

Table of Contents

Chapter 1: Getting Started with Go for Data Structures and Algorithms	1
Technical Requirements	1
Arrays	2
Slices	3
Two Dimensional Slices	6
Maps	11
Database Operations	12
Variadic Functions	16
CRUD Web Forms	21
Defer and Panic	26
Summary	35
Q&A	36
Reference	36
Index	37

1

Getting Started with Go for Data Structures and Algorithms

Slices are similar to arrays except that they have unusual properties. `make` function is used to create a slice of specific type and with capacity. `len` and `cap` functions of slice return length and capacity of slices respectively. `Copy` and `append` functions are used to increase the capacity and contents of the slice. Slice operations such as enlarging a slice using `append` and `copy` methods, assigning parts of a slice, appending a slice, and appending a part of a slice are presented with code samples.

Database operations and CRUD web forms are the scenarios in which Go data structures and algorithms are demonstrated.

In this chapter, we will discuss the following Go language specific data structures:

- Arrays
- Slices
- Two dimensional slices
- Maps

Technical Requirements

Install Go version `go1.10` from the [link on golang.org](http://link.on.golang.org) depending on your os version.

The github url for the code in chapter 2 will be at [Hands-On Data Structures and Algorithms with Go](#), published by Packt

In this chapter, database operations require “github.com/go-sql-driver/mysql”. In addition to this, MySQL (4.1+) needs to be installed from [Dev Mysql site](#).

Run commands are shown below:

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

Arrays

Arrays are the most famous data structures in different programming languages. Different data types can be handled as elements in Arrays such as int, float32, double, and others. The following code snippet shows the initialisation of an array:

arrays.go

```
var arr = [5]int {1,2,4,5,6}
```

An array's size can be found with the len() function. A for loop is used for accessing all the elements in an array as shown below:

```
var i int
for i=0; i< len(arr); i++ {
    fmt.Println("printing elements ",arr[i])
}
```

In the next code snippet, the Range keyword is explained in detail. The Range keyword can be used to access index and value for each element:

```
var value int
for i, value = range arr{
    fmt.Println(" range ",value)
}
```

_blank identifier is used if index is ignored. The code exhibit shows how _blank identifier can be used:

```
for _, value = range arr{
    fmt.Println("blank range",value)
}
```

Run the following commands:

```
go run arrays.go
```

The screenshot of the output is attached below:

```
chapter2 -- -bash -- 80x30
Bhagvans-MacBook-Pro:chapter2 bhagvankomadi$ go run arrays.go
printing elements 1
printing elements 2
printing elements 4
printing elements 5
printing elements 6
  range 1
  range 2
  range 4
  range 5
  range 6
blank range 1
blank range 2
blank range 4
blank range 5
blank range 6
Bhagvans-MacBook-Pro:chapter2 bhagvankomadi$
```

Go arrays are not dynamic but fixed in size. To add more elements than the size, a bigger array needs to be created and all the elements of the old one needs to be copied. Array is passed as a value through functions by copying the array. Passing a big array to a function might be a performance issue.

Slices

A Go Slice can be appended with elements after the capacity has reached its size. The `len()` function gives the current length of the slice, and the capacity of the slice can be obtained using the `cap()` function.

Slices are dynamic and can double the current capacity in order to add more elements. The code sample below shows the basic slice creation and appending a slice.

basic_slice.go

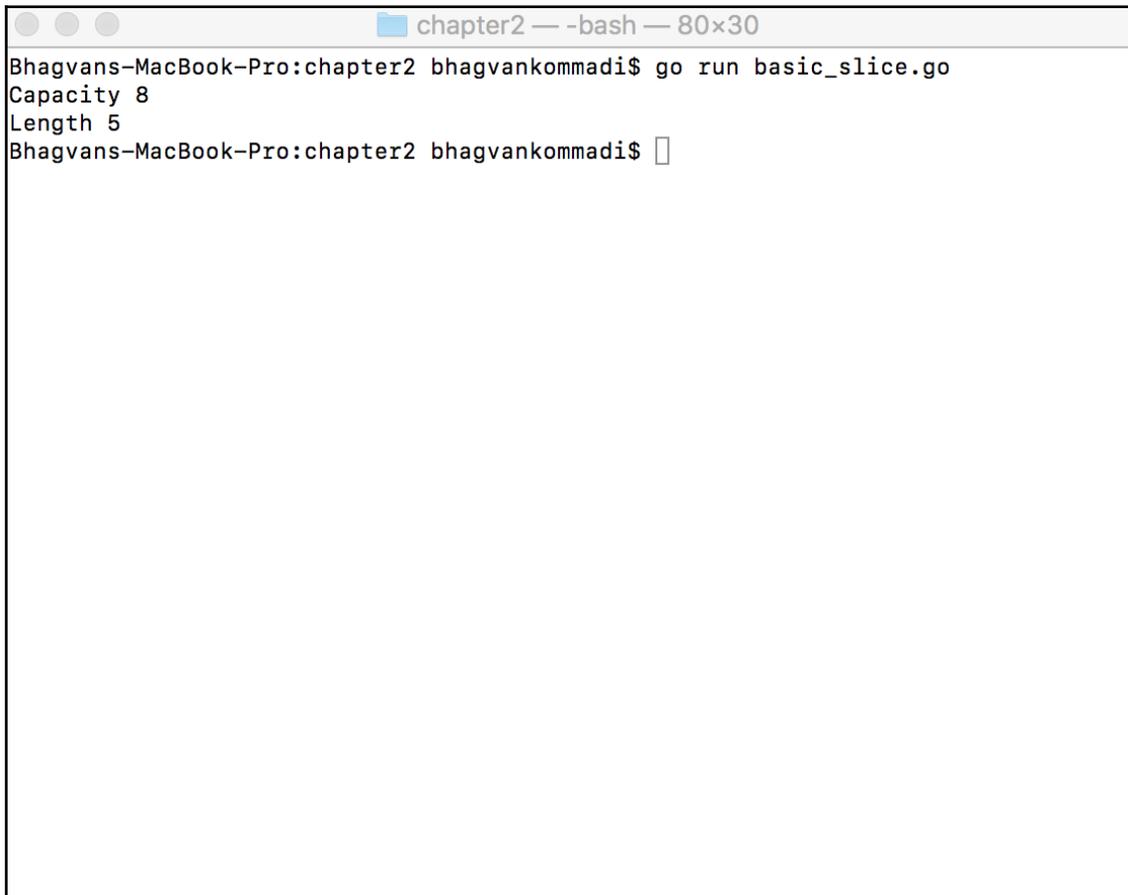
```
var slice = []int{1,3,5,6}
slice = append(slice, 8)
fmt.Println("Capacity", cap(slice))
```

```
fmt.Println("Length", len(slice))
```

Run the following commands to execute the above code exhibit:

```
go run basic_slice.go
```

The screenshot of the output is attached below:



```
chapter2 --bash-- 80x30
Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$ go run basic_slice.go
Capacity 8
Length 5
Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$
```

Slices are passed by reference to functions. Big slices can be passed to the functions without impacting the performance. Passing a slice as a reference to a function is demonstrated in the code below:

slices.go

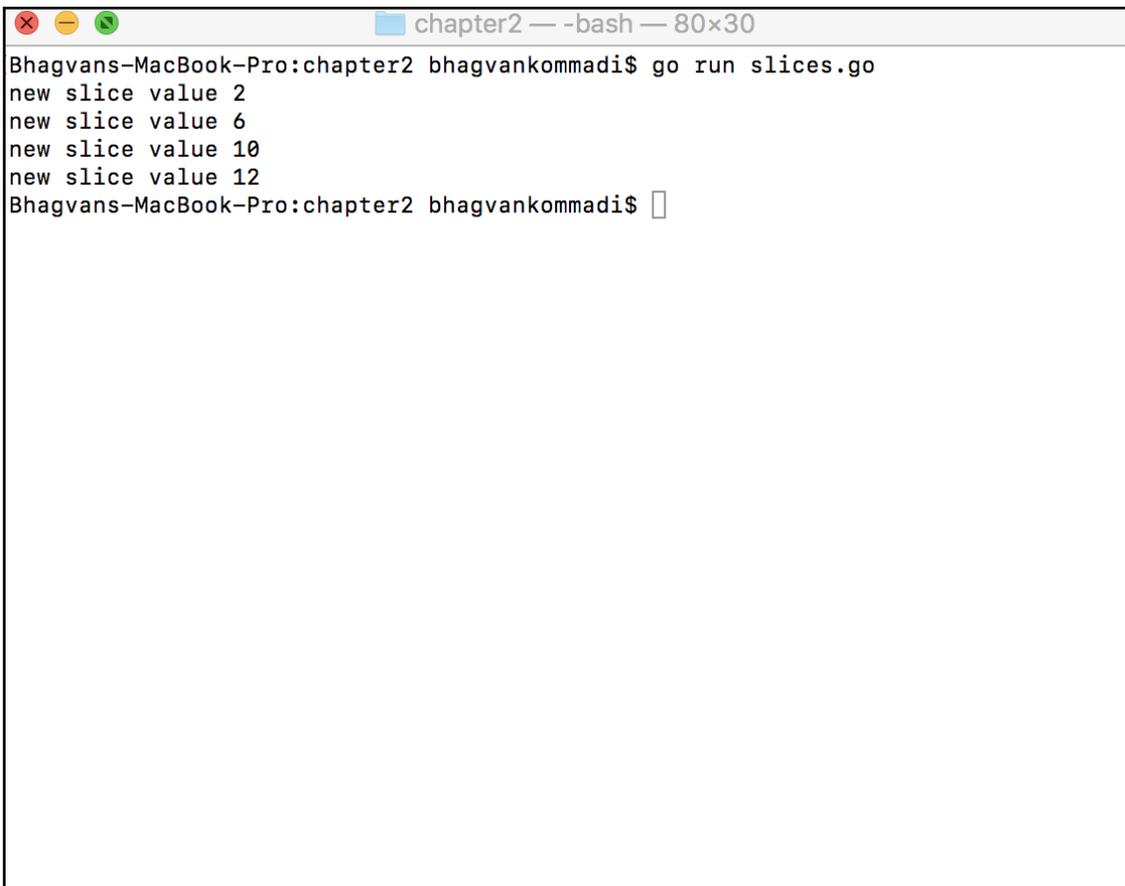
```
//twiceValue method given slice of int type
```

```
func twiceValue(slice []int) {
    var i int
    var value int
    for i, value = range slice {
        slice[i] = 2*value
    }
}
// main method
func main() {
    var slice = []int{1,3,5,6}
    twiceValue(slice)
    var i int
    for i=0; i< len(slice); i++ {
        fmt.Println("new slice value", slice[i])
    }
}
```

Run the following commands:

```
go run slices.go
```

The screenshot of the output is attached below:

A terminal window titled "chapter2 — -bash — 80x30" on a Mac. The prompt is "Bhagvans-MacBook-Pro:chapter2 bhagvankommadi\$". The user has entered "go run slices.go". The output shows four lines: "new slice value 2", "new slice value 6", "new slice value 10", and "new slice value 12". The prompt is now "Bhagvans-MacBook-Pro:chapter2 bhagvankommadi\$" with a cursor.

```
chapter2 — -bash — 80x30
Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$ go run slices.go
new slice value 2
new slice value 6
new slice value 10
new slice value 12
Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$
```

Two Dimensional Slices

Two dimensional slices are descriptors of a two dimensional array. This is a contiguous section of an array which is stored away from the slice itself. They hold references to an underlying array. A two dimensional slice will be an array of arrays, while the capacity of a slice can be increased by creating a new slice and copying the contents of the initial slice into the new one. This is also referred to as a slice of slices. Below is an example of two dimensional arrays. 2D array is created and the array elements are initialised with values.

twodarray.go is the code exhibit presented in the code section as follows:

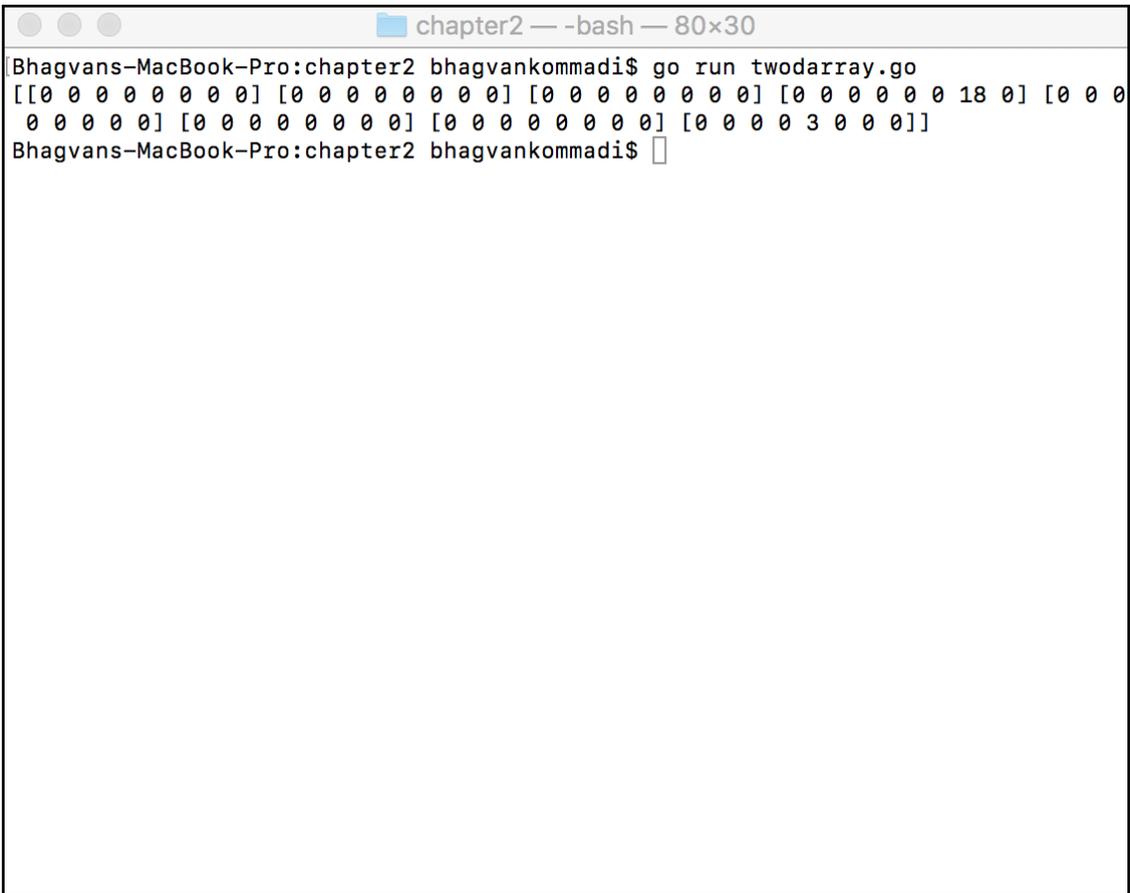
```
//main package has examples shown
// in Go Data Structures and algorithms book
```

```
package main
// importing fmt package
import (
    "fmt"
)
// main method
func main() {
    var TwoDArray [8][8]int
    TwoDArray[3][6] = 18
    TwoDArray[7][4] = 3
    fmt.Println(TwoDArray)
}
```

Run the following commands:

```
go run twodarray.go
```

The screenshot of the output is attached below:



```
chapter2 — -bash — 80x30
[Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$ go run twodarray.go
[[0 0 0 0 0 0 0] [0 0 0 0 0 0 0] [0 0 0 0 0 0 0] [0 0 0 0 0 18 0] [0 0 0
0 0 0 0] [0 0 0 0 0 0 0] [0 0 0 0 0 0 0] [0 0 0 0 3 0 0 0]]
Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$
```

For dynamic allocation, we use slice of slices. Slice of slices is explained below as two dimensional slices in the code exhibit - twodslices.go

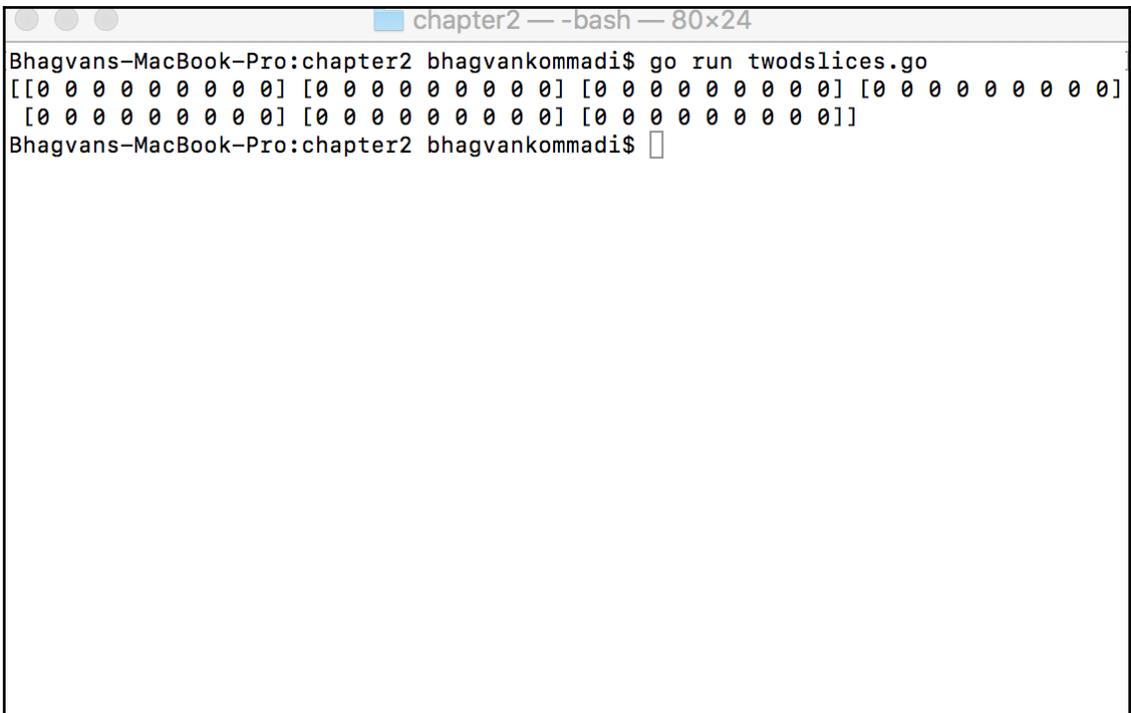
```
// in Go Data Structures and algorithms book
package main
// importing fmt package
import (
    "fmt"
)
// main method
func main() {
    var rows int
    var cols int
    rows = 7
    cols = 9
```

```
var twodslices = make([][]int, rows)
var i int
for i = range twodslices {
    twodslices[i] = make([]int, cols)
}
fmt.Println(twodslices)
}
```

Run the following commands:

```
go run twodslices.go
```

The screenshot of the output is attached below:



```
chapter2 — -bash — 80x24
Bhagvans-MacBook-Pro:chapter2 bhagvankomadi$ go run twodslices.go
[[0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0]]
Bhagvans-MacBook-Pro:chapter2 bhagvankomadi$
```

The "append" method on slice is used to append new elements in the slice. If the slice capacity has reached the size of the underlying array, then append increases the size by creating a new underlying array and adding the new element. slic1 is a sub slice of arr starting from 0 to 3 (excluded), while slic2 is a sub slice of arr starting from 1 (inclusive) to 5 (excluded). In the snippet below, the append method calls on slic2 in order to add a new '12' element:

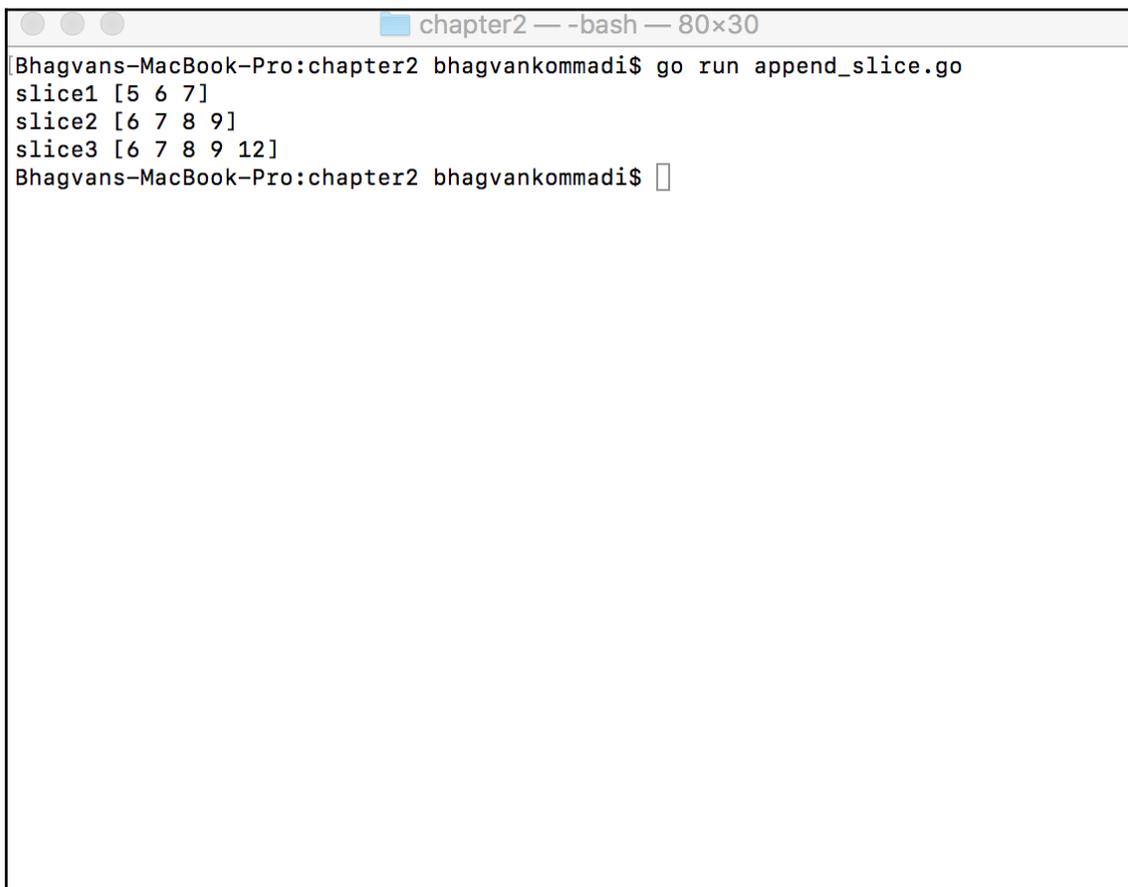
append_slice.go

```
var arr = [] int{5,6,7,8,9}
var slic1 = arr[: 3]
fmt.Println("slice1",slic1)
var slic2 = arr[1:5]
fmt.Println("slice2",slic2)
var slic3 = append(slic2, 12)
fmt.Println("slice3",slic3)
```

Run the following commands:

```
go run append_slice.go
```

The screenshot of the output is attached below:



```
chapter2 -- -bash -- 80x30
[Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$ go run append_slice.go
slice1 [5 6 7]
slice2 [6 7 8 9]
slice3 [6 7 8 9 12]
Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$ ]
```

Maps

Maps are used to keep track of keys which are types such as integers, strings, float, double, pointers, interfaces, structs and arrays. The value can be of different types. In the example below, languages of map type with a key integer and value string is created:

maps.go

```
var languages = map[int]string {
    3: "English",
    4: "French",
    5: "Spanish"
}
```

Maps can be created using the "make" method, specifying the key type and value type. Products of map type with key integer and value string are shown below in the code snippet:

```
var products = make(map[int]string)
products[1] = "chair"
products[2] = "table"
```

A for loop is used for iterating through the map. The languages map is iterated below:

```
var i int
var value string
for i, value = range languages {
    fmt.Println("language", i, ":", value)
}
fmt.Println("product with key 2", products[2])
```

Retrieving value and deleting slice operations are shown below using the products map:

```
fmt.Println(products[2])
delete(products, "chair")
fmt.Println("products", products)
```

Run the following commands:

```
go run maps.go
```

The screenshot of the output is attached below:

```
chapter2 — -bash — 80x30
Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$ go run maps.go
language 3 : English
language 4 : French
language 5 : Spanish
product with key 2 table
products map[2:table]
Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$
```

Database Operations

GetCustomer method retrieves the Customer data from the database. To start with, the Create Database operation is shown in the example below. Customer is the table with CustomerId, CustomerName and SSN attributes. The GetConnection method returns the database connection, which is used to query the database. The query then returns the rows from the database table. In the code exhibit below, database operations are explained in detail:

database_operations.go

```
//main package has examples shown
// in Hands-On Data Structures and algorithms with Go book
```

```
package main

// importing fmt,database/sql, net/http, text/template package
import (
    "fmt"
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
)

// Customer Class
type Customer struct {
    CustomerId int
    CustomerName string
    SSN string
}

// GetConnection method which returns sql.DB
func GetConnection() (database *sql.DB) {
    databaseDriver := "mysql"
    databaseUser := "newuser"
    databasePass := "newuser"
    databaseName := "crm"
    database, error := sql.Open(databaseDriver,
databaseUser+": "+databasePass+"@"+"/"+databaseName)
    if error != nil {
        panic(error.Error())
    }
    return database
}

// GetCustomers method returns Customer Array
func GetCustomers() []Customer {
    var database *sql.DB
    database = GetConnection()

    var error error
    var rows *sql.Rows
    rows, error = database.Query("SELECT * FROM Customer ORDER BY
CustomerId DESC")
    if error != nil {
        panic(error.Error())
    }
    var customer Customer
    customer = Customer{}

    var customers []Customer
    customers= []Customer{}
    for rows.Next() {
        var customerId int
        var customerName string
```

```
        var ssn string
        error = rows.Scan(&customerId, &customerName, &ssn)
        if error != nil {
            panic(error.Error())
        }
        customer.CustomerId = customerId
        customer.CustomerName = customerName
        customer.SSN = ssn
        customers = append(customers, customer)
    }

    defer database.Close()

    return customers
}

//main method
func main() {

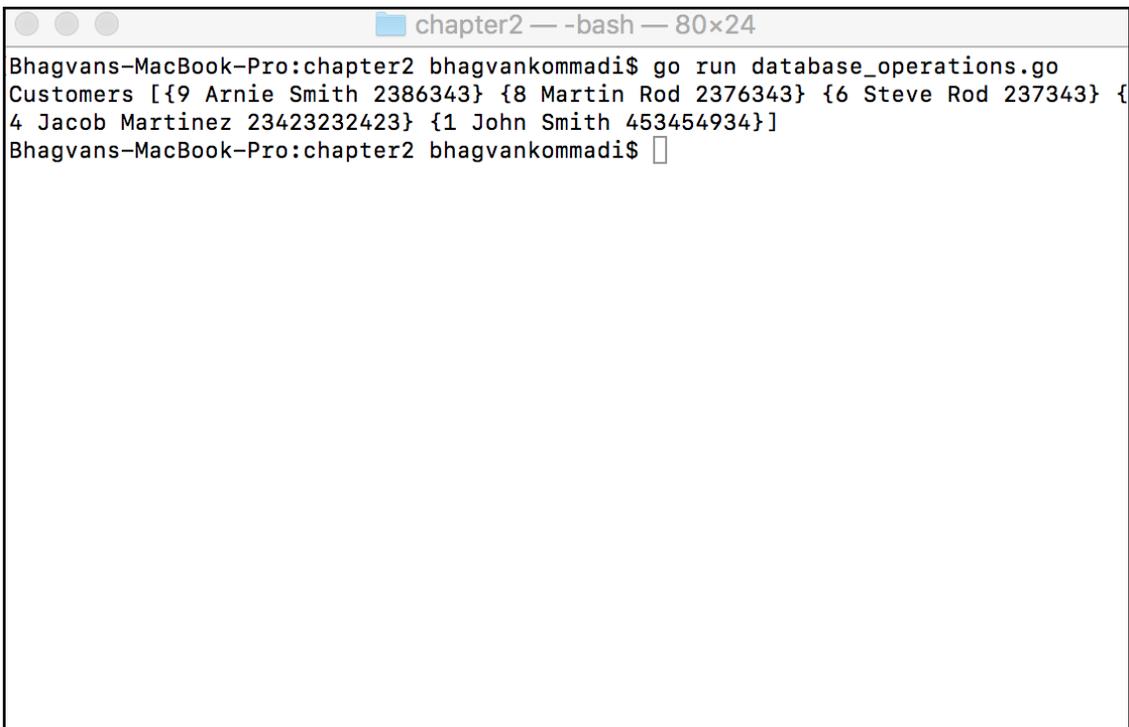
    var customers []Customer
    customers = GetCustomers()
    fmt.Println("Customers",customers)

}
```

Run the following commands:

```
go run database_operations.go
```

The screenshot of the output is attached below:

A terminal window titled "chapter2 -- -bash -- 80x24" on a Mac. The prompt is "Bhagvans-MacBook-Pro:chapter2 bhagvankommadi\$". The command "go run database_operations.go" has been executed, resulting in the output: "Customers [{9 Arnie Smith 2386343} {8 Martin Rod 2376343} {6 Steve Rod 237343} {4 Jacob Martinez 23423232423} {1 John Smith 453454934}]". The prompt is now "Bhagvans-MacBook-Pro:chapter2 bhagvankommadi\$".

```
Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$ go run database_operations.go
Customers [{9 Arnie Smith 2386343} {8 Martin Rod 2376343} {6 Steve Rod 237343} {
4 Jacob Martinez 23423232423} {1 John Smith 453454934}]
Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$
```

The Insert operation is shown below. The InsertCustomer method takes the Customer parameter and creates a prepared statement for the Insert statement. The statement is used to execute the insertion of customer rows into the table, as shown in the snippet below:

```
// InsertCustomer method with parameter customer
func InsertCustomer(customer Customer) {
    var database *sql.DB
    database= GetConnection()

    var error error
    var insert *sql.Stmt
    insert,error = database.Prepare("INSERT INTO
CUSTOMER(CustomerName,SSN) VALUES(?,?) ")
    if error != nil {
        panic(error.Error())
    }
    insert.Exec(customer.CustomerName,customer.SSN)

    defer database.Close()
```

```
}
```

Variadic Functions

Variadic functions can be invoked with a variable number of parameters. `fmt.Println` is a common variadic function as shown below:

```
//main method
func main() {
    var customers []Customer
    customers = GetCustomers()
    fmt.Println("Before Insert",customers)
    var customer Customer
    customer.CustomerName = "Arnie Smith"
    customer.SSN = "2386343"
    InsertCustomer(customer)
    customers = GetCustomers()
    fmt.Println("After Insert",customers)
}
```

Run the following commands:

```
go run database_operations.go
```

The screenshot of the output is attached below:

```
chapter2 — -bash — 80x30
Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$ go run database_operations.go
Before Insert [{8 Martin Rod 2376343} {6 Steve Rod 237343} {5 Michael Thompson
8834344} {4 Jacob Martinez 23423232423} {1 John Smith 453454934}]
After Insert [{9 Arnie Smith 2386343} {8 Martin Rod 2376343} {6 Steve Rod 237343
} {5 Michael Thompson 8834344} {4 Jacob Martinez 23423232423} {1 John Smith 453
454934}]
Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$
```

The update operation is shown below. UpdateCustomer method takes the parameter Customer and creates a prepared statement for the Update statement. The statement is used to update a customer row in the table:

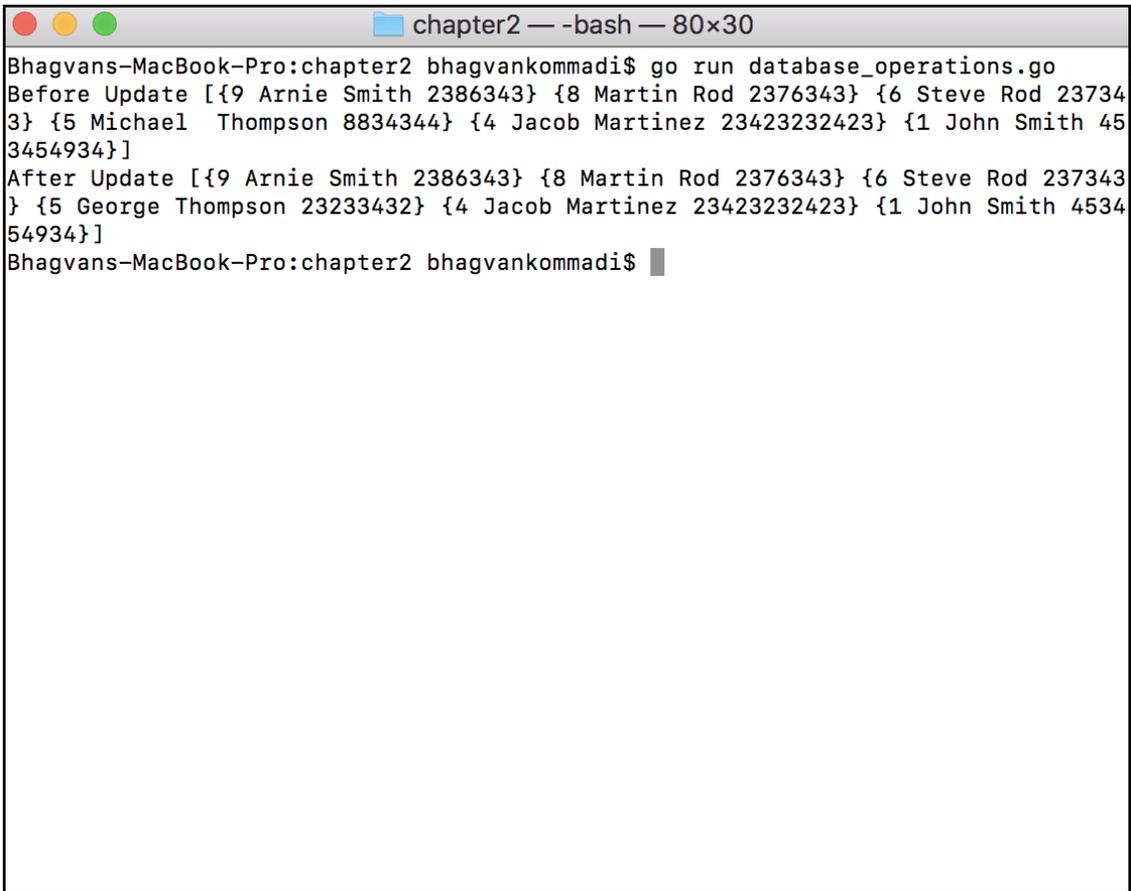
```
// Update Customer method with parameter customer
func UpdateCustomer(customer Customer) {
    var database *sql.DB
    database= GetConnection()
    var error error
    var update *sql.Stmt
    update,error = database.Prepare("UPDATE CUSTOMER SET CustomerName=?,
SSN=? WHERE CustomerId=?")
    if error != nil {
        panic(error.Error())
    }
    update.Exec(customer.CustomerName,customer.SSN,customer.CustomerId)
```

```
defer database.Close()
}
// main method
func main() {
    var customers []Customer
    customers = GetCustomers()
    fmt.Println("Before Update",customers)
    var customer Customer
    customer.CustomerName = "George Thompson"
    customer.SSN = "23233432"
    customer.CustomerId = 5
    UpdateCustomer(customer)
    customers = GetCustomers()
    fmt.Println("After Update",customers)
}
```

Run the following commands:

```
go run database_operations.go
```

The screenshot of the output is attached below:

A terminal window titled "chapter2 — -bash — 80x30" showing the execution of a Go program. The output displays two arrays of customer records. The first array, labeled "Before Update", contains five records: Arnie Smith (2386343), Martin Rod (2376343), Steve Rod (237343), Michael Thompson (8834344), and Jacob Martinez (23423232423). The second array, labeled "After Update", contains five records: Arnie Smith (2386343), Martin Rod (2376343), Steve Rod (237343), George Thompson (23233432), and Jacob Martinez (23423232423). The terminal prompt is "Bhagvans-MacBook-Pro:chapter2 bhagvankommadi\$".

```
Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$ go run database_operations.go
Before Update [{9 Arnie Smith 2386343} {8 Martin Rod 2376343} {6 Steve Rod 237343} {5 Michael Thompson 8834344} {4 Jacob Martinez 23423232423} {1 John Smith 453454934}]
After Update [{9 Arnie Smith 2386343} {8 Martin Rod 2376343} {6 Steve Rod 237343} {5 George Thompson 23233432} {4 Jacob Martinez 23423232423} {1 John Smith 453454934}]
Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$
```

The delete operation is shown below. The `DeleteCustomer` method takes the `Customer` parameter and creates a prepared statement for the Delete statement. The statement is used to execute the deletion of a customer row in the table:

```
// Delete Customer method with parameter customer
func deleteCustomer(customer Customer) {
    var database *sql.DB
    database= GetConnection()
    var error error
    var delete *sql.Stmt
    delete,error = database.Prepare("DELETE FROM Customer WHERE
Customerid=?")
    if error != nil {
        panic(error.Error())
    }
}
```

```
        delete.Exec(customer.CustomerId)
    defer database.Close()
}
// main method
func main() {
    var customers []Customer
    customers = GetCustomers()
    fmt.Println("Before Delete", customers)
    var customer Customer
    customer.CustomerName = "George Thompson"
    customer.SSN = "23233432"
    customer.CustomerId = 5
    deleteCustomer(customer)
    customers = GetCustomers()
    fmt.Println("After Delete", customers)
}
```

Run the following commands:

```
go run database_operations.go
```

The screenshot of the output is attached below:

```
chapter2 -- -bash -- 80x30
Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$ go run database_operations.go
Before Delete [{9 Arnie Smith 2386343} {8 Martin Rod 2376343} {6 Steve Rod 237343} {5 George Thompson 23233432} {4 Jacob Martinez 23423232423} {1 John Smith 453454934}]
After Delete [{9 Arnie Smith 2386343} {8 Martin Rod 2376343} {6 Steve Rod 237343} {4 Jacob Martinez 23423232423} {1 John Smith 453454934}]
Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$
```

CRUD Web Forms

To start a basic html page with GO net/http package, the webforms example is shown below. This has a welcome greeting in main.html

webforms.go

```
//main package has examples shown
// in Hands-On Data Structures and algorithms with Go book
package main
// importing fmt, database/sql, net/http, text/template package
import (
    "net/http"
```

```
    "text/template"
    "log")
// Home method renders the main.html
func Home(writer http.ResponseWriter, reader *http.Request) {
    var template_html *template.Template
    template_html = template.Must(template.ParseFiles("main.html"))
    template_html.Execute(writer, nil)
}
// main method
func main() {
    log.Println("Server started on: http://localhost:8000")
    http.HandleFunc("/", Home)
    http.ListenAndServe(":8000", nil)
}
```

main.html

```
<html>
<body>
<p> Welcome to Web Forms</p>
</body>
</html>
```

Run the following commands:

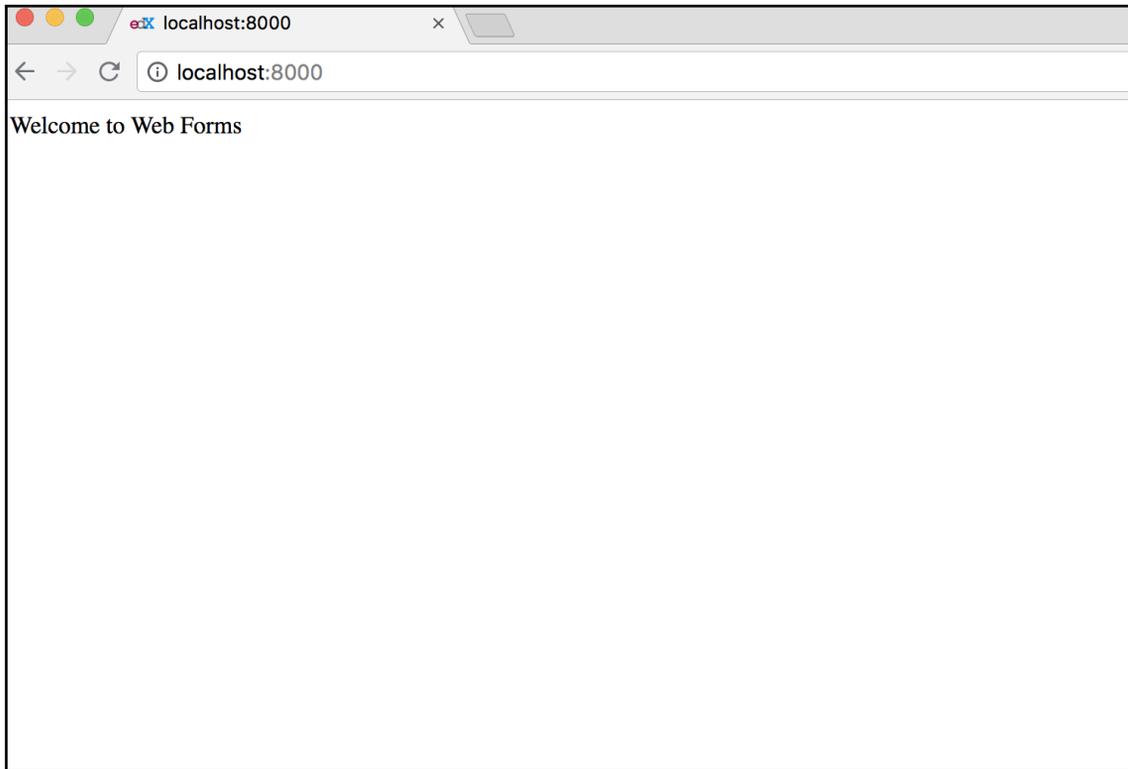
```
go run webforms.go
```

The screenshot of the output is attached below:

A terminal window titled "chapter2 — webforms < go run webforms.go — 80x24". The terminal shows the command "go run webforms.go" being executed. The output is "2018/08/24 17:30:02 Server started on: http://localhost:8000".

```
Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$ go run webforms.go
2018/08/24 17:30:02 Server started on: http://localhost:8000
```

The web browser output will look something like this:



The CRM application is built with web forms as an example in order to demonstrate CRUD operations. We can use database operations built in the previous section. In the code sample below, *crm* database operations are presented. CRM Database operations consists of CRUD methods such as Create, Read , Update and Delete Customer operations.

GetConnection method retrieves the database connection for performing the database operations.

crm_database_operations.go

```
//main package has examples shown
// in Hands-On Data Structures and algorithms with Go book
package main
// importing fmt,database/sql, net/http, text/template package
import (
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
)
// Customer Class
type Customer struct {
```

```
    CustomerId    int
    CustomerName  string
    SSN           string
}
// GetConnection method which returns sql.DB
func GetConnection() (database *sql.DB) {
    databaseDriver := "mysql"
    databaseUser   := "newuser"
    databasePass   := "newuser"
    databaseName   := "crm"
    database, error := sql.Open(databaseDriver,
databaseUser+": "+databasePass+"@"+"/"+databaseName)
    if error != nil {
        panic(error.Error())
    }
    return database
}
```

`GetCustomerById` method takes `customerId` parameter to look up in the database the customer. The method returns the customer object.

```
//GetCustomerById with parameter customerId returns Customer
func GetCustomerById(customerId int) Customer {
    var database *sql.DB
    database = GetConnection()
    var error error
    var rows *sql.Rows
    rows, error = database.Query("SELECT * FROM Customer WHERE
CustomerId=?",customerId)
    if error != nil {
        panic(error.Error())
    }
    var customer Customer
    customer = Customer{}
    for rows.Next() {
        var customerId int
        var customerName string
        var SSN string
        error = rows.Scan(&customerId, &customerName, &SSN)
        if error != nil {
            panic(error.Error())
        }
        customer.CustomerId = customerId
        customer.CustomerName = customerName
        customer.SSN = SSN
    }
}
```

Defer and Panic

The defer statement defers the execution of the function until the surrounding function returns. Panic function stops the current flow and control. Deferred Functions are executed normally after the panic call. In the code below, the Defer call gets executed even when the panic call is invoked.

```
        defer database.Close()
        return customer
    }
    // GetCustomers method returns Customer Array
    func GetCustomers() []Customer {
        var database *sql.DB
        database = GetConnection()
        var error error
        var rows *sql.Rows
        rows, error = database.Query("SELECT * FROM Customer ORDER BY
Customerid DESC")
        if error != nil {
            panic(error.Error())
        }
        var customer Customer
        customer = Customer{}
        var customers []Customer
        customers= []Customer{}
        for rows.Next() {
            var customerId int
            var customerName string
            var ssn string
            error = rows.Scan(&customerId, &customerName, &ssn)
            if error != nil {
                panic(error.Error())
            }
            customer.CustomerId = customerId
            customer.CustomerName = customerName
            customer.SSN = ssn
            customers = append(customers, customer)
        }
        defer database.Close()
        return customers
    }
```

InsertCustomer method takes customer as a parameter to execute the SQL statement for inserting into customer table.

```
// InsertCustomer method with parameter customer
func InsertCustomer(customer Customer) {
```

```
var database *sql.DB
database= GetConnection()
var error error
var insert *sql.Stmt
insert,error = database.Prepare("INSERT INTO
CUSTOMER(CustomerName,SSN) VALUES(?,?) ")
    if error != nil {
        panic(error.Error())
    }
insert.Exec(customer.CustomerName,customer.SSN)
defer database.Close()
}
```

UpdateCustomer method prepares the update statement by passing the customer name and SSN from customer object.

```
// Update Customer method with parameter customer
func UpdateCustomer(customer Customer) {
    var database *sql.DB
    database= GetConnection()
    var error error
    var update *sql.Stmt
    update,error = database.Prepare("UPDATE CUSTOMER SET CustomerName=?,
SSN=? WHERE CustomerId=?")
        if error != nil {
            panic(error.Error())
        }
    update.Exec(customer.CustomerName,customer.SSN,customer.CustomerId)
    defer database.Close()
}
```

DeleteCustomer method deletes the customer passed by executing the delete statement.

```
// Delete Customer method with parameter customer
func DeleteCustomer(customer Customer) {
    var database *sql.DB
    database= GetConnection()
    var error error
    var delete *sql.Stmt
    delete,error = database.Prepare("DELETE FROM Customer WHERE
Customerid=?")
        if error != nil {
            panic(error.Error())
        }
    delete.Exec(customer.CustomerId)
    defer database.Close()
}
```

The CRM web application is shown below with various web paths handled. CRM application code is shown in the code sample below. Home function executes the Home template with parameters `writer` and `customers` array.

crm_app.go

```
//main package has examples shown
// in Hands-On Data Structures and algorithms with Go book
package main

// importing fmt,database/sql, net/http, text/template package
import (
    "fmt"
    "net/http"
    "text/template"
    "log"
)

var template_html = template.Must(template.ParseGlob("templates/*"))

// Home - execute Template
func Home(writer http.ResponseWriter, request *http.Request) {
    var customers []Customer
    customers = GetCustomers()
    log.Println(customers)
    template_html.ExecuteTemplate(writer, "Home", customers)
}
}
```

Create function takes `writer` and `request` parameters to render Create Template.

```
// Create - execute Template
func Create(writer http.ResponseWriter, request *http.Request) {

    template_html.ExecuteTemplate(writer, "Create", nil)

}
}
```

Insert function invokes `GetCustomers` method to get an array of customers and renders Home template with `writer` and `customers` array as parameters by invoking `ExecuteTemplate` method.

```
// Insert - execute template
func Insert(writer http.ResponseWriter, request *http.Request) {
```

```
    var customer Customer
    customer.CustomerName = request.FormValue("customername")
    customer.SSN = request.FormValue("ssn")
    InsertCustomer(customer)
    var customers []Customer
    customers = GetCustomers()
    template_html.ExecuteTemplate(writer, "Home", customers)
}
```

Alter function renders Home template by invoking ExecuteTemplate method with writer and customers array as parameters.

```
// Alter - execute template
func Alter(writer http.ResponseWriter, request *http.Request) {

    var customer Customer
    var customerId int
    var customerIdStr string
    customerIdStr = request.FormValue("id")
    fmt.Sscanf(customerIdStr, "%d", &customerId)
    customer.CustomerId = customerId
    customer.CustomerName = request.FormValue("customername")
    customer.SSN = request.FormValue("ssn")
    UpdateCustomer(customer)
    var customers []Customer
    customers = GetCustomers()
    template_html.ExecuteTemplate(writer, "Home", customers)

}
```

Update function invokes the ExecuteTemplate method with writer and customer looked up by Id. ExecuteTemplate method renders Update Template.

```
// Update - execute template
func Update(writer http.ResponseWriter, request *http.Request) {

    var customerId int
    var customerIdStr string
    customerIdStr = request.FormValue("id")
    fmt.Sscanf(customerIdStr, "%d", &customerId)
    var customer Customer
    customer = GetCustomerById(customerId)

    template_html.ExecuteTemplate(writer, "Update", customer)

}
```

Delete method renders the Home template after deleting the customer found by GetCustomerById method. View method renders the View template after finding the customer by invoking GetCustomerById metho

```
// Delete - execute Template
func Delete(writer http.ResponseWriter, request *http.Request) {
    var customerId int
    var customerIdStr string
    customerIdStr = request.FormValue("id")
    fmt.Sscanf(customerIdStr, "%d", &customerId)
    var customer Customer
    customer = GetCustomerById(customerId)
    DeleteCustomer(customer)
    var customers []Customer
    customers = GetCustomers()
    template_html.ExecuteTemplate(writer, "Home", customers)
}

// View - execute Template
func View(writer http.ResponseWriter, request *http.Request) {
    var customerId int
    var customerIdStr string
    customerIdStr = request.FormValue("id")
    fmt.Sscanf(customerIdStr, "%d", &customerId)
    var customer Customer
    customer = GetCustomerById(customerId)
    fmt.Println(customer)
    var customers []Customer
    customers= []Customer{customer}
    // customers.append(customer)
    template_html.ExecuteTemplate(writer, "View", customers)
}
}
```

Main method handles Home, Alter, Create,Update, View, Insert and Delete functions with different aliases for lookup and rendering the templates appropriately. HttpServer listens to the port 8000 and waits for template alias invocation.

```
// main method
func main() {
    log.Println("Server started on: http://localhost:8000")
    http.HandleFunc("/", Home)
    http.HandleFunc("/alter", Alter)
    http.HandleFunc("/create", Create)
    http.HandleFunc("/update", Update)
    http.HandleFunc("/view", View)
    http.HandleFunc("/insert", Insert)
    http.HandleFunc("/delete", Delete)
}
```

```
    http.ListenAndServe(":8000", nil)
}
```

The header template has the html head and body defined in the code snippet below. The title of the web page is set to CRM and the web page has Customer Management - CRM as content.

Header.tmpl

```
{{ define "Header" }}
<!DOCTYPE html>
<html>
  <head>
    <title>CRM</title>
    <meta charset="UTF-8" />
  </head>
  <body>
    <h1>Customer Management- CRM</h1>
  </body>
</html>
{{ end }}
```

The footer template has the html and body close tags defined. Footer template is presented in the code snippet below:

Footer.tmpl

```
{{ define "Footer" }}
  </body>
</html>
{{ end }}
```

The menu template has the links defined for Home and Create Customer as shown in the code below:

Menu.tmpl

```
{{ define "Menu" }}
<a href="/">Home</a> | <a href="/create">Create Customer</a>
{{ end }}
```

The Create template consists of Header, Menu and Footer templates. The form to create customer fields is found in the create template. This form is submitted to a web path - Insert, as show in the code snippet below:

Create.tmpl

```
{{ define "Create" }}
  {{ template "Header" }}
  <form>
```

```

    {{ template "Menu" }}
    <br>
    <h1>Create Customer</h1>
    <br>
    <br>
    <form method="post" action="/insert">
        Customer Name: <input type="text" name="customername"
placeholder="customername" autofocus/>
        <br>
        <br>
        SSN: <input type="text" name="ssn" placeholder="ssn"/>
        <br>
        <br>
        <input type="submit" value="Create Customer"/>
    </form>
    {{ template "Footer" }}
    {{ end }}

```

The Update template consists of Header, Menu, and Footer templates, as shown below. The form to update customer fields is found in the update template. This form is submitted to a web path - Alter:

Update.tmpl

```

    {{ define "Update" }}
        {{ template "Header" }}
        {{ template "Menu" }}
    <br>
    <h1>Update Customer</h1>
    <br>
    <br>
    <form method="post" action="/alter">
        <input type="hidden" name="id" value="{{ .CustomerId }}" />
        Customer Name: <input type="text" name="customername"
placeholder="customername" value="{{ .CustomerName }}" autofocus>
        <br>
        <br>
        SSN: <input type="text" name="ssn" value="{{ .SSN }}"
placeholder="ssn"/>
        <br>
        <br>
        <input type="submit" value="Update Customer"/>
    </form>
    {{ template "Footer" }}
    {{ end }}

```

The View template consists of Header, Menu, and Footer templates. The form to view customer fields is found in the View template presented in the code below:

View.tmpl

```
{{ define "View" }}
  {{ template "Header" }}
  {{ template "Menu" }}
  <br>
  <h1>View Customer</h1>
  <br>
  <br>
<table border="1">
<tr>
<td>CustomerId</td>
<td>CustomerName</td>
<td>SSN</td>
<td>Update</td>
<td>Delete</td>
</tr>
{{ if . }}
  {{ range . }}
<tr>
<td>{{ .CustomerId }}</td>
<td>{{ .CustomerName }}</td>
<td>{{ .SSN }}</td>
<td><a href="/delete?id={{.CustomerId}}" onclick="return confirm('Are you
sure you want to delete?');">Delete</a> </td>
<td><a href="/update?id={{.CustomerId}}">Update</a> </td>
</tr>
{{ end }}
  {{ end }}
</table>
{{ template "Footer" }}
{{ end }}
```

Run the following commands:

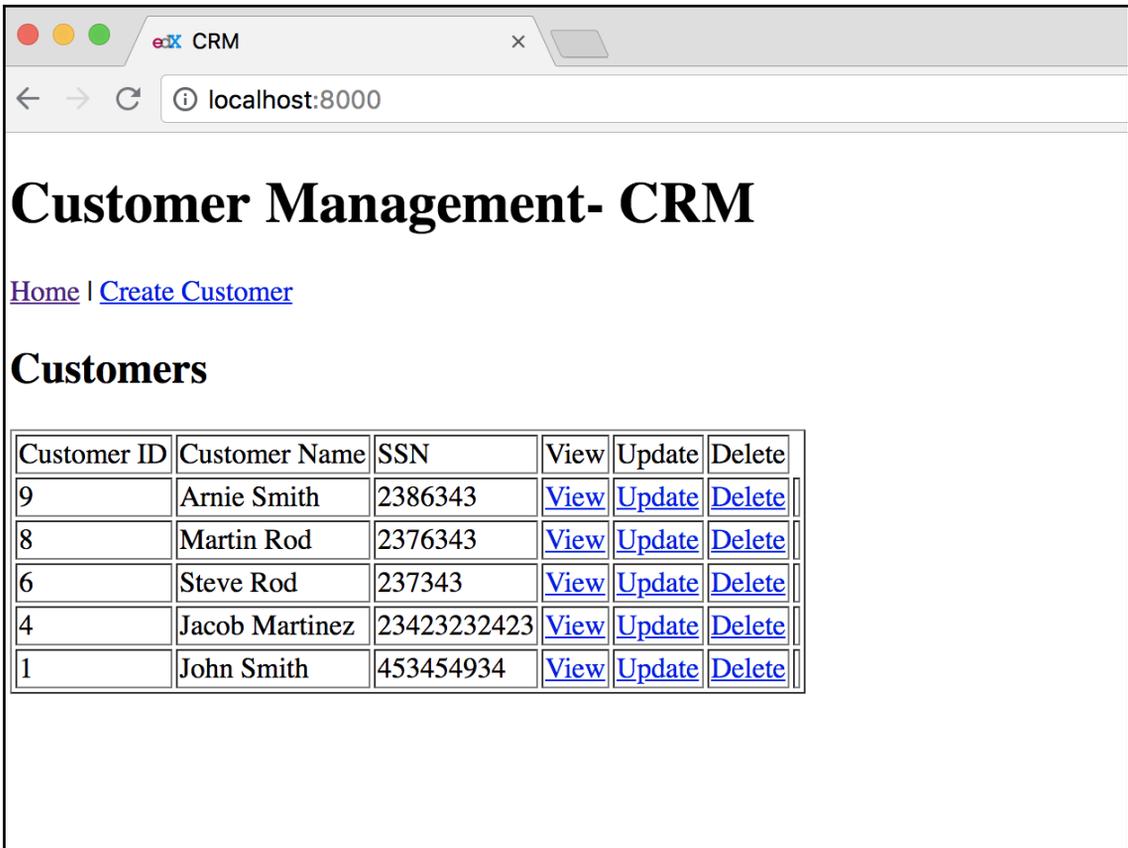
```
go run crmapp.go crm_database_operations.go
```

The screenshot of the output is attached below:



```
chapter2 — crm_app ◀ go run crm_app.go crm_database_operations.go — 94x24
Bhagvans-MacBook-Pro:chapter2 bhagvankommadi$ go run crm_app.go crm_database_operations.go
2018/08/24 17:38:14 Server started on: http://localhost:8000
```

The web browser output will look as follows:



Customer Management- CRM

[Home](#) | [Create Customer](#)

Customers

Customer ID	Customer Name	SSN	View	Update	Delete
9	Arnie Smith	2386343	View	Update	Delete
8	Martin Rod	2376343	View	Update	Delete
6	Steve Rod	237343	View	Update	Delete
4	Jacob Martinez	23423232423	View	Update	Delete
1	John Smith	453454934	View	Update	Delete

Summary

This chapter introduced database operations and Web Forms. The reader will be able to build the web applications which can store the data in databases. Arrays, Slices, two dimensional slices and maps were covered with code samples. Array methods such as len, iterating through array using for, and range were explained in this chapter using code snippets. Two dimensional arrays and slice of slices were discussed in the slices section.

Maps are explained with various scenarios such as adding keys and values as well as retrieving and deleting values. Maps of different types such as string and integer were also discussed in this chapter. Furthermore, variadic functions, deferred function calls, and panic and recover operations are demonstrated in the sections of database operations and web forms.

The CRM application was built as a web application with data persisted in the MySQL database. Database operations for adding, deleting, updating, and retrieving data are shown in code snippets. In addition, WebForms for Creation, Updating, Deleting and Viewing Customer Data is presented using webforms with templates. My sql driver and installation details were provided in the technical requirements section in this chapter. How to create a web application using go is demonstrated with execution details.

The next chapter will have the topics related to linear data structures such as Lists, Sets, Tuples, and Stacks.

Q&A

1. What is the name of the method to get the size of an array?
2. How do you find the capacity of the slice?
3. How do you initialise the 2D slice of type string?
4. How do you add an element to the slice?
5. In the code, can you demonstrate how to create a map of key strings and value strings? Initialize the map with keys and values in the code, iterate them in a loop, and print the keys and values in the code.
6. How do you delete a value in a map?
7. What are the parameters required for getting a database connection?
8. Which sql.Rows class method makes it possible to read the attributes of the entity in a table?
9. What does defer do when a database connection is closed ?
10. Which method allows the sql.DB class to create a prepared statement?

Reference

To read more about Arrays, Maps and Slices, the following books are recommended:

1. Learning Go DataStructures & Algorithms [Video]
2. Mastering Go

Index