

# Getting GoLangAPI to work with MYSQL

## For Installation of MYSQL on MAC M1

["https://www.youtube.com/watch?v=nj3nBCwZaql"](https://www.youtube.com/watch?v=nj3nBCwZaql)

[https://www.tutorialspoint.com/python3/python\\_database\\_access.htm](https://www.tutorialspoint.com/python3/python_database_access.htm)

Installing mySQL on Mac

<https://www.youtube.com/watch?v=n1zT1OZcgnw>

### Setting your path after installation

<https://www.youtube.com/watch?v=oxToe-4c6OM>

It's different for M1

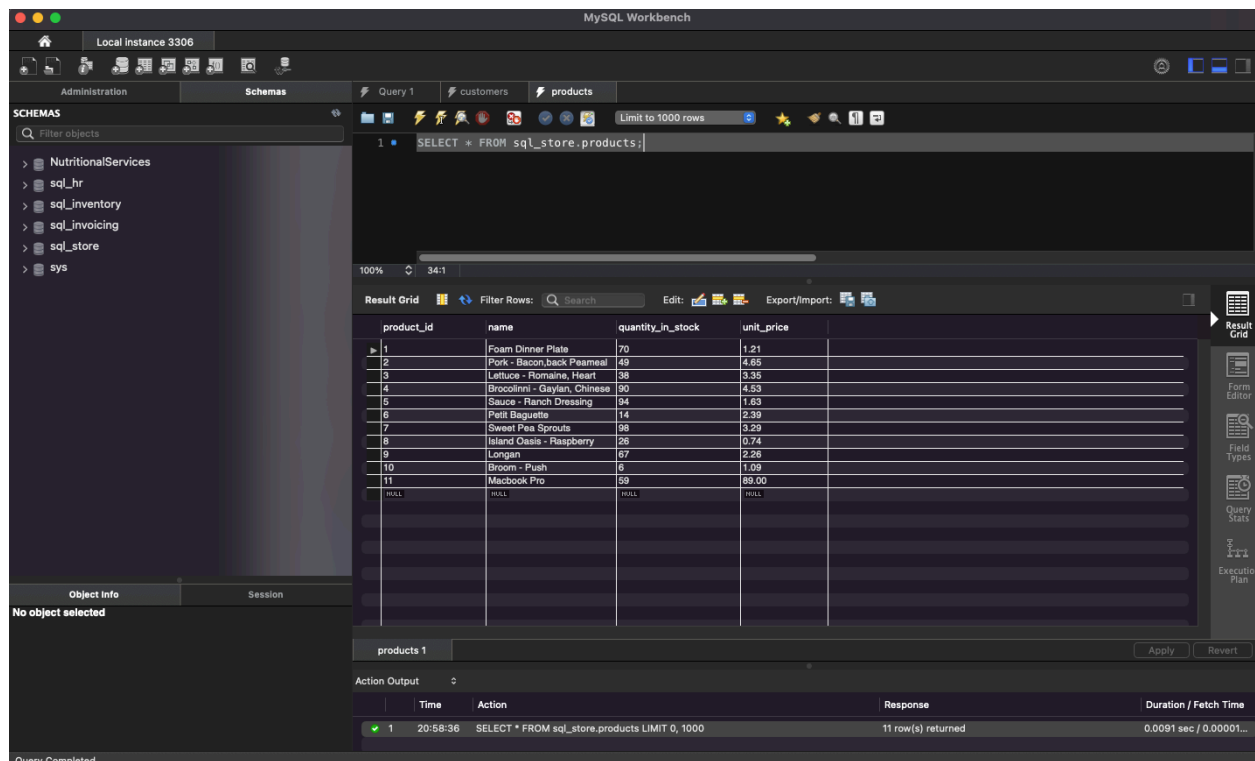
**MySQL course YouTube (Complete with instructions on using SQL Workbench and good sample data included) - Very good "FREE COURSE"**

[https://www.youtube.com/watch?v=7S\\_tz1z\\_5bA](https://www.youtube.com/watch?v=7S_tz1z_5bA)

The SQL Scripts for the test data is attached - See asana notes

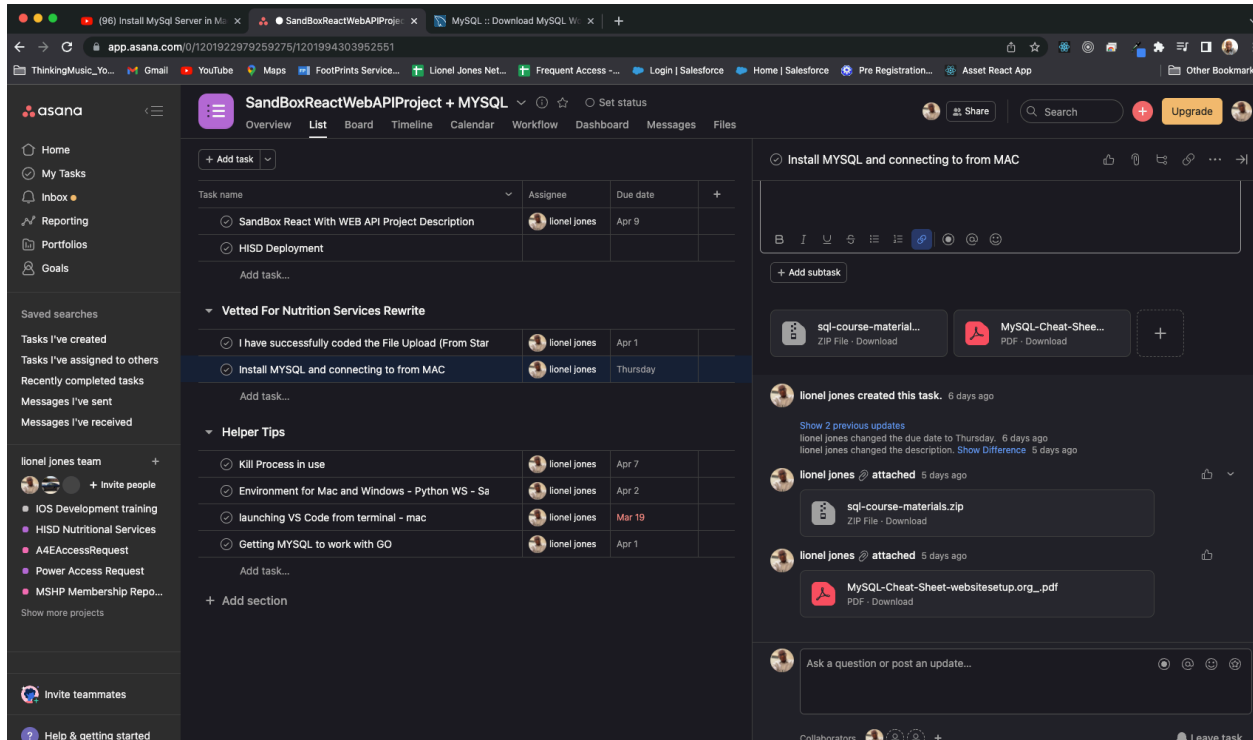
### MYSQL Workbench

<https://dev.mysql.com/downloads/workbench/>



ALSO MYSQL CHEAT SHEET IS ATTACHED

Also in my ASANA Note: - I have the course materials and SQL Scripts to use for the SQL Workbench Training



Good reference(s)

**Creating a RESTful API**

Below is a link to the docs that explains how to write an API with Go

<https://go.dev/doc/tutorial/web-service-gin>

The GitHub Project is located: **(I coded it in the Udemmy Coarse)**

<https://github.com/lionel5116/golangApiExample.git>

**MySQL Tutorial with Go**

<https://tutorialedge.net/golang/golang-mysql-tutorial/>

<https://zetcode.com/golang/mysql/>

**The best Youtube Video**

"[https://www.youtube.com/watch?v=DWNNozbn\\_fuk](https://www.youtube.com/watch?v=DWNNozbn_fuk)"

**GitHub - go-sql-driver/mysql: Go MySQL Driver is a MySQL driver for Go's (golang) database/sql package**

<https://github.com/go-sql-driver/mysql#installation>

## To install the package

\$ go get -u [github.com/go-sql-driver/mysql](https://github.com/go-sql-driver/mysql)

```
$ go get -u github.com/go-sql-driver/mysql
```

Make sure [Git is installed](#) on your machine and in your system's `PATH` .

## Usage

Go MySQL Driver is an implementation of Go's `database/sql/driver` interface. You only need to import the driver and can use the full `database/sql` API then.

Use `mysql` as `driverName` and a valid DSN as `dataSourceName` :

```
import (
    "database/sql"
    "time"

    _ "github.com/go-sql-driver/mysql"
)

// ...

db, err := sql.Open("mysql", "user:password@dbname")
if err != nil {
    panic(err)
}
// See "Important settings" section.
db.SetConnMaxLifetime(time.Minute * 3)
db.SetMaxOpenConns(10)
db.SetMaxIdleConns(10)
```

[Examples are available in our Wiki](#)

```
//you have to name your fields "Uppercase" so that they will be
exportable *
//add the json:"<tags>" in lowercase
type Product struct {
    Product_id    int    `json:"product_id"`
    Name          string `json:"name"`
    Quantity_in_stock int    `json:"quantity_in_stock"`
    Unit_price    float32 `json:"unit_price"`
}
```

One of the key things to note is that when you declare your structure, you have to have your field names in UpperCase in order for the structure to be exportable when you write it out to your response.

In order to append objects to an array:

```
91
92 //the syntax below is how you append objects to a struct
93 func addProduct(_product Product) {
94     var my_product = new(Product)
95     my_product.Product_id = _product.Product_id
96     my_product.Name = _product.Name
97     my_product.Quantity_in_stock = _product.Quantity_in_stock
98     my_product.Unit_price = _product.Unit_price
99     arrProducts = append(arrProducts, *my_product)
100 }
101
```

**The code for calling MYSQL**  
**Your imports for MYSQL**

When you browse:

```
func {
//
dt
if
}
de
//
re
if
}
vz
//
fc
1
}
c.Inc
}

[
{
  "product_id": 1,
  "name": "Foam Dinner Plate",
  "quantity_in_stock": 70,
  "unit_price": 1.21
},
{
  "product_id": 2,
  "name": "Pork - Bacon,back Peameal",
  "quantity_in_stock": 49,
  "unit_price": 4.65
},
{
  "product_id": 3,
  "name": "Lettuce - Romaine, Heart",
  "quantity_in_stock": 38,
  "unit_price": 3.35
},
{
  "product_id": 4,
  "name": "Brocolinni - Gaylan, Chinese",
  "quantity_in_stock": 90,
  "unit_price": 4.53
},
{
  "product_id": 5,
  "name": "Sauce - Ranch Dressing",
  "quantity_in_stock": 94,
  "unit_price": 1.63
},
{
  "product_id": 6,
  "name": "Petit Baguette",
  "quantity_in_stock": 14,
  "unit_price": 2.39
},
{
  "product_id": 7,
  "name": "Sweet Pea Sprouts",
  "quantity_in_stock": 98,
  "unit_price": 3.29
},
{
  "product_id": 8,
  "name": "Island Oasis - Raspberry",
  "quantity_in_stock": 100,
  "unit_price": 2.50
}
]
```

3031/a3..1/3dfc3 main -> main

# Overview of Go

Every Go program is made up of packages

In your main file (i.e **main.go**), you import package main

## Dependencies

When you start a new go project, you want to add a module to it by typing:

**go mod init *myapp***

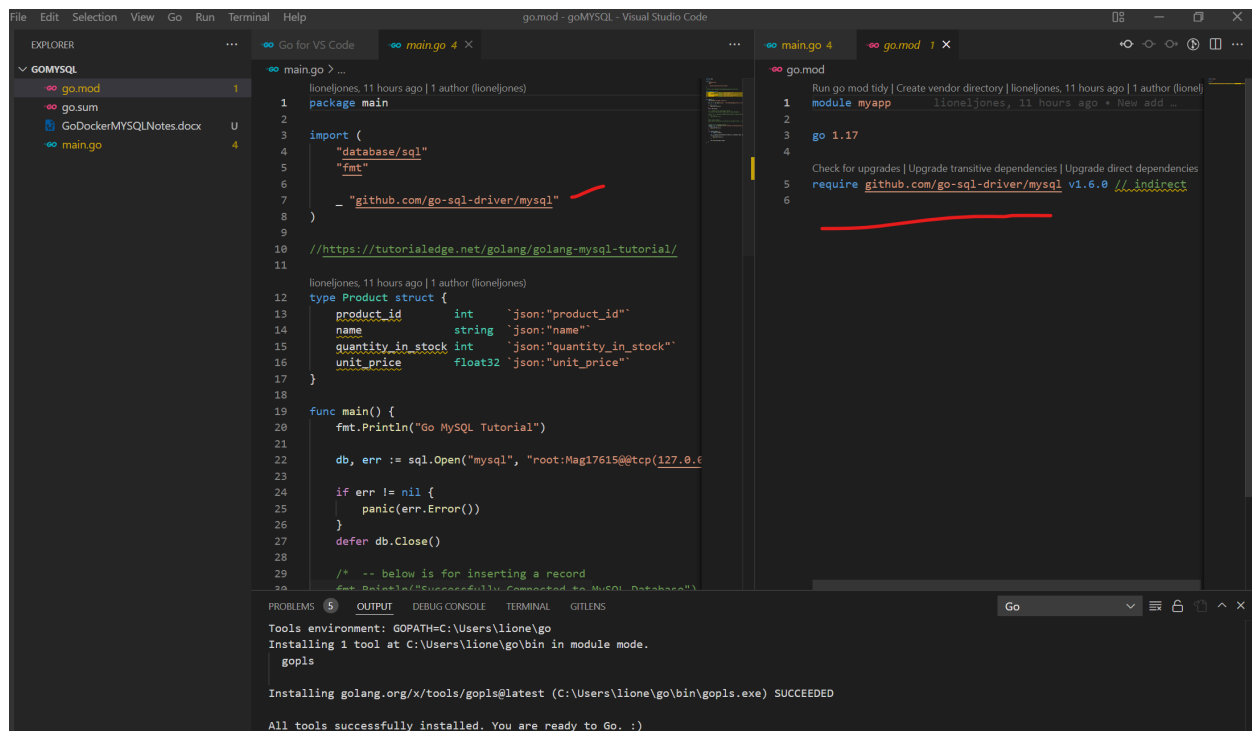
What this does is:

The go mod init command **creates a go.mod file to track your code's dependencies**

This is like a node\_modules file (it is called **go.mod**)

When you add imports to your program, if go does not find the correct dependency in your project, it will download it and add the reference to the go.mod file

For example:



The screenshot shows a Visual Studio Code editor with two files open: `main.go` and `go.mod`. The `main.go` file contains a Go program that imports the `database/sql` package and the `github.com/go-sql-driver/mysql` package. It defines a `Product` struct and a `main` function that connects to a MySQL database. The `go.mod` file shows the module name `myapp` and the dependency `github.com/go-sql-driver/mysql v1.6.0` with the `indirect` tag. The terminal at the bottom shows the output of the `go mod tidy` command, which successfully installed the `mysql` dependency.

```
1 package main
2
3 import (
4     "database/sql"
5     "fmt"
6     _ "github.com/go-sql-driver/mysql"
7 )
8
9 //https://tutorialedge.net/golang/golang-mysql-tutorial/
10
11 lioneljones, 11 hours ago | 1 author (lioneljones)
12 type Product struct {
13     product_id int    `json:"product_id"`
14     name        string `json:"name"`
15     quantity_in_stock int  `json:"quantity_in_stock"`
16     unit_price  float32 `json:"unit_price"`
17 }
18
19 func main() {
20     fmt.Println("Go MySQL Tutorial")
21
22     db, err := sql.Open("mysql", "root:Mag17615@tcp(127.0.0.1:3306)/myapp")
23
24     if err != nil {
25         panic(err.Error())
26     }
27     defer db.Close()
28
29     /* -- below is for inserting a record
30     db.Exec("INSERT INTO product (product_id, name, quantity_in_stock, unit_price) VALUES (1, 'Go MySQL Tutorial', 1, 1.0)")
31     */
32 }
```

```
1 module myapp
2
3 go 1.17
4
5 require github.com/go-sql-driver/mysql v1.6.0 // indirect
```

Tools environment: GOPATH=C:\Users\lione\go  
Installing 1 tool at C:\Users\lione\go\bin in module mode.  
gopls  
Installing golang.org/x/tools/gopls@latest (C:\Users\lione\go\bin\gopls.exe) SUCCEEDED  
All tools successfully installed. You are ready to Go. :)

In my MySQL go program, I added an import reference to mysql, when I tried to run the program, go did not recognize mysql. It prompted me to install it, once I installed it, it placed the dependency in the go.mod file.

## The import keyword

import (

"fmt"

"math/rand"

)

This is the same as any other import statement in other languages. It is where you add the packages that you want to use in your program

### Data Types:

[https://www.w3schools.com/go/go\\_data\\_types.php](https://www.w3schools.com/go/go_data_types.php)

**Go has three basic data types:**

**bool:** represents a boolean value and is either true or false

**Numeric:** represents integer types, floating point values, and complex types

**string:** represents a string value

### Integers

[https://www.w3schools.com/go/go\\_integer\\_data\\_type.php](https://www.w3schools.com/go/go_integer_data_type.php)

int,int8,int16,int32,int64

### Floats

[https://www.w3schools.com/go/go\\_float\\_data\\_type.php](https://www.w3schools.com/go/go_float_data_type.php)

float32,float64

### Strings

[https://www.w3schools.com/go/go\\_string\\_data\\_type.php](https://www.w3schools.com/go/go_string_data_type.php)

### Variables

[https://www.w3schools.com/go/go\\_variables.php](https://www.w3schools.com/go/go_variables.php)

In Go, there are two ways to declare a variable

```
var variablename type = value
```

or

```
variablename := value (this method uses type inference)
```

## Example

```
package main
import ("fmt")

func main() {
    var student1 string = "John" //type is string
    var student2 = "Jane" //type is inferred
    x := 2 //type is inferred

    fmt.Println(student1)
    fmt.Println(student2)
    fmt.Println(x)
}
```

[Try it Yourself »](#)

## Functions

functions are declared as below:

The way functions work is a little different in go:

```
functions.go
1 package main
2
3 import "fmt"
4
5 func add(x int, y int) int {
6     return x + y
7 }
8
9 func main() {
10     fmt.Println(add(42, 13))
11 }
12
```

Notice how the type comes after the variable name, also the return type comes after the method signature

## Loops

[https://www.w3schools.com/go/go\\_loops.php](https://www.w3schools.com/go/go_loops.php)

The **for** loop is **the only** loop available in Go.



```

package main
import ("fmt")

func main() {
    for i:=0; i < 5; i++ {
        fmt.Println(i)
    }
}

```

## Classes

Go **does not** have Classes, but it has **Structures**

[https://www.w3schools.com/go/go\\_struct.php](https://www.w3schools.com/go/go_struct.php)

Syntax:

```

type Person struct {
    name string
    age int
    job string
    salary int
}

```

Structures are declared with the **type** keyword

When structs are used, we don't declare them with a "new" keyword or with a (). We use the var keyword. And remember since the structure is a type, we note the type after the variable name.

Example:

```

type Person struct {
    name string
    age int
    job string
    salary int
}

```

```

func main() {
    var pers1 Person

    // Pers1 specification
    pers1.name = "Hege"
}

```

```
pers1.age = 45
pers1.job = "Teacher"
pers1.salary = 6000

// Access and print Pers1 info
fmt.Println("Name: ", pers1.name)
fmt.Println("Age: ", pers1.age)
fmt.Println("Job: ", pers1.job)
fmt.Println("Salary: ", pers1.salary)

}
```

## go Build

See the command list below

This builds an executable of your program. You can distribute the program and the target computer “**does not**” have to have Go installed to run the go program

### Quick List

### Tip: Go Commands

Command	Description
<code>go run <u>file.go</u></code>	runs your go program
<code>go mod init <u>myapp</u></code>	creates a module for you to use your own custom modules
<code>var <u>&lt;variablename&gt;</u> string</code>	creating variables
<code><u>&lt;variablename&gt;:= ""</u></code>	creating variables using type inference
<code>go build <u>main.go</u></code> or  <b>mac</b> <code>go build -o <u>eliza</u> <u>main.go</u></code> <b>windows</b> <code>go build -o <u>eliza.exe</u> <u>main.go</u></code>	builds your program  <b>To build with a different name</b>
<code>.\main</code>	to run your program
<code>go mod init <u>myapp</u></code>	command for Creating a custom mod file for your application's modules

## Adding and calling your own package

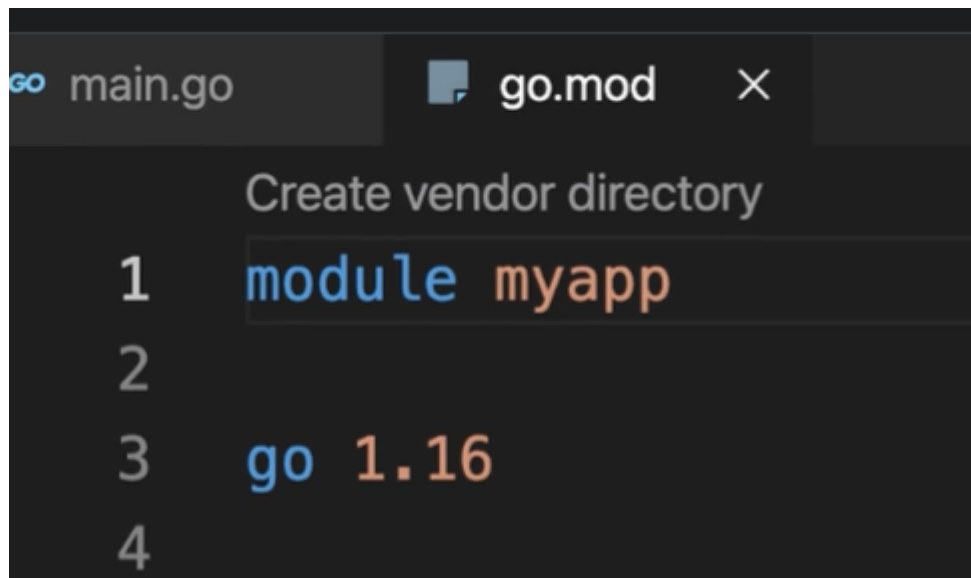
<https://www.udemy.com/course/go-programming-language-crash-course/learn/lecture/26161716#overview>

To allow for this, you have to create the mod file as discussed earlier

At the command line, type:

`go mod init myapp`

Once you do that, it creates the module file:

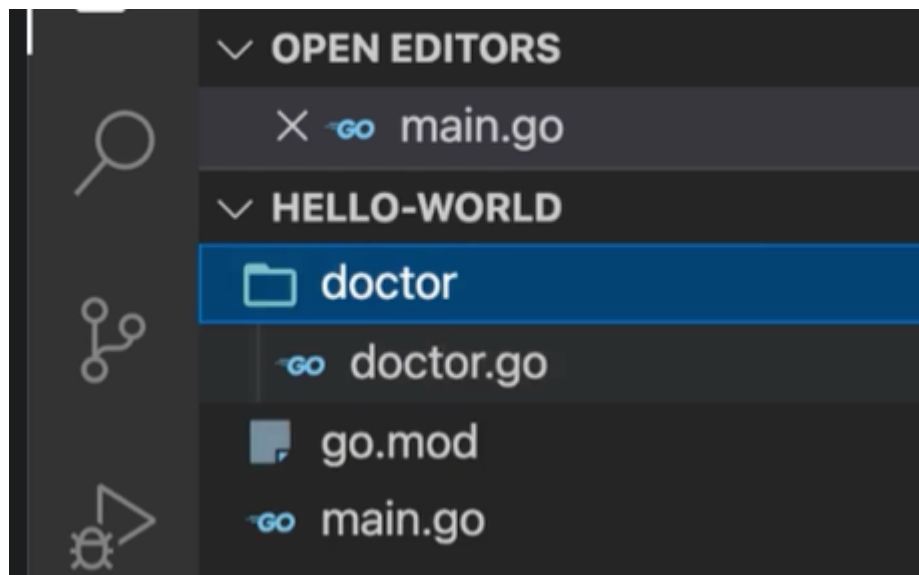


The screenshot shows a code editor with two tabs at the top: 'main.go' and 'go.mod'. The 'go.mod' tab is active, displaying the following content:

```
Create vendor directory
1  module myapp
2
3  go 1.16
4
```

Notice the module is named “myApp” based on the name you used at the command line.

Next create a directory and create .go file in it:



Paste some code in the file

In your main program, when you want to use this module, you make a reference to the module in your import statement (closure)

```

1  package main
2
3  import (
4      "fmt"
5      "myapp/doctor"
6  )
7
8  func main() {
9      var whatToSay string
10
11     whatToSay = doctor.Intro()
12 }
13

```

{LP} Learn Programming academy

This allows you to reference a method in the package you created. (Notice how we use the “myapp” keyword along with reference to the package)

```

// Intro returns the intro text
func Intro() string {
    return `
I'm Eliza
-----

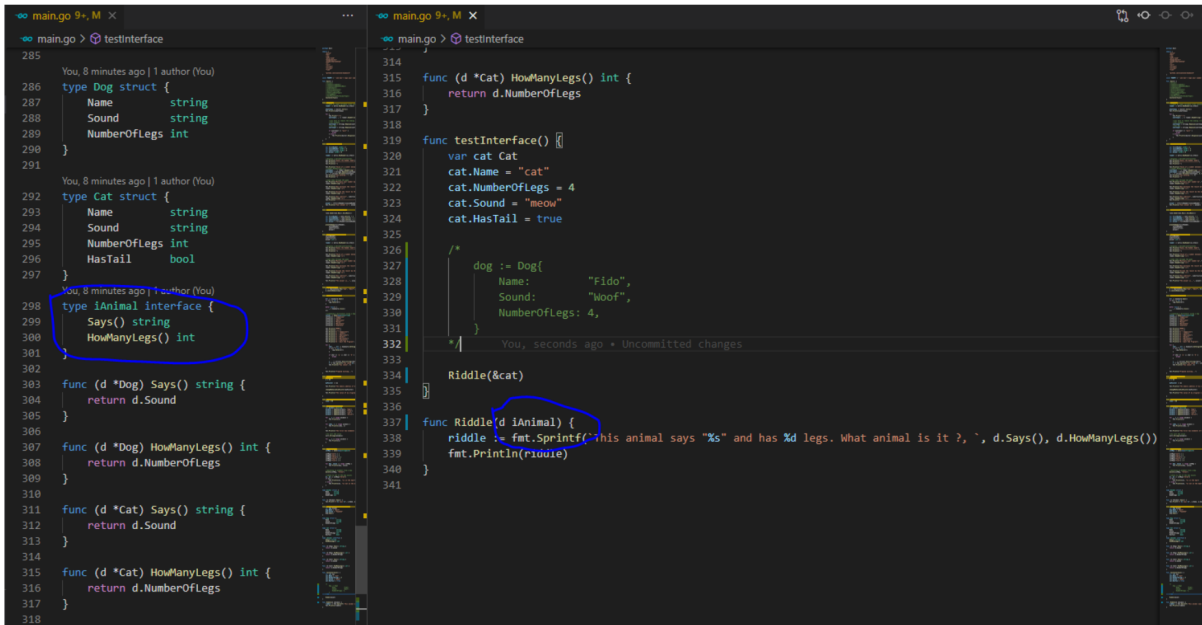
Talk to the program by typing in plain English, using normal
and lower-case letters and punctuation. Enter 'quit' when done.

Hello. How are you feeling today?`
}

```

**Tip:** Interfaces

As you see below, interfaces "can" have methods as opposed to structures. Also you will note below that this demonstrates a good use case to use a pointer in [golang](https://golang.org/). Interfaces use the "attach" a function to a structure methodology. You don't use the keyword "implement" or: like in inheritance. You just meet the requirements of creating the functions the interface requires.



The image shows two side-by-side windows from a Go IDE. The left window, titled 'main.go 9+, M X', contains the following code:

```
285 // You, 8 minutes ago | 1 author (You)
286 type Dog struct {
287     Name      string
288     Sound     string
289     NumberOfLegs int
290 }
291
292 // You, 8 minutes ago | 1 author (You)
293 type Cat struct {
294     Name      string
295     Sound     string
296     NumberOfLegs int
297     HasTail   bool
298 }
299
300 // You, 8 minutes ago | 1 author (You)
301 type IAnimal interface {
302     Says() string
303     HowManyLegs() int
304 }
305
306 func (d *Dog) Says() string {
307     return d.Sound
308 }
309
310 func (d *Dog) HowManyLegs() int {
311     return d.NumberOfLegs
312 }
313
314 func (d *Cat) Says() string {
315     return d.Sound
316 }
317
318 func (d *Cat) HowManyLegs() int {
319     return d.NumberOfLegs
320 }
```

The right window, also titled 'main.go 9+, M X', contains the following code:

```
314
315 func (d *Cat) HowManyLegs() int {
316     return d.NumberOfLegs
317 }
318
319 func testInterface() {
320     var cat Cat
321     cat.Name = "cat"
322     cat.NumberOfLegs = 4
323     cat.Sound = "meow"
324     cat.HasTail = true
325
326     /*
327     dog := Dog{
328         Name:      "Fido",
329         Sound:     "Woof",
330         NumberOfLegs: 4,
331     }
332     */
333     Riddle(&cat)
334 }
335
336 func Riddle(iAnimal) {
337     riddle := fmt.Sprintf("this animal says \"%s\" and has %d legs. What animal is it ?, ", iAnimal.Says(), iAnimal.HowManyLegs())
338     fmt.Println(riddle)
339 }
340
341
```

## The \_ in Golang

What is Blank Identifier(underscore) in Golang?

\_ (underscore) in Golang is known as the Blank Identifier.

Golang has a special feature to define and use the unused variable using Blank Identifier.

**The real use of Blank Identifier comes when a function returns multiple values**

```
func main ()
{
    reader:= bufio.NewReader(os.Stdin)
    whatToSay := doctor.Intro()
    fmt.Println(whatToSay)

    userInput, _ := reader.ReadString()

}
```

**Above**

The reader.ReadString() function returns "two" return values. But we "don't" have to use the value for the \_, the underscore is just a "placeholder"

## Functions that return two values

```
reg, err := regexp.Compile("[^a-zA-Z0-9"])
```

```
if err != nil {
```

```
    log.Fatal(err)
}
userInput = reg.ReplaceAllString(userInput, "")
```

Above is typical, you will see error handling handled where you have the err as the 2<sup>nd</sup> return value

**Stripping of the carriage return placeholder when reading a string from user input (from iO reader)**

<https://www.udemy.com/course/go-programming-language-crash-course/learn/lecture/26161724#overview>

```
userInput = strings.Replace(userInput, "\r\n", "", -1)
```