RX-M Cloud Native Consulting

Go

Overview

Go (often referred to as golang) is a free and open source programming language created at Google in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson. It is a compiled, statically typed language in the tradition of Algol and C, with garbage collection, limited structural typing, memory safety features, and CSP-style concurrent programming features (CSP = communicating sequential processes).

In this lab we will install Go and create our first Go program.

0. Preparation

The lab exercises for this course are designed for completion on a base Ubuntu 64 bit 16.04 system. The system should have 2+ CPUs, 2+ GBs of RAM and 20+ GB of disk. Students who have access to such a system (e.g. a typical cloud instance) can perform the lab exercises on that system.

Alternately students can use the RX-M preconfigured lab virtual machine.

https://github.com/RX-M/classfiles/blob/master/lab-setup.md

1. Install the latest version of Go

Many Go developers choose to install Go from source. You can also install Go using your system's package manager. In this course we are not going to use the package manager because these versions tend to be out of date, and may not use the official builds. We will take a third path and use the official binary installation of the latest stable build.

Download the prebuilt package (tools, standard library, documentation).

```
user@ubuntu:~$ curl -sLO https://storage.googleapis.com/golang/go1.8.3.linux-amd64.tar.gz
user@ubuntu:~$
```

Decompress the package.

```
user@ubuntu:~$ tar zxf go1.8.3.linux-amd64.tar.gz
user@ubuntu:~$
```

Place the tooling in the standard location (`/usr/local/go`). We can change this location using GOROOT, but at this point it is not recommended.

```
user@ubuntu:~$ sudo mv ~/go/ /usr/local/
user@ubuntu:~$
```

Add the binaries to our PATH. Due to potentially many ways to use the VM (GUI, SSH, etc.) we need to update two files, `.bash_profile` and `.bashrc` .

For non-interactive logins (like SSH).

```
user@ubuntu:~$ echo "export PATH=/usr/local/go/bin:$PATH" >> .bashrc
user@ubuntu:~$
```

To capture interactive logins (like the VM GUI).

```
user@ubuntu:~$ echo "[[ -r ~/.bashrc ]] && . ~/.bashrc" >> ~/.bash_profile
user@ubuntu:~$
```

Update our current shell.

```
user@ubuntu:~$ source ~/.bash_profile
user@ubuntu:~$
```

Confirm the PATH modification is working (also try opening a new terminal).

```
user@ubuntu:~$ which go
/usr/local/go/bin/go
user@ubuntu:~$
```

Finally, confirm Go is available and working.

```
user@ubuntu:~$ go version
go version go1.8.3 linux/amd64
user@ubuntu:~$
```

You can learn more about the process and customizing operations here https://golang.org/dl/ and https://golang.org/doc/install#install.

2. Hello World

With Go installed we're ready to try the simplest program we can write that we can prove ran, hello world. This will allow us to verify that all of our tools work and it will give us a basic skeleton to build on.

First create a working directory for your lab-overview code.

```
user@ubuntu:~$ mkdir -p ~user/go/src/lab-overview/

user@ubuntu:~$ cd ~user/go/src/lab-overview/
user@ubuntu:~/go/src/lab-overview$
```

Now enter the code for your hello world Go app:

```
user@ubuntu:~/go/src/lab-overview$ vim hello.go
user@ubuntu:~/go/src/lab-overview$ cat hello.go
```

```
package main

import "fmt"

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

```
user@ubuntu:~/go/src/lab-overview$
```

Go code is organized into packages, which are similar to libraries or modules in other languages. A package consists of one or more `.go` source files in a single directory that define what the package does. Packages are Go's way of organizing and reusing code. Each source file begins with a `package` declaration. Our hello world program declares the "main" package.

Other packages can be accessed by first referencing them with the `import` statement. In the `hello.go` code we reference the `fmt` package. Package fmt implements formatted I/O with functions analogous to C's printf and scanf. The format 'verbs' are derived from C's but are simpler. Here we use the Println function to display text to the console. Println inserts blanks between operands and appends a newline.

Functions are the building blocks of a Go program. They have inputs, outputs, and a series of steps called statements which are executed in order. All functions start with the keyword func followed by the name of the function (main in this case), a list of zero or more “parameters” surrounded by parentheses, an optional return type, and a “body” which is surrounded by curly braces. Our main function has no parameters, doesn't return anything and has only one statement in its body. The function name "main" is special because it is the function that gets invoked when you execute the program.

Go is a compiled language so we need to use the Go toolchain to convert our source into an executable program. All of the build tools used with Go are accessed through a single command called `go`. The `go` command has a number of subcommands. The `run` subcommand compiles the source code, links it with any needed libraries, and then runs the resulting executable file.

Try it:

```
user@ubuntu:~/go/src/lab-overview$ ls -l
total 4
-rw-rw-r-- 1 user user 72 May 21 16:30 hello.go
```

```
user@ubuntu:~/go/src/lab-overview$ go run hello.go
hello world
user@ubuntu:~/go/src/lab-overview$
```

```
user@ubuntu:~/go/src/lab-overview$ ls -l
total 4
-rw-rw-r-- 1 user user 72 May 21 16:30 hello.go
user@ubuntu:~/go/src/lab-overview$
```

As you can see the `go run` subcommand leaves no trace of the compilation on disk.

The `go build` subcommand can be used to produce an executable file.

Try it:

```
user@ubuntu:~/go/src/lab-overview$ go build hello.go
user@ubuntu:~/go/src/lab-overview$
```

```
user@ubuntu:~/go/src/lab-overview$ ls -l
total 1520
-rwxrwxr-x 1 user user 1551621 May 21 16:33 hello
-rw-rw-r-- 1 user user      72 May 21 16:30 hello.go
user@ubuntu:~/go/src/lab-overview$
```

```
user@ubuntu:~/go/src/lab-overview$ ./hello
hello world
user@ubuntu:~/go/src/lab-overview$
```

3. Building a "hello world" web service

Go is a modern language and tries to make it easy to do the kinds of things programmers generally need to do. For example imagine you need to build an HTTP based web service (you probably don't need to imagine).

Here's all it takes in Go, enter the following program:

```
user@ubuntu:~/go/src/lab-overview$ vim ws.go
user@ubuntu:~/go/src/lab-overview$ cat ws.go
package main

import "net/http"

func main() {
        http.Handle("/", http.HandlerFunc(hi))
        http.ListenAndServe("localhost:9999", nil)
}

func hi(resp http.ResponseWriter, req *http.Request) {
        resp.Write([]byte("Hello World!\n"))
}
user@ubuntu:~/go/src/lab-overview$
```

This program uses the net/http package, defines a handler for requests and maps it to an IRI, then starts a listener to accept connections and process requests. The handler receives a ResponseWriter and a Request. These are pretty common idioms in web programming frameworks, however we're not using a framework! These features are built into Go, many Go programmers avoid web frameworks, preferring the flexibility and absence of lock-in that programming in Go directly provides.

We use the Write() method of the ResponseWriter object to return our "Hello World" message. The Write method wants an array of bytes rather than a string, which we handily convert on the fly with elegant and safe go cast syntax.

Our listener is waiting for connections on the localhost interface (127.0.0.1) and on port 9999. The listener also requires a second parameter, representing the "global handler". This parameter is usually set to nil, which means, use the DefaultServeMux global handler, which directs requests as specified by individual calls to Handle(). In our case, we called Handle() to tell the ServeMux to send requests for the root ("/") IRI to hi(). You can create custom global handlers if you would like to implement some special routing logic. You can learn more here https://golang.org/pkg/net/http/#ServeMux.

In Go, nil is the zero value for pointers, interfaces, maps, slices, channels, and function types; representing an uninitialized value. nil is a proper value in itself. An object in Go is nil if and only if it's value is nil, which it can only be if it's one of the aforementioned types. Another interesting aspect of Go is that error is an interface type, so nil is a valid value for the error interface, a nil error represents no error.

Now run your simple service:

```
user@ubuntu:~/go/src/lab-overview$ go run ws.go
```

Perfect, the server is waiting (silently) for requests. Now that the service is running, open a second terminal and `curl` the localhost interface on port 9999:

```
user@ubuntu:~$ curl -v localhost:9999
* Rebuilt URL to: localhost:9999/
*   Trying ::1...
* connect to ::1 port 9999 failed: Connection refused
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 9999 (#0)
> GET / HTTP/1.1
> Host: localhost:9999
> User-Agent: curl/7.47.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Sat, 24 Jun 2017 18:36:23 GMT
< Content-Length: 13
< Content-Type: text/plain; charset=utf-8
<
Hello World!
* Connection #0 to host localhost left intact
user@ubuntu:~$
```

A web service in 9 lines of code, not bad. Now consider how many lines of Python, Java, C#, C or C++ you would have needed to do the same thing (without a framework!).

You can press ^C to terminate your server.

```
user@ubuntu:~/go/src/lab-overview$ go run ws.go
^Csignal: interrupt
user@ubuntu:~/go/src/lab-overview$
```

Peruse the Go `net/http` package reference if you have time here https://golang.org/pkg/net/http/

4. Simple operators, expressions and constants

Create the follow Go program, `op.go`

```
user@ubuntu:~/go/src/lab-overview$ vi op.go
user@ubuntu:~/go/src/lab-overview$ cat op.go
```

```
package main

import "fmt"
import "math"

func main() {
        fmt.Println("go" + "lang")
        fmt.Println("1+1 =", 1+1)
        fmt.Println("7/3 =", 7/3)
        fmt.Println("7%3 =", 7%3)
        fmt.Println("7.0/3.0 =", 7.0/3.0)
        fmt.Println(true && false)
        fmt.Println(true || false)
        fmt.Println(!true)
        fmt.Println("--------------")
        const s string = "constant"
        fmt.Println(s)
        const n = 500000000
        const d = 3e20 / n
        fmt.Println(3e20)
        fmt.Println(d)
        fmt.Println(int64(d))
        fmt.Println(math.Sin(n))
}
```

```
user@ubuntu:~/go/src/lab-overview$
```

Before running the program jot down the output you expect.

Now run the program and compare the actual results to your expectations.

5. Challenge

Create a Go program that uses the `time` pacakge to print a current timestamp.

Congratulations you have completed the lab, welcome to Go!!