Go Employment Test
Explained In Color

REVISION 1

Hawthorne-Press.com

Go Employment Test Explained in Color

Published by

Hawthorne-Press.com

10310 Moorberry Lane
Houston, Texas 77043, USA

Table of Contents

GO EMPLOYMENT TEST EXPLAINED IN COLOR

INTRODUCTION

The following code example is the result of an employment code test. The criteria stated were as follows:

> "***Design & code a very simple in-memory file system and expose the file system operations over HTTP"***

I was given an afternoon to code against this requirement. As can be seen, this is a wide-open description. I, perhaps foolishly, took it a little farther than would be expected. I declined to use Go Collections and other packages to ease the project. I only used two major packages, HTTP and JSON. The other packages were standard, such as FMT, IOUTIL, OS, TIME and STRINGS. .

I discovered several subtle bugs after submitting the program. Some I fixed others I left to the reader to correct!

The only code I have modified from the test was to fix a minor bug when you have two similar *Names* in the database. The names were searched using the string "Contains" method for EDIT and VIEW. This worked, but unfortunate side effect is that the program produced an error if more than one name matched. I fixed this problem by reverting to string "equality" if the search returns more than one entry. The DELETE Method always uses string "equality".

Because of time constraints, I did not do any refactoring. However, this document notes refactoring opportunities.

BASIC HTTP

The first thing to do is start with a basic HTTP Server function. We will build the test program on this foundation.

```
package main
import (
       "fmt"
       "http"
)
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "Hello, world")
}
func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

Run this program in one window and in a browser enter "localhost:8080/". The browser will display "Hello, world".

The main function creates a handler for the Web "root" page, which is "localhost:8080/", and starts a Server to listen for traffic on port 8080. When it receives a request, it calls the handler function write the message to the Client.

NEXT STEPS

From this point, we will add new handlers for VIEW, EDIT, DELETE, and SAVE to replace the simple Web Root handler. We will also create a number of utility routines to handle common tasks!

.The server will continue to listen on port 8080 and call the appropriate new handlers..

Program Database Structure

The required *in-memory filesystem* is a slice of the structure type **Page**.

```
var xMem []Page         // In Memory Database File  (Note that it is a Global!)

type Page struct {      // Database Page
    Index int          // Index of Database Page
    Name  string       // KEY:   Name as Search Key
    Body  []byte       // VALUE: Data associated with the Key
```

The JSON Package packs the in-memory database for external storage and unpacks external storage to restore the in-memory database. Using JSON consistently, rather than substituting slice operations, reduces errors.

Main()

The main() function is modified to load the Database, setup the handler functions, and start the server on port 8080. The changes in the original code our shown in red.

```
func main() {
    loadDatabase()                          // Load Database

    http.HandleFunc("/view/", viewHandler)   // Setup Handler Functions
    http.HandleFunc("/edit/", editHandler)
    http.HandleFunc("/save/", saveHandler)
    http.HandleFunc("/delete/", deleteHandler)
    http.ListenAndServe(":8080", nil)        // Setup up Server to listen on port 8080
}
```

As shown, the main() function only loads to database and sets up our new handlers. What we add from here, are the handlers themselves and the utility support functions. Normally, these additional functions would be in their own Go source file. However, in this case all the code resides in one source file.

Utility Support Functions

Function save()

The *save* function is used by the *saveHandler* Function to create a new entry in the database and write it to disk.

```
//
// SaveHandler helper to create and store a new page in the database
//
func (p *Page) save() {
    np := Page{Index: p.Index, Name: p.Name, Body: p.Body}     // Create a database Page
    xMem[p.Index] = np                                        // Append it to the in-memory database
    data, err := json.Marshal(xMem)                           // Marshal the database
    check("Marshalling Failed", err)                          // Check for error
    writeData(data)                                           // Write database to the disk
    return                                                    // Return
}
```

FUNCTION LOADDATABASE()

LoadDatabase reads the external database file and unmarshals the data into the in-memory database.

If the external database is missing, it is created along with five test records. After creation and marshalling, the new data set is written to disk. The data set is then unmarshalled into the in-memory database.

A slice copy could have replaced the unmarshalling function, but I decided that all external database operations would use JSON functionality for consistency.

```
//
// Load Database
//
func loadDatabase() {

    var jMem []Page                                 // Temporary Blank in-memory Database

    data, err := ioutil.ReadFile("Data.db")         // Load Database
    if err != nil {                                 // If missing - Create
//
// Create the Initial Database with Test Data - Remove appends below for empty database
//
        jMem = append(jMem, Page{Index: 0, Name: "Charles", Body: []byte("Charles Data")})
        jMem = append(jMem, Page{Index: 1, Name: "Ann", Body: []byte("Ann Data")})
        jMem = append(jMem, Page{Index: 2, Name: "Jack", Body: []byte("Jack Data")})
        jMem = append(jMem, Page{Index: 3, Name: "Mike", Body: []byte("Mike Data")})
        jMem = append(jMem, Page{Index: 4, Name: "Jacky", Body: []byte("Jacky Data")})

        data, err = json.Marshal(jMem)              // Marshall Database
        check("Marshalling Failed", err)
        _, err := os.Create("Data.db")              // Create Database
        check("Create File Failed", err)
        writeData(data)                             // Write Database
    }
    err = json.Unmarshal(data, &xMem)               // Unmarshall into in-memory Database
    check("Unmarshal Failed", err)
}
```

FUNCTION FINDNAME(XMEM, NAME)

This function is called with a pointer to the in-memory Database, a Global Value, and the “name” to find.

The first search uses the *string.Contains* method from the strings Package. If this search only matches one “name” in the memory, the appropriate page element is returned along with the bool return value set to True.

If the first search returns no value, an error message and a bool False are returned.

If more than one page found for the search string, a second search is executed. This search uses the *string.Compare* method and tests for string “equality”. If a match is found, the page is returned with a bool return value set to True.

If the second search finds no exact match, an error message and a bool False are returned.

```
//
//  Find Name Function - Locates by string.Contains.
//  If more than one name matches, it searches again with string.Compare Function (equality match)
//
func findName(xMem []Page, name string) (Page, bool) {
    var retPg Page                                  // Create Variables
    dup := 0
    for i, key := range xMem {                      // Search using string.Contains method
        if strings.Contains(key.Name, name) {
            dup++                                   // Increment variable dup for each match
            retPg = xMem[i]
        }
    }
    if dup != 1 {                                    // Duplicate Search Result Check!
        for i, key := range xMem {                   // Find exact match!
            if strings.Compare(key.Name, name) == 0 {
                retPg = xMem[i]                      // If found - Store returned page
                return retPg, true                   // Return Result and True (exact match found)
            }
        }
        return retPg, false                          // If no exact match return False
    }
    return retPg, true                               // Return Result and True (only one "contains"
Result found)
}
```

Function findExactName(xMem, name)

The findExactName function only returns a database page if the specified "name" is an exact match. This function is only used by the *deleteHandler*. Why? We must delete only one specific page in the Database.

```
//
//  Find Name Function - Locates by string.Compare "equality" match
//
func findExactName(xMem []Page, name string) (Page, bool) {
    var retPg Page                              // Create Variables
    for i, key := range xMem {                  // Search in-memory database for an exact match
        if strings.Compare(key.Name, name) == 0 {
            retPg = xMem[i]                     // Match Found
            return retPg, true                  // Return Page and bool return value of True
        }
    }
    return retPg, false                         // No match found -- Return bool return value of False
}
```

Function writeData(data []byte)

As the function name indicates, this routine writes the marshalled data set to the disk and checks for errors.

```
//
// Write Data Set to Disk
//
func writeData(data []byte) {                   // Write "Mashalled" data to external device
    err := ioutil.WriteFile("Data.db", data, 0644)
    check("Write File Failed", err)             // Error Check -- Panic if write fails
}
```

FUNCTION VIEWHANDLER(W HTTP.RESPONSEWRITER, R *HTTP.REQUEST)

The *viewHandler* processes the following URL entries and checks for error conditions:

- Localhost:8080/view — List all names in database
- Localhost:8080/view/name — Display full record for element "name"

Database contains the following Names:

```
Record  0 :  Charles
Record  1 :  Ann
Record  2 :  Jack
Record  3 :  Mike
Record  4 :  Jacky
Record  5 :  bob
```

View: Ann

```
Ann Data
```

The processing of the */view* URL without a "name" is the only handler that checks for database corruption. While this command is the one most likely to detect a corrupted database, it should be "Refactored" out of view and made part of the function *loadDatabase(). Since I failed to do so at the time, I left it in as an example of an opportunity for refactoring your code.*

After checking for an empty database or using ALL as a "name", the function looks up the "name". If there is only one result, a <form> is constructed and returned to the client via *http.ResponseWriter*.

```
//
//  View Handler --
//    localhost:8080/view/          -- Lists all names in the database
//    localhost:8080/view/name      -- Displays the Page for "name"
//
func viewHandler(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Path[len("/view/"):]               // Extract "name" from URL path
    var body string                                  // Create Variables
    if len(name) <= 0 {                              // Check for /view without name
        if len(xMem) <= 0 {                          // Empty Database Check
            body = fmt.Sprintln("")                  // Create Empty TextArea
        } else {
                                                     // Page Validation Check --
            for i := 0; i < len(xMem); i++ {
                if i != xMem[i].Index {              // Make sure in-memory index equals internal
page index
                    // Database corruption error!
                    error := fmt.Sprintln("Database Indexing Failure: Name = ", xMem[i].Name)
                    fmt.Fprintf(w, "<h1>View: %s</h1>", error)
                    time.Sleep(time.Second)
                    return
                }
                                                      // Display Page Elements in Testarea of <form>
                body += fmt.Sprintln("Record ", xMem[i].Index, ": ", xMem[i].Name, string(""))
            }
                                                     // Send constructed Display to the Client!
            fmt.Fprintf(w, "<h1>Database contains the following Names:</h1>"+
                "<textarea nameM=\"body\" rows=\"20\" cols=\"80\">%s</textarea><br>"+
                "</form>",
```

```
                body)
            return
        }
    }
                                                    // Display Empty Database Message
    if len(xMem) <= 0 {
        fmt.Fprintf(w, "<h1>View: %s</h1>", "Empty Database")
        return
    }

                                                    // Display "ALL" Error (Only for /delete/ALL)
    if name == "ALL" {
        fmt.Fprintf(w, "<h1>View Error: %s</h1>", "'ALL' is a Command!")
        return
    }
                                                    // Handle Display of a "Named" Page
    p, ok := findName(xMem, name)
    if !ok {                                        // Too many matches or Name not found
        if len(p.Name) > 0 {
            fmt.Fprintf(w, "<h1>View: %s</h1>", "Name Matches > 1!")
            return
        }
        fmt.Fprintf(w, "<h1>View: %s</h1>", "Name not found!")
        return
    }

    fmt.Fprintf(w, "<h1>View: %s</h1>"+                 // Create <form> for viable "name" result
        "<form action=\"/load/%s\" method=\"POST\">"+
        "<textarea nameM=\"body\" rows=\"20\" cols=\"80\">%s</textarea><br>"+
        "</form>",
        p.Name, p.Name, p.Body)
}
```

FUNCTION SAVEHANDLER(W HTTP.RESPONSEWRITER, R *HTTP.REQUEST)

This handler is different from the *View*, *Edit*, and *Delete* handlers in one important respect. The *saveHandler* is not a Client function. While the URL */localhost:8080/save* and /save/name are valid URL's, they perform erroneous operations. Without a "name" the function returns the Blank Screen. If you supply a name, it actually deletes the data value for that "name". Why? Because a direct "/save/name" returns an empty "body" value for the save function.

This is another opportunity to refactor and correct this problem in case the client directly issues a "/save".

The *saveHandler's* only valid purpose is to support the *editHandler* internally. When the client selects Save on the edit form, the <form action> executes a "/save/name" redirection operation.

```
//
// Save Handler Function -- Should not be used by the Client
//
func saveHandler(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Path[len("/save/"):] // Get "name" value if present
    if len(name) <= 0 {                // If no name - redirect to /view
        http.Redirect(w, r, "/view/", http.StatusFound)
    }
    pg, ok := findName(xMem, name)     // Find "name"
```

```
    if !ok {                            // If error - report it and panic
        fmt.Println("Name Lookup Failed")
        panic("error")
    }
    body := r.FormValue("body")         // Get <form> value for "body"
    p := &Page{Index: pg.Index, Name: name, Body: []byte(body)}
    p.save()                            // Call Helper function save()
    http.Redirect(w, r, "/view/"+name, http.StatusFound)     // Redirect to /view/name
}
```

FUNCTION EDITHANDLER(W HTTP.RESPONSEWRITER, R *HTTP.REQUEST)

The *editHandler* processes the following URL entry and checks for error conditions:

- Localhost:8080/edit/name Display scomplete record for "name"

The *editHandler* checks that "name" is not ALL, and calls *findName* routine to get the correct page element.

If the specified "name" is not in the database, the function will create an entry with the correct index and name values. However, the body value will be blank. After adding the new value to the in-memory database, it marshals it into a data set, and writes it to the disk. Lastly, it then performs a f*indName* on the new "name".

At this point, unless we encountered an error, we have a valid page element in both cases. Using the collected data a <form> is created and populated with the appropriate data, and sent to the client.

The client can edit the data in the textWindow and press the Save Button to store the result.

```
//
// Edit Handler
//   localhost:8080/edit/name
//
func editHandler(w http.ResponseWriter, r *http.Request) {
    var np Page                                     // Create Variables
    name := r.URL.Path[len("/edit/"):]              // Extract Name Portion
    if name == "ALL" {                              // Name = ALL -- Print Error
        fmt.Fprintf(w, "<h1>Edit Error: %s</h1>", "'ALL' is a Command!")
        return
    }
```

```
    p, ok := findName(xMem, string(name))          // Find Name
    if !ok {                                       // If no name -
                                                   //   Create name with empty body
        np = Page{Index: len(xMem), Name: name, Body: []byte("")}
        xMem = append(xMem, np)                    // Append the in-memory database
        data, err := json.Marshal(xMem)            // Marshal xMem ==> Data set
        check("Marshalling Failed", err)
        writeData(data)                            // Write Data Set to disk
        p, ok = findName(xMem, string(name))       // Find newly created name!
        if !ok {
            fmt.Println("Update Failure")          // Notify of failure
            return
        }
    }
    fmt.Fprintf(w, "<h1>Editing %s</h1>"+          // Build Form and send to client
        "<form action=\"/save/%s\" method=\"POST\">"+
        "<textarea name=\"body\" rows=\"20\" cols=\"80\">%s</textarea><br>"+
        "<input type=\"submit\" value=\"Save\">"+
        "</form>",
        p.Name, p.Name, p.Body)
}
```

FUNCTION DELETEHANDLER(W HTTP.RESPONSEWRITER, R *HTTP.REQUEST)

The *deleteHandler* processes the following URL entries and checks for error conditions:

- localhost:8080/delete/name Deletes “name” and lists remaining names in database
- localhost:8080/delete/ALL Deletes all database entries

In both cases above, after the selected database elements are deleted, the processing redirects to “/view”.

While deleting all database elements is straight forward, deleting a single element requires some care. Since the database elements contain an index synchronized with slice index, just deleting to slice element will corrupt the database.

An empty database slice is created and the actual database slice is ‘walked’ to copy desired elements. This is necessary to “re-index” the indexes in each page record.

Including an index in the Database elements was an original design decision that while interesting, was not necessary. Since we used JSON to encode/decode the database, it was unlikely to be corrupted.

Again, this is an opportunity to refactor.

```
//
// Delete Handler
//    Delete/ALL
//    Delete/Name
//
func deleteHandler(w http.ResponseWriter, r *http.Request) {
    var zMem []Page                                    // Empty Database
    name := r.URL.Path[len("/delete/"):]
```

```
    if name == "ALL" {                                          // Process ALL
        data, err := json.Marshal(zMem)                         // Marshal empty Database
        check("Marshalling Failed", err)
        writeData(data)                                         // Write Data Set to disk
        err = json.Unmarshal(data, &xMem) //Reload In-Memory Copy
        check("Unmarshal Failed", err)
        http.Redirect(w, r, "/view/", http.StatusFound)     // Redirect to /view
    }
    p, ok := findExactName(xMem, string(name))              // Not ALL - Find Name
    if !ok {                                                // Report Failure
        fmt.Fprintf(w, "<h1>View: '%s' %s</h1>", name, "not found!")
        return
    } else {
        // Deletion has to be done this way to insure that internal index updated!
        i := 0
        for j, v := range xMem {        // Walk old Database and Create new Database
            if j != p.Index {
                zMem = append(zMem, Page{Index: i, Name: v.Name, Body: v.Body})
                i++
            }
        }
        data, err := json.Marshal(zMem)                      // Marshal new Database
        check("Marshalling Failed", err)
        writeData(data)                                      // Write to disk
        err = json.Unmarshal(data, &xMem)                    //Reload In-Memory Copy
        check("Unmarshal Failed", err)
        http.Redirect(w, r, "/view/", http.StatusFound)      // Redirect to /view
    }
}
```

A Final Word

This White Paper shows what can be done in a relatively short time. While the program worked for its original purpose, it is not a "great" program in many respects. Some of the original design decisions were less than perfect. In a normal working environment, these decisions would revisited and corrected by *Refactoring* the code. I suggest you study this example and attempt to refine it.

This will give you some practice in *refactoring and in depth testing*. I have noted several places in the code that should be refactored.

As a developer, never get "married" to a design decision. If it becomes obvious that a piece of code is awkward or problematic, do not let pride keep you from changing it. *The only people who do not make mistakes are the ones who are not working.*

Strive for simplicity in your code. The *Occam's Razor principle* applies as strongly for software as it does for many other walks of life. In most situations, giving up a little in efficiency to simply your code is often the right decision. Code Simplicity reduces the chances of subtle errors.

As always, comments are welcome at hawthornepresscom@gmail.com.

As to why I have an email address on **gmail** instead of on my website, has to do with AT&T and their inability to handle the concept that not everyone is running a Microsoft OS and my stubborn insistence on running Linux.

Revision History

Date	By	Section	Changes
04/1/2016	C.E. Thornton	All	Initial Document