Go

The Go standard library

In this lab, we will look at several stand library packages. The primary goal is to see what is available with the standard library, and second is to review there characteristics.

Preparation

```
user@ubuntu:~$ mkdir -p $(go env GOPATH)/src/lab-std-lib/{cmd,mylib}

user@ubuntu:~$ cd $(go env GOPATH)/src/lab-std-lib

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

1. Sample of standard library

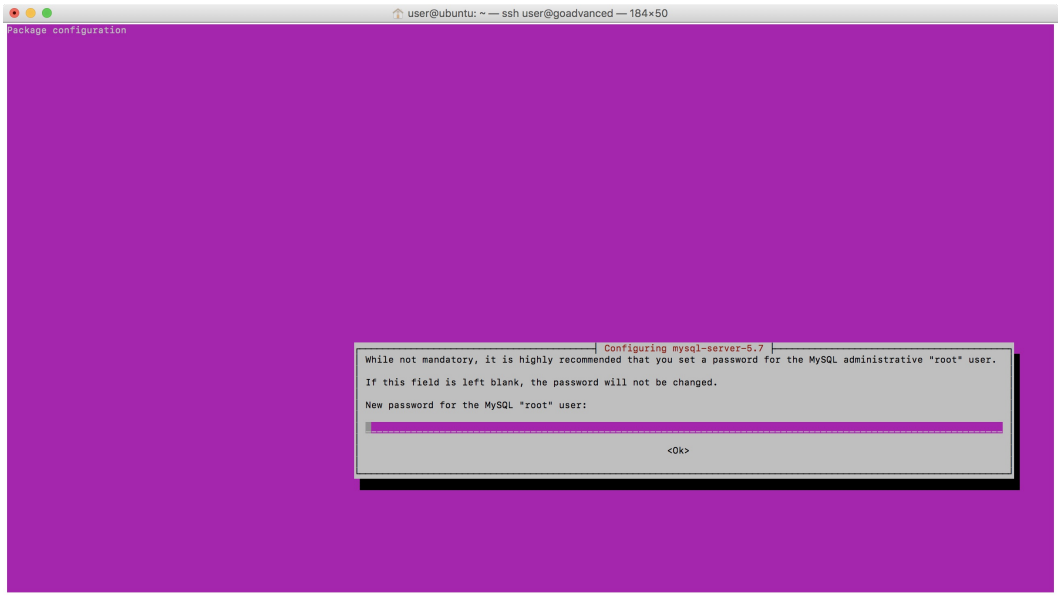
database

In this example, we will look into accessing a MySQL server from a Go program.

Go provides a package defining the SQL and related driver interface. We will need a driver that implements this to access the database.

First, we install MySQL.

```
user@ubuntu:~/go/src/lab-std-lib$ sudo apt-get install mysql-server
...
```



``` ... user@ubuntu:~/go/src/lab-std-lib$ ```
Confirm the MySQL service is up and running.

```
user@ubuntu:~/go/src/lab-std-lib$ systemctl status mysql.service
● mysql.service - MySQL Community Server
   Loaded: loaded (/lib/systemd/system/mysql.service; enabled; vendor preset: enabled)
   Active: active (running) since Wed 2017-05-17 21:10:08 PDT; 3min 41s ago
 Main PID: 58666 (mysqld)
    Tasks: 28
   Memory: 148.3M
```
```

```
       CPU: 2.373s
    CGroup: /system.slice/mysql.service
            └─58666 /usr/sbin/mysqld

May 17 21:10:05 ubuntu systemd[1]: Starting MySQL Community Server...
May 17 21:10:08 ubuntu systemd[1]: Started MySQL Community Server.
user@ubuntu:~$
```

Now run a basic command to confirm connectivity.

```
user@ubuntu:~/go/src/lab-std-lib$ mysqladmin -p -u root version
Enter password:
mysqladmin  Ver 8.42 Distrib 5.7.18, for Linux on x86_64
Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Server version          5.7.18-0ubuntu0.16.04.1
Protocol version        10
Connection              Localhost via UNIX socket
UNIX socket             /var/run/mysqld/mysqld.sock
Uptime:                 4 min 17 sec

Threads: 1  Questions: 3  Slow queries: 0  Opens: 107  Flush tables: 1  Open tables: 26  Queries per second avg: 0.011
user@ubuntu:~$
```

With the server running, next we need to install a database driver that implements the Go `database` driver interface. Using `go get`, we will use the `go-mysql-driver`.

```
user@ubuntu:~/go/src/lab-std-lib$ go get github.com/go-sql-driver/mysql

user@ubuntu:~/go/src/lab-std-lib$ tree $(go env GOPATH)/src/github.com/go-sql-driver/mysql/
/home/user/go/src/github.com/go-sql-driver/mysql/
├── appengine.go
├── AUTHORS
├── benchmark_test.go
├── buffer.go
├── CHANGELOG.md
├── collations.go
├── connection.go
├── connection_test.go
├── const.go
├── CONTRIBUTING.md
├── driver.go
├── driver_go18_test.go
├── driver_test.go
├── dsn.go
├── dsn_test.go
├── errors.go
├── errors_test.go
├── infile.go
├── LICENSE
├── packets.go
├── packets_test.go
├── README.md
├── result.go
├── rows.go
├── statement.go
├── transaction.go
├── utils.go
└── utils_test.go

0 directories, 28 files
user@ubuntu:~$
```

Before we review the implementation, lets try using it.

```
user@ubuntu:~/go/src/lab-std-lib$ vi cmd/main.go
user@ubuntu:~/go/src/lab-std-lib$ cat cmd/main.go
package main

import (
  "database/sql"
  _ "github.com/go-sql-driver/mysql"
  "fmt"
```

```
)

func main() {

  db, err := sql.Open("mysql", "root:root@/mysql")

  if err != nil {
    fmt.Printf("Failed to connect", err)
  }

  err = db.Ping()
  if err != nil {
    panic(err.Error()) // proper error handling instead of panic in your app
  }
}
user@ubuntu:~/go/src/lab-std-lib$
```

If we run the aforementioned code, there should be no error. The `Ping` function, will only error if we can't connect to the database.

```
user@ubuntu:~/go/src/lab-std-lib$ go run cmd/main.go
user@ubuntu:~/go/src/lab-std-lib$
```

In another terminal try turning off the MySQL database and rerun the program, it should fail.

```
user@ubuntu:~$ sudo systemctl stop mysql
[sudo] password for user:
user@ubuntu:~$
```

```
user@ubuntu:~/go/src/lab-std-lib$ go run cmd/main.go
panic: dial tcp 127.0.0.1:3306: getsockopt: connection refused

goroutine 1 [running]:
main.main()
        /home/user/go/src/lab-std-lib/cmd/main.go:19 +0x136
exit status 2
user@ubuntu:~/go/src/lab-std-lib$
```

Review the following functions from the Go `database` package.

- https://golang.org/pkg/database/sql/#Open
- https://golang.org/pkg/database/sql/#DB.Close
- https://golang.org/pkg/database/sql/#DB.Ping
- https://golang.org/pkg/database/sql/#DB.Query

Review the following type from the Go `database` package.

- https://golang.org/pkg/database/sql/#Rows

Now that we have a general idea of the package purpose, lets review the Go package database.

```
user@ubuntu:~$ cd ~/go1.8.1/src/database/sql/
user@ubuntu:~/go1.8.1/src/database/sql$
```

In `sql.go`, we can find the implementations, for example here is the code to `Open`

```
user@ubuntu:~/go1.8.1/src/database/sql$ grep -A 16 "func Open" sql.go
func Open(driverName, dataSourceName string) (*DB, error) {
        driversMu.RLock()
        driveri, ok := drivers[driverName]
        driversMu.RUnlock()
        if !ok {
                return nil, fmt.Errorf("sql: unknown driver %q (forgotten import?)", driverName)
        }
        db := &DB{
                driver:       driveri,
                dsn:          dataSourceName,
                openerCh:     make(chan struct{}, connectionRequestQueueSize),
                lastPut:      make(map[*driverConn]string),
                connRequests: make(map[uint64]chan connRequest),
        }
        go db.connectionOpener()
        return db, nil
}
```

```
user@ubuntu:~/go1.8.1/src/database/sql$
```

- In `sql.go`, review `func Open(driverName, dataSourceName string) (*DB, error)`
- In `sql.go`, review `func (db *DB) Ping() error`
- In `sql.go`, review `func (db *DB) Close() error`
- In `sql.go`, review `func (db *DB) Query(query string, args ...interface{}) (*Rows, error)`
- In `sql.go`, review `type Rows struct`

Now that we have a basic idea how the database package looks, lets see how the particular driver (`go-sql-driver`) works.

How does a database vendor supply a driver via the Go `database` package? Remembering the point of the package, is to normalize the interface to the database but allowing the user to change the implementation. In the case of the database package, a driver needs to register itself in order to be used.

- https://golang.org/pkg/database/sql/#Register

```
user@ubuntu:~/go/src/lab-std-lib$ grep -nr "\.Register(" $(go env GOPATH)/src/github.com/go-sql-driver/mysql/
/home/user/go/src/github.com/go-sql-driver/mysql/driver.go:182: sql.Register("mysql", &MySQLDriver{})
user@ubuntu:~/go/src/lab-std-lib$
```

We see a call to `Register` in the the `driver.go`, lets take a look.

```
user@ubuntu:~/go/src/lab-std-lib$ grep -C 1 -nr "\.Register(" /home/user/go/src/github.com/go-sql-driver/mysql/driver.go
181-func init() {
182:    sql.Register("mysql", &MySQLDriver{})
183-}
user@ubuntu:~/go/src/lab-std-lib$
```

Remembering that `init()` has a special meaning, it is called when a package is loaded. If you remember when we referenced our the driver from our code, it must call `init()`; specifically here:

```
user@ubuntu:~/go/src/lab-std-lib$ grep -C 2 _ cmd/main.go
import (
  "database/sql"
  _ "github.com/go-sql-driver/mysql"
  "fmt"
)
user@ubuntu:~/go/src/lab-std-lib$
```

- Review https://golang.org/doc/effective_go.html#blank_import

We did not discuss database design here, the goal is to get an understanding on how the `database` package and related implementation relate.

- Review an alternative driver, examples https://github.com/golang/go/wiki/SQLDrivers

net

We next use the `net` package, which provides us low-level network primitives.

In our first example, we will create a client and server that perform an echo service.

```
user@ubuntu:~/go/src/lab-std-lib$ mkdir {client,server}
```

First the client.

```
user@ubuntu:~/go/src/lab-std-lib$ vi client/main.go
user@ubuntu:~/go/src/lab-std-lib$ cat client/main.go
package main

import (
  "net"
  "fmt"
)

func main() {
  conn, _ := net.Dial("tcp", "localhost:8080")

  d := []byte("Here is a string....")

  conn.Write(d)

  conn.Read(d)
```

```
  fmt.Println(string(d[:]))

  conn.Close()
}
user@ubuntu:~/go/src/lab-std-lib$
```

Now the server.

```
user@ubuntu:~/go/src/lab-std-lib$ vi server/main.go
user@ubuntu:~/go/src/lab-std-lib$ cat server/main.go
package main

import (
  "net"
)

func main() {
  ln, _ := net.Listen("tcp", "localhost:8080")

  conn, _ := ln.Accept()

  tmp := make([]byte, 256)

  conn.Read(tmp)

  conn.Write(tmp)

  conn.Close()
}
user@ubuntu:~/go/src/lab-std-lib$
```

To run the program, we will compile the server and client, running each in a separate terminal.

Normally we would use `go install` to compile and install the commands. This requires us to set GOBIN, here is an example:

```
  user@ubuntu:~/go/src/lab-std-lib$ export GOBIN=$(go env GOPATH)/bin/
  user@ubuntu:~/go/src/lab-std-lib$
```

Instead, lets use `go build` and place the executable ourself.

```
  user@ubuntu:~/go/src/lab-std-lib$ go build -o /home/user/go/bin/client client/main.go
  user@ubuntu:~/go/src/lab-std-lib$ go build -o /home/user/go/bin/server server/main.go
  user@ubuntu:~/go/src/lab-std-lib$
```

Now in two different terminals we can see run the programs.

step	terminal1	terminal2
1	user@ubuntu:~/go/src/lab-std-lib$ server	
2		user@ubuntu:~$ client
3		(sends data)
4	(receives->echos)	
5	exits	displays
6		exits

```
  user@ubuntu:~/go/src/lab-std-lib$ server
```

```
  user@ubuntu:~$ client
  Here is a string....
  user@ubuntu:~$
```

- Do connections block?

- Does the server handle two or more connections?
- How can we make it handle multiple connections?

net/http

Go provides HTTP related functionality for client and server via `net/http` package.

```
user@ubuntu:~/go/src/lab-std-lib$ mkdir -p http/{client,server}
user@ubuntu:~/go/src/lab-std-lib$
```

Basic client usage example:

```
user@ubuntu:~/go/src/lab-std-lib$ vi http/client/main.go
user@ubuntu:~/go/src/lab-std-lib$ cat http/server/main.go
package main

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

func main() {
  resp, _ := http.Get("http://example.com/")

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

  fmt.Printf(string(body[:]))

  for k,v := range resp.Header {
    fmt.Printf(k,":",v)
  }
}
user@ubuntu:~/go/src/lab-std-lib$
```

Run the client.

```
user@ubuntu:~/go/src/lab-std-lib$ go run http/client/main.go | tail
<body>
<div>
    <h1>Example Domain</h1>
    <p>This domain is established to be used for illustrative examples in documents. You may use this
    domain in examples without prior coordination or asking for permission.</p>
    <p><a href="http://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
Content-Type%!(EXTRA string=:, []string=[text/html])Date%!(EXTRA string=:, []string=[Thu, 18 May 2017 07:39:43 GMT])Expires%!(EXTRA
string=:, []string=[Thu, 25 May 2017 07:39:43 GMT])Last-Modified%!(EXTRA string=:, []string=[Fri, 09 Aug 2013 23:54:35 GMT])Server%!
(EXTRA string=:, []string=[ECS (rhv/81A7)])Vary%!(EXTRA string=:, []string=[Accept-Encoding])Accept-Ranges%!(EXTRA string=:, []string=
[bytes])Cache-Control%!(EXTRA string=:, []string=[max-age=604800])Etag%!(EXTRA string=:, []string=["359670651"])X-Cache%!(EXTRA
string=:, []string=[HIT])user@ubuntu:~/go/src/lab-std-lib$
```

- Modify the client to request headers only.

Basic server usage example:

```
user@ubuntu:~/go/src/lab-std-lib$ vi http/server/main.go
user@ubuntu:~/go/src/lab-std-lib$ cat http/server/main.go
package main

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

func fooHandler() http.Handler {
  fn := func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Foo"))
  }
  return http.HandlerFunc(fn)
}

func main() {
```

```
  http.Handle("/foo", fooHandler())

  http.HandleFunc("/bar", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
  })

  log.Fatal(http.ListenAndServe(":8080", nil))
}
user@ubuntu:~/go/src/lab-std-lib$
```

Run the server in one terminal, and access it from another, hitting the specified paths.

```
user@ubuntu:~/go/src/lab-std-lib$ go run http/server/main.go
^Csignal: interrupt
user@ubuntu:~/go/src/lab-std-lib$
```

```
user@ubuntu:~$ curl localhost:8080/bar
Hello, "/bar"user@ubuntu:~$
```

- Modify the client to access our server.
- How does Handle and HandleFunc differ?
- Review https://golang.org/src/net/http/server.go

While it is nice to use the default HTTP service, we might need to deploy more complex (ports, nested paths) services. We don't go into depth, but two areas to review include:

- ServeMux - https://golang.org/pkg/net/http/#ServeMux
- Handlers - https://golang.org/pkg/net/http/#Handler

For a nice overview of using custom mux and handlers, see http://www.alexedwards.net/blog/a-recap-of-request-handling

encoding/json

Another useful package, `encoding/json` is to encode and decode JSON data.

- Marshall/Unmarshall a Go struct by following this tutorial https://blog.golang.org/json-and-go

2. Complex service

- Create http or net client and server that marshalls and unmarshalls JSON data.

Congratulations, you have successfully completed the lab!