ECGR 4090/5090 Cloud Native Application Architecture
Lab 4: Git and Go Webserver

Git and Github/Gitlab

From this lab onwards, we'll start using an online code hosting platform for version control. You may be doing this already. I would recommend using either *gitlab.com* or *github.com* and keeping your repositories private.
I am using Gitlab for the labs repository.
If you don't have an account, sign up on gitlab.com (or github.com)

If you are unfamiliar with Git and Github please watch the following YouTube tutorial - https://www.youtube.com/watch?v=RGOj5yH7evk

You will need to add SSH keys so that your machine can talk to Github/Gitlab

Create a new repository on Gitub/Gitlab - call it *CloudNativeCourse* and initialize it with *README.md*

You will need to add SSH keys so that your machine can authenticate itself to Github/Gitlab. The video explains how to do it. Here are instructions from Gitlab and Github

https://docs.gitlab.com/ee/user/ssh.html
https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent

After the SSH keys are setup,
In the *labs* directory on your machine,

$ *git clone https://gitlab.com/<your username>/cloudnativecourse*
Install git if needed -
($ sudo apt update; $ sudo apt install; $ git --version
This should create a *cloudnativecourse* directory, with a README.md inside.

Let's do a test update of the online repository. Open the README.md file (use your favorite editor), and add some text (say Go Rocks!).
Display state of working directory and staging area
$ git status
Add the README.md to staging
$ *git add README.md*
# Capture snapshots of staged changes. The message should describe the change
$ *git commit -m "Initial commit"*
# Push to online repository (origin)
$ *git push origin main*

Refresh your browser page for the online repository, and check to see if the *README.md* is updated.

From now, your development flow will involve *git status*, *git add*, *git commit*, and *git push* after you've coded for a bit to keep the local copy and the remote copy synced. Be sure to use meaningful commit messages.

If you change your dev machine (or if others are modifying the repo), start with a *git pull* to sync up your local copy with the online repo.

$ *git pull origin master*

Now add a .gitignore file under *cloudnativecourse* directory with the following content to instruct git not to add the following to the repository. These are typically editor related files that are not part of the project. Any other file could be listed as well (for example, executables)

```
# For vim
.*.sw*

# For vscode
.vscode/*
!.vscode/settings.json
!.vscode/tasks.json
!.vscode/launch.json
!.vscode/extensions.json
*.code-workspace

# Local History for Visual Studio Code
.history/
```

Push .gitignore to your repository.

Important:

1. Your remote repo (the one in Gitlab or Github) must be the true source of the project. Assume that your local machine could fail any time, and all data could be lost!

2. If you collaboratively work by sharing your repo with your teammates, be sure to work on different parts of the code. If there are code conflicts (say 2 people do different changes to the same function), then git will warn you about merge conflicts, and you will need to resolve this manually. I suggest not doing collaborative updates, unless you are really familiar with git.

3. If you need to delete files tracked by Git you will need to do it through Git. Else the origin and local repos will be out of sync.

$ git rm -r <directory/filename>

$ git commit -m "remove file1.txt"

For more on Git -
https://www.atlassian.com/git/tutorials/setting-up-a-repository

Go web server

Building a Go web server with the net/http package (from The Go Programming Language book)

Here's the basic outline -

```
package http

type handler interface {
        ServerHTTP(w ResponseWriter, r *Request)
}

func ListenAndServe(address string, h Handler) error
```

The *ListenAndServer* function requires a server address such as "localhost:8000", and an instance of the Handler interface to which all requests should be dispatched. It runs forever, or until the server fails.

To enable a user defined data type to be able to serve http requests, we need to attach a ServeHTTP method to it.
For example, consider a map of (items:price) that we want to serve http requests from

```
package main

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

type dollars float32

func (d dollars) String() string { return fmt.Sprintf("$%.2f", d) }

type database map[string]dollars
```

```
func (db database) ServeHTTP(w http.ResponseWriter, req *http.Request) {
	for item, price := range db {
		fmt.Fprintf(w, "%s: %s\n", item, price)
	}
}

func main() {
	db := database{"shoes": 50, "socks": 5}
	log.Fatal(http.ListenAndServe("localhost:8000", db))
}
```

Install *curl* if needed. *curl* is a command-line tool for transferring data using various network protocols. The name stands for "Client for URL".

$ *sudo apt update*
$ *sudo apt install curl*

Run the webserver
$ *go run webserver.go*
In a second terminal, use a *curl* client to access the server
$ *curl "http://localhost:8000"*

Note: If you are copy-pasting, the quotes may not copy correctly.

To incorporate multiple URL endpoints, we could use a switch statement

```
func (db database) ServeHTTP(w http.ResponseWriter, req *http.Request) {
	switch req.URL.Path {
	case "/list":
		for item, price := range db {
			fmt.Fprintf(w, "%s: %s\n", item, price)
		}
	case "/price":
		item := req.URL.Query().Get("item")
		price, ok := db[item]
		if !ok {
			w.WriteHeader(http.StatusNotFound) // 404
			fmt.Fprintf(w, "no such item: %q\n", item)
			return
		}
		fmt.Fprintf(w, "%s\n", price)
	default:
```

```
            w.WriteHeader(http.StatusNotFound) // 404
            fmt.Fprintf(w, "no such page: %s\n", req.URL)
        }
}
```

Test with curl -
$ curl “http://localhost:8000/list”
$ curl “http://localhost:8000/price?item=socks”

Alternatively, we could use a request multiplexer to simplify the association between URLs and handlers. A *ServeMux* aggregates a collection of *http.Handlers* into a single *http.Handler*

```
import (
        "fmt"
        "log"
        "net/http"
)

type dollars float32

func (d dollars) String() string { return fmt.Sprintf("$%.2f", d) }

func main() {
        db := database{"shoes": 50, "socks": 5}
        mux := http.NewServeMux()
        mux.Handle("/list", http.HandlerFunc(db.list))
        mux.Handle("/price", http.HandlerFunc(db.price))
        log.Fatal(http.ListenAndServe("localhost:8000", mux))
}

type database map[string]dollars

func (db database) list(w http.ResponseWriter, req *http.Request) {
        for item, price := range db {
            fmt.Fprintf(w, "%s: %s\n", item, price)
        }
}

func (db database) price(w http.ResponseWriter, req *http.Request) {
        item := req.URL.Query().Get("item")
        price, ok := db[item]
        if !ok {
            w.WriteHeader(http.StatusNotFound) // 404
```

```
            fmt.Fprintf(w, "no such item: %q\n", item)
            return
        }
        fmt.Fprintf(w, "%s\n", price)
}
```

Note:
db.list and db.price do not have a ServerHTTP method, and hence cannot be passed directly to mux.Handle. So an adapter function is needed to add the ServerHTTP method (decorator pattern). The net/http package defines the HandlerFunc type for this purpose

```
package http

type HandlerFunc func(w ResponseWriter, r *Request)

func (f HandlerFunc) ServerHTTP(w ResponseWriter, r *Request {
        f(w, r)
}
```

HandlerFunc is a function type that has methods that satisfies the *http.Handler* interface. Its behavior is to call the underlying function.

ServerMux has a convenience function HandleFunc to simplify the code for handler registration -

```
mux.HandleFunc("/list", db.list)
mux.HandleFunc("/price", db.price)
```

Important:
The web server invoked each handler in a new goroutine, so take precautions to handle concurrent operations on shared data structures using locks and channels.

To do -

Add additional handlers so that clients can create, read, update, and delete (CRUD) database entries. For example, a request of the form /update?item=socks&price=6 will update the price of an item in the inventory and report an error if the item does not exist or if the price is invalid. Note that the list method already implements read. You only need to implement the create, update, and delete methods
Hint: https://gobyexample.com/number-parsing

Hint: Multiple clients could be doing CRUD operations on an item. You will need to use mutexes from the sync package to synchronize access.
Hint: Use RWMutexes to improve read performance

Test your implementation of the webserver with *curl.*