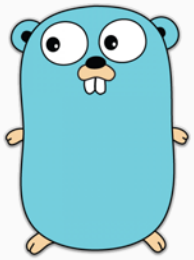
Programming in Go
Lesson 3: Structs, Networking, Testing & Tools

Matt Holiday

30 April 2019

Cardinal Peak



Lesson #2

What we'll cover today:

- Homework #2
- Structs
- JSON
- Simple web clients & servers
- Unit tests
- Static analysis tools

Homework #2

```
package main

import (
    "bytes"
    "fmt"
    "os"
    "strings"

    "golang.org/x/net/html"
)
```

Homework #2

```
var raw = `
<!DOCTYPE html>
<html>
  <body>
    <h1>My First Heading</h1>
      <p>My first paragraph.</p>
      <p>HTML images are defined with the img tag:</p>
      <img src="xxx.jpg" width="104" height="142">
  </body>
</html>`
```

Homework #2

```
func main() {
    doc, err := html.Parse(bytes.NewReader([]byte(raw)))

    if err != nil {
        fmt.Fprintf(os.Stderr, "parse failed: %s\n", err)
        os.Exit(-1)
    }

    words, pics := countWordsAndImages(doc)

    fmt.Printf("%d words and %d images\n", words, pics)
}

// outputs "14 words and 1 images"
```

Homework #2

```
func countWordsAndImages(doc *html.Node) (int, int) {
    var words, images int

    visit(doc, &words, &images)

    return words, images
}
```

Homework #2

```
func visit(n *html.Node, words, pics *int) {
    // if it's an element node then see what tag it has

    if n.Type == html.TextNode {
        *words += len(strings.Fields(n.Data))
    } else if n.Type == html.ElementNode && n.Data == "img" {
        *pics++
    }

    // then visit all the children using recursion

    for c := n.FirstChild; c != nil; c = c.NextSibling {
        visit(c, words, pics)
    }
}
```

Structs

Structs

We've already seen a couple of aggregate types:

- slices & arrays group a sequence of the same type
- maps use one type to index a collection of another type

A `struct` is an aggregate of possibly disparate types

```
type Employee struct {
    Name   string
    Number int
    Boss   *Employee
    Hired  time.Time
}
```

Notice that we can have a pointer to the type we're defining

Structs

In other languages it's a "record" (using database terminology)

It's parts are called "fields" and each must have a unique name

Access to the fields is with "dot" notation

```
employees := make(map[string]*Employee)

var matt = Employee{
    Name:   "Matt",
    Number: 72,
    Boss:   employees["Bernard"],
    Hired:  time.Date(2017, 12, 9, 16, 30, 0, 0, time.UTC),
}

employees[matt.Name] = &matt
```

Struct initialization

With names, selected fields can be initialized

The others default to “zero”

```
employees := make(map[string]*Employee)

var matt = Employee{
    Name:   "Matt",
    Number: 72,
    Hired:  time.Date(2017, 12, 9, 16, 30, 0, 0, time.UTC),
}

employees[matt.Name] = &matt
```

Here `matt.Boss` is left to the default value `nil`

Anonymous Structs

Anonymous structs are possible:

```
var album = struct {
    title  string
    artist string
    year   int,
    copies int,
}{
    "The White Album",
    "The Beatles",
    1968,
    10000000,
}
```

Initialization can be done without names by setting all fields in the correct order

Struct compatibility

Two `struct` types are compatible if

- the fields have the same types and names
- in the same order
- and the same tags (more on this later)

A `struct` may be copied or passed as a parameter in its entirety

A `struct` is comparable if all its fields are comparable

The zero value for a `struct` is “zero” for each field in turn

Passing structs

Structs are passed by value unless a pointer is used

```
var white album

func soldAnother(a album) {
    // oops
    a.copies++
}

func soldAnother(a *album) {
    // what you intended
    a.copies++
}
```

Note that “dot” notation works on pointers too, same as `(*a).copies`

Struct gotcha

Here's an annoying little issue in Go:

```
// Note -- not a pointer anymore
employees := make(map[string]Employee)

var matt Employee{
    Name:   "Matt",
    Number: 72,
    Boss:   &bernard,
    Hired:  time.Date( . . . ),
}

employees[matt.Name] = matt

employees["Matt"].Number++          // can't do this
```

A map entry is not addressable; issue #3117

Make the zero value useful

"It is usually desirable that the zero value be a natural or sensible default.

For example, in `bytes.Buffer`, the initial value of the struct is a ready-to-use empty buffer, and the zero value of `sync.Mutex` . . . is a ready-to-use unlocked mutex." (*GOPL* §4.4)

Empty structs

A `struct` with no fields is useful; it takes up no space

These examples draw on future material:

```
// a set type (instead of bool)

var isPresent map[int]struct{}

// a way to warn on bad struct copies (go vet)

type noCopy struct{}

func (*noCopy) Lock()   {}
func (*noCopy) Unlock() {}

// a very cheap channel type

done := make(chan struct{})
```

Struct Tags and JSON

Struct tags

Tags are a part of a `struct` definition captured by the compiler

They are available to code that uses *reflection*

```
type Response struct {
    Page  int      `json:"page"`
    Words []string `json:"words,omitempty"`
}
```

Sometimes multiple tags are appropriate, separated by a space

```
type Response struct {
    Page  int      `json:"page" db:"page"`
    Words []string `json:"words,omitempty" db:"words"`
}
```

Reflection in action: JSON support

The JSON package in Go uses reflection on Go objects

```
r := &Response{Page: 1, Words: []string{"up", "lo", "an"}}

j, _ := json.Marshal(r)                    // ignoring errs
fmt.Println(string(j))

var r2 Response

json.Unmarshal(j, &r2)                     // ignoring errs
fmt.Printf("%#v\n", r2)

// {"page":1,"words":["up","lo","an"]}
// main.Response{Page:1, Words:[]string{"up", "lo", "an"}}
```

Reflection in action: JSON support

"omitempty" causes a nil object to be ignored by `Marshal`

```
r := &Response{Page: 1, Words: []string{}}

j, _ := json.Marshal(r)                    // ignoring errs
fmt.Println(string(j))

var r2 Response

json.Unmarshal(j, &r2)                     // ignoring errs
fmt.Printf("%#v\n", r2)

// {"page":1}
// main.Response{Page:1, Words:[]string(nil)}
```

Struct tags have many uses

Tags can also be used in conjunction with SQL queries

```
import "github.com/jmoiron/sqlx"

type item struct {
    Name string `db:"name"`
    When string `db:"created"`
}

func PutStats(db *sqlx.DB, item *item) error {
    stmt := `INSERT INTO items (name, created)
             VALUES (:name, :created);`
    _, err := db.NamedExec(stmt, item)

    return err
}
```

Struct tag gotcha

Only **exported** (capitalized) field names are convertible

```go
import "github.com/jmoiron/sqlx"

type item struct {
    Name string `db:"name"`
    when string `db:"created"` // oops
}

func PutItem(db *sqlx.DB, item *item) error {
    stmt := `INSERT INTO items (name, created)
             VALUES (:name, :created);`

    // fails, missing when
    _, err := db.NamedExec(stmt, item)

    return err
}
```

Networking with HTTP

Go network libraries

The Go standard library has many packages for making web servers:

That includes:

- client & server sockets
- route multiplexing
- HTTP and HTML, including HTML templates
- JSON and other data formats
- cryptographic security
- SQL database access
- compression utilities
- image generation

There are also lots of 3rd-party packages with improvements

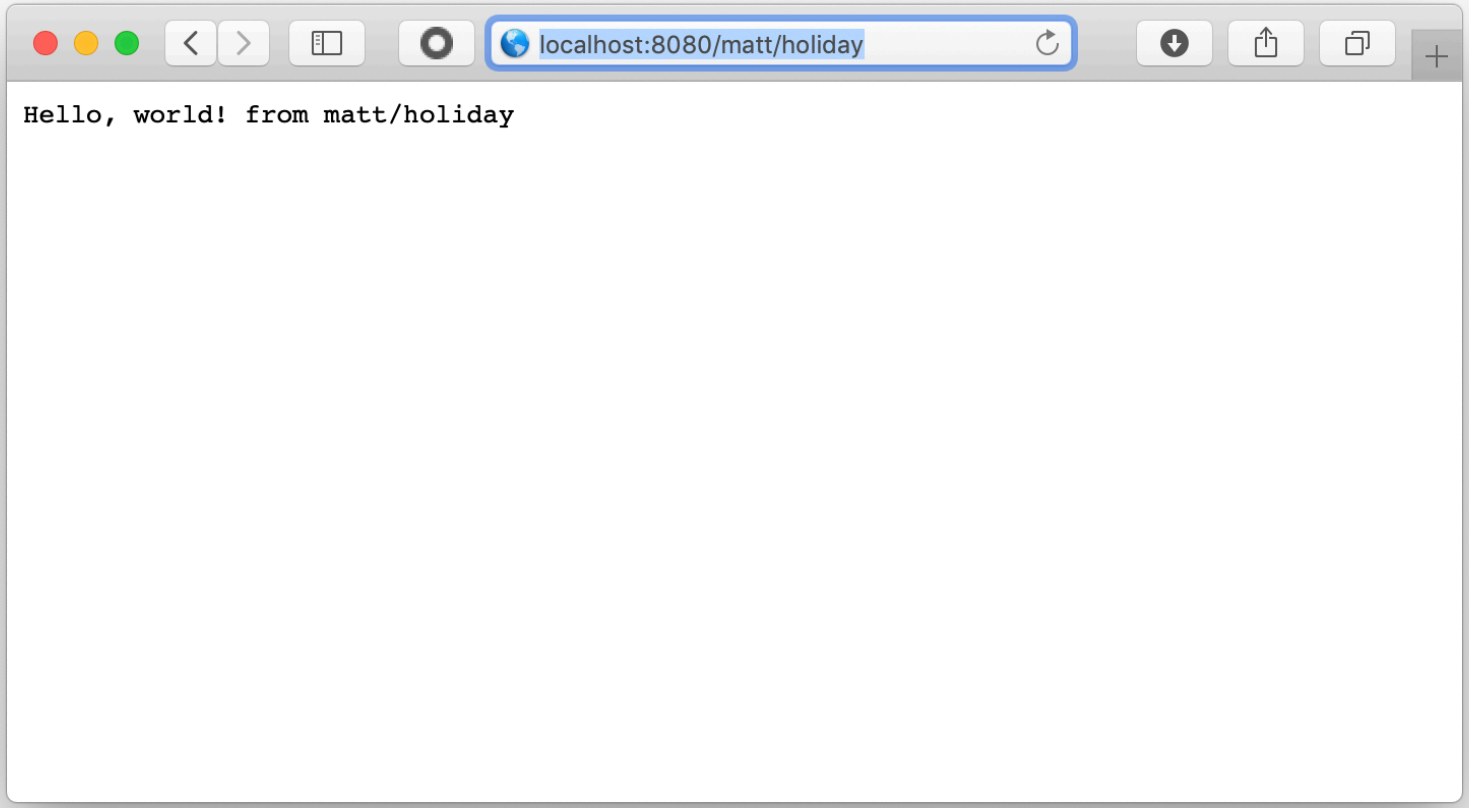
A very short web server

A simple web server is almost a one-line program

```
package main

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

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, world! from %s\n", r.URL.Path[1:])
}

func main() {
    http.HandleFunc("/", handler)
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

A very short web server

Go run a web server:



Go HTTP design

An HTTP handler function is an instance of an interface

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}

type HandlerFunc func(ResponseWriter, *Request)

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

// handler matches type HandlerFunc and so interface Handler
// so the HTTP framework can call ServeHTTP on it

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, world! from %s\n", r.URL.Path[1:])
}
```

A very short web client

A simple web client takes a little bit more

```
package main

import ("fmt"; "io/ioutil"; "net/http"; "os")

func main() {
    resp, _ := http.Get("http://localhost:8080/" + os.Args[1])
    defer resp.Body.Close()

    if resp.StatusCode == http.StatusOK {
        body, _ := ioutil.ReadAll(resp.Body)
        fmt.Println(string(body))
    }
}

// $ go run client.go matt/holiday
// Hello, world! from matt/holiday
```

A simple JSON REST client

```go
package main

import ("fmt"; "io/ioutil"; "net/http")
const url = "https://jsonplaceholder.typicode.com"

func main() {
    resp, _ := http.Get(url + "/todos/1")
    defer resp.Body.Close()

    if resp.StatusCode == http.StatusOK {
        body, _ := ioutil.ReadAll(resp.Body)
        fmt.Println(string(body))
    }
}

// $ go run client.go
// {"userId": 1, "id": 1, "title": "delectus aut autem",
//  "completed": false}
```

A simple JSON REST client

```
package main

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "net/http"
)

type todo struct {
    UserID    int    `json:"userID"`
    ID        int    `json:"id"`
    Title     string `json:"title"`
    Completed bool   `json:"completed"`
}

const base = "https://jsonplaceholder.typicode.com"
```

A simple JSON REST client

```
func main() {
    var item todo

    resp, _ := http.Get(base + "/todos/1")

    defer resp.Body.Close()

    body, _ := ioutil.ReadAll(resp.Body)

    _ := json.Unmarshal(body, &item)

    fmt.Printf("%#v\n", item)
}

// $ go run client.go
// main.todo{UserID:1, ID:1, Title:"delectus aut autem",
//           Completed:false}
```

Serving from a template

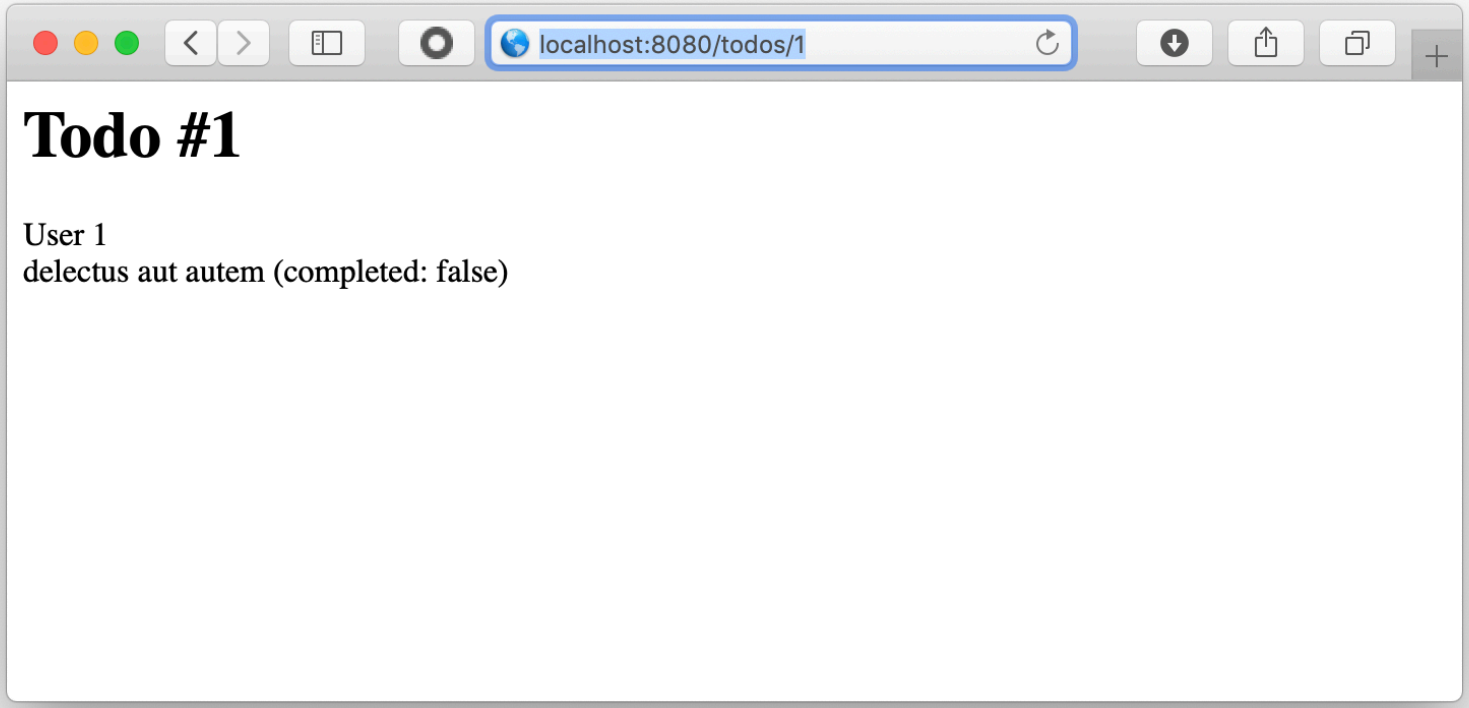
```
package main

import (
    "encoding/json"
    "html/template"
    "io/ioutil"
    "log"
    "net/http"
)

type todo struct {
    UserID    int    `json:"userID"`
    ID        int    `json:"id"`
    Title     string `json:"title"`
    Completed bool   `json:"completed"`
}

const base = "https://jsonplaceholder.typicode.com/"
```

Serving from a template

```
var form = `
<h1>Todo #{{.ID}}</h1>
<div>{{printf "User %d" .UserID}}</div>
<div>{{printf "%s (completed: %t)" .Title .Completed}}</div>`

func handler(w http.ResponseWriter, r *http.Request) {
    var item todo

    resp, _ := http.Get(base + r.URL.Path[1:])
    defer resp.Body.Close()
    body, _ := ioutil.ReadAll(resp.Body)
    _ = json.Unmarshal(body, &item)

    tmpl := template.New("mine")

    tmpl.Parse(form)
    tmpl.Execute(w, item)
}
```

Serving from a template

```
func main() {
    http.HandleFunc("/", handler)
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```



Serving up an error

```
func handler(w http.ResponseWriter, r *http.Request) {
    var item todo

    resp, _ := http.Get(base + r.URL.Path[1:])
    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK {
        http.NotFound(w, r)
        return
    }

    body, _ := ioutil.ReadAll(resp.Body)
    _ = json.Unmarshal(body, &item)

    tmpl := template.New("mine")
    tmpl.Parse(form)
    tmpl.Execute(w, item)
}
```

Serving up a wiki

See the Golang article Writing Web Applications

The tutorial includes:

- creating a data structure with load and save methods
- using the net/http package to build web applications
- using the html/template package to process HTML templates
- using the regexp package to validate user input
- using closures

Testing in Go

Go unit tests

Go has standard tools and conventions for unit testing

Test files end with `_test.go` and have `TestXXX` functions

Typically they're in a separate directory, but not necessarily

You run tests with `go test`

```
go test ./test
ok   bvi/test 56.841s
go test ./pkg/acedb
ok   bvi/pkg/acedb (cached)
```

Tests aren't run if the source wasn't changed since the last test

Unit test functions

Test functions have the same signature using `testing.T`

```
func TestCrypto(t *testing.T) {
    uuid := "650b5cc5-5c0b-4c00-ad97-36b08553c91d"
    key1 := "75abbabc1f9f8d28d55200b43fd95962"
    key2 := "75abbabc1f9f8d28d66200b43fd95962"

    ct, err := secrets.MakeAppKey(key1, uuid)

    if err != nil {
        t.Errorf("make failed: %s", err)
    }

    . . .
}
```

Errors are reported through that parameter and fail the test

Static Analysis

Static analysis

"Static" means we're not running the program

We can analyze source files and find many possible defects

- style issues
- security issues
- things that might compile but be wrong
- excessive code complexity
- inefficient structure layout

We can also automatically

- reformat code
- update import lists

Start clean, stay clean

Cruft and “technical debt” can build up pretty quickly

Not all code smells will be found by these static tools

Pay heed to the tools on every file save and every commit

Put little bits of time in every day to prevent lots of work later

Gofmt and Goimports

`gofmt` will put your code in standard form (spacing, indentation)

It will notify you if something was changed

`goimports` will do that and also update import lists

The standard practice is to run one or the other on every save in your IDE/editor (as a save file hook)

They can also be run as a pre-commit hook in your local repo

Golint

`golint` will check for non-format style isses, for example:

- exported names should have comments for `godoc`
- names shouldn't have `under_scores` or be in `ALLCAPS`
- `panic` shouldn't be used for normal error handling
- the error flow should be indented, the happy path not
- variable declarations shouldn't have redundant type info

The "rules" are based on Effective Go and Google's
Go Code Review Comments

Go vet

`go vet` will find some issues the compiler won't

For example, it will report suspicious "printf" format strings

It can flag cases were you've accidentally copied a mutex type

It can find cases of "shadowing" that might be in error
(Go 1.12 makes this part harder to run, but still worth it)

Run it before you build

Other tools

There are too many tools to list

`gosec` looks for possible security issues

`ineffasign` finds assignments that are “ineffective” (shadowed?)

`gocyclo` reports high cyclomatic complexity in functions

Treat these things as **warnings** because there are false positives

Many of these can be run by the uber-tool `metalinter`

(I let the GoLand IDE run it on every save)

Download with go get

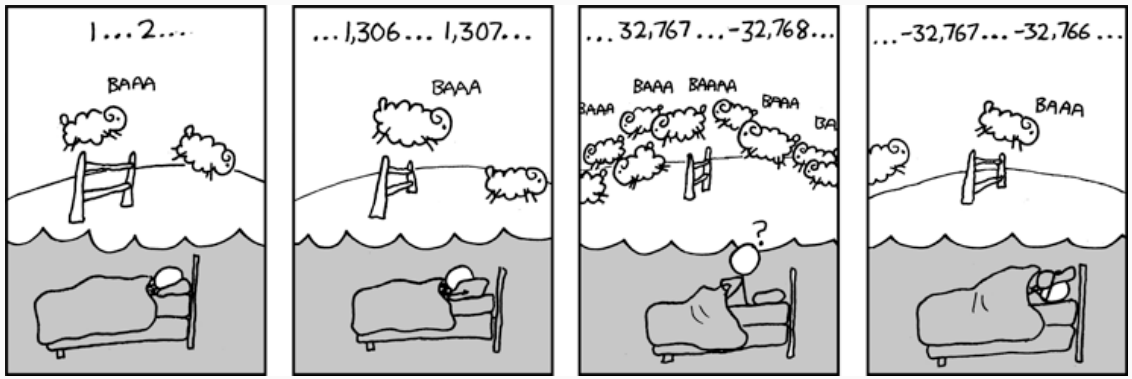
```
go get golang.org/x/lint/golint
go get golang.org/x/tools/cmd/goimports
go get github.com/fzipp/gocyclo
go get github.com/gordonklaus/ineffassign
go get github.com/securego/gosec/cmd/gosec
go get honnef.co/go/tools/cmd/staticcheck
```

Homework

Homework #3

Exercise 4.12 from *GOPL:* fetching from the web

The popular web comic “xkcd” has a JSON interface. For example, a request to https://xkcd.com/571/info.0.json produces a detailed description of comic 571, one of many favorites. Download each URL (once!) and build an offline index. Write a tool xkcd that, using this index, prints the URL *and date* of each comic whose *transcript* matches a search string provided on the command line.



Homework #3

What the raw data looks like:

```
{
 "month":      "4",
 "num":        571,
 "link":       "",
 "year":       "2009",
 . . .
 "transcript": "[[Someone is in bed, . . . long int.",
 "img":        "https://imgs.xkcd.com/comics/cant_sleep.png",
 "title":      "Can't Sleep",
 "day":        "20"
}
```

For the query “Someone is in bed” we get the URL and date:
`https://xkcd.com/571/ 4/20/2009`