` is the remote repo and `master` is the branch name which you want to push the commits to.

This book isn't about git, there already is an amazing book on git.

## Gitignore

At times in a project we want a few files to be not tracked, such files are placed in the `.gitignore` file in the root of the git repository. Any file or file pattern which is present in `.gitignore` is ignored by Git.

### Links

-Previous section -Next section

**Links**

-Previous section

As of 28'th August 2016.

https://github.com/benkasminbullock https://github.com/sprstnd https://github.com/appleboy https://github.com/datyayu https://github.com/roebuk https://github.com/smihir https://github.com/ghurley https://github.com/dgrosenblatt

Following users raised issues: https://github.com/norbertfuhs https://github.com/astropanic https://github.com/gernest https://github.com/raggi