Golang stdlib - Webserver

> This is just some notes, nothing serious.

We need `"net/http"` module to create webserver and listen to the specified port.

```
import (
    "net/http"
)
```

we also might need these modules as well:

- `"encoding/json"` too for parsing JSON data
- `"database/sql"` and `"github.com/go-sql-driver/mysql"` to handle (my)sql/mariadb part
- `"github.com/redi/go-redis/v9"` to handle redis-cache

We need a request handler struct for the http server to be able to work.

```
type RequestHandler struct {}
```

And now we define the http server. it needs a `ServeHTTP()` function and paths that needs to be handled (HTTP requests goes to the specified paths)

main funcion

```
func main() {
    mux := http.NewServeMx()
    mux.Handle("/path1", &RequestHandler{})
    mux.Handle("/path1/", &RequestHandler{})
    mux.Handle("/path2", &RequestHandler{})
    mux.Handle("/path2/", &RequestHandler{})
    /* continue */
    http.ListenAndServe(":8080", mux)
}
```

serve function

```
func (h *RequestHandler) ServeHTTP(w http.ResponseWriter, r *http.Requset) {
    request_type := r.Method
    tmp := strings.SplitN(r.URL.String(), "/", 3)
    section := tmp[1] // path
    key := tmp[2] // add data after the path
    params := r.URL.Query()
    value := params.Get(key)
    body, err := io.ReadAll(r.Body) // values from curl's `--data` flag (spected to be JSON)
    /* continue */
}
```

1. Redis Handler

```
func cache_handler() {
    client := redis.NewClient(&redis.Options{
        Addr: "localhost:6349"
        Password: "" // No password, to get it from env variables use: `os.Getenv("ENV")`
        DB: 0 // default db
    })
}
```

2. MySQL Handler

```
func database_handler() {
    sql_url := "root@localhost:3360"
    db, err := sql.Open("mysql", sql_url)
    if err != nil {
        pandic(err.Error())
    }
    defer db.Close()
    /* continue */
}
```

3. JSON Parser - Unmarshal

```
type Movie struct {
    Name string `json:"name"`
    Publisher string `json:"publisher"`
    Year int `json:"year"`
}

func HandleMovie() {
    // var target map[string]any /* not optimal */
    var movie Movie

    input := `{
        "name": "Jocker",
        "publisher": "WB",
        "year": 2019
    }`

    // err := json.Unmarshal([]byte(input), &target) /* not optimal */
    err := json.Unmarshal([]byte(input), &movie) /* not optimal */
    if err != nil {
        log.Fatalf("Unable to marshal JSON due to %s", err)
    }

    /* not optimal */
    // for k, v := range target {
    //     fmt.Printf("k: %s, v: %v\n", k, v)
    // }
    fmt.Printf(
        "Name: %s, Publisher: %s, Year: %d\n",
        movie.Name, movie.Publisher, movie.Year,
    )
}
```

3.1. Complex JSON

assets/complex.json

```
{
  "name": "James Peterson",
  "age": 37,
  "address": {
    "line1": "Block 78 Woodgrove Avenue 5",
    "line2": "Unit #05-111",
    "postal": "654378"
  },
  "pets": [
    {
      "name": "Lex",
      "kind": "Dog",
      "age": 4,
      "color": "Gray"
    },
    {
      "name": "Faye",
      "kind": "Cat",
      "age": 6,
      "color": "Orange"
    }
  ]
}
```

examples/complex_json/main.go

```
type (
    FullPerson struct {
        Address Address
        Name    string
        Pets    []Pet
        Age     int
    }

    Pet struct {
        Name  string
        Kind  string
        Color string
        Age   int
    }

    Address struct {
        Line1  string
        Line2  string
        Postal string
    }
)


func main() {
    b, err := os.ReadFile("assets/complex.json")
    if err != nil {
        log.Fatalf("Unable to read file due to %s\n", err)
    }

    var person FullPerson

    err = json.Unmarshal(b, &person)
    if err != nil {
        log.Fatalf("Unable to marshal JSON due to %s", err)
    }

    litter.Dump(person)
}
```

3.2. Common pitfalls with JSON unmarshalling in Go

1. Extra fields are omitted in the target struct
2. Missing fields fallback to zero values
3. Unmarshalling is case insensitive

4. Field names must match JSON keys exactly
5. Type aliases are preserved

4. JSON Parser - Marshal

The `json.Marshal()` method does the opposite of `Unmarshal()` by converting a given data structure into a JSON.

examples/basic_marshal/main.go

```go
func marshal(in any) []byte {
    out, err := json.Marshal(in)

    if err != nil {
        log.Fatalf("Unable to marshal due to %s\n", err)
    }

    return out
}

func main() {
    first := marshal(14)
    second := marshal("Hello world")
    third := marshal([]float32{1.66, 6.86, 10.1})
    fourth := marshal(map[string]int{"num": 15, "other": 17})
    fmt.Printf(
        "first: %s\nsecond: %s\nthird: %s\nfourth: %s\n",
        first,
        second,
        third,
        fourth,
    )
}
```

4.1. structs

```go
func main() {
    p := Person{
        Name:  "John Jones",
        Age:   26,
        Email: "johnjones@email.com",
        Phone: "89910119",
        Hobbies: []string{
            "Swimming",
            "Badminton",
        },
    }

    b, err := json.Marshal(p)
    if err != nil {
        log.Fatalf("Unable to marshal due to %s\n", err)
    }

    fmt.Println(string(b))
}
```

If you wish to format the JSON object, you can use the `MarshalIndent()` method which performs the same function as `Marshal()` but applies some indentation to format the output.

4.2. Customizing JSON field names with struct tags

```go
type Dog struct {
    Breed         string
    Name          string
    FavoriteTreat string
    Age           int
}

var dog = Dog{
  Breed: "Golden Retriever",
  Age: 8,
  Name: "Paws",
  FavoriteTreat: "Kibble",
}
```

```go
type Dog struct {
    Breed         string `json:"breed"`
    Name          string `json:"name"`
    FavoriteTreat string `json:"favorite_treat"`
    Age           int    `json:"age"`
}
```

```go
func main() {
    input := `{
  "name": "Coffee",
  "breed": "Toy Poodle",
  "age": 5,
  "favorite_treat": "Kibble"
}`

    var coffee Dog

    err := json.Unmarshal([]byte(input), &coffee)
    if err != nil {
        log.Fatalf("Unable to marshal JSON due to %s", err)
    }

    litter.Dump(coffee)
}
```

4.3. Other uses of struct tags

Omit an empty field (one with its zero value in Go)

```go
type User struct {
    Username string   `json:"username"`
    Password string   `json:"-"`

    Email    string   `json:"email"`
    Hobbies  []string `json:"hobbies,omitempty"`
}
```

5. Validating JSON data

```go
func main() {
    good := `{"name": "John Doe"}`
    bad := `{name: "John Doe"}`

    fmt.Println(json.Valid([]byte(good)))
    fmt.Println(json.Valid([]byte(bad)))
}
```

6. Defining custom behavior - Marshal / Unmarshal data

In Go, you can define custom behavior for marshalling data by implementing the `json.Marshaler` interface. This interface defines a single method, `MarshalJSON()` which takes no arguments and returns a byte slice and an error.

To implement the `json.Marshaler` interface, you need to define a new type that wraps the original type you want to marshal. This new type should have a method named `MarshalJSON()` that returns a byte slice and an error.

examples/custom_timestamp/main.go

```go
type (
    CustomTime struct {
        time.Time
    }

    Baby struct {
        BirthDate CustomTime `json:"birth_date"`
        Name      string     `json:"name"`
        Gender    string     `json:"gender"`
    }
)
```

In the above snippet, we defined a new `CustomTime` type that wraps a `time.Time` value. In is subsequently used in the `Baby` struct as the type of the `BirthDate` value.

Here's an example that marshals a value of type `Baby` below:

```go
func main() {
    baby := Baby{
        Name:   "johnny",
        Gender: "male",
        BirthDate: CustomTime{
            time.Date(2023, 1, 1, 12, 0, 0, 0, time.Now().Location()),
        },
    }

    b, err := json.Marshal(baby)
    if err != nil {
        log.Fatalf("Unable to marshal due to %s\n", err)
    }

    fmt.Println(string(b))
}
```

Notice how the `birth_date` presented in the RFC 3339 format. You can now define the custom marshalling behavior that will return a different format for `CustomTime` values (such as `DD-MM-YYYY`) instead of the default RFC 3339 timestamp format.

You only need to define a `MarshalJSON()` method for the type as shown below:

examples/custom_timestamp/main.go

```go
func (ct CustomTime) MarshalJSON() ([]byte, error) {
    return []byte(fmt.Sprintf(`%q`, ct.Time.Format("02-01-2006"))), nil
}
```