The XML Data Format

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

WE'LL COVER THE FOLLOWING

What is XML?

XML is a markup language that sets some rules to encode data in both human and machine-readable format. It stands for e**X**tensible **M**arkup **L**anguage. The XML equivalent for the JSON example used in the last lesson is:


```
<Person>
<FirstName>Laura</FirstName>
<LastName>Lynn</LastName>
</Person>
```

Marshalling and unmarshaling

Like the `json` package `xml` contains a `Marshal()` and an `UnMarshal()` function to encode and decode data to and from XML. In the same way as with JSON, XML-data can be marshaled or un-marshaled to/from structs.

Here is an example where we see this in action:

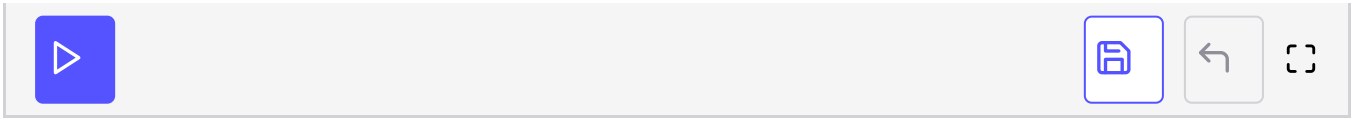
```
package main
import (
```

```
"encoding/xml"
"fmt"
)


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

func main() {

  b := []byte(`<Person><personName>Obama</personName><personAge>57</personAge></Person>`)
  var p Person
  // Unmarshalling
  xml.Unmarshal(b, &p)
  fmt.Println(p)
  // Marshalling
  xmlString, _ := xml.Marshal(p)
  fmt.Printf("%s\n", xmlString)
}
```



XML Tree

The XML functionality resides in the `encoding/xml` package which is imported at **line 3**. At **line** 7, we define a struct `Person`; notice that the field names have an alias XML name, which is the name that data will have in XML strings or files.

At **line 14**, we have XML data as a string `b`. At **line 15**, we declare a variable `p` of type `Person`. At **line 17**, we decode the XML data from `b` to `p` (referenced here as `&p`) with `Unmarshal`. At **line 20**, we do the reverse, encoding the struct `p` to an `xmlString` variable, which is printed out at **line 21**.

> **Remark:** Use `Unmarshal` to deserialize a short text you already have in memory (e.g., some user input) and `NewDecoder` when you are reading from a stream (e.g., from a file).

Parsing XML-data

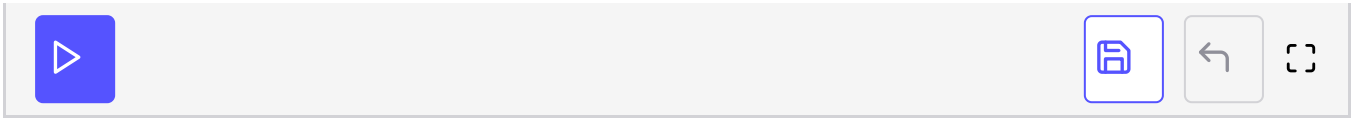
The `encoding/xml` package also implements a simple XML parser (SAX) to read XML-data and parse it into its constituents. The following code illustrates how this parser can be used:

```
package main
import (
        "fmt"
        "strings"
        "encoding/xml"
)

var t, token    xml.Token
var     err                     error

func main() {
        input := "<Person><FirstName>Laura</FirstName><LastName>Lynn</LastName></Person>"
        inputReader := strings.NewReader(input)
        p := xml.NewDecoder(inputReader)

        for t, err = p.Token(); err == nil; t, err = p.Token() {
                switch token := t.(type) {
                        case xml.StartElement:
                                name := token.Name.Local
                                fmt.Printf("Token name: %s\n", name)
                                for _, attr := range token.Attr {
                                        attrName := attr.Name.Local
                                        attrValue := attr.Value
                                        fmt.Printf("An attribute is: %s %s\n", attrName, attr
                                        // ...
                                }
                        case xml.EndElement:
                                fmt.Println("End of token")
                        case xml.CharData:
                                content := string([]byte(token))
                                fmt.Printf("This is the content: %v\n", content )
                                // ...
                        default:
                                // ...
                }
        }
}
```



Parsing XML

In the code above, the **line 12** initializes an XML string called `input`. At **line 13**, we construct a new string reader `inputReader` on top of it, and at **line 14**, we start the XML decoding by defining a decoder `p` on `intputReader`.

The decoding is done in a for-loop starting at **line 16**. We read in a token `p` with the `Token()` method. As long as the error `err` returned by token remains `nil`, we execute the for-loop and then read the next token.

The loop ends when the `Token()` method returns an error at the end of the file because there is no token left to parse. Further processing is done in a *type-switch* at **line 17**. It is defined according to the current kind of XML-tag that

was found in token `t`.

If it is a `StartElement` (see **line 18**), we print out its XML name. Then in the for-loop (from **line 21** to **line 26**), we retrieve all of its attributes and print their names and values. When an XML element is closed, **End of token** is printed at **line 28**. **Line 29** converts the content in the XML data (`Chardata`) to a `[ ]bytes`, making it readable with a string conversion, and prints it out.

The package defines a number of types for XML-tags: `StartElement`, `Chardata` (this is the actual text between the start- and end-tag), `EndElement`, `Comment`, `Directive` or `ProcInst`. It also defines a struct `Decoder`. The method `NewDecoder` takes an `io.Reader` (in this case a `strings.NewReader`) and produces an object of type `Parser`. This has a method `Token()` that returns the next XML token in the input stream. At the end of the input stream, `Token()` returns nil (`io.EOF`).

The XML-text is walked through in a for-loop, which ends when the `Token()` method returns an error at the end of the file because there is no token left to parse. Further processing can be defined according to the current kind of XML-tag through a type-switch. Content in `Chardata` is just a [ ]bytes, make it readable with a string conversion.

Now that you are familiar with how the `xml` package deals with XML type data, let's see how Go handles binary type data in the next lesson.