

# 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 Area Networks) and even BANs (Body Area 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

```
1 package main
2 import (
3     "fmt"
4     "net/http"
5 )
6 func main() {
7     http.HandleFunc("/", handler)
8     http.ListenAndServe(":8081", nil)
9 }
10 func handler(w http.ResponseWriter, r *http.Request) {
11     fmt.Fprintf(w, "Hello World!")
12 }
```

# 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 `/`.

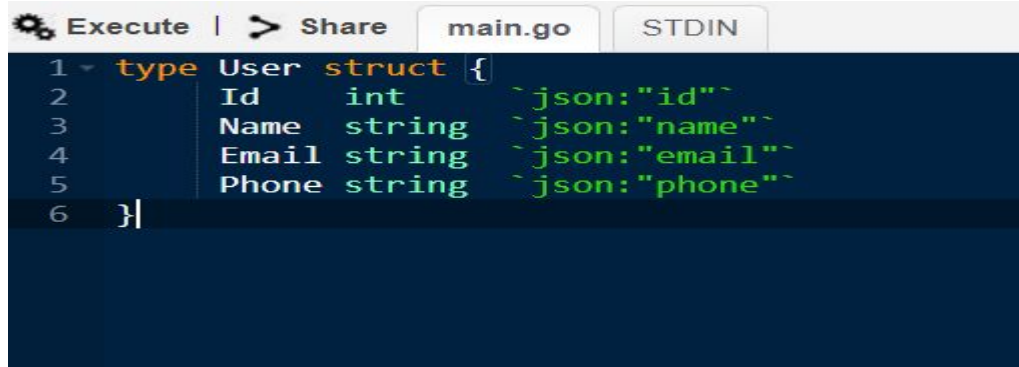
# JSON Response

---

- 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

---



The screenshot shows a Go Playground interface with a dark blue background. At the top, there are tabs for 'Execute' (with a gear icon), 'Share' (with a share icon), 'main.go', and 'STDIN'. Below the tabs, a Go struct definition is written in a light green monospace font. The code is as follows:

```
1 type User struct {  
2     Id      int    `json:"id"`  
3     Name   string `json:"name"`  
4     Email  string `json:"email"`  
5     Phone  string `json:"phone"`  
6 }
```

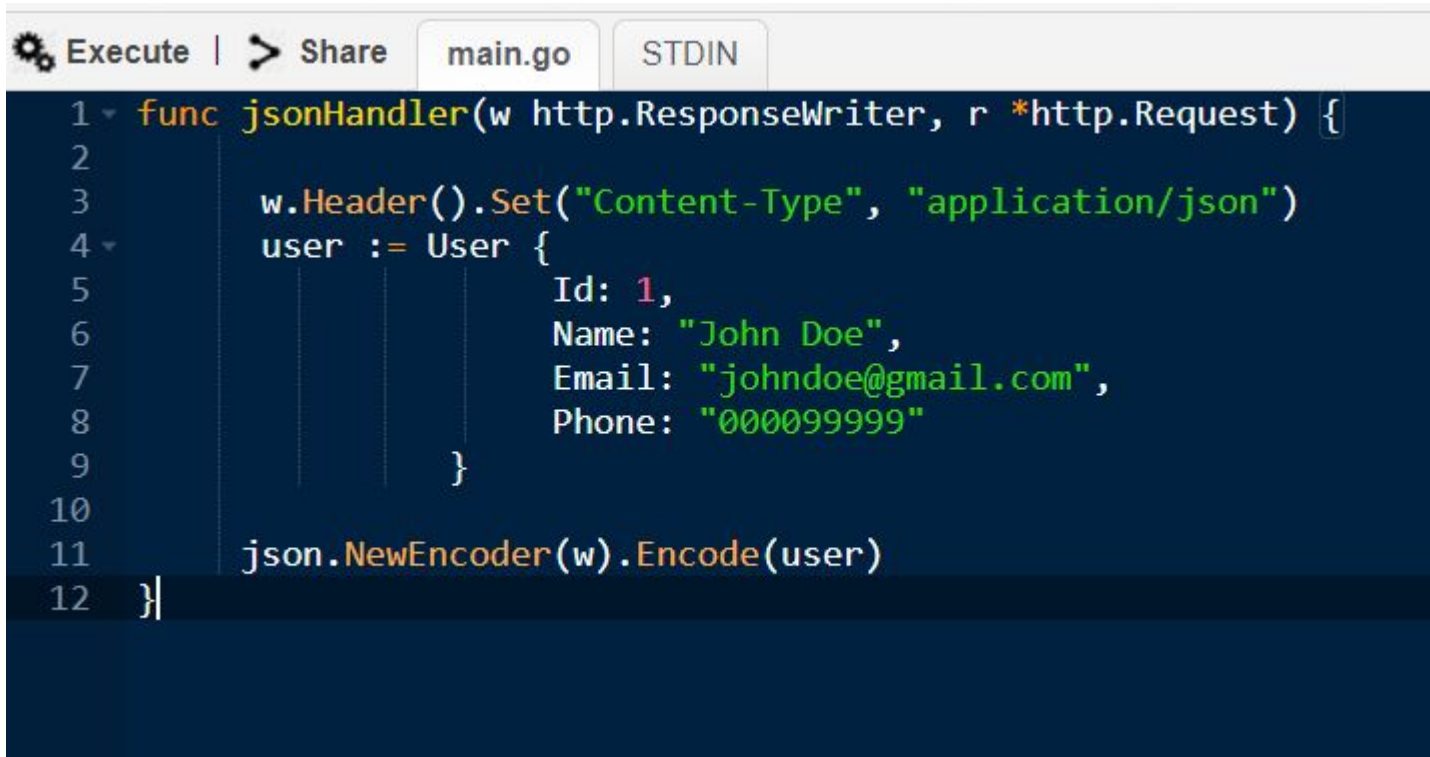
# JSON Response

---

- 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

---



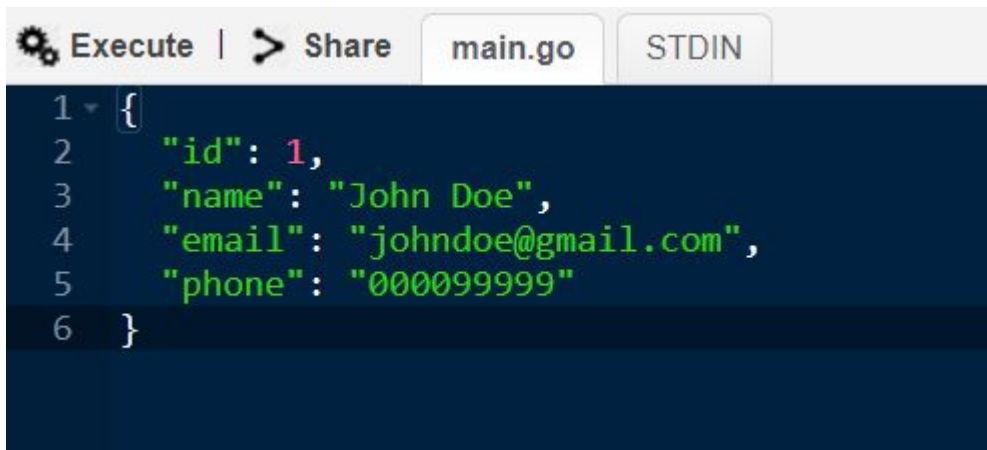
The image shows a code editor window with a dark blue background. At the top, there is a toolbar with a gear icon, the text 'Execute', a share icon, the text 'Share', and two tabs labeled 'main.go' and 'STDIN'. Below the toolbar, the code is written in Go. It defines a function 'jsonHandler' that takes an 'http.ResponseWriter' and an '\*http.Request' as arguments. The function sets the 'Content-Type' header to 'application/json', creates a 'User' struct with fields 'Id' (1), 'Name' ('John Doe'), 'Email' ('johndoe@gmail.com'), and 'Phone' ('000099999'), and then encodes this struct into JSON using 'json.NewEncoder(w).Encode(user)'.

```
1 func jsonHandler(w http.ResponseWriter, r *http.Request) {  
2  
3     w.Header().Set("Content-Type", "application/json")  
4     user := User {  
5         Id: 1,  
6         Name: "John Doe",  
7         Email: "johndoe@gmail.com",  
8         Phone: "000099999"  
9     }  
10  
11     json.NewEncoder(w).Encode(user)  
12 }
```

# 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.



The screenshot shows a code editor interface with a dark blue background. At the top, there are tabs for 'main.go' and 'STDIN'. To the left of the tabs are buttons for 'Execute' (with a gear icon) and 'Share' (with a share icon). The code is a JSON object displayed on a line-by-line basis:

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

# HTML or Template as Response

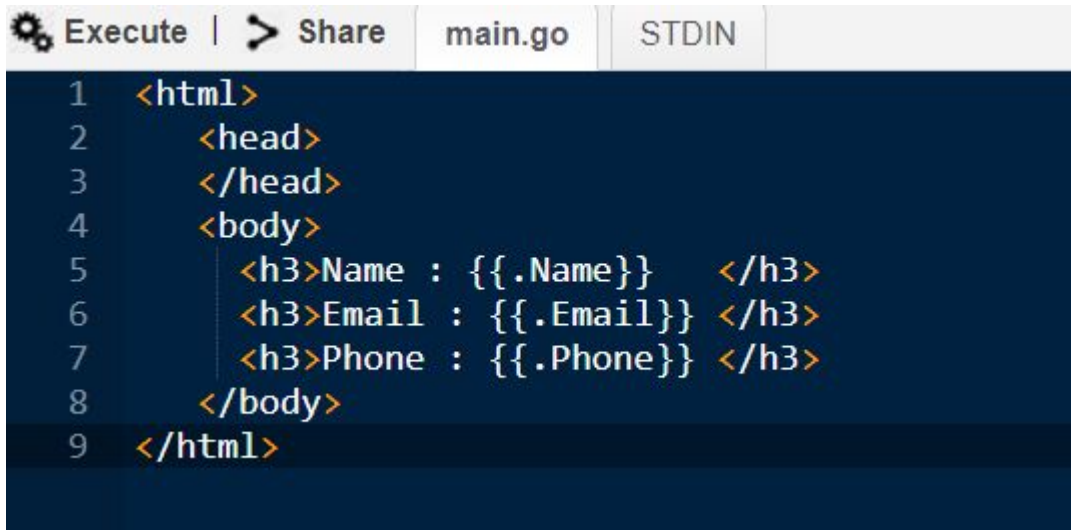
---

- 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

---



The image shows a code editor interface with a dark blue background. At the top, there is a toolbar with a gear icon, the text 'Execute', a share icon, the text 'Share', a file tab labeled 'main.go', and a tab labeled 'STDIN'. Below the toolbar, the code is displayed in a monospaced font with syntax highlighting. The code is an HTML document template using Go's templating syntax. It consists of a root <html> tag containing a <head> section and a <body> section. The <body> section contains three <h3> tags, each displaying a field from a Go struct: Name, Email, and Phone. The fields are accessed using the {{.Field}} syntax. The code is as follows:

```
1 <html>
2   <head>
3   </head>
4   <body>
5     <h3>Name : {{.Name}} </h3>
6     <h3>Email : {{.Email}} </h3>
7     <h3>Phone : {{.Phone}} </h3>
8   </body>
9 </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
1 func templateHandler(w http.ResponseWriter, r *http.Request) {
2     w.Header().Set("Content-Type", "text/html; charset=utf-8")
3
4     t, err := template.ParseFiles("template.html")
5     if err != nil {
6         fmt.Fprintf(w, "Unable to load template")
7     }
8
9     user := User{
10         Id: 1,
11         Name: "John Doe",
12         Email: "johndoe@gmail.com",
13         Phone: "000099999"
14     }
15
16     t.Execute(w, user)
17 }
```

# 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](#)

```
1 // Golang program to illustrate the
2 // concept of encoding using JSON
3 package main
4
5 import (
6     "fmt"
7     "encoding/json"
8 )
9
10 // declaring a struct
11 type Human struct{
12     // defining struct variables
13     Name string
14     Age int
15     Address string
16 }
17
18
19 // main function
20 func main() {
21     // defining a struct instance
22     human1 := Human{"Ankit", 23, "New Delhi"}
```

```
24
25 // encoding human1 struct
26 // into json format
27 human_enc, err := json.Marshal(human1)
28
29 if err != nil {
30     // if error is not nil
31     // print error
32     fmt.Println(err)
33 }
34
35
36 // as human_enc is in a byte array
37 // format, it needs to be
38 // converted into a string
39 fmt.Println(string(human_enc))
40
41 // converting slices from
42 // goLang to JSON format
43
44 // defining an array
45 // of struct instance
46 human2 := []Human{
47     {Name: "Rahul", Age: 23, Address: "New Delhi"},
48     {Name: "Priyanshi", Age: 20, Address: "Pune"},
49     {Name: "Shivam", Age: 24, Address: "Bangalore"},
50 }
```



```
50 }
51
52 // encoding into JSON format
53 human2_enc, err := json.Marshal(human2)
54
55 if err != nil {
56
57     // if error is not nil
58     // print error
59     fmt.Println(err)
60 }
61
62 // printing encoded array
63 fmt.Println()
64 fmt.Println(string(human2_enc))
65 }
66
```

# 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`

```
1 // Golang program to illustrate the
2 // concept of decoding using JSON
3 package main
4
5 import (
6     "fmt"
7     "encoding/json"
8 )
9
10 // declaring a struct
11 type Human struct{
12     // defining struct variables
13     Name string
14     Address string
15     Age int
16 }
17
18 // main function
19 func main() {
20     // defining a struct instance
21     var human1 Human
22
23     // creating a JSON string
24     jsonString := `{"Name": "John", "Address": "New York", "Age": 30}`
25
26     // parsing the JSON string into a struct
27     err := json.Unmarshal([]byte(jsonString), &human1)
28     if err != nil {
29         fmt.Println("Error in decoding JSON:", err)
30     }
31
32     // printing the struct variables
33     fmt.Println("Name: ", human1.Name)
34     fmt.Println("Address: ", human1.Address)
35     fmt.Println("Age: ", human1.Age)
36 }
```

```
25 // data in JSON format which
26 // is to be decoded
27 Data := []byte(`{
28     "Name": "Deeksha",
29     "Address": "Hyderabad",
30     "Age": 21
31 }`)
32
33 // decoding human1 struct
34 // from json format
35 err := json.Unmarshal(Data, &human1)
36
37 if err != nil {
38     // if error is not nil
39     // print error
40     fmt.Println(err)
41 }
42
43
44 // printing details of
45 // decoded data
46 fmt.Println("Struct is:", human1)
47 fmt.Printf("%s lives in %s.\n", human1.Name, human1.Address)
```

```

48
49 // unmarshaling a JSON array
50 // to array type in Golang
51
52 // defining an array instance
53 // of struct type
54 var human2 []Human
55
56 // JSON array to be decoded
57 // to an array
58 Data2 := []byte(`
59 [
60     {"Name": "Vani", "Address": "Delhi", "Age": 21},
61     {"Name": "Rashi", "Address": "Noida", "Age": 24},
62     {"Name": "Rohit", "Address": "Pune", "Age": 25}
63 ]`)
64
65 // decoding JSON array to
66 // human2 array
67 err2 := json.Unmarshal(Data2, &human2)
68
69 if err2 != nil {
70
71     // if error is not nil
72     // print error
73     fmt.Println(err2)
74 }
75
76 // printing decoded array
77 // values one by one
78 for i := range human2{
79
80     fmt.Println(human2[i])
81 }
82 }
83

```

# 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

```
1  package main
2
3  import (
4      "fmt"
5      "log"
6      "net/http"
7  )
8
9  const (
10     // Host name of the HTTP Server
11     Host = "localhost"
12     // Port of the HTTP Server
13     Port = "8080"
14 )
15
16 func home(w http.ResponseWriter, r *http.Request) {
17     fmt.Fprintf(w, "HOME Page")
18 }
19
20 func about(w http.ResponseWriter, r *http.Request) {
21     fmt.Fprintf(w, "ABOUT Page")
22 }
```



```
22 }
23
24 func services(w http.ResponseWriter, r *http.Request) {
25     fmt.Fprintf(w, "SERVICES Page")
26 }
27
28 func main() {
29     http.HandleFunc("/", home)
30     http.HandleFunc("/about", about)
31     http.HandleFunc("/services", services)
32     err := http.ListenAndServe(Host+": "+Port, nil)
33     if err != nil {
34         log.Fatal("Error Starting the HTTP Server : ", err)
35         return
36     }
37
38 }
```

---

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.