Backend Programming

M. Yauri M. Attamimi

Upgrade your potential !

Foundation Class

- WEEK 2 (Session 2) -

Package

1. Package Declaration
2. Init Function
3. Importing Package
4. Export and Import Types

Package Declaration

- Packages are used to organize source code for better readability and maintanability hence it can be easily reusabled.

- Every executable go application must contain a main function. This function is the entry point for execution. The main function should reside in the main package.

- The line of code to specify that a particular source file belongs to a particular package is: `package <package_name>` which should be written on the first line of every go source file.

Init Function

- Every package contains an init function. The init function should not have any return type and should not have any parameters.
- The init function cannot be explicitly called in our source code.
- Here what it looks like :

```
func init() {
}
```

- The init function can be used to perform initialization tasks and can also be used to verify the correctness of the program before the execution starts.
- The order of initialization of a package is as follows:
  1. Package level variables are initialized first
  2. init function is called next
- if a package imports other packages, the imported packages are initialized first.
- A package will be initialized only once even if it is imported from multiple packages.

Export and Import Types

- We can make any type to be publicly available (exported types) by capitalized their first letter. e.g.:

```
// exported function (can be accessed from outside the package / publicly available)
func CountTotalOrder() int

// un-exported function (cannot be accessed from outside the package / private)
func countTotalOrder() int
```

- Any variables or functions which starts with a capital letter are exported names in Go, hence it can be used (imported) from other packages (outside of the package where they were being defined).

Reading and Writing Files

Writing to File (I)

```
import (
    “io/ioutil”
    “fmt”
)

data := []byte(“Init data”)
err := ioutil.WriteFile(“localfile.data”, data, 0777)
if err != nil {
    fmt.Println(err)
}
```

Writing to File (II)

```
import (
    “io/ioutil”
    “os”
)

f, err := os.OpenFile(
    “localfile.data”, os.O_APPEND|o.OWRONLY, 0600)
if err != nil {
    fmt.Println(err)
}
defer f.Close()
if _, err = f.WriteString(“\nnew line\n”); err != nil {
    fmt.Println(err)
}
```

Reading from File

```
import (
    “io/ioutil”
    “fmt”
)

data, err := ioutil.ReadFile(“localfile.data”)
if err != nil {
    fmt.Println(err)
}

fmt.Println(string(data))
```

Permission File and File Mode

https://golang.org/pkg/os/#FileMode

Working with JSON

The JSON data-interchange format is easy for humans to read and write, and efficient for machines to parse and generate.

Go Basic Types for JSON

- bool for JSON booleans
- float64 for JSON numbers
- string for JSON strings
- nil for JSON null

Additionally `time.Time` and the numeric types in the `math/big` package can be automatically coded and encoded as JSON strings.

We can use `encoding/json` to easily create JSON from our data structure.

Struct to JSON

The `json.Marshal` function in package `encoding/json` generates JSON data.

```
type Customer struct {
    Name string
    Phone []string
    Id int64 `json:"ref"`
    private string // An unexported field is not encoded.
    Created time.Time

}

customer := Customer{
    Name: "Jackie",
    Phone: []string{"+6282225251437", "+602174368"},
    Id: 999,
    private: "First-class",
    Created: time.Now(),

}

var jsonData []byte
jsonData, err := json.Marshal(customer)
if err != nil {
    log.Println(err)

}

fmt.Println(string(jsonData))
```

Encoded Rules

- Only data than can be represented as JSON will be encoded
- Only the exported (public) fields of a struct will be presented in the JSON output
- A field with a json : tag is stored with its tag name instead of its variable name
- Pointers will be encoded as the values they point to, or null if the pointer is nil

JSON Pretty Print

Replace `json.Marshal` with `json.MarshalIndent`

```
jsonData, err := json.MarshalIndent(customer, "", "    ")
```

```
type Customer struct {
        Name        string `json:"name,omitempty"`
        Phone       []string `json:"phones,omitempty"`
        Id          int64 `json:"ref"` `json:"id"`
        private string // An unexported field is not encoded.
        Created time.Time `json:"created,omitempty"`
}
```

Skipping Fields

```
type Customer struct {
    Name   string `json:”name,omitempty”`
    Phone  []string `json:”phones,omitempty”`
    Id     int64 `json:"ref"` `json:”id”`
    private string // An unexported field is not encoded.
    Created time.Time `json:”-”` // not produced in JSON response
}
```

Maps to JSON

```
customer := map[string]interface{}{
    "name": "Jackie",
    "id": 999,
    "created": time.Now()
}

jsonBytes, _ := json.Marshal(customer)
fmt.Printf("%s", jsonBytes)
```

Slices to JSON

```
emails := []string{"yauritux@gmail.com", "yauri.attamimi@automate.id"}
json_bytes := json.Marshal(emails)
fmt.Printf("%s", json_bytes)
```

Struct to JSON

The `json.Marshal` function in package `encoding/json` parses JSON data.

```
type Customer struct {
    Name string `json:”name”`
    Phone    []string `json:”phones”`
    Id  int64 `json:"ref"`
    private string // An unexported field is not encoded.
    Created time.Time `json:”created”`

}

jsonData := []byte(`{
    “name”:  "Jackie",
    “phones”:    [“+6282225251437”, “+602137468”],
    “ref”: 999,
    “created”: "2019-02-05T23:00:00Z"

}`)

var customer Customer
err := json.Unmarshal(jsonData, &customer)
if err != nil {
    log.Println(err)
}
fmt.Println(customer.Name, customer.Phone, customer.Id)
```

Check the exercises at https://exercism.io

SUMMARY

You have learned the concept of package in Go, why do we need it and how to use it properly in order to make your code more maintainable, including various concepts related to package. You have learned how to wrap and present your data with JSON.

Thank You