Collaboration and Crawling W/ Google's Go (Golang) Language

Golang Modules, Git, gRPC, and Scraping Twitter

Todd McLeod & Daniel Hoffmann

Table of Contents

Getting started

Welcome

Welcome to the course. You are taking a great step by enrolling in this course. Better skills create a better life. ***You are on your way to a better life.*** As you learn new skills, you are building a better life. ***I commend you for your efforts to improve your life.*** As you improve your life, you are improving the world --- one person at a time. You are making the world better, and you are making your life better. This is a win-win for everybody. Great work! Also, ***this is your course. Use it in the way which is best for you.*** If you want to skip ahead, skip ahead. As your teacher, my job is to help you succeed. The content here is designed to help you succeed both with visual studio code, and also as a student and in life.
Video file: 01 welcome

Course outline

You can find everything I use in the course and all of the courses resources in the COURSE OUTLINE which is attached to this lecture on UDEMY

- github - https://github.com/GoesToEleven/golang-project

Video file: 03

Credentials

I am ***a tenured professor in California.*** I have taught at both the college and university level. I have also taught in multiple disciplines including business, information systems, computer science, and online education. In 1997, I was one of the first professors at the university to teach online. I did post-graduate work in online education at UC San Diego. I co-founded and taught in the “Online Teacher’s Training Program” which trained professors how to teach online. In 2008, I was selected as one of the best instructors in the entire California Community College system. Currently, when measured by the number of students served, I am the world’s leading

trainer in Google's new programming language which is one of the fastest growing, highest paying programming languages in America. My background in business, information systems, computer science, and online education has prepared me to teach this course!
Video file: 02 credentials

A little philosophy

Built for humans

This course has been built for humans. Are you human? If "yes" then great! This course has been built for you. What does that mean to be built for humans? As software engineers, we are creating tools for humans. The more we understand humans, ourselves and others, the better we are going to do in life and with the products we create. Too frequently in software engineering, and in many other endeavors too, the human element is ignored. When I speak with managers from companies, I like to ask them, "What can I do as a professor to better prepare software engineers for you?" The answer I invariably receive is to teach human skills; soft skills. So ***this course has been built to include a human component;*** ***this course has been built for humans.*** Some examples relevant to mention now:

- multiple intelligences
    - CSCI also equals, for me: collectively stupid, collectively intelligent
    - collaboration is key to greater success; collaboration entails
        - listening
        - learning
        - being humble
        - being comfortable being wrong
        - working with others
    - "The more I admit I do not know, the higher I go; the higher I go, the harder it becomes to admit I do not know."

Video file: 04 Built For Humans

How to be successful

Understanding what has made others successful can help you become successful. These are principles which have helped me become successful. I learned these principles from others and from my own experience. I share these **principles to help you succeed in this course and in life:**

- **GRIT**
    - Grit - Angela Duckworth
- **DRIP LEARNING**
    - Time on task
    - Small frequent engagements

    - Multiple perspectives, multiple engagements
    - my teachers
        - drop by drop, the bucket gets filled
        - persistently, patiently, you are bound to succeed
- **FOCUS**
    - Bill Gates & Warren Buffett
- **PLANNING**
    - Bill Gates, “If you want to be successful, get in front of what’s coming and let it hit you.”
- **RESOURCES**
    - The 7 Habits of Highly Effective People
        - https://drive.google.com/file/d/0B22KXlqHz6ZNQ1VwM21ZR1FiOGM/view?usp=sharing&resourcekey=0-o4aHZPlgaITJUSAI9XsEjw

Video file: 06 Success

Git essentials

Introduction

This is an introduction to Daniel and git
video: 13 introduction

Understanding ssh encryption

- working in windows on the terminal / bash / shell
    - as opposed to command prompt / cmd / dos prompt / powershell
- add an ssh public key to github

video: 14 ssh encryption

Creating our github repo

- working in windows
- git, and our git repo

video: 15 creating github repo

Adding a collaborator to github

adding Daniel as a collaborator to the github repo
video: 16 adding github collaborator

Cloning a public project

Daniel clones the repo for our class

- cloning and forking explained
- how branches work with this

video: 17 cloning a public project

Forking a public project

When you fork a project, you make a copy of someone else's repo and it gets put under your account. Here is how you fork a public project, and submit a pull request from the master branch

video: 18 forking a public project

Git branches in action

Working with merge conflicts

An example of a merge conflict and how it gets resolved. Using branches helps you avoid merge conflicts.

video: 19 merge conflict

Creating a branch

- Commands used in video
    - show branches — git branch
        - all branches — git branch -a
    - create a new branch — git checkout -b <branch name>

before creating a branch, get your current state of the remote repository

- **git pull**

when you create a branch it's created off of whatever branch you're already on

video: 20 creating a branch

Working with branches

- checkout out a branch from master — git checkout <branch name>

video 21 working with branches

Merging branches

- on github
- on terminal

    - get onto the branch you want to merge INTO, then PULL, then merge
    - merge a branch — git merge <branch name>

video: 22 merging branches

Deleting branches

It is always a good idea to make things current before you work on branches. Do this with a GIT PULL. Once you have done that, switch to the master branch, then delete whatever branches you no longer want. **Make sure you have merged any changes you want from a branch before you delete it.**

- delete a branch — git branch -d <branch name>

video: 23 delete branch

Workflow summary

Prefixing branches with an issue number from github is a useful practice.

- create branch — git checkout -b <branch name>
    - see branches — git branch
    - see all branches — git branch -a
- work on a branch — git checkout <branch name>
- see status of branch — git status
- stage files — git add -A
    - see staged files — git status
- commit files — git commit -m <some message>
- switch back to master branch — git checkout master
- merge your branch — git merge <branch name>
- push your code to remote repo — git push [origin master]
- delete your branch — git branch -d <branch name>
- clean up cache — git remote prune origin

video: 24 summary

Hands-on exercises - ninja level 1

Introduction

Hands-on exercises will help you learn more efficiently and effectively. By doing these hands-on exercises, you will **learn faster and remember longer.** We encourage you to join us on these hands-on exercises, and to have fun while doing them!

- Housekeeping
    - two computers - we're both on a computer
        - sometimes you see my screen
        - sometimes you see Daniel's screen

    - we've created some structure to our github repo; we added folders to segment the code and moved files into those folders.
    - twitter request is how we're partly determining what we're teaching in this course

video 25 intro hands on

Hands-on exercise #1

An introduction in a class is a common practice. This lets the students learn about the other students in the course. In this hands-on exercise, we will introduce ourselves.

In this exercise, you will merge

FROM

- the **master on your fork**

TO

- **master on the repo from which you forked**
    - **(** https://github.com/GoesToEleven/golang-project **)**

fork this repo

- https://github.com/GoesToEleven/golang-project
- on the branch **master**, go into the folder "**002-hands-on-exercises/01-about**"
- make a copy of the file "**000-TEMPLATE-about-me.txt"**
- rename ***your copy*** to
    - "**<your github username>.txt**"
- in the file, to the extent that you feel comfortable, introduce and share about yourself
- stage this file
- commit this file
    - provide a descriptive commit message
- push this commit to your fork
- create a pull request to branch **master** on this repo
    - https://github.com/GoesToEleven/golang-project

Note: the structure of this file will allow us to later parse the information into other forms of storage, eg, take it into a database; do things with it. Second note: if for some reason the template is no longer there, simply provide a description.

video: 26 hands on 01

Hands-on exercise #1 - todd's solution

Here is Todd's solution.

video: 27 todds solution

Hands-on exercise #2

In this exercise, you will merge

FROM

- the **branch (off master) on your fork**

TO

- **master on the repo from which you forked**
    - **(** https://github.com/GoesToEleven/golang-project **)**

At your fork of this repo https://github.com/GoesToEleven/golang-project do the following

- create a branch titled "**myhumble**"
    - switch to the branch
- in the folder "**002-hands-on-exercises/02-humble**" create a file called "**<your github username>-humble.txt**"
- in the file, share what "being humble" means to you
- stage this file
- commit this file
    - provide a descriptive commit message
- push this file to your fork
- create a pull request to branch **master** on this repo
    - https://github.com/GoesToEleven/golang-project

video: 28 hands on 02

Exploring git rebase

- git remote
- git remote show origin
- git remote add upstream <ssh url>
- git fetch upstream
- git checkout master
    - confirm with
        - git branch -a
- git pull
- two ways to pull from FORKED repo
    - **git rebase upstream/master**
        - modifies history
        - can cause issue if people have FORKED your FORK
    - **git merge upstream/master**
        - doesn't modify history
        - safer if people have forked your fork
- now push to your FORKED repo on github - either the first or, if needed, the second
    - git push
    - git push -f origin master

Notes from StackOverflow:

In your local clone of your forked repository, you can add the original GitHub repository as a "remote". ("Remotes" are like nicknames for the URLs of repositories - `origin` is one, for example.) Then you can fetch all the branches from that upstream repository, and rebase your work to continue working on the upstream version. In terms of commands that might look like:

```
# Add the remote, call it "upstream": git remote add upstream
https://github.com/whoever/whatever.git # Fetch all the branches of that remote
into remote-tracking branches, # such as upstream/master: git fetch upstream #
Make sure that you're on your master branch: git checkout master # Rewrite your
master branch so that any commits of yours that # aren't already in
upstream/master are replayed on top of that # other branch: git rebase
upstream/master
```

If you don't want to rewrite the history of your master branch, (for example because other people may have cloned it) then you should replace the last command with `git merge upstream/master`. However, for making further pull requests that are as clean as possible, it's probably better to rebase.

If you've rebased your branch onto `upstream/master` you may need to force the push in order to push it to your own forked repository on GitHub. You'd do that with:

```
git push -f origin master
```

You only need to use the `-f` the first time after you've rebased.
source
video: 29 rebase

Hands-on exercise #2 - todd's solution

Here is Todd's solution
video 30 hoe 02 todd

Hands-on exercise #3

In this exercise, you will merge

FROM:

- the **master on your fork**

TO

- **master on the repo from which you forked**
  - **(** https://github.com/GoesToEleven/golang-project **)**

At your fork of this repo https://github.com/GoesToEleven/golang-project in the folder "**002-hands-on-exercises/03-purpose**" do the following

- create a branch titled "**mypurpose**"
  - switch to the branch
  - create a file called "**<your github username>-purpose.txt**"
    - share what you believe gives purpose and meaning to human life
  - stage this file
  - commit this file
    - provide a descriptive commit message

    - merge this branch to the **master** branch ***ON YOUR FORK***
- create a pull request from branch **master on your fork** to branch **master** on this repo
    - https://github.com/GoesToEleven/golang-project

video: 31 hoe 03 daniel

Hands-on exercise #3 - todd's solution

Here is Todd's solution
video 32 hoe03 todd

Hands-on exercise #4

Delete your branches.

- git branch -d <branch name>

Once we accept your pull request (which might take a few months), you can delete your branch on your fork on github, and then prune locally.

- git remote prune origin

video: 33 hoe4

Go modules - intro

Overview

An overview of go modules.

- GOPATH will always still work for backwards compatibility
- Modules is the new recommended way of doing things however

video: 34 overview

Package management

Understanding package management and dependencies. Different names, mostly the same meaning:

- packages
- libraries
- groupings of other people's code (OPC)
- groupings of your own code

*"A **module** is a collection of related Go **packages**. Modules are the unit of source code **interchange and versioning.** The go command has direct support for working with modules, including recording and resolving dependencies on other modules. Modules replace the old GOPATH-based approach to specifying which source files are used in a given build."*

- source - at the terminal: **go help modules**

Using other people's code comes with risks. Great article by Russ Cox:

- Our Software Dependency Problem

    - *"Note that while the go command makes adding a new dependency quick and easy, it is not without cost. Your module now literally depends on the new dependency in critical areas such as **correctness, security, and proper licensing**, just to name a few."*
        - ***THIS APPLIES TO ALL DEPENDENCIES / USING OTHER PEOPLE'S CODE***
- npm "left-pad" story
    - direct dependencies
    - indirect dependencies

Modules are usually one-to-one with a repo, though there are ways to put multiple modules in one repo.
video 35 package management

Reading Documentation

- ***A module is a collection of Go packages stored in a file tree with a go.mod file at its root. The go.mod file defines the module's module path, which is also the import path used for the root directory, and its dependency requirements, which are the other modules needed for a successful build. Each dependency requirement is written as a module path and a specific semantic version.***
    - ***NAMESPACING***
        - ***not: you forgot someone's name (spaced on their name)***
        - ***is: you have a unique name to identify different things***
            - ***people example***
            - ***code example***
- ***As of Go 1.11, the go command enables the use of modules when the current directory or any parent directory has a go.mod, provided the directory is outside $GOPATH/src. (Inside $GOPATH/src, for compatibility, the go command still runs in the old GOPATH mode, even if a go.mod is found.***
- ***GO COMMAND DOCUMENTATION***
    - ctrl + f → "module"
- Starting in Go 1.13, module mode will be the default for all development.
    - source
- **Golang blog**
    - this section of the curriculum draws from, and in some cases closely parallels, these go blog articles:
    - Using Go Modules Tyler Bui-Palsulich and Eno Compton
    - Migrating to Go Modules Jean de Klerk
    - Module Mirror and Checksum Database Launched Katie Hockman

video 36 documentation

Creating a new module

Create a new, empty directory somewhere outside $GOPATH/src.

- **go mod init <name spacing, eg, example.com/hello>**

file	code

hello.go	`package hello`
	
	`func Hello() string {`
	`  return "Hello, world."`
	`}`

hello_test.go	`package hello`
	
	`import "testing"`
	
	`func TestHello(t *testing.T) {`
	`  want := "Hello, world."`
	`  if got := Hello(); got != want {`
	`  t.Errorf("Hello() = %q, want %q", got, want)`
	`  }`
	`}`

- **go test**

The go.mod file only appears in the root of the module. Packages in subdirectories have import paths consisting of the module path plus the path to the subdirectory.

- **cat go.mod**

video 37 creating module

Adding a dependency

The primary motivation for Go modules was to improve the experience of using (that is, adding a dependency on) code written by other developers. ***Go resolves imports by using the specific dependency module versions listed in go.mod.***

- Let's update our hello.go to import rsc.io/quote and use it to implement Hello

file	code

hello.go	`package hello`
	
	`import "rsc.io/quote"`
	
	`func Hello() string {`
	`  return quote.Hello()`
	`}`

hello_test.go	package hello
	
	import "testing"
	
	func TestHello(t *testing.T) {
	    want := "Hello, world."
	    if got := Hello(); got != want {
	        t.Errorf("Hello() = %q, want %q", got, want)
	    }
	}

- **go test**
- **cat go.mod**

NOTES:

- if you import a package/library (call it what you will) not yet tracked by go.mod, by default:
  - **go adds it to go.mod, using the latest version.**
  - "Latest" is defined as
    - latest tagged stable (non-prerelease) version, or
    - latest tagged prerelease version, or
    - latest untagged version.
- see the version of rsc.io/quote
  - **cat go.mod**
  - only direct dependencies are recorded in the go.mod file
- two types of dependencies
  - ***DIRECT dependency***
  - ***INDIRECT dependency***
- see all ***direct*** and ***indirect*** dependencies
  - **go list -m all**

video 38 adding dependency

Security assurance with go.sum

In addition to go.mod, the go command maintains a file named go.sum containing the expected cryptographic hashes of the content of specific module versions:

- **cat go.sum**

go.sum ensures that future downloads of modules retrieve the same bits as the first downloads. This ***ensures the modules your project depends on do not change unexpectedly,*** whether for ***malicious***, ***accidental***, or other reasons.

- ***Both go.mod and go.sum should be checked into version control.***

DISCUSSION

- **Why check go.mod and go.sum into version control?**

DISCUSSION & EXPLORATION

- Modules already downloaded are cached locally (in $GOPATH/pkg/mod).

video 39 go sum

Go modules - deeper dive

Understanding versioning

With Go modules, **versions are referenced with semantic version tags.** A semantic version has three parts:

- **major.minor.patch**

For example, for v0.1.2,

- major version is 0
- minor version is 1
- patch version is 2

Here is a description of each part:

- MAJOR
  - ***backwards incompatible*** changes added
- MINOR
  - ***backwards compatible*** changes added
- PATCH
  - ***backwards compatible*** bug fixes

Learn more about semantic versioning.

DISCUSSION

- If we see a dependency go **from v1.8.9 to v2.0.0**, what should we do?

video: 40 versioning

Upgrading a minor dependency

Let's walk through a couple minor version upgrades. Then, we'll consider a major version upgrade. From the output of ...

- **go list -m all**

... we can see we're using an untagged version of golang.org/x/text. The golang.org/x/text version is something like this ...

- v0.0.0-20170915032832-14c0d48ead0c

… and is an example of a **pseudo-version,** which is the go command's version syntax for a specific untagged commit. Let's upgrade to the latest tagged version - either of these commands work (and we will see how we can do **@v1.3.1** in the next video):

- **go get golang.org/x/text**
- **go get golang.org/x/text@latest**

Each argument passed to go get can take an explicit version. The default is **@latest** which resolves to the latest version as defined earlier. Now let's test that everything still works:

- **go test**

What do our dependencies look like now?

- **go list -m all**

The golang.org/x/text package has been upgraded to the latest tagged version. What does our go.mod file look like now?

- **cat go.mod**

The indirect comment indicates a dependency is not used directly by this module, only indirectly by other module dependencies. See **go help modules** for details.
video 41 upgrade minor dependency

Specifying dependency version

See all of your direct and indirect dependencies

- **go list -m all**

Now let's try ***upgrading the rsc.io/sampler minor version.***

- **go get rsc.io/sampler@latest**
- **go test**

Uh, oh! The test failure shows that the latest version of rsc.io/sampler is incompatible with our usage. Let's list the available tagged versions of that module.

- **go list -m -versions rsc.io/sampler**

rsc.io/sampler v1.99.99 should have been backwards-compatible with rsc.io/sampler v1.3.0, but bugs or incorrect client assumptions about module behavior can both happen.
Let's get a different version

- **go get rsc.io/sampler@v1.3.1**

Each argument passed to go get can take an explicit version

- the default is **@latest** which resolves to the latest version as defined earlier.
- or you can specify the version like the above **@v1.3.1**

video 42 specify version

Adding a new major dependency

Let's add a new function to our package:

file	code

hello.go	`package hello`
	
	`import (`
	`  "rsc.io/quote"`
	`  quoteV3 "rsc.io/quote/v3"`
	`)`
	
	`func Hello() string {`
	`  return quote.Hello()`
	`}`

	`func Proverb() string {`
	`    return quoteV3.Concurrency()`
	`}`

`hello_test.go`	`package hello`
	
	`import "testing"`
	
	`func TestHello(t *testing.T) {`
	`    want := "Hello, world."`
	`    if got := Hello(); got != want {`
	`        t.Errorf("Hello() = %q, want %q", got, want)`
	`    }`
	`}`
	
	`func TestProverb(t *testing.T) {`
	`    want := "Concurrency is not parallelism."`
	`    if got := Proverb(); got != want {`
	`        t.Errorf("Proverb() = %q, want %q", got, want)`
	`    }`
	`}`

- **go test**
- **cat go.mod**
- **go list -m all**
- **go list -m rsc.io/q...**

Semantic import versioning

- you can have **different versions of the same package** / library / code (call it what you will)
- Each different major version (v1, v2, and so on) uses a different module path
    - starting at v2, the path must end in the major version
    - this gives incompatible packages (different major versions) different names
    - **you can only use one of each major version** - example:
        - rsc.io/quote
        - rsc.io/quote/v2
        - rsc.io/quote/v3, and so on.
    - it is impossible for a program to build with both rsc.io/quote v1.5.2 and rsc.io/quote v1.6.0

- allowing different major versions of a module (because they have different paths) gives module consumers the ability to **upgrade** to a new major version **incrementally**.
    - The ability to **migrate incrementally** is especially important in a large program or codebase.

video: 43 adding major dependency

Upgrading a major dependency

Let's complete our conversion **from rsc.io/quote to only rsc.io/quote/v3.** If we look at the documentation of the package using ***go doc*:**

- **go doc rsc.io/quote/v3**

Because of the major version change, we should expect that some APIs may have been removed, renamed, or otherwise changed in incompatible ways. Reading the docs, we can see that ***Hello has become HelloV3.***

file	code

hello.go	`package hello`
	
	`import "rsc.io/quote/v3"`
	
	`func Hello() string {`
	`  return quote.HelloV3()`
	`}`
	
	`func Proverb() string {`
	`  return quote.Concurrency()`
	`}`

hello_test.go	`package hello`
	
	`import "testing"`
	
	`func TestHello(t *testing.T) {`
	`  want := "Hello, world."`
	`  if got := Hello(); got != want {`
	`  t.Errorf("Hello() = %q, want %q", got, want)`
	`  }`
	`}`
	
	`func TestProverb(t *testing.T) {`

```
    want := "Concurrency is not parallelism."
    if got := Proverb(); got != want {
        t.Errorf("Proverb() = %q, want %q", got,
want)
    }
}
```

- **go test**

video 44 upgrading dependency

Removing unused dependencies

We've removed all our uses of rsc.io/quote, but it still shows up in **go list -m all** and in our **go.mod** file:

- **cat go.mod**
- **go list -m all**

Building a single package, like with go build or go test, can easily tell when something is missing and needs to be added, but not when something can safely be removed. ***Removing a dependency can only be done after checking all packages in a module, and all possible build tag combinations for those packages.*** An ordinary build command does not load this information, and so it cannot safely remove dependencies. The ***go mod tidy*** command cleans up these unused dependencies:

- **go mod tidy**
- **cat go.mod**
- **go list -m all**

video 45 tidy

Go modules - good to know

Automatic conversion

If you are already using dependency manager like **dep** or **glide** or something else, the command **go mod init** will automatically convert your dependency management to go modules. These are the [older dependency management tools] that automatically convert to go modules. To convert,

- navigate to the root of your project
- **go mod init**
- **go mod tidy**

Note that other dependency managers may specify dependencies at the level of individual packages or entire repositories (not modules), and generally do not recognize the requirements specified in the go.mod files of dependencies. Consequently, ***you may not get exactly the***

same version of every package as before, and there's some risk of upgrading past breaking changes. Therefore, it's important to follow the above commands with an audit of the resulting dependencies. To do so, run

- **go list -m all**

Upgrade or downgrade to the correct version using **go get** as needed.
video: 46 auto conversion

The module mirror & privacy

- A module mirror is a special kind of module proxy that caches metadata and source code in its own storage system, allowing the mirror to continue to serve source code that is no longer available from the original locations. This can speed up downloads and protect you from disappearing dependencies.
- The Go team maintains a module mirror, served at proxy.golang.org, which the go command will use by default for module users as of Go 1.13. If you are running an earlier version of the go command, then you can use this service by setting GOPROXY=https://proxy.golang.org in your local environment.
- GOPRIVATE

source & video: https://www.youtube.com/watch?v=KqTySYYhPUE
video: 47 module mirror

Modules and git commit versioning

A “git tag” allows you to tag a particular commit with some name. With golang modules, you tag should follow semantic versioning. Here are the commands to use:

- create a tag
    - git tag <tag name>
- show a list of all the tags
    - git tag
- push a tag
    - git push --tags
    - git push origin <tag name>

video: 48 git versioning

Releasing major versions in go modules

video 49 major versions

Git forks and module path replacement

You can alias one package name with another package name
video 50 module path replacement

Review commands

- **go mod init <add your name space>**
  - creates a new module, initializing the go.mod file that describes it.
- **go build, go test**
  - and other package-building commands add new dependencies to go.mod as needed.
- **cat go.mod**
  - shows the contents of go.mod file
- **go list -m all**
  - prints the current module's dependencies.
    - direct
    - indirect
- **go list -m -versions <package name>**
  - lists all of the versions of a package
- **go get <package name>**
  - changes the required version of a dependency (or adds a new dependency).
  - example: **go get rsc.io/sampler@v1.3.1**
  - defaults to **@latest**
- **go mod tidy**
  - removes unused dependencies.
  - adds dependencies for other platforms
- **go doc <package name>**
  - shows us the documentation of a package.
  - example: go doc fmt
- tags
  - create a tag
    - **git tag <tag name>**
  - show a list of all the tags
    - **git tag**
  - push a tag
    - **git push --tags**
    - **git push origin <tag name>**

video: 51 review commands

Hands-on exercises - ninja level 2

Hands-on exercise #1

- create a go module
- import rsc.io/quote

    - use Glass() from rsc.io/quote
- run your program
- look at go.mod
    - What version of rsc.io/quote are you using?
- Do you have any indirect dependencies?
    - **go list -m all**
- look at go.sum

video: 52 Hands On 1

Hands-on exercise #2

Working with the same code that you built in hands-on exercise #1:

- go get v3.1.0 of rsc.io/quote
- Does your code still run?
- What is the folder structure of v3.1.0 on github?
- run these commands
    - **go list -m all**
    - **go mod tidy**
    - **go list -m all**
- https://github.com/golang/go/wiki/Modules#releasing-modules-v2-or-higher

video: 53 Hands On 2

Hands-on exercise #3

- from v3.1.0 of rsc.io/quote
    - use the function Glass()
- from v2.0.1 of rsc.io/quote
    - use the function Opt()

video 54 Hands On 3

Hands-on exercise #4

- repo #1
    - create a github repo called **quotes**
    - clone your repo
    - add go module
    - add a function **Favs() []string**
    - add a git tag version of v0.1.0
    - push your code
- repo #2
    - create a new project with a go module
        - this can be on your computer and does not need to be on github
    - from your github repo #1 "**quotes**"
        - use the function **Favs() []string**

    - run your code
    - look at
        - go.mod
        - go.sum

video 55 Hands On 4

Summary

video 56 Summary

Git head

Introduction

video: 57 intro

Git syncing code

Using “git checkout” to check files out from HEAD on my local computer / my local clone
video: 58 Git Sync

Understanding git head

also covered:

- .gitignore
- git fetch

video: 59 Understanding Git Head

Using git head

also covered:

- .gitignore
- git fetch

video: 60 Using Git Head

Crawl #1

Approaching crawling

Here are some things to think about when crawling:

- First thing to think when wanting to crawl a website: Is there an API from the website you want to crawl that provides the data you want?
- Second thing to think about - How do you get the data from the web page sent to you?
    - robots.txt
        - good to look at
        - tells you what you can do at the site
- introducing colly
    - http://go-colly.org/

video: 61 Approaching Crawling

How to approach a package

- look at the examples
- look at the documentation

video: 62 Approaching package

Question and answer

- css class names?
- wow! colly is free!?!
- did this get all of the tweets?
- anything else?

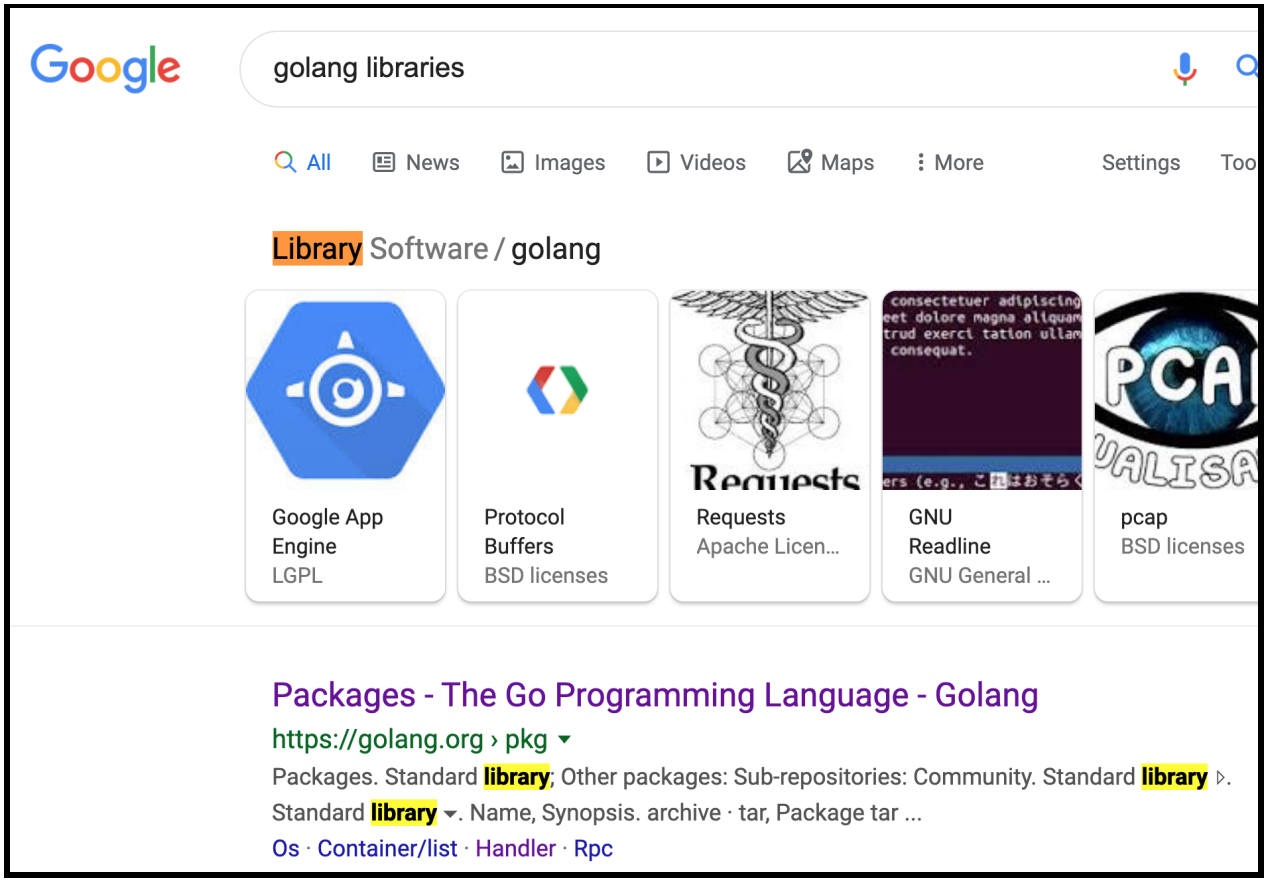
video: 63 Q&A

Reflections

- two versions of twitter
    - logged in
    - incognito
- Daniel's solution - overview
    - found a good library
    - looked at the examples
        - how much of your code was just copy-pasta of the example?
- terminology disambiguation revisited
    - **module**

        - *"In software, a **module** is a part of a program. Programs are composed of one or more independently developed **modules** that are not combined until the program is linked. A single **module** can contain one or several routines. (2) In hardware, a **module** is a self-contained component."*
            - https://www.webopedia.com/TERM/M/module.html
        - ***IN GO***
            - ***A "module" or "go module" is***
                - ***namespacing for your workspace.***
                - ***dependency management***
                    - ***direct dependencies***
                    - ***indirect dependencies***
                    - ***versioning***
            - ***can have 1+ packages***
            - ***usually 1-to-1 with a repo***
    - **package**
        - *"A **package** is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of **packages** as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another."*
            - https://docs.oracle.com/javase/tutorial/java/concepts/package.html
        - ***IN GO***
            - ***higher level grouping of code***
            - ***scope in go:***
                - ***universe***
                - ***package***
                - ***file***
                - ***code block***
                    - ***function***
                    - ***curlies***
                    - ***Go is lexically scoped using blocks: The scope of a predeclared identifier is the universe block. The scope of an identifier denoting a constant, type, variable, or function (but not method) declared at top level (outside any function) is the package block. The scope of the package name of an imported package is the file block of the file containing the import declaration. The scope of an identifier denoting a method receiver, function parameter, or result variable is the function body. The scope of a constant or variable identifier declared inside a function begins at the end of the ConstSpec or VarSpec (ShortVarDecl for short variable declarations) and ends at the end of the innermost containing block. The scope of a type identifier declared inside a function begins at the identifier in the TypeSpec and ends at the end of the innermost containing block.***
                        - ***source: golang spec***
    - library

- *"In computer science, a **library** is a collection of non-volatile resources used by computer programs, often for software development. These may include configuration data, documentation, help data, message templates, pre-written code and subroutines, classes, values or type specifications."*
  - https://en.wikipedia.org/wiki/Library_(computing)
- ***IN GO***
  - ***standard library***
    - ***which is filled with packages***
  - ***code you can use from others***



video: 64 reflections

Crawl #2

Crawling infinite scroll

- in dev tools
  - network tab
  - filter to: XHR

        - scroll to get infinite scroll
        - see the requests
    - click on one of the requests
        - HEADERS
        - exam the URL
            - query parameters separated by “&”
        - PARAMS
        - RESPONSE
        - looks promising
            - min_position
            - max_position
            - has_more_items
            - items_html
                - looks like it has the html of the tweets

video: 65 crawl infinite

robots.txt revisited

Google for a “robots.txt validator” or “robots.txt checker” - here’s the one we used:

- https://technicalseo.com/tools/robots-txt/

video: 66 robot txt revisited

Making a conversation request

Looking at solution #2 and the **makeConversationRequest** function:

- this function makes a single request
    - http.Get()

video: 67 Convo Request

Making all conversation requests

Looking at the **getConversation** function:

- this calls **makeConversationRequest** multiple times until all of the requests have been retrieved

video: 68 make all convo request

Parsing html

Looking at the **parseHTML** function

video: 69 parse html

Exploring parsing html

Todd's talk-through on the **parseHTML** function:

- a reflection on
    - how long did it take you to write this?
    - if you feel overwhelmed as a newbie
        - how long did it take you to get to this level

video: 70 explore parsing html

Exploring main and running

video: 71 explore main and execute

Reflections

- infinite for loops
    - have a bad reputation, but good in practice!
- modularizing
    - how did you think about breaking it all up?

video: 72 reflections

Analysis I - word count

in this analysis we use

- creating a struct to hold
    - word
    - count
- creating a map to hold
    - key: word
    - value: count
- using the map to count the words
- ranging over the map
    - putting the word & count into a struct

video: 73 word count

Analysis II - sorting

in this analysis we use

- sort.Slice
    - takes in two parameters
        - slice

        - less func(i, j int) bool)

and the results!

video: 74 sorting

Hands-on exercises - ninja level 3

Hands-on exercise #1

For this hands-on exercise:

- determine whether or not this url can be crawled:
- https://www.reddit.com/r/funny/

video: 75 hands on 1

Hands-on exercise #2

For this hands-on exercise:

- use gocolly
- crawl https://www.reddit.com/r/golang/
    - hint - take a look at: https://old.reddit.com/r/golang/

video: 76 hands on 2

Hands-on exercise #3

For this hands-on exercise:

- split all of the text crawled in the previous version into words
- count the words
- sort the words by count, largest to smallest
- display the words in sorted order

video: 77 hands on 3

gRPC - getting started

Introduction

video: 78 grpc intro

installation - windows

- https://grpc.io
    - https://grpc.io/docs/quickstart/go/
- install PROTOC program

    - the protoc compiler is used to generate gRPC service code
- take the ***executable*** from PROTOC, store it some location, have that location on your PATH environment variable
    - echo $PATH
    - mine for example:
        - mv protoc /usr/local/bin
- import http://github.com/golang/protobuf/protoc-gen-go
    - using go modules
        - create a file like this in your project

```
// +build tools
package main

import _ "http://github.com/golang/protobuf/protoc-gen-go"
```

- **go mod tidy**
    - confirm this "http://github.com/golang/protobuf/protoc-gen-go" is in go.mod
- set "**GOBIN**" environment variable to something
- **go install http://github.com/golang/protobuf/protoc-gen-go**

video: 79 installation - windows

Installation - Mac

video: 80 installation - mac

gRPC - in action

Defining the service

video: 81 define service

DSL vs IDL, and versions

Todd's commentary:

- what is DSL
    - DSL vs IDL
- version of proto to use
    - https://grpc.io/docs/guides/

video: 82 DSL vs IDL

Compiling the service

- what is DSL

    - DSL vs IDL
- version of proto to use
    - https://grpc.io/docs/guides/

video: 83 compiling service

Protocol buffer, folders, & packages

- “protocol buffer” I think “json”
- package name = folder name

video: 84 proto buff, folder, packages

Setting up a grpc server

- commit
    - pg.go file
    - echo.proto

video: 85 grpc server setup

Understanding the code

- “empty struct takes no memory”
- line 27
    - initializing
- !=

video: 86 understanding the code

Setting up a grpc client

video: 87 grpc client setup

Discussing the service

- what are we doing
- response vs Response

video: 88 discussing the service

gRPC - chat server

Introduction

- nice article

video: 89 intro

First steps

- start with the **chat.proto** file
- generate your code and take a look at it
    - **chat.pb.go**

video: 90 first steps

Creating a client

video: 91 creating a client

Foundations of a server

video: 92 server foundation

Handling multiple connections

video: 93 handle muti connections

Running the grpc chat server

video: 94 running chat server

Tracing the path of a message

video: 95 trace message path

Farewell

Congratulations!

Great job completing the course!
Video file: 96 congratulations

Bonus lecture