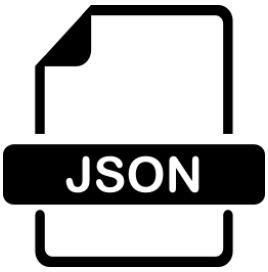
The JSON Data Format

This lesson discusses specifically the JSON type of data, and how Go reads and writes data in the JSON format efficiently.

WE'LL COVER THE FOLLOWING

Introduction

Structs can contain binary data; if this were printed as the text, it would not be readable for a human. Moreover, the data does not include the name of the struct field, so we don't know what the data means. To remedy this, several formats have been devised, which transform the data into plain text, but annotated by their field names, so humans can read and understand the data.



Data in these formats can be transmitted over a network. In this way, the data is platform-independent and can be used as input/output in all kinds of applications, no matter what the programming language or the operating system is. The following terms are common here:

- **data structure to special format string** = marshaling or encoding (before transmission at the sender or source)
- **special format string to data structure** = unmarshaling or decoding (after transmission at the receiver or destination)

Marshaling is used to convert in-memory data to the special format (data -> string), and vice versa for **unmarshaling** (string -> data structure).

Explanation

Encoding does the same thing, but the output is a stream of data (implementing io.Writer); decoding starts from a stream of data (implementing `io.Reader`) and populates a data structure. Well known is the XML-format (which we'll study later). Another popular format sometimes preferred for its simplicity is JSON (JavaScript Object Notation, see here). It is most commonly used for communication between web back-ends and JavaScript programs running in the browser, but it is used in many other places too. This is a short piece of JSON:

```
{
  "Person":
  { "FirstName": "Laura",
    "LastName": "Lynn"
  }
}
```

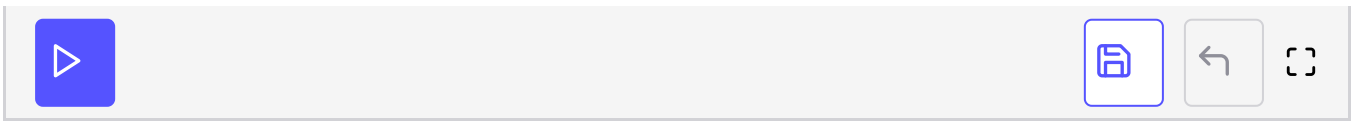
Though XML is widely used, JSON is less verbose (thus taking less memory space, disk space, and network bandwidth) and more readable. This explains why it is the de facto standard in most web applications for async communication.

The JSON package provides an easy way to read and write JSON data from your Go programs. We will use it in the following example:

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

type Address struct {
  Type string
  City string
  Country string
}
```

```
type VCard struct {
  FirstName string
  LastName string
  Addresses []*Address
  Remark string
}
func main() {
  pa := &Address{"private", "Aartselaar","Belgium"}
  wa := &Address{"work", "Boom", "Belgium"}
  vc := VCard{"Jan", "Kersschot", []*Address{pa,wa}, "none"}
  // fmt.Printf("%v: \n", vc) // {Jan Kersschot [0x126d2b80 0x126d2be0] none}:
  // JSON format:
  js, _ := json.Marshal(vc)
  fmt.Printf("JSON format: %s", js)
  // using an encoder:
  file, _ := os.OpenFile("output/vcard.json", os.O_CREATE|os.O_WRONLY, 0)
  defer file.Close()
  enc := json.NewEncoder(file)
  err := enc.Encode(vc)
  if err != nil {
    log.Println("Error in encoding json")
  }
}
```



Reading and Writing JSON

To use this functionality, we have to import the package `encoding/json` (see **line 4**). Two struct types are defined here:

- `Address` is defined at **line 9**. It contains *three* fields: `Type`, `City`, and `Country`.
- `VCard` is defined at **line 15**. It contains *four* fields: `FirstName`, `LastName`, `Addresses`, and `Remark`.

At **line 22** and **line 23**, we make two addresses `pa` and `wa`. At **line 24**, a `VCard` instance `vc` is made with the field `Addresses` containing the previously made `Address` instances.

At **line 27**, we call the `Marshal` function to make a *json* string `js` from `vc`. The second left-hand side argument is `_`, thereby indicating that we discard any errors from `Marshal()`. The JSON string `js` is printed at **line 28**.

At **line 30**, we open a file **vcard.json**, and close it at the end of the program with `defer` at **line 31**. At **line 32**, we construct a new *json* encoder for this file and call the encode method on it with the struct `vc` as a parameter. This makes a json string from `vc` and writes it into the file. The return value is `nil`

or an error, on which we test from **line 34** to **line 36**.

Marshal

The `json.Marshal()` function with signature `func Marshal(v interface{}) ([]byte, error)` encodes the data into the following json-text (in fact a []bytes):

```
{"FirstName":"Jan","LastName":"Kersschot","Addresses":[{"Type":"private",
"City":"Aartselaar","Country":"Belgium"},{"Type":"work","City":"Boo
m","Country":"Belgium"}],"Remark":"none"}
```

For security reasons, in web applications, it is better to use the `json.MarshalForHTML()` function, which performs an `HTMLEscape` on the data, so that the text will be safe to embed inside HTML `<script>` tags. The default Go types used in JSON are:

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

Not everything can be JSON-encoded though; only data structures that can be represented as valid JSON will be encoded:

- JSON objects only support strings as keys; to encode a Go map type, it must be of the form `map[string]T` (where `T` is any Go type supported by the `json` package).
- Channel, complex, and function types cannot be encoded.
- Cyclic data structures are not supported; they will cause `Marshal` to go into an infinite loop.
- Pointers will be encoded as the values they point to (or 'null' if the pointer is nil).

The `json` package only accesses the exported fields of struct types; only those will be present in the JSON output. This is necessary because `json` uses reflection on them.

UnMarshal

The `UnMarshal()` function with signature `func Unmarshal(data []byte, v interface{}) error` performs the decoding from JSON to program data-structures. First, we create a struct, for example, `Message`, where the decoded data will be stored in:

```
var m Message
```

and call `Unmarshal()`, passing it a []byte of JSON data `b` and a pointer to `m`:

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

Through reflection, it tries to match the JSON-fields with the destination struct fields; only the matched fields are filled with data. So no error occurs if there are fields that do not match; they are disregarded.

Now that you're familiar with the basics of JSON in Golang, let's see how encoding and decoding work with JSON.