The Go Projects

In this part of the lab section of the course, we move from simple examples and exercises that are connected with the presentation content into a series of mini-projects that illustrate how Go is used to create some typical applications. Because these projects do use a lot of the standard Go package functionality, they are an opportunity to explore the content of these libraries in an applied setting

These projects are based on or derived from standard applications, demos and tutorials that are easily accessible from multiple sources and are all either well knows or somewhat traditional as Go sample projects. For some of the projects that follow, you will find similar applications to what is presented here, but the code may not be identical, nor the design quite the same. It is a good idea to contrast and compare what is here with other versions to understand the different design choices made by the different authors.

Except where noted, the projects are independent of each other and may be done in any particular sequence, however discussions of the functionality of the Go packages are presented in each project so if you do skip a project, you may have to backtrack to read some of the commentary on the Go packages in the skipped project.

Project 1: Command Line Chat Server

This project implements a command line chat program which implements the functionality of a chat room. This particular version was inspired by Brian Kernigan's version since it does not involve any web interface – the focus of the project is on the concurrency and i/o operations.

Desired Functionality

The chat server is a program that allows remote clients to connect into a common chat room where they can exchange messages with the other clients. Each client should be able to see all of the messages sent by all the other clients which means each client's messages are broadcast to all the other clients and each message sent by a client should be seen by everyone else.

In addition, all the clients should be notified by the arrival of a new client and the departure of an existing client.

The server will have to accept new clients, keep track of all of the existing clients and relay any input from a client to all of the other clients. The server will also have to do some sort of cleanup after each client leaves.

Part One: A Chat Client

The client has to do three things.

1. Establish a connection to the server
2. Print output from the connection to the terminal
3. Sent input from the terminal to the connection

In this first version of the client, we will implement functionality to perform the first two tasks.

1.1 The chatCopy() function.

This will be the general utility function that will copy input from connection to terminal and vice versa and should terminate when the connection is broken. There will be two packages that we will use, the io package and the log package.

The io Package

To implement the functionality, we use the Copy function from the io package. This is a function that copies a stream of bytes from a source that is readable to a source that is writeable. The method signature is given by.

```
func Copy(dst Writer, src Reader) (written int64, err error)
```

One of the useful features of the io.Copy() function is that it reads until it encounters an EOF but does not report encountering EOF as an error.

The log Package

One of the problems we have may be that the io.Copy() function fails. If this happens, then we will log the error that caused the failure and exit the program. To do this, we use the log.Fatal() function from the log package which is is described as:

The logger writes to standard error and prints the date and time of each logged message. The Fatal functions call os.Exit(1) after writing the log message.

The ChatCopy() function looks like this:

```
 // chatCopy() function
func chatCopy(destination io.Writer, source io.Reader) {
   if _, err := io.Copy(destination, source); err != nil {
        log.Fatal(err)
   }
 }
```

The next step is to implement the functionality to connect to a server and read from the connection then write to the stdout or terminal output.. In order to do this, we will assume that our server is running on port 3000.

The net Package

The net package provides basic networking connectivity. The function that we are going to be using here is net.Dial() which establishes a connection to a server. The Dial() function takes two arguments, the first is the network which in this case will be “tcp” and the second is the server location.

The first part of the client connects to the server and if the connection is not successful, exits and logs an error message.

Notice the use of defer to makes sure the connection is closed before the client program exits. Since we could exit for a variety of reasons, this makes sure the connection is closed no matter how we exit. This is an example of exactly what the defer operator is for.

```
// Listen Only Client
func main() {
   connection, err := net.Dial("tcp", "localhost:3000")
   if err != nil {
        log.Fatal(err)
   }
   defer connection.Close()
   chatCopy(os.Stdout, connection)
 }
```

1.2 Testing the read functionality

1. Compile and run the client. We should get the error that the program exited with the exit code 1 because the connection was refused. Of course it was because there was nothing listening on port 3000 to establish a connection to.

```
[chatclient1]$ ./chatclient1
2016/08/18 06:18:54 dial tcp [::1]:3000: getsockopt:
connection refused
```

2. In order to give the chat client something to test against, there is a timeserver app provided for testing purposes in the directory ClockServer. Build and run Clockserver in a separate window, this is a server that just writes the time to port 3000.

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

3. Now rerun the chatclient1 in a different window from the ClockServer while the ClockServer is running and you should see the time printing out at the console.

```
[chatclient1]$ ./chatclient1
06:39:53
06:39:54
06:39:55
^C
```

4. Exit both programs.

1.3 Adding the Write Functionality

Now we have to pick up what is typed in at the console and send it to the server.

In order to add write functionality, we are going to move what we have already done into an anonymous goroutine that will be started by the client. The reason why we are using a goroutine is that we want the two directions of data transmission to be concurrent but independent of each other. While the main chat client routine is sending what is typed to the chat server, the anonymous goroutine is reading from the server and writing it to standard output.

1. To ensure that we get a clean exit from the chat, the two goroutines are synchronized with a done channel. When the user terminates the main goroutine, it executes the Close() command on the connection. It then waits and listens on the done channel for the anonymous goroutine to signal it is finished.
2. Closing the connection in the main goroutine causes the connection to be closed for the anonymous goroutine – remember that the connection variable is a free variable in the anonymous goroutine.
3. Closing the connection used by the anonymous goroutine causes the io.Copy() command to terminate since it is now trying to read from a non-existent connection.
4. The anonymous gorountine now logs that it is done and sends a signal over the done channel to the main goroutine that it is finished its cleanup. On receiving this message, the main goroutine exits.

```
// Full Chat Client

func main() {
    connection, err := net.Dial("tcp", "localhost:3000")
    if err != nil {
         log.Fatal(err)
    }
    done := make(chan bool)
    go func() {
         io.Copy(os.Stdout, connection)
         log.Println("done")
         done <- true
    }()
    chatCopy(connection, os.Stdin)
    connection.Close()
    <-done
 }
```

Part Two: The Chat Server

The server has three main tasks to accomplish.

1. Accept incoming connection requests and keep track of clients.
2. Manage each connection by receiving input from each of the clients it is connected to.
3. Broadcast to all clients whatever is received from each connection.

The the second and third of these tasks are handled in separate functions that we will look at individually. We start with looking at the main server.

2.1 The Main Server Routine.

```
// Chat Server Main Routine

func main() {
  listener, err := net.Listen("tcp", "localhost:3000")
  if err != nil {
       log.Fatal(err)
  }
  go broadcaster()
  for {
       conn, err := listener.Accept()
       if err != nil {
            log.Print(err)
            continue
       }
       go handleConn(conn)
  }
}
```

1. The first part of the main routine starts by listening for connection requests on port 3000 using the net.Listen() function. If it cannot listen on that port, then it logs and error and exits.
2. A goroutine called broadcaster() is launched. In the same way that we wanted the input and output of the chat client to be concurrent, we want the broadcaster routine to operate concurrently with any input that comes in from any of the clients.

3. In the for loop, the chat main function will listen for a client to establish a connection. If the connection fails for some reason, then an error is logged and listening resumes.
4. One a connection is established, then a new goroutine is spawned to manage all of the interactions with that client, and the main routine goes back to listening.

The completed code for this is in the file chat.go.

2.3 The Broadcaster.

The broadcast routine is responsible for sending out the messages from each client to all of the other clients.

1. The broadcaster needs to be updated when a client leaves or enters the chat so that it can keep track of who to send messages to. To facilitate this, there are two channels set up to communicate those events to the broadcasters, and entering channel and a leaving channel. ***One of the critical points to note is that the entering and receiving channels are channels that send and receive string channels.***
2. There is also a message channel that provides the input for broadcasting as well as an outgoing message channel.

```
// The broadcaster channels.

// an outgoing message channel
type client chan<- string

var (
  entering = make(chan client)
  leaving  = make(chan client)
  // all incoming client messages
  messages = make(chan string)
)
```

3. The broadcaster keep track of clients by maintaining a map of the client channels with a status flag to identify whether or not the client is active.
4. When a chat client connection is created, it sends a message on the entering channel. Once a message has been received on the entering channel by the broadcaster, it creates a new client channel in the map and sets the flag to true.
5. When a chat client leaves, it sends a message on the leaving channel which causes the broadcaster to remove the client channel from the map and close the outbound connection for that client.

```
func broadcaster() {
  clients := make(map[client]bool) // all connected clients
  for {
       select {
       case msg := <-messages:
            // Broadcast incoming message to all
            // clients' outgoing message channels.
            for cli := range clients {
                 cli <- msg
            }

       case cli := <-entering:
            clients[cli] = true

       case cli := <-leaving:
            delete(clients, cli)
            close(cli)
       }
  }
}
```

6. The rest of the time, the select loop listens on the inbound messages channel and for each item, loops through the array and sends it to every client.
7. The full code is in broadcast.go

2.4 Handling the Connection

This is the part of the server that the chat client talks to. Clearly each client talks to one connection object. Remember that this is not the client, this is the goroutine that talks to the client through the network connection.

1. When first spawned, the goroutine creates an outgoing message channel on which to send the messages received from the client to the broadcaster.
2. It then sends that channel it just created over the entering channel so that the broadcaster can add it to the map. Of course it also sends a message to announce the arrival of the client.

3. It also creates a Scanner object. A scanner reads a newline delimited stream of tokens (ie. lines) from the connection with each call to the Scan() method and then adds an identifying prefix. The resulting string is dumped onto the messages channel to be sent to the broadcaster.

```
func handleConn(conn net.Conn) {
  ch := make(chan string) // outgoing client messages
  go clientWriter(conn, ch)

  who := conn.RemoteAddr().String()
  ch <- "You are " + who
  messages <- who + " has arrived"
  entering <- ch
  input := bufio.NewScanner(conn)
  for input.Scan() {
       messages <- who + ": " + input.Text()
  }
  leaving <- ch
  messages <- who + " has left"
  conn.Close()
}

func clientWriter(conn net.Conn, ch <-chan string) {
  for msg := range ch {
       fmt.Fprintln(conn, msg)
  }
}
```

4. One the EOF occurs because the conversation is terminated by the client (ie. the connection is broken or the client exits), then the client channel is sent over leaving channel to ensure that the client is removed from the broadcaster map.
5. The clientWriter() goroutine runs concurrently and just ensures that all messages sent to the client by the broadcaster and sends them over the connection.
6. The clientWriter() goroutine terminates when the handleConn() goroutine closes the connection.

7. The full code is in handlecon.go

2.5 Build and Execute

Following the directions in the project build and test the chat server with two clients. This project illustrates a concurrent program design. You may find it difficult to think concurrently at the start, but it helps to think of this as a manual messaging system where pieces of paper are passed around instead of connections and the goroutines as people with specific tasks.

Project 2: Building a Wiki

This second project will build a simple wiki and introduce some of the basic building blocks of the Go net/http package that is used in web development. In the next project, we will build a restful web service that incorporates using JSON data.

Part One: Hello World

Building a web application is almost trivial using the Go net//http package. Just like in the chat server project, we can create a listener that listens on a specific port, in this case 3000 although you should change it to whatever port works for you in your environment.

In this first example, we are going to build a basic Hello World application to see the basic structure of an HTTP server using the GO packages.

The application deployment consists of two basic steps. First we register what are traditionally called “call back” function or handlers to respond to specific URLs. Go will create a map of URLs and functions so that when a specific URL is received, the associated call back function is executed to complete the http request. Once the handlers or call back functions have been written, the second step is just to listen on a port for HTTP requests.

The call back function in the code on the next page is called “hello()” and takes as a first parameter an http.ResponseWriter, which is the same type of object as a response writer in Java or a similar language. Basically the writer is like a file we write to and the resulting content is then displayed as the html page. The second argument is a pointer to the http.Request object generated by the HTTP request from the client to the server.

The http.HandleFunc() method specifies that for a specific URL, in this case “/” means all URLS, the hello function will be called to handle it.

The hello() function just takes the URL, removes the leading slash and then prints out the rest as on the page.

If you compile and run this code, then in a browser enter http”//localhost:3000/world you should see a hello world message displayed.

The full code is in the directory HelloWeb

```go
// A simple Web based "Hello World"

package main

import (
  "fmt"
  "net/http"
)
func hello(wr http.ResponseWriter, req *http.Request) {
     fmt.Fprintf(wr, "<h1>Hello %s!</h1>",req.URL.Path[1:])
}

func main() {
  http.HandleFunc("/", hello)
  http.ListenAndServe(":3000", nil)
}
```

Part Two: The Wiki Server

We assume that you also are quite comfortable with the idea of what a wiki is, which is basically a set of editable web pages.

2.1 The Wiki Pages

The a wiki page itself is defined as a struct that looks like the code. Of course a real wiki page would be a lot more complex, but this will work for now.

Since we need to have editable content, we keep the content of the wiki in a text file. This means that we need two functions to read and write the wiki page. To be able to find the pages, we name the file that has the content with the title of the wiki page plus the file extension ".txt"

```
 // Basic Wiki Support Code

package main

import (
    "fmt"
    "io/ioutil"
)

type Page struct {
    Title string
    Body  []byte
}

func (p *Page) save() error {
    filename := p.Title + ".txt"
    return ioutil.WriteFile(filename, p.Body, 0600)
}

func loadPage(title string) (*Page, error) {
    filename := title + ".txt"
    body, err := ioutil.ReadFile(filename)
    if err != nil {
         return nil, err
    }
    return &Page{Title: title, Body: body}, nil
 }
```

The full code is in wiki.go

Test the code by writing a few test pages, saving them and retriving them.

2.2 Displaying Wiki Pages

The first version of the server is in the directory Wiki in the file server.go. Copying the basic idea of the Hello World App, we can now create a viewPage() handler that gets the page and displays it. You should try out the following code to make sure it works and that you understand what is happening.

```
  // Basic Display Wiki Pages Server

package main

import (
    "fmt"
    "net/http"
)

func viewHandler(wr http.ResponseWriter, req *http.Request) {
    title := req.URL.Path[len("/view/"):]
    pg, _ := loadPage(title)
    fmt.Fprintf(wr,"<h1>%s</h1><div>%s</div>",pg.Title,
                                                    pg.Body)
}

func main() {
    http.HandleFunc("/", hello)
    http.ListenAndServe(":3001", nil)
  }
```

2.3 Editing the WIki Pages

The edit functionality will be implemented via an HTML form, however the HTML is starting to get too complex to be maintainable by placing in line like we have been doing. In the go packages, there is Go package called html/template that allows us to write template code, sort of like JSPs, which can be populated dynamically.

The template for the edit page looks like this and is in the file "edit.html" in the Wiki2 directory.

```
<h1>Editing {{.Title}}</h1>

<form action="/save/{{.Title}}" method="POST">
    <div>
         <textareaname="body" rows="20" cols="80">
              {{printf "%s" .Body}}
         </textarea>
    </div>
    <div>
         <input type="submit" value="Save">
    </div>
 </form>
```

The template for the view.html looks like this:

```
<h1>{{.Title}}</h1>

<p>[<a href="/edit/{{.Title}}">edit</a>]</p>
<div>{{printf "%s" .Body}}</div>
```

Notice that the view template has a link to the edit.html

The code in the "{ }" are directives that are dynamically inserted into the templates before they are used.

The templates are used by creating a template object, derived from parsing a template file. The template is then executed, which means that the template structure directives are executed and the resulting HTML page written to the http.ResponseWriter. This takes place in these two lines of code:

```
    t, _ := template.ParseFiles("edit.html")
    t.Execute(wr, pg)
```

Now we can write the two handlers for viewing and editing a wiki page, deferring the saving the edited page part until the next section. The view and edit handlers should look familiar even through we have introduced the template.

```
 // Basic Display Wiki Pages Server

package main

import (
    "fmt"
    "net/http"
)

func viewHandler(wr http.ResponseWriter, req *http.Request) {
    title := req.URL.Path[len("/view/"):]
    pg, _ := loadPage(title)
    t,+ := template.ParseFiles("view.html")
    t.Execute(wr,pg)
}
func editHandler(wr http.ResponseWriter, req *http.Request) {
    title := req.URL.Path[len("/edit/"):]
    pg, err := loadPage(title)
    if err != nil {
        pg = &Page{Title: title}
    }
    t,+ := template.ParseFiles("edit.html")
    t.Execute(wr,pg)
}

func main() {
    http.HandleFunc("/view/", viewHandler)
    http.HandleFunc("/edit/", editHandler)
    http.ListenAndServe(":3001", nil)
 }
```

One addition that should be noted is that in the editHandler, if the page cannot be loaded, then a new blank page is provided with the supplied title.

Build the code as we have done so far and test it out. Once you have it working properly, then we can finish up the rest of the tasks to be completed.

The full code for this section is in the directory Wiki2

2.4 Finishing up the Project

There are a number of features that still need to be added.

1. When a non-existent page is requested, the view handler should open up the editor so the page can be created.
2. The saveHandler() has to be implemented in order that edited pages can be saved.
3. We need to put some security on the server to avoid malicious URLs from getting out of the "sandbox"
4. Adding error responses.

2.4.1 Viewing a non-existent Wiki Page

When the page requested does not exist, then we redirect the request to the edit page. The modified viewHandler code looks like this

```
func viewHandler(wr http.ResponseWriter, req *http.Request) {
    title := req.URL.Path[len("/view/"):]
    pg, err := loadPage(title)
    if err != nil {
       http.Redirect(wr, req, "/edit/"+title, http.StatusFound)
       return
    }
    t, _ := template.ParseFiles("view.html")
    t.Execute(wr,pg)
}
```

1. Make the code change and test it out to see that it works

2.4.2 Adding a saveHandler

Nothing happens yet when we click the "save" button in the edit box. We can logically figure out that what we want to have happen is for the contents of the form to be saved to a file, then the viewHandler called to display the contents.

We can now add in the save handler to look like this:

Once you have the save handler added, then compile and try out the full functionality of the wiki server.

The complete code is in the directory Wiki3

```
func saveHandler(wr http.ResponseWriter, req *http.Request) {
    title := req.URL.Path[len("/save/"):]
    body := req.FormValue("Body")
    pg := &Page{Title: title, Body: []byte(body)}
    p.save()
    http.Redirect(w, r, "/view/"+title, http.StatusFound)
}
func main() {
    http.HandleFunc("/view/", viewHandler)
    http.HandleFunc("/edit/", editHandler)
    http.HandleFunc("/save/", saveHandler)
    http.ListenAndServe(":3001", nil)
  }
```

2.4.3 Validation Security

Users are only supposed to use URLs that start with "view" or "edit" or "save." However, we know that there will be those who will try and break the wiki.

A standard way of validating any sort of path, string or value that fits a known pattern is to use a regular expression. We are assuming that everyone is familiar with regular expressions, so we will use them without explanation, however if you are unsure as to what they are, then just Google “Regular Expression” and find a quick tutorial or summary.

The regular expression pattern we want for validation is “A leading / followed by one of the words ‘edit’,’save’ or ‘view’, followed by another slash and then followed by a string consisting of one of more letters and number.”

The actual pattern is:

```
"^/(edit|save|view)/([a-zA-Z0-9]+)$"
```

The way we will do this is to create a regular expression that we can validate the supplied URL against. We create a validPath regexp object and define a getTitle() method that extracts the URL from the http.Request and sees if it matches the pattern.

The method “FindStringSubMatch()” returns a slice of strings with the first string the left most match and the then subsequent strings the substrings that matched the sub expressions. That means that the string “/view/TestPage” would result in the following slice being returned

```
[“/view/TestPage”, “view”,”TestPage”].
```

This means the title will be the last element in the slice.

```
 var validPath = regexp.MustCompile(
                          "^/(edit|save|view)/([a-zA-Z0-9]+)$")

 func getTitle(wr http.ResponseWriter, req *http.Request)
                                 (string, error){
     url := validPath.FindStringSubmatch(r.URL.Path)
     if url == nil {
         http.NotFound(wr, req)
         return "", errors.New("Invalid Page Title")
     }
     return url[2], nil
}
```

If the url is not well formed, then a “404” error is generated.

We can now place the getTitle() function in the viewHandler to ensure that only well formed URLs will be accepted by the handler.

```
 func viewHandler(wr http.ResponseWriter, req *http.Request) {
     title, err := getTitle(wr, req)
     if err != nil {
          return
     }
     pg, err := loadPage(title)
     if err != nil {
        http.Redirect(wr, req, "/edit/"+title, http.StatusFound)
        return
     }
     t, _ := template.ParseFiles("view.html")
     t.Execute(wr,pg)
 }
```

We can make the similar substituion of the getTitle() function into the other two handlers. This will be left for you to do on your own.

A complete set of code is in the directory Wiki4