Golang for Sysadmins - A Practical Introduction

Francis Begyn

2023/04/28

O11y

Introduction

- workshop that gets you started with Go in a practical manner
- first workshop, be gentle!
- follow along: https://gitlab.inuits.io/o11y/o11y-presentations :
  - presentations/intro-to-go-workshop.md

Installation

- through the system packager:
  - `apt install golang`
  - `yay go`
  - `pacman -Syu go`
  - `...`
- from the website:
  1. Download the tarbal from https://go.dev/dl/
  2. Run `tar -C /usr/local -xzf go1.20.3.linux-amd64.tar.gz`
  3. Add the path to `PATH`: `export PATH=$PATH:/usr/local/go/bin`
  4. Test by running `go version`

compiling and running go programs

- initialze a go module `go mod init workshop`
  - Go module is a collection of packages in a file tree
- running go programs without building `go run main.go`
- building go programs `go build main.go`
  - `go build -o workshop main.go`
  - `./workshop`

basics - structure

```
package main

import "fmt"

func main() {
    fmt.Println("hello world")
}
```

basics - variables

- variables can be explicitly declared `var X int`
- but the types can also be infered from the variable `var a = "test"`

```
var a string
a = "test"
var b, c int
c = 2

var d = "initial"
var e = true
```

basics - variables

- `:=` is shorthand for declaring and initializing a variable

```
var a = "apple"
var b, c int = 1, 2
var d = true
```

is equivalent

```
a := "apple"
b, c := 1, 2
d := true
```

basics - constants

- keyword `const`

```
const n = "test"
const a = 1
```

basics - data types

- string
- int, int16, int32, int64
- uint, uint16, uint32, uint64
- float32, float64
- string
- bool
- rune (this is basically the Go equivalent of `char`)

basics - basic data structures

- array:
  - `[2]string`
  - declare and initiliaze `[2]string{"hello","world"}`
  - defined by type AND size
- slice:
  - `[]string`
  - defined by type
  - has `append` function
- map:
  - `map[string]int`
  - accessing the map `var[key]` returns the value
  - check if key exists `exist, val := map[key]`

basics - conditionals

```
if a == b {
  fmt.Println("hello world")
}

if a == b {
  fmt.Println("hello world")
} else {
  fmt.Println("foo bar")
}
```

basics - conditionals

```
if a == b {
  fmt.Println("hello world")
} else if a == c {
  fmt.Println("foo bar")
} else {
  fmt.Println("ooopss")
}
```

basics - loops

- Go only has `for` as a loop keyword

```
for i:=0;i++;i<=10 {
    fmt.Println("hello world")
}
```

- `for each` is implemented by using the `range` keyword

```
words := []string{"hello", "world", "test", "oops"}
for ind, val := range words {
    fmt.Printf("%s - %s"/n, ind, val)
}
```

basics - loops

- The `for` keyword also accepts a condition (turning it into a `while`)

```
search := true
for search {
    fmt.Println("searching...")
    if var == "looking for" {
      search = false
    }
}
```

- An infite loop is very short in Go

```
for {
    fmt.Println("going forever!")
}
```

basics - functions

```
func hello(name string) string {
    return "hello " + name
}

func main() {
    fmt.Println(hello("francis"))
}
```

basics - error handling

- functions can return `error` as a data type

```
func hello(name string) (string, error) {
    if name != "francis" {
      return "", fmt.Errof("woops, you're not the right per
    }
    return "hello " + name, nil
}

func main() {
    fmt.Println(hello("francis"))
    fmt.Println(hello("tom"))
}
```

basics - error handling

```
func hello(name string) (string, error) {
    if name != "francis" {
      return "", fmt.Errof("woops, you're not the right per
    }
    return "hello " + name, nil
}

func main() {
    welcome, err := hello("tom")
    if err != nil {
        fmt.Println("hey, you don't seem the have acces to
        os.Exit(0)
    }
}
```

standard lib

- very extensive
- holds almost anything you want for simple programs

```
import (
    "fmt"
    "io"
    "os"
    "net/http"
)
```

packages and modules

- get dependencies with `go get <dependency path>`
- a module can contain multiple packages
- tracked in `go.mod` and `go.sum` files

basics - up to you now!

1. Install Go
    - through the system packager:
        - `apt install golang`
        - `yay go`
        - `pacman -Syu go`
        - ...
    - from the website:
        1.1 Download the tarbal from https://go.dev/dl/
        1.2 Run `tar -C /usr/local -xzf go1.20.3.linux-amd64.tar.gz`
        1.3 Add the path to `PATH`: `export PATH=$PATH:/usr/local/go/bin`
        1.4 Test by running `go version`

basics - up to you now!

2. Write a small program that will check if the passed `name` is in an array of strings and greet the person when they are. If they are not, kindly inform them of this and handle the error.
    - check if `name` is in array
    - return error when they are not
    - greet them when they are

- 20 min time box
- links:
    - https://gobyexample.com/
    - https://pkg.go.dev/
    - https://gobyexample.com/if-else
    - https://gobyexample.com/slices
    - https://gobyexample.com/maps
    - https://gobyexample.com/functions

basics - solution

```
func authenticate(name string, auth map[string]bool) error
    ok, val := auth[name]
    if !ok {
        return fmt.Errorf("you are not in the authenticated
    }
    if !val {
        return fmt.Errorf("you don't have permission to acc
    }
    return nil
}
```

basics - solution

```
func main() {
    users := make(map[string]bool)
    users["francis"] = true
    users["vince"] = true
    users["tom"] = false
    users["kerre"] = true

    attempt := "francis"
    err := authenticate(attempt, users)
    if err != nil {
        fmt.Printf("we had an error when authenticating: %v
    }
    fmt.Printf("welcome %s!\n", attempt)
}
```

basics - solution

```
func main() {
    users := make(map[string]bool)
    users["francis"] = true
    users["vince"] = true
    users["tom"] = false
    users["kerre"] = true

    attempt = "leeroy"
    err = authenticate(attempt, users)
    if err != nil {
        fmt.Printf("we had an error when authenticating: %v
        os.Exit(0)
    }
    fmt.Printf("welcome %s!\n", attempt)
}
```

Break

- 15 minutes

Go - reading/writing files

- reading/writing file is spread over a selection of packages
- helper functions to easily parse complete files or write files
- can also “open” the file and then manually handle the parsing

```
import (
    "bufio"
    "fmt"
    "io"
    "os"
)
```

reading/writing files

- option to just read the file into memory `os.Readfile`
- same for writing `os.WriteFile`

```
dat, err := os.ReadFile("/tmp/dat")
fmt.Print(string(dat))

d1 := []byte("hello\ngo\n")
err := os.WriteFile("/tmp/dat1", d1, 0644)
```

reading/writing files

- handle the files manually
  - `open()`
  - `create()`
  - `read()`
  - `...`

```
func main() {
    f, err := os.Open("/tmp/dat")
    b1 := make([]byte, 5)
    n1, err := f.Read(b1)
    r4 := bufio.NewReader(f)
    b4, err := r4.Peek(5)
    f.Close()

    f, err := os.Create("/tmp/dat2")
    d2 := []byte{115, 111, 109, 101, 10}
```

HTTP client

- standard library “net/http”

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

- standard request helper functions
  - `.Get(url string)`
  - `.Post(url string, body io.Reader)`
  - `...`

Simple HTTP client

```
func main() {
    resp, err := http.Get("https://gobyexample.com")
    if err != nil {
        panic(err)
    }
    defer resp.Body.Close()


    fmt.Println("Response status:", resp.Status)
    ... handle body parsing ...
}
```

HTTP client

- standard library “net/http”
- standard request helper functions
  - `.Get(url string)`
  - `.Post(url string, body io.Reader)`
- complex queries:
  1. Create client with the right config: `client := http.Client{}`
  2. Create the request: `http.NewRequest(method, ur string, body io.Reader)`
  3. Let the client execute the request: `client.Do(request)`

Slightly more complex HTTP client

```
func (bw *BitwardenClient) CreateItem(item Item) error {
    jsonItem, err := json.Marshal(item)
    req, err := http.NewRequest("POST",
        bw.BaseURL+"/object/item",
        bytes.NewBuffer(jsonItem)
    )
    req.Header.Add("Content-Type", "application/json")
    resp, err := bw.Client.Do(req)
    defer resp.Body.Close()
    var createResp ItemCreateResp
    json.NewDecoder(resp.Body).Decode(&createResp)
    ...
```

HTTP server

- HTTP server using `net/http` is relatively easy
- based on the “handler” concept

HTTP server

- HTTP server using `net/http` is relatively easy
- based on the “handler” concept
- server started with `http.ListenAndServe(":8080", nil)`

HTTP server - handlers

- handlers implement the `http.Handler` interface
- handler functions take `http.Responsewrite` & `htt.Request` arguments
  - `ResponseWriter` can be used to answer

```
func hello(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintf(w, "hello\n") # just send "hello" back
}
```

HTTP server - handlers

- handlers implement the `http.Handler` interface
- handler functions take `http.Responsewrite` & `htt.Request` arguments
  - `ResponseWriter` can be used to answer
  - `Request` holds the request data

```
# This handler does something a little more sophisticated b
# reading all the HTTP request headers and echoing them int
# the response body.
func headers(w http.ResponseWriter, req *http.Request) {
    for name, headers := range req.Header {
        for _, h := range headers {
            fmt.Fprintf(w, "%v: %v\n", name, h)
        }
    }
}
```

HTTP server - starting the server

```
func main() {
    http.HandleFunc("/hello", hello)
    http.HandleFunc("/headers", headers)
    http.ListenAndServe(":8090", nil)
}
```

HTTP excersise

1. Create a little program that uses HTTP client to talk with you're favorite API! No ideas, see the API's below
2. Create HTTP server that combines the code from before (the authentication excersise) and only answers if a correct user is mentioned in the headers.

- remainder of the workshop
- links:
  - https://gobyexample.com/
  - https://pkg.go.dev/net/http
  - https://pkg.go.dev/encoding/json
  - https://gobyexample.com/json
  - https://gobyexample.com/http-client
  - https://gobyexample.com/http-server