JSON decoding in Go

29 Nov 2013

Incidentally, decoding JSON data (or really, almost any data structure) is really easy in Go (golang). We simply call `json.Unmarshal(…)` and boom! We have nice data structures.

Well, except if our input source is not very well defined (meaning *not strictly typed*).

Objects with loose schema

Take this example. We want to decode a JSON object that looks like this:

```
{
  "author": "attila@attilaolah.eu",
  "title":  "My Blog",
  "url":    "http://attilaolah.eu"
}
```

The usual way to go is to decode it into a struct:

```
type Record struct {
    Author string `json:"author"`
    Title  string `json:"title"`
    URL    string `json:"url"`
}

func Decode(r io.Reader) (x *Record, err error) {
    x = new(Record)
    err = json.NewDecoder(r).Decode(x)
    return
}
```

That’s fairly easy. But what happens if suddenly we add a new data source that uses numeric author IDs instead of emails? For example, we might have an input stream that looks like this:

```
[{
```

```
  "author": "attila@attilaolah.eu",
  "title":  "My Blog",
  "url":    "http://attilaolah.eu"
}, {
  "author": 1234567890,
  "title":  "Westartup",
  "url":    "http://www.westartup.eu"
}]
```

Decoding to `interface{}`

An quick & easy fix is to decode the `author` field to an `interface{}` and then do a type switch. Something like this:

```go
type Record struct {
    AuthorRaw interface{} `json:"author"`
    Title     string      `json:"title"`
    URL       string      `json:"url"`

    AuthorEmail string
    AuthorID    uint64
}

func Decode(r io.Reader) (x *Record, err error) {
    x = new(Record)
    if err = json.NewDecoder(r).Decode(x); err != nil {
        return
    }
    switch t := x.AuthorRaw.(type) {
    case string:
        x.AuthorEmail = t
    case float64:
        x.AuthorID = uint64(t)
    }
    return
}
```

That was easy… except, it doesn't work. What happens when our IDs get close to 2^{64}-1? Their precision will not fit in a `float64`, so our decoder will end up rounding some IDs. Too bad.

`Decoder.UseNumber()` to the rescue!

Luckily there's an easy way to fix this: by calling `Decoder.UseNumber()`. "UseNumber causes the `Decoder` to unmarshal a number into an `interface{}` as a [`Number`] instead of as a `float64`" — from [the docs].

Now our previous example would look something like this:

```
func Decode(r io.Reader) (x *Record, err error) {
    x = new(Record)
    if err = json.NewDecoder(r).Decode(x); err != nil {
        return
    }
    switch t := x.AuthorRaw.(type) {
    case string:
        x.AuthorEmail = t
    case json.Number:
        var n uint64
        // We would shadow the outer `err` here by using `:=`
        n, err = t.Int64()
        x.AuthorID = n
    }
    return
}
```

Seems fine, now, right? Nope! This will still fail for numbers > 2^{63}, as they would overflow the `int64`.

No we see that if we want to decode a JSON a number into an `uint64`, we really have ta call `Decoder.Decode(…)` (or `json.Unmarshal(…)`) with a `*uint64` argument (a pointer to a `uint64`). We could do that simply by directly decoding the string representation of the number. Instead of:

```
        n, err = t.Int64()
```

...we could write:

```
        err = json.Unmarshal([]byte(t.String()), &n)
```

Wait... Let's use `json.RawMessage` instead.

Now we're correctly decoding large numbers into `uint64`. But now we're also just using the `json.Number` type to delay decoding of a particular value. To do that, the `json` package provides a more powerful type:

json.RawMessage. `RawMessage` simply delays the decoding of part of a message, so we can do it ourselves later. (We can also use it to special-case encoding of a value.)

Here is our example, using `json.RawMessage`:

```
type Record struct {
    AuthorRaw json.RawMessage `json:"author"`
    Title     string          `json:"title"`
    URL       string          `json:"url"`

    AuthorEmail string
    AuthorID    uint64
}

func Decode(r io.Reader) (x *Record, err error) {
    x = new(Record)
    if err = json.NewDecoder(r).Decode(x); err != nil {
        return
    }
    var s string
    err = json.Unmarshal(x.AuthorRaw, &s); err == nil {
        x.AuthorEmail = s
        return
    }
    var n uint64
    err = json.Unmarshal(x.AuthorRaw, &n); err == nil {
        x.AuthorID = n
    }
    return
}
```

This looks better. Now we can even extend it to accept more schemas. Say we want to accept a third format:

```
[{
  "author": "attila@attilaolah.eu",
  "title":  "My Blog",
  "url":    "http://attilaolah.eu"
}, {
  "author": 1234567890,
  "title":  "Westartup",
  "url":    "http://www.westartup.eu"
}, {
  "author": {
    "id":    1234567890,
    "email": "nospam@westartup.eu"
  },
  "title":  "Westartup",
```

```
  "url":    "http://www.westartup.eu"
}]
```

It seems obvious that we are going to need an `Author` type. Let's define one.

```
type Record struct {
    AuthorRaw json.RawMessage `json:"author"`
    Title     string          `json:"title"`
    URL       string          `json:"url"`

    Author Author
}

type Author struct {
    ID    uint64 `json:"id"`
    Email string `json:"email"`
}

func Decode(r io.Reader) (x *Record, err error) {
    x = new(Record)
    if err = json.NewDecoder(r).Decode(x); err != nil {
        return
    }
    err = json.Unmarshal(x.AuthorRaw, &x.Author); err == nil {
        return
    }
    var s string
    err = json.Unmarshal(x.AuthorRaw, &s); err == nil {
        x.Author.Email = s
        return
    }
    var n uint64
    err = json.Unmarshal(x.AuthorRaw, &n); err == nil {
        x.Author.ID = n
    }
    return
}
```

This looks fine... Except that now we're doing all the decoding of the Author type in the function that decodes the `Record` object. And we can see that with time, our `Record` object's decoder will grow bigger and bigger. Wouldn't it be nice if the `Author` type could somehow *decode itself*?

Behold, the [`json.Unmarshaler`]() interface!

Implement that by any type, and the `json` package will use that to unmarshal your object.

Let's move the decode logic to the `Author` struct:

```
type Record struct {
    Author Author `json:"author"`
    Title  string `json:"title"`
    URL    string `json:"url"`
}

type Author struct {
    ID    uint64 `json:"id"`
    Email string `json:"email"`
}

// Used to avoid recursion in UnmarshalJSON below.
type author Author

func (a *Author) UnmarshalJSON(b []byte) (err error) {
        j, s, n := author{}, "", uint64(0)
    if err = json.Unmarshal(b, &j); err == nil {
                *a = Author(j)
        return
    }
    if err = json.Unmarshal(b, &s); err == nil {
        a.Email = s
        return
    }
    if err = json.Unmarshal(b, &n); err == nil {
        a.ID = n
    }
    return
}

func Decode(r io.Reader) (x *Record, err error) {
    x = new(Record)
    err = json.NewDecoder(r).Decode(x)
    return
}
```

Much better. Now that the `Author` object knows how to decode itself, we don't have to worry about it any more (and we can extend `Author.UnmarshalJSON` when we want to support extra schemas, e.g. username or email).

Furthermore, now that Record objects can be decoded without any additional work, we can move one more level higher:

```
type Records []Record
```

```
func Decode(r io.Reader) (x Records, err error) {
    err = json.NewDecoder(r).Decode(&x)
    return
}
```

You can go play with this.

NOTE: Thanks to Riobard Zhan for pointing out a mistake in the previous version of this article. The reason I have two types above, `Author` and `author`, is to avoid an infinite recursion when unmarshalling into an `Author` instance. The private `author` type is used to trigger the built-in JSON unmarshal machinery, while the exported `Author` type is used to implement the `json.Unmarshaler` interface. The trick with the conversion near the top of the `Unmarshal` is used to avoid the recursion.

What about encoding?

Let's say we want to normalise all these data sources in our API and *always* return the `author` field as an object. With the above implementation, we don't have to do anything: re-encoding records will normalise all objects for us.

However, we might want to save some bandwidth by not sending defaults. For that, we can tag our fields with `json:",omitempty"`:

```
type Author struct {
    ID    uint64 `json:"id,omitempty"`
    Email string `json:"email,omitempty"`
}
```

Now `1234` will be turned into `{"id":1234}`, `"attila@attilaolah.eu"` to `{"email":"attila@attilaolah.eu"}`, and `{"id":1234,"email":"attila@attilaolah.eu"}` will be left intact when re-encoding objects.

Using `json.Marshaler`

For encoding custom stuff, there's the `json.Marshaler` interface. It's works similarly to `json.Unmarshaler`. You implement it, and the `json` package uses it.

Let's say that we want to save some bandwidth, and always transfer the minimal information required to reconstruct the objects by the `json.Unmarshaler`. We could implement something like this:

```
func (a *Author) MarshalJSON() ([]byte, error) {
    if a.ID != 0 && a.Email != "" {
        return json.Marshal(map[string]interface{}{
            "id":    a.ID,
            "email": a.Email,
        })
    }
    if a.ID != 0 {
        return json.Marshal(a.ID)
    }
    if a.Email != "" {
        return json.Marshal(a.Email)
    }
    return json.Marshal(nil)
}
```

Now `1234`, `"attila@attilaolah.eu"` and `{"id":1234,"email":"attila@attilaolah.eu"}` are left intact, but `{"id":1234}` is turned into `1234` and `{"email":"attila@attilaolah.eu"}` is turned into `"attila@attilaolah.eu"`.

Another way to do the same would be to have two types, one that always encodes to an object (`Author`), and one that encodes to the minimal representation (`CompactAuthor`):

```
type Author struct {
    ID    uint64 `json:"id,omitempty"`
    Email string `json:"email,omitempty"`
}

type CompactAuthor Author

func (a *CompactAuthor) MarshalJSON() ([]byte, error) {
    if a.ID != 0 && a.Email != "" {
        return json.Marshal(Author(a))
```

```
    }
    if a.ID != 0 {
        return json.Marshal(a.ID)
    }
    if a.Email != "" {
        return json.Marshal(a.Email)
    }
    return json.Marshal(nil)
}
```

Using pointers

We see now that the `omitempty` tag is pretty neat. But we can't use it with the `author` field, because `json` can't tell if an `Author` object is "empty" or not (for `json`, a struct is always non-empty).

To fix that, we can turn the `author` field into an `*Author` instead. Now `json.Unmarshal(…)` will leave that field `nil` when the author information is completely missing, and it will not include the field in the output when `Record.Author` is `nil`.

Example:

```
type Record struct {
    Author *Author `json:"author"`
    Title  string  `json:"title"`
    URL    string  `json:"url"`
}
```

Timestamps

[`time.Time`] implements both `json.Marshaler` and `json.Unmarshaler`. Timestamps are formatted as [RFC3339].

However, it is important to remember that `time.Time` is a struct type, hence `json` will never consider it "empty" (`json` will *not* consult [`Time.IsZero()`])

To omit zero timestamps with the `omitempty` tag, use a pointer (i.e.

`*time.Time`) instead.

Conclusion

Interfaces are awesome. Even more awesome than you probably think. And tags. Combine these two, and you have objects that you can expose through an API supporting various encoding formats. Let me finish with a simple yet powerful example:

```
type Author struct {
    XMLName xml.Name `json:"-" xml:"author"`
    ID      uint64   `json:"id,omitempty" xml:"id,attr"`
    Email   string   `json:"email,omitempty" xml:"email"`
}
```

Authors can now be encoded/decoded to/from a number of formats:

```
[{
  "id": 1234,
  "email": "attila@attilaolah.eu"
},
  "nospam@westartup.eu",
  5678
]
```

```
<author id="1234">
  <email>attila@attilaolah.eu</email>
</author>
<author>
  <email>nospam@westartup.eu</email>
</author>
<author id="5678"/>
```

Related posts:

- JSON and struct composition in Go