Networking

- A network is a communications system for connecting end systems called hosts.
- The mechanisms of connection might be copper wire, ethernet, fiber optic or wireless, but that won't concern us here.
- A local area network (LAN) connects computers that are close together, typically belonging to a home, smaA Wide Area Network (WAN) connects computers across a larger physical area, such as between cities.

Networking

- There are other types as well, such as MANs (Metropolitan Area Network), PANs (Personal Are Networks) and even BANs (Body Are Network).
- An internet is a connection of two or more distinct networks, typically LANs or WANs.
- An intranet is an internet with all networks belonging to a single organization.
- There are significant differences between an internet and an intranet.

Networking

- Typically an intranet will be under a single administrative control, which will impose a single set of coherent policies.
- An internet on the other hand will not be under the control of a single body, and the controls exercised over different parts may not even be compatible.

Networking

- A trivial example of such differences is that an intranet will often be restricted to computers by a small number of vendors running a standardized version of a particular operating system.
- On the other hand, an internet will often have a smorgasbord of different computers and operating systems.
- The techniques of this book will be applicable to internets.
- They will also be valid for intranets, but there you will also find specialized, non-portable systems.

Networking

- And then there is the "mother" of all internets: The Internet.
- This is just a very, very large internet that connects us to Google, my computer to your computer and so on.

Request processing

- Parsing both url and body request data in Golang is very easy but also very tricky if things aren’t done in the right order.
- Always set content type to application/x-www-form-urlencoded if-and-only-if you want to process request body as form-encoded data.
- However, note that once your HTML form’s method is set to “POST” or “PUT”, this will be done for you automatically.

Request processing

- Always invoke ParseForm or ParseMultipartForm on the request object in your handler before attempting to read url or body data from the request object.
- Note : call ParseMultipartForm if your form supports file upload else, ParseForm will do just fine.

Generating responses

Plain text response:

- Creating a web server in Go is very simple and we can do it by writing just a few lines of code.
- We need to use `net/http` package to create an HTTP server. This is how a simple HTTP server code looks like in Go.

Execute | Share main.go STDIN

```go
package main
import (
        "fmt"
        "net/http"
    )
func main() {
        http.HandleFunc("/", handler)
        http.ListenAndServe(":8081", nil)
}
func handler(w http.ResponseWriter, r *http.Request) {

    fmt.Fprintf(w, "Hello World!")
}
```

Generating responses

- Once we run the file with command `go run server.go`, we will have a web server listening on port `8081`.
- Open a browser and access `http://localhost:8081/` you will get plain text response as `Hello World!`.
- Above web server is very simple and it will respond with `Hello World!` no matter what path you type after `/`.

JSON Response

- Most of the web services use JSON to communicate with clients and one web service can be used for different clients like Web Application, Android Application or iOS Application.
- Now, we will see how to return a JSON response in Go.At first, I will create `User` struct which will store user information.
- We will use the same struct to return user information as JSON response.
- In your application, you might want to get data from some database or file and return it as a `JSON` response.

JSON Response

Execute | Share main.go STDIN

```go
type User struct {
    Id    int    `json:"id"`
    Name  string `json:"name"`
    Email string `json:"email"`
    Phone string `json:"phone"`
}
```

JSON Response

- Let’s create a handler name `jsonHandler` which will handle `/json` and return user information as JSON response.
- To send JSON response, we need to encode the user information using JSON encoder and set the response header `Content-Type` to `application/json` while sending the response. This is how `jsonHandler` code should look like.

JSON Response

Execute | Share main.go STDIN

```go
func jsonHandler(w http.ResponseWriter, r *http.Request) {

    w.Header().Set("Content-Type", "application/json")
    user := User {
            Id: 1,
            Name: "John Doe",
            Email: "johndoe@gmail.com",
            Phone: "000099999"
        }

    json.NewEncoder(w).Encode(user)
}
```

JSON Response

- Register the jsonHandler, run the file with go run server.go and access http://localhost:8081/json in browser, you should get a JSON response as below.

Execute | Share main.go STDIN

```
1 {
2   "id": 1,
3   "name": "John Doe",
4   "email": "johndoe@gmail.com",
5   "phone": "000099999"
6 }
```

HTML or Template as Response

- We will add a new handler `templateHandler` which will handle `/template` and respond with HTML file (template) as a response.
- Let’s create a template file as `template.html` which will display user information and will be used by `templateHandler`.

HTML or Template as Response

Execute | Share main.go STDIN

```html
<html>
    <head>
    </head>
    <body>
      <h3>Name : {{.Name}}   </h3>
      <h3>Email : {{.Email}} </h3>
      <h3>Phone : {{.Phone}} </h3>
    </body>
</html>
```

HTML or Template as Response

- New handler will use `html/template` package to parse and execute template files.
- Also, we need to set `Content-Type` header to `text/html; charset=utf-8` otherwise template will be returned as plain text.

HTML or Template as Response

Execute | Share main.go STDIN

```go
func templateHandler(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "text/html; charset=utf-8")

        t, err := template.ParseFiles("template.html")
        if err != nil {
                fmt.Fprintf(w, "Unable to load template")
          }

      user := User{
                    Id: 1,
                    Name: "John Doe",
                    Email: "johndoe@gmail.com",
                    Phone: "000099999"
                  }

        t.Execute(w, user)
}
```

Working with JSON

- JSON is a widely used format for data interchange.
- Golang provides multiple encoding and decoding APIs to work with JSON including to and from built-in and custom data types using the encoding/json package.

Data Types:

- The default Golang data types for decoding and encoding JSON are as follows:

Working with JSON

- bool for JSON booleans
- float64 for JSON numbers
- string for JSON strings
- nil for JSON null
- array as JSON array
- map or struct as JSON Object

Encoding/Marshaling structs:

- The Marshal() function in package encoding/json is used to encode the data into JSON.
- **Syntax:** func Marshal(v interface{}) ([]byte, error)

Execute | Share main.go STDIN

```go
// Golang program to illustrate the
// concept of encoding using JSON
package main

import (
    "fmt"
    "encoding/json"
)

// declaring a struct
type Human struct{

    // defining struct variables
    Name string
    Age int
    Address string
}

// main function
func main() {

    // defining a struct instance
    human1 := Human{"Ankit", 23, "New Delhi"}

```

```go
    // encoding human1 struct
    // into json format
    human_enc, err := json.Marshal(human1)

    if err != nil {

        // if error is not nil
        // print error
        fmt.Println(err)
    }

    // as human_enc is in a byte array
    // format, it needs to be
    // converted into a string
    fmt.Println(string(human_enc))

    // converting slices from
    // golang to JSON format

    // defining an array
    // of struct instance
    human2 := []Human{
        {Name: "Rahul", Age: 23, Address: "New Delhi"},
        {Name: "Priyanshi", Age: 20, Address: "Pune"},
        {Name: "Shivam", Age: 24, Address: "Bangalore"},
    }
```

```go
    }

    // encoding into JSON format
    human2_enc, err := json.Marshal(human2)

        if err != nil {

        // if error is not nil
        // print error
            fmt.Println(err)
        }

    // printing encoded array
    fmt.Println()
        fmt.Println(string(human2_enc))
}
```

Decoding/Unmarshaling structs:

- The Unmarshal() function in package encoding/json is used to unpack or decode the data from JSON to struct.

Syntax: func Unmarshal(data []byte, v interface{}) error

Execute | Share main.go STDIN

```go
// Golang program to illustrate the
// concept of decoding using JSON
package main

import (
    "fmt"
    "encoding/json"
)

// declaring a struct
type Human struct{

    // defining struct variables
    Name string
    Address string
    Age int
}

// main function
func main() {

    // defining a struct instance
    var human1 Human

```

```go
    // data in JSON format which
    // is to be decoded
    Data := []byte(`{
        "Name": "Deeksha",
        "Address": "Hyderabad",
        "Age": 21
    }`)

    // decoding human1 struct
    // from json format
    err := json.Unmarshal(Data, &human1)

    if err != nil {

        // if error is not nil
        // print error
            fmt.Println(err)
    }

    // printing details of
    // decoded data
    fmt.Println("Struct is:", human1)
    fmt.Printf("%s lives in %s.\n", human1.Name, human1.Address)
```

```go

    // unmarshaling a JSON array
    // to array type in Golang

    // defining an array instance
    // of struct type
    var human2 []Human

    // JSON array to be decoded
    // to an array
    Data2 := []byte(`
    [
        {"Name": "Vani", "Address": "Delhi", "Age": 21},
        {"Name": "Rashi", "Address": "Noida", "Age": 24},
        {"Name": "Rohit", "Address": "Pune", "Age": 25}
    ]`)

    // decoding JSON array to
    // human2 array
    err2 := json.Unmarshal(Data2, &human2)

        if err2 != nil {

        // if error is not nil
        // print error
            fmt.Println(err2)
        }

    // printing decoded array
    // values one by one
    for i := range human2{

        fmt.Println(human2[i])
    }
}
```

Routing requests

- Web Development is all about multiple routing, so we have to define more than one URL route in a Go Web Application using Golang net/http HandleFunc to enable multiple Request Routing.
- This also includes mapping of the path to the respective Handlers and resources.
- In this example, we are going to make three endpoints, such as '/', '/about', '/services' along with their handlers.

Routing requests

- If you don’t know how to create a simple HTTP Server in Golang must Read to understand it better.

Execute | Share main.go STDIN

```go
package main

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

const (
    // Host name of the HTTP Server
    Host = "localhost"
    // Port of the HTTP Server
    Port = "8080"
)

func home(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "HOME Page")
}

func about(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "ABOUT Page")
}
```

```go
}

func services(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "SERVICES Page")
}

func main() {
    http.HandleFunc("/", home)
    http.HandleFunc("/about", about)
    http.HandleFunc("/services", services)
    err := http.ListenAndServe(Host+":"+Port, nil)
    if err != nil {
        log.Fatal("Error Starting the HTTP Server : ", err)
        return
    }

}
```

Run the program using:

```
$  go run http-request-routing.go
```

Output:

- Open the browser and go to http://localhost:8080, then add /about to the URL, and after all, go to /services URL.
- The browser will render the messages defined in the respective handlers.
- This is the extension of the previous blog where we learned about the creation of a simple **HTTP Server in Golang**.
- Previously we were had one handler i.e '/' but in this blog, we have multiple endpoints.