Decoding with JSON

This lesson provides a detailed explanation on decoding data with JSON by providing coded examples.

WE'LL COVER THE FOLLOWING

Decoding arbitrary data

The `json` package uses `map[string]interface{}` and `[]interface{}` values to store arbitrary JSON objects and arrays; it will happily unmarshal any valid JSON blob into a plain interface{} value.

Consider this JSON data, stored in the variable `b`:

```
b := []byte(`{"Name": "Wednesday", "Age": 6, "Parents": ["Gomez", "Mortici
a"]}`)
```

Without knowing this data's structure, we can decode it into an interface{} value with Unmarshal:

```
var f interface{}
err := json.Unmarshal(b, &f)
```

At this point, the value in `f` would be a map, whose keys are strings and whose values are themselves stored as empty interface values:

```
map[string]interface{}{
  "Name": "Wednesday",
  "Age": 6,
  "Parents": []interface{}{
```

```
    "Parents": []interface{}{
      "Gomez",
      "Morticia",
    },
}
```

To access this data we can use a type assertion to access `f`'s underlying `map[string]interface{}`:

```
m := f.(map[string]interface{})
```

We can then iterate through the map with a range statement and use a type switch to access its values as their concrete types:

```
for k, v := range m {
  switch vv := v.(type) {
  case string:
    fmt.Println(k, "is string", vv)
  case int:
    fmt.Println(k, "is int", vv)
  case []interface{}:
    fmt.Println(k, "is an array:")
    for i, u := range vv {
      fmt.Println(i, u)
    }
  default:
    fmt.Println(k, "is of a type I don't know how to handle")
  }
}
```

In this way, you can work with unknown JSON data while still enjoying the benefits of type safety.

Decoding data into a struct

If we know beforehand the semantics of the json-data, we can then define an appropriate struct and unmarshal into it. For example, in the previous section, we would define:

```
type FamilyMember struct {
  Name string
  Age int
  Parents []string
}
```

and then do the unmarshaling with:

```
var m FamilyMember
err := json.Unmarshal(b, &m)
```

This allocates a new slice behind the scenes. This is typical of how unmarshal works with the supported reference types (pointers, slices, and maps).

Here is an example that shows how it works:

```
package main
import (
  "fmt"
  "encoding/json"
)

type Node struct {
  Left *Node
  Value interface{}
  Right *Node
}
func main() {
  b := []byte(`{"Value": "Father", "Left": {"Value": "Left child"}, "Right": {"Value": "Right child"}}`)
  var f Node
  json.Unmarshal(b, &f)
  fmt.Println(f, f.Left, f.Right)
}
```

JSON Tree

In the code above, we define a struct called `Node` at **line 7** that refers to a binary tree. It has *three* fields. `Left` and `Right` being pointers to other `Nodes`, while `Value` contains the data. `Value` can contain anything; that's why its type is `interface{}`.

At **line 13**, we define an array of bytes `b`, containing raw data for a map with keys `Value`, `Left` and `Right`.

We declare variable `f` to be of type `Node` at **line 15**, and at **line 16**, we decode `b` into the `f` structure. We see that the pointers are all memory addresses: real ones in the top nodes, and `nil` for the left and right nodes.

Working with JSON and structs

The `json` package also allows to encode and decode `json`, having fields names different from the ones in a struct, for example:

```
package main
import (
  "fmt"
  "encoding/json"
)

type Person struct {
  Name string `json:"personName"`
  Age int `json:"personAge"`
}

func main() {
  b := []byte(`{"personName": "Obama", "personAge": 57}`)
  var p Person
  // Unmarshalling
  json.Unmarshal(b, &p)
  fmt.Println(p)
  // Marshalling
  js, _ := json.Marshal(p)
  fmt.Printf("%s\n", js)
}
```

JSON and Structs

Here, we define a struct `Person` at **line 7** while we define alternative json names as a raw string enclosed in backticks, e.g. `Name` string `json:"personName"` means the field `Name` has as name `personName` in JSON.

At **line 13**, we make some JSON data in variable `b`, and at **line 14**, we make an instance `p` of struct `Person`. Then we decode `b` into `p` at **line 16** with `Unmarshal(b, &p)` and print the resulting struct out.

At **line 19**, we use `Marshal` to again make a JSON string `js` from the struct `p`, and we print it out, showing that the JSON field names are used:

Streaming encoders and decoders

The `json` package provides `Decoder` and `Encoder` types to support the common operation of reading and writing streams of JSON data. The `NewDecoder` and `NewEncoder` functions wrap the `io.Reader` and `io.Writer` interface types.

```
func NewDecoder(r io.Reader) *Decoder
func NewEncoder(w io.Writer) *Encoder
```

To write the json directly to a file, use `json.NewEncoder` on a file (or any other type which implements `io.Writer`) and call `Encode()` on the program data. The reverse is done by using a `json.Decoder` and the `Decode()` function:

```
func NewDecoder(r io.Reader) *Decoder
func (dec *Decoder) Decode(v interface{}) error
```

See how the use of interfaces generalizes the implementation. The data structures can be everything because they only have to implement interface{}. The targets or sources to/from which the data is encoded must implement `io.Writer` or `io.Reader`. Due to the ubiquity of readers and writers, these `Encoder` and `Decoder` types can be used in a broad range of scenarios, such as reading and writing to HTTP connections, web-sockets, or files.

Now that you are familiar with how encoding and decoding works with JSON, let's see how Go handles XML type data in the next lesson.