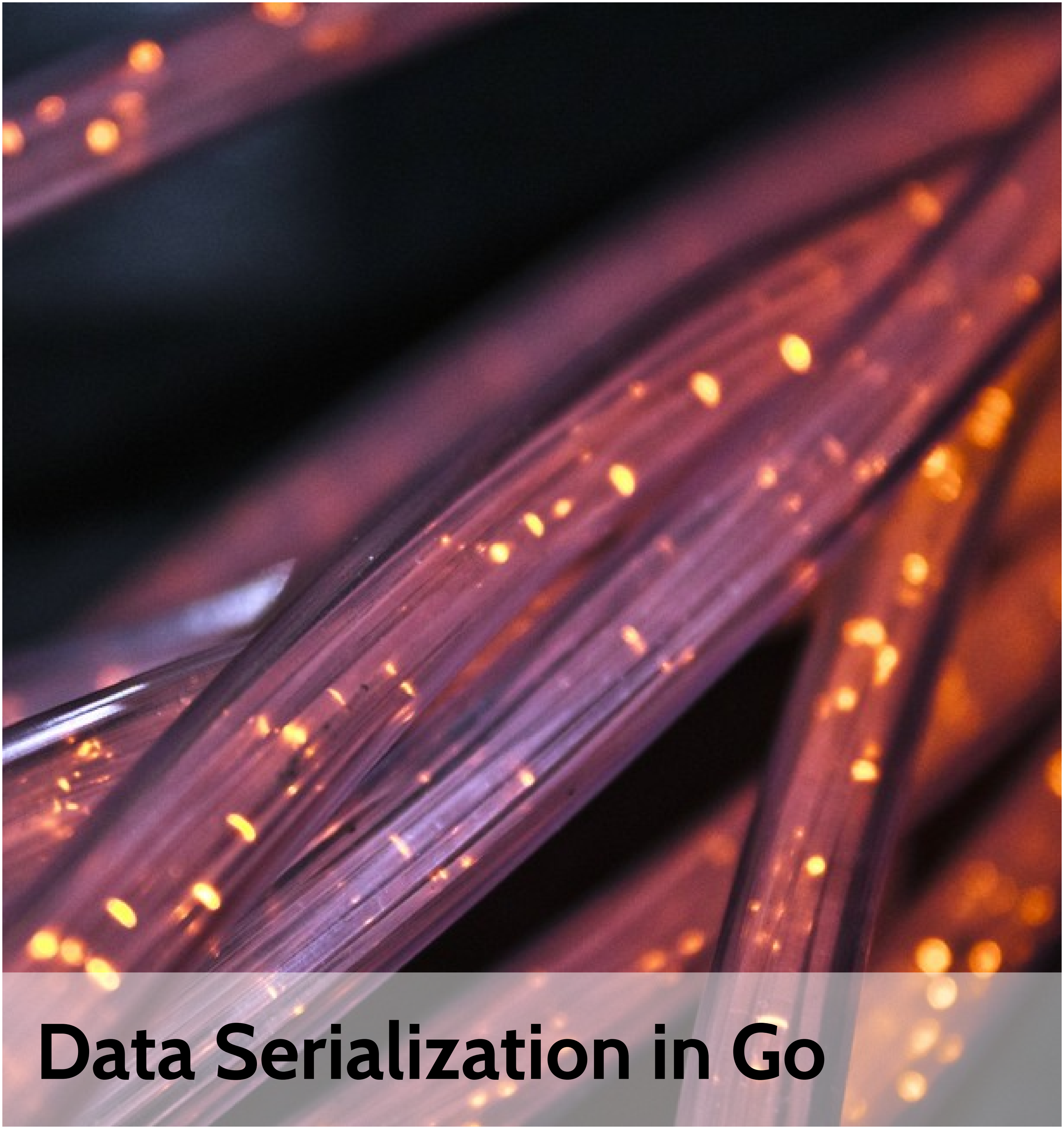

Data Serialization in Go

The Complete Guide to JSON and Other Serialization Formats in Go

Jonathan Hall

Data Serialization in Go

The Complete Guide to JSON and Other Serialization Formats in Go

Jonathan Hall

December 12, 2020

Data Serialization in Go

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.
The source code of this book template is available at:
`https://github.com/fmarotta/kaobook`

Preface

This book has been running around in the back of my mind for several years. And it's still very much a work in progress, as you can see, from the many incomplete sections.

In August of 2015, I began work on a wrapper for PouchDB[1], to allow me to use it with GopherJS[2]. PouchDB is inspired by Apache CouchDB[3], a mult-master, JSON-based NoSQL database. This work eventually grew into Kivik[4], a common Go and GopherJS client library for CouchDB, PouchDB, and related databases, which I still actively develop and maintain.

My work on Kivik forced me to face a wide variety of JSON-related challenges in Go. So in many ways, this is the book I wish I could have read when beginning that project several years ago.

In 2016 I asked a question on StackOverflow: "What is the idiomatic way to embed a struct with a custom `MarshalJSON()` method?"[5] It never received a satisfying answer. That is, until Section 8.3 (Extending an Embedded Marshaler).

Many of these topics I have previously discussed elsewhere: on my blog[6], on my YouTube channel[7], or just on the `go` tag on StackOverflow[8]. But I feel the time has finally come to bring it all together, into one, authoritative resource.

1: PouchDB is a JavaScript database designed to run in the browser. https://pouchdb.com/

2: GopherJS compiles Go to Javascript, to run in a browser. https://github.com/gopherjs/gopherjs

3: https://couchdb.apache.org/

4: http://kivik.io/

5: https://stackoverflow.com/q/38489776/13860

6: https://jhall.io/posts/

7: https://www.youtube.com/channel/UC5UfX0EgUWlcdQ2RDsq_fcA

8: https://stackoverflow.com/questions/tagged/go

Intended audience

This book assumes that you are already familiar with the basics of Go programming, but not necessarily with data serialization (in or out of Go).

This book's primary audience is the developer comfortable with Go, but struggling with some of the nuances of data serialization, particularly in a statically-typed language, or without the use of generics, as this describes myself in 2016, when I began wishing I could read this book!

JSON is the starting point

This book generally uses JSON as a baseline for all data serializations. I made this decision for a number of reasons:

- JSON is the most ubiquitous data serialization format in use today, so nearly everyone should have at least basic familiarity

- JSON is (basically) human-readable, which makes it easy use in examples
- Using a single standard, then discussing differences where appropriate, seems much easier than selecting a different random format for each example.

For this reason, most examples and discussion center around the `encoding/json` package in the Go standard library, and this is the assumed base-line against which other comparisons are made.

How to use this book

I have divided the book into a number of parts, to make navigation easier.

Part I (The Basics) on page 16 covers the basics concepts of data serialization, particularly as it relates to Go. This section is intended primarily for the reader inexperienced with serialized data, or serializing data with the Go standard library.

Part II (Data Serialization Toolbox) on page 20 dives deeper into some of the more subtle capabilities of the Go standard library, and builds some foundational concepts for use later in the book. While it does contain many code samples, this section is the most theoretical of the entire book.

Part III (Recipes) on page 27 is the "cook book" section of this book. It contains concrete answers to common data serialization questions in Go, along with a step-by-step explanation, and a complete, working code example for each one. This section builds on material discussed in Part II (Data Serialization Toolbox), but I provide plenty of references, so don't feel obligated to read it first.

If you find this section dry to read page-by-page, I won't be offended. It is intended to serve mainly as reference. It would be good to at least skim this section, or its table of contents, though, so you know what it covers, so that you know where to look when you run across a question later on.

Part IV (Working with YAML, BSON, MessagePack, and Others) on page 60 digs into the many non-JSON data serialization formats, and discusses how their use differs in Go, how to convert between them, and other related subjects.

And finally, **Part V (Third Party-Libraries) on page 69** talks about third-party libraries for data serialization, for niche purposes.

A work in progress

This book is still a work in progress. As such, many sections are still missing (most are marked with **(TODO)** or **(WIP)**, and are subject to change or even be removed, as the project matures). There are also no doubt content errors, including spelling and formatting mistakes, which will be corrected as the project matures.

I intend to add new material at a healthy pace. If you find yourself facing a problem you wish this book would cover, please do not hesitate to contact me—I can make an effort to get that material into the book sooner than later.

And, of course, if you have purchased this book on LeanPub, all updates are free, so you'll get the updated content as soon as I have made it available!

Thank you

Thank you for reading. I hope you find this material helpful. I would love to hear your questions and feedback.

Jonathan Hall

jonathan@jhall.io

Contents

Introduction (WIP) 1

1.1 What is Data Serialization

The concept of data serialization is at the same both straight forward, and very broad and wrought with subtly.

According to Wikipedia[1]:

> Serialization is the process of translating data structures or object state into a format that can be stored or transmitted and reconstructed later.

1: http
Serial

That sounds simple enough! Why would you want to read a book that just expands on that single sentence?

Subtleties

As with many things, data serialization is much easier said than done.

“I want to transfer this set of data objects to another computer, so please serialize it for me," seems like a simple enough request.

But what are those data objects? What kinds of data do they contain? Can that data be expressed efficiently—or at all—over the transmission medium? How will the receiver know how to transform the serialized data back into meaningful objects?

When data isn’t just data

One of the most obvious limitations of data serialization is when a data object isn’t just a data object. As programmers, we are constantly using data types that refer to external state: Files, network connections, even OS processes, are all accessed via data structures, but trying to serialize those structures is, at best non-straight forward (i.e. when dealing with a file on disk), and at worst, completely impossible (serializing an actual network connection doesn’t really even make sense).

How to handle such types of data is not at all straight forward. Let’s consider the different ways we might handle the serialization of an open file in Go, represented by the `*os.File`[2] type:

2: https://golang.org/pkg/os/#File

1. We could simply skip it
2. We could abort the serialization operation, and return an error
3. We could encode simple metadata about the file (file name, size, ownership, permission, etc)
4. We could encode the file contents

Every one of these options could be valid, depending on your needs. So the challenge then becomes determining our needs, and how to express them in, and to our software. But at least in principle, it's possible to serialize an on-disk format. This is, after all, what compression and archive programs like `zip` and `tar` do, isn't it?

But let's consider one other hairy situation: Encoding a network connection, represented by the Go data type `net.Conn`[3]. Here we have some of the same options as for the `*os.File` example: skip, return an error, or encode some metadata about the connection, such as the remote address and port. But the final option, of encoding the "contents" of the network connection really isn't possible, except in certain very specific, applications (such as network capturing as done by Wireshark[4]), or if you know that the network connection is meant to read a data stream—perhaps video or audio.

3: https://golang.org/pkg/net/#Conn

4: https://www.wireshark.org/

Because of the ambiguity that surrounds serializing data types that refer to ephemeral state, such as these, the common approach is to return an error, rather than attempting serialization. In Go this also applies to channels, which by nature contain ephemeral data.

Fortunately, Go gives us the flexibility to add our own (de-)serialization methods when we have a legitimate need to operate serially on such data types, by way of custom `Marshal` and `Unmarshal` methods.

When data shouldn't be shared

There are also times when data simply shouldn't be shared. This might be a matter of privacy, but from a programmatic standpoint, it's more often a matter of just not airing our dirty laundry.

Many data structures contain internal state that is both meaningless outside the context of the data structure, and also unnecessary to faithfully transmit the data serially. Examples of this are;

- Data caches
- Internal counters and flags
- Duplicated data (usually for convenience)
- References to ephemeral state, as discussed above

Let's look at one simple example from the Go standard library: `url.URL`[5]. Although this type does not contain any unexported fields, it does contain examples of data caching, internal flags, and duplicated data. The struct definition looks like this:

5: https://golang.org/pkg/net/url/#URL

```
type URL struct {
    Scheme     string
    Opaque     string    // encoded opaque data
    User       *Userinfo // username and password information
    Host       string    // host or host:port
    Path       string    // path (relative paths may omit leading
    slash)
    RawPath    string    // encoded path hint (see EscapedPath
    method)
    ForceQuery bool      // append a query ('?') even if RawQuery
    is empty
    RawQuery   string    // encoded query values, without '?'
    Fragment    string    // fragment for references, without '#'
}
```

It may not be entirely obvious by this definition alone, but nearly half of the fields in this struct contain redundant data in one form or another. Read the documentation for the (String)[6] method for complete details, but here's a brief summary:

6: https://golang.org/pkg/net/url/#URL.String

- `Opaque` duplicates `Host` and `Path`
- `RawPath` duplicates `Path`
- `ForceQuery` is a formatting flag

While it would be possible, in principle, to serialize this entire data structure, it would rarely, if ever, make sense to do so. If you have a `url.URL` object in memory, and want to transmit it over a network, or store it in a file, the natural approach would be to store it in the normalized text format, as produced by the `String` method. This method applies all of the proper de-duplication rules to faithfully produce a textual representation. And the reverse is achieved with the `Parse`[7] method.

7: https://golang.org/pkg/net/url/#Parse

This is incredibly common, both with standard data types, as well as with custom data types.

When data doesn't have a single, unambiguous representation

Putting aside the differences between serialization formats for the time (JSON vs YAML vs XML), quite often there are multiple valid representations of a single piece of data. Choosing which format to use isn't always straight forward.

Dates are an obvious example. Today's date, as I'm writing this, can be expressed in a number of valid ways:

- July 11, 2020
- Saturday, 11th of July
- 2020/07/11
- 19 Tamuz 5780[8]

8: https://en.wikipedia.org/wiki/Hebrew_calendar

And of course, many, many more.

"But that's a bad example," you may be thinking. "Dates are especially ambiguous, because they're primarily consumed by humans."

Dates *are* an extreme example, to be sure. So let's consider something more innocuous: A simple string. There's only one possible representation of the string "Hello, World!", right?

Of course you know my answer: No.

ASCII (hex)	`48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21`
CDC 1604 Punch Card (hex)	`48 45 4C 4C 4F 2C 20 57 4F 52 4C 44`
Morse code	`.... . .-.. .-.. -- -..- .- -- .-. .-.. -.. -.-.-`

The first is standard ASCII, with which we are all abundantly familiar. The second is an old 6-bit encoding[9], which only supports capital letters, and no exclamation point. And the final example is the even older Morse code[10], invented well before the first computer.

9: https://en.wikipedia.org/wiki/Six-bit_character_code#Examples_of_BCD_six-bit_codes

10: https://en.wikipedia.org/wiki/Morse_code

"Okay, okay, you've made your point. But nobody uses 6-bit character sets, or Morse code these days."

You may have a point. But only barely. In fact, until fairly recently, character encodings[11] were a common nightmare for programmers working with human-readable text, as every language needed a different encoding, and in many cases, different computer manufactures had competing standards. Fortunately, by now most computers use Unicode[12], so even using non-English character sets is relatively easy in many common cases.

11: `https://en.wikipedia.org/wiki/Character_encoding`

12: `https://en.wikipedia.org/wiki/Unicode`

So let's use an even more basic data type: the integer. There should be no ambiguity at all when transmitting a number like, say, 4200, right?

Wrong again. This time, really.

Consider these *common* integer formats: Little- and big-endian[13], JSON (which is just the ASCII representation of the decimal number), BSON[14], and varint[15].

13: `https://en.wikipedia.org/wiki/Endianness`

14: `http://bsonspec.org/`

15: `https://developers.google.com/protocol-buffers/docs/encoding`

Little-endian	`00010000 01101000`
Big-endian	`01101000 00010000`
JSON	`00110100 00110010 00110000 00110000`
BSON	`00000000 00000000 00010000 01101000`
Varint	`11010000 01000001`

Even at the most fundamental level, of representing raw numbers, choices must be made as to how data should be represented in serial format.

For the most part, we don't need to worry about this level of detail, although there are times when it definitely matters, which we'll discuss as they arise.

Serialization is key

After this little detour into some of the nuances of data serialization, you could be forgiven for feeling it may not be worth the effort.

But please, don't do that.

Data serialization is, after all, at the heart of computers. What is a computer, if not a device that consumes serialized input, and produces some other serialized output? In a sense. a computer is fundamentally a machine that simply de-serializes, processes, then re-serializes data. Nothing else, really.

So if we want to make effective use of our computers, learning to effectively serialize and deserialize data is, in my opinion, very much worth the effort.

It's also quite fun. For my taste, anyway.

1.2 Uses of Data Serialization

As discussed in Section 1.1 (What is Data Serialization), data serialization is ubiquitous in computing. It is used for practically all input and output operations, from typing (input of serial textual data), to reading and writing files, to network communication, and many more applications.

As such, we can expect that there are countless applications for data serialization. I can't pretend to imagine all of them, and I won't even try to discuss some I can think of—I'm not particularly interested in the wire protocols used for USB or SATA, for example.

Instead, here I'll focus on the uses of data serialization, that I will cover in this book. Later, in Section **??** (**??**), I'll dive deeper into how to select a format for a particular application, along with some personal recommendations.

These categories are not strictly defined, and there are many overlaps. The point is to provide a general overview of the important uses, not to build a precisely-defined dictionary of terms.

Messaging

I don't mean messaging in the sense of instant messaging—although that is clearly an example of messaging.

I'm talking about sending messages between computers, such as through a REST API or SOAP[16].

16: `https://en.wikipedia.org/wiki/SOAP`

This is the first thing many people think of when considering data serialization. Especially in the modern Internet-connected world, this is where many of us spend most of our time considering data serialization.

File storage

Although it's perhaps less visible to most programmers these days, as we largely use well-defined formats, handled by existing libraries, file formats are data serialization formats.

Configuration

Closely related, and perhaps a subset of, file storage, I think configuration deserves its own heading. The important difference, in my mind, is that while most file formats are intended to be written *and* read by computers, configuration formats must be read by computers, but are *written* (and to some extent read) by humans.

This asymmetry leads to many unique considerations when dealing with configuration formats. Some formats can *only* be written by humans; or at the very least, computers lose, or distort information, when they attempt to write them. For example, it's often difficult to programmatically update a YAML file that contains comments, without discarding or re-ordering the comments.

Data comparison

Sometimes the reason to serialize data is simply to perform a comparison between two like data structures. By serializing, the comparison can often be made simpler.

Human readability

This might seem like a corner case, but in truth, practically all computer output is the result of serializing data for human consumption. From the text you're reading right now, to a video on YouTube, or the frames produced by your GPU in your favorite video game–the computer had to serialize that data in some way, for the benefit of your consumption.

Some less banal examples might be generating readable log output, or producing a human-readable diff[17] (relates to Data comparison).

17: `https://en.wikipedia.org/wiki/Diff`

Whatever the use case, when the primary consumer of serialized data is a human, special considerations come into play.

Hashing

Hashing is an often-overlooked application of data serialization. This comes in handy when, for example, we need to know the identity of a complex data structure. We can calculate a hash of the serialized representation of the object, to act as its identity—even if we don't keep the serialized data around.

This is essentially how CouchDB calculates document revisions[18], and how git calculates commit hashes[19].

18: `https://stackoverflow.com/a/5961218/13860`

19: `https://gist.github.com/masak/2415865`

Data compression and encryption

Arguably, data compression is not an application of data serialization, since it simply modifies already-serialized data, but I think it warrants mentioning because it so often comes up when working with serialized data. Whether trying to save network bandwidth with our messaging protocols, or saving disk space with our disk formats, data compression is highly relevant.

In theory, encryption could be done solely for the sake of hiding data that otherwise would not be serialized (i.e. encrypting the in-memory copy of some data), but in practice, it tends to operate on data that has already been serialized, same as data compression. And of course, most encryption topis are well beyond the scope of this book, but it does relate, so it's worth mentioning.

1.3 History (TODO)

WIP: A brief history of data serialization formats, and how JSON fits on the map, along with related formats (particularly: XML, YAML).

Data Formats (WIP) 2

Choosing a data format for your project can be a daunting task. There are almost as many data formats as there are uses of data. Some are open standards. Some are language-specific. Some are completely proprietary.

This chapter provides an overview of some of the more common data formats, and their relative strengths and weaknesses, at a very high level. For very specialized use cases, you are encouraged to do your own research, or possibly even create your own format.

This chapter is divided into sections, one for each format or group of formats. Within each section, I provide a very brief "elevator pitch" for the format before diving into the more fine-grained strengths and weaknesses.

When the decision has been made for you

Some times, you'll find that the decision has already been made for you. Perhaps you're integrating with a third-party application or API, which dictates the format. You might think you're entirely limited here; and in a sense you'd be right. But if the pre-determined requirements are onerous enough, you may desire a work-around.

Writing a translation layer between the third-party format, and your main application can make your application programming much simpler.

This may be most common when reading files produced by some third-party application. Perhaps you're required to read files produced by Excel 97. Rather than having your application read the Excel 97 format directly, perhaps you can convert the data using an external tool (or write one yourself), then consume a friendlier format.

This can also happen, although hopefully more rarely, when integrating with certain APIs. One project I worked on a few years ago needed to integrate with an API whose vendor only provided a C# SDK. Our application was written in Node.js. Our solution was to implement a C# API that exposed the service to our Node.js using a REST API. With this proxy in place, we had the flexibility to expose whatever data format we wanted to our internal application.

2.1 JSON

Spec:	RFC 8259
MIME:	application/json
File Ext:	.json, .jsn

JSON is not a great format, but it's simple, and widely supported, making it a great go-to format, for just about anything—or at least the first version.

Great for

- Communicating with web browsers
- REST APIs
- Simple messaging
- Easy debugging

Reasonable for

- Configuration files
- On-disk storage of small or simple data
- Human-readable data

Bad for

- Random access files
- Binary data
- Semantic data

The current version of JavaScript Object Notation, or JSON, as defined by RFC 8259[1], is a light-weight data interchange format that has its roots, obviously, in JavaScript. It's ubiquity, and human readability make it a reasonable default choice for data formats, despite its many shortcomings, which I will discuss shortly.

1: `https://tools.ietf.org/html/rfc8259`

Because of its ubiquity, I expect many of you will use JSON frequently—even if you use other formats as well.

For this reason, this book uses JSON, and specifically, Go's `encoding/json` library, as the basis for discussion of data serialization. The majority of concepts discussed in this book apply equally to all data formats, so using a common base-line makes sense. When there are differences, they will be mentioned explicitly.

Strengths

JSON has a lot going for it. In a technical sense, it's not the best format for most things, but all things considered, it's key strengths make it a the go-to format for many uses. Only when one of the shortcomings of JSON begins to give you headaches would I recommend considering another format.

JSON is simple

RFC 8259, which defines JSON, is only 16 pages long. And of those 16 pages, 3 are just links to references, one is a copyright notice, and two are just introductory text. That leaves 10 pages of actual specifications. You can read the entire thing in half an hour. If you prefer diagrams and flow charts, www.json.org breaks it down nicely.

The point is, JSON is simple enough for practically anyone to understand the basics quickly. And any subtle nuances are not much harder to pick up (some on that in a moment). This makes it an attractive format for a wide variety of use cases.

JSON is human-readable

JSON isn't always *fun* to read, but the fact that it is a text-based format does provide a lot of advantages when debugging.

Because of this benefit, many applications and libraries that support more efficient data formats, often support JSON as well, to allow the user to turn on JSON mode for debugging purposes.

Twirp (`https://github.com/twitchtv/twirp`) is one such library, which supports both Protobuf and JSON

JSON is ubiquitous

JSON is everywhere. This means that practically every software developer should be familiar with it. If you're writing software that others will use or interact with, JSON is an excellent choice for reducing the learning curve.

JSON is native to browser

If you're building a backend service that talks to a web browser, then JSON is likely the best choice, because the browser can natively parse and produce JSON. Other formats are possible, but require additional libraries, which can bloat the front-end code and add complexity.

Weaknesses

Sadly, the list of problems with JSON is a lot longer than the list of its strengths. But it strengths are pretty strong, so don't dismiss JSON so quickly!

JSON does not handle binary data

JSON is purely a text format.

It supports strings, numbers, boolean values, and null. These types can then be composed into objects or arrays. That's it.

If you wish to encode some bit of binary data, such as an encryption key, or an image, there is no proper way to do this with JSON—although convention, and `encoding/json`, is to Base64-encode binary data as a JSON string. While this does allow for JSON-representation of binary data, it has two big drawbacks:

1. It increases the size of the data by 33%
2. It is up to the consumer of the data to know that the data should be Base64-decoded, and not treated as a raw string

JSON is verbose

A second drawback to JSON is that it is very verbose. This is the nature of practically all human-readable formats.

An obvious example is that JSON represents numbers as decimal values in plain ASCII. As we saw on page 15, the number 4200 requires four bytes in JSON, but only two bytes in memory.

Also object keys must be quoted, which adds an overhead of 2 bytes per key, and are transmitted in plain text (i.e. human-readable), which can use a lot of space. And what's more, object keys are repeated for every instance of an object.

There are projects, such as JSONH (https://github.com/WebReflection/JSONH), which eliminate repetitive object keys, while still using JSON, but they lose a lot of the human readability of JSON.

And of course, as mentioned above, when encoding binary data, the size actually *increases* by 33% due to Base64 overhead.

A lot of this downside can be overcome by using data compression. A recent analysis by Lucidchart[2] found that JSON + Brotli compression[3] provided the best performance compared to more compact formats, for their needs. So just because JSON is verbose doesn't necessarily mean you should avoid it.

2: `https://www.lucidchart.com/techblog/2019/12/06/json-compression-alternative-binary-formats-and-`

3: `https://en.wikipedia.org/wiki/Brotli`

JSON has limited data types

I already discussed the lack of binary data support, but the impact of limited data types reaches much further. With only three fundamental types: string, number, and boolean (four if you count null separately), there is no way to express numerical precision (i.e. a 16-bit int versus a 32-bit int), or to indicate that a string represents a date, rather than someone's name.

In practice, this means that one of the most complicated parts of handling JSON (in a language other than JavaScript, anyway), is coercing your data into a string that faithfully represents your data—or parsing a string you've received, into the desired data format.

JSON handles numbers very poorly

Closely related, JSON's number handling is poor. Or more to the point, it practically doesn't exist.

The JSON specification itself offers no limit, but also no guarantee, about the precision of numbers. In theory, any number can be faithfully represented in JSON. But in practice, it is generally considered unsafe to use JSON integers less than -(2^{53}-1) or greater than 2^{53}-1, because this is the limitation imposed by JavaScript[4].

4: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/MAX_SAFE_INTEGER`

Unless you are certain your JSON will never be modified by a JavaScript interpreter, it is safest to encode very small or very large integers, or floating-point numbers of high precision, as a string. See Section 7.2 on page 27 for more on this topic.

JSON offers no semantic data

XML used to be the buzz in the 2000s. Then JSON came along and stole the show. But it left semantics behind.

Simply put, this means that the *meaning* of the data contained in JSON, is not part of the JSON representation. Where XML will tell you exactly what a piece of data is used for, with its tags, JSON leaves that information out completely.

There are efforts to standardize JSON semantics for particular applications. See JSON-LD (`https://en.wikipedia.org/wiki/JSON-LD` for one such example.)

JSON doesn't use length markers

This means that while reading JSON, you must read all of the data, before you know how much to read. For many applications this doesn't matter, but if you're trying to store data in an on-disk format for easy random access, it spells disaster. To read the twelfth item in a JSON file, you must read the preceding eleven elements in their entirety, to even know where the twelfth one begins.

JSON doesn't have comments

Comments existed in early versions of JSON, but the author removed them[5] because he saw them being abused.

While this doesn't matter for many applications, but for configuration files, comments are often considered a necessity.

5: `https://web.archive.org/web/20120506232618/https://plus.google.com/118095276221607585885/posts/RK8qyGVaGSr`

2.2 YAML

Spec:	YAML 1.2
MIME:	application/x-yaml, text/yaml
File Ext:	.yaml, .yml

YAML is most popular for configuration files, but can be used for any sort of generic data interchange, must as JSON. The complex parsing rules, though, often lead to subtle incompatibilities between parsers, so using it as a data interchange between different implementations can be dangerous.

Great for

- Configuration files
- Human-readable data

Reasonable for

- On-disk storage of small or simple data

Bad for

- REST APIs
- Random access files
- Binary data
- Semantic data
- Data interchange between different parsers/languages

YAML, which stands for "YAML Ain't Markup Language", as of version 1.2[6], is officially a superset of JSON, meaning that a valid JSON document can be parsed as YAML (but not the other way around). You can even mix JSON into a YAML file, and it should parse correctly.

6: `https://yaml.org/spec/1.2/spec.html`

In large part due to its support for comments (in contrast to JSON) files, YAML is often preferred for configuration files over JSON.

YAML's syntax is significantly more complex than JSON, however, which makes it both more expressive, but also more difficult to learn.

Strengths

YAML supports comments

As already mentioned, support for comments is one of YAML's biggest strengths, particularly for use as a configuration format, or any other format that is written by humans.

YAML can be less verbose than JSON

YAML offers several features that can make it more efficient than JSON, in terms of space usage. Most substantially, YAML offers aliases[7], which provides a way to reference one portion of a document in multiple other places, reducing duplication.

7: `https://yaml.org/spec/1.2/spec.html#id2765878`

At a lower level, YAML also allows for more concise representation of many types of data, as compared to JSON:

- Object keys are not quoted
- Strings may be unquoted (in many cases)
- Numbers may be represented using hexadecimal
- Nulls can be shortened to

YAML has more extensive types

YAML has explicit support for dates (expressed in ISO 8601[8] format).

YAML also supports extended types, such as binary data, ordered maps, and sets[9].

8: `https://www.iso.org/iso-8601-date-and-time-format.html`

9: `https://yaml.org/spec/1.2/spec.html#id2761292`

Weaknesses

YAML is complex

YAML has a reputation for being over-complex. As an example, there are as many as 63[10] ways to write multi-line strings in YAML.

10: https://stackoverflow.com/a/21699210/13860

And as a practical matter, indentation is significant in YAML, which leads to frequent typos and frustration, when a line is not indented properly.

The rules about parsing simple values are often confusing, too, leading to different parsers making different decisions[11].

11: https://github.com/cblp/yaml-sucks

YAML is not efficient for binary data

Although YAML, unlike JSON, has direct support for binary data, it's still a text format, so that data must be Base64-encoded, which bloats it by 33%.

2.3 TOML

Spec:	TOML v1.0.0-rc1
MIME:	application/toml
File Ext:	.toml

TOML is designed from the ground up for writing configuration files, and it does that very well. You should not consider TOML for any other purpose.

Great for

- Configuration files

Bad for

- Everything else

TOML[12] stands for "Tom's Obvious Minimal Language", named after the format's creator, Tom Preston-Werner. It is specifically designed to be obvious for humans to read and write, while also being easy to parse programmatically.

12: https://toml.io/

Strengths

TOML data types

TOML expands on both JSON and YAML with respect to fundamental data types, by offering strings, integers, floats, booleans, timestamps (with and without timezones) in RFC 3339[13] format, stand-alone dates and times.

13: https://tools.ietf.org/html/rfc3339

The supported composite types are arrays (which may be of mixed types), and tables, which approximate JSON objects, but in a more config-centric format.

Weaknesses

TOML's limited focus on configuration files makes it unsuitable for practically any other use.

2.4 INI (TODO)

2.5 BSON (TODO)

2.6 MessagePack (TODO)

2.7 Smile (TODO)

2.8 Ion (TODO)

2.9 CBOR (TODO)

2.10 XML (TODO)

2.11 Protobuf (TODO)

2.12 CSV (TODO)

2.13 Custom & proprietary formats (TODO)

Part I

The Basics

Getting Started with JSON (TODO) 3

3.1 The `encoding/json` Package (TODO)

WIP: Brief discussion of the standard library `encoding/json` package.

Throughout this book, I use the standard Go spelling of terms, as defined at `https://github.com/golang/go/wiki/Spelling`, even though this seems to prefer the UK-centric spelling of certain words (i.e. *canceled* and *marshaling*), while choosing American spellings everywhere else.

3.2 Marshaling JSON (TODO)

WIP: How to marshal simple data to JSON.

- Simple data types (`int`s, `string`s, etc)
- Slices, arrays, maps
- Structs

Exported vs Unexported Fields

WIP: A discussion of why exported fields are required for JSON (un)marshaling.

3.3 Unmarshaling JSON (TODO)

WIP: How to unmarshal simple data to JSON.

- Simple data types (`int`s, `string`s, etc)
- Slices, arrays, maps
- Structs

3.4 Creating a Custom Marshaler (TODO)

WIP: How to implement a `json.Marshaler`.

3.5 Creating a Custom Unmarshaler (TODO)

WIP: How to implement a `json.Unmarshaler`.

3.6 Relationship to the `encoding` Package (TODO)

WIP: When and how to use `encoding.TextMarshaler` and `encoding.TextUnmarshaler` instead of the JSON counterparts.

Streaming JSON (TODO) 4

4.1 `json.Encoder` Options (TODO)

WIP: Discussion of all the features of the `json.Encoder` options and behaviors.

See Chapter 10 on page 48 for recipes related to streaming to a JSON encoder.

4.2 `json.Decoder` Options (TODO)

WIP: Discussion of all the features of the `json.Decoder` options and behaviors.

See Chapter 11 on page 58 for recipes related to streaming to a JSON decoder.

4.3 Streaming Limitations (TODO)

```
func joinJSONObjects(obj1, obj2 []byte) []byte {
    obj2[0] = ',' // Set the first character of obj2 to a comma
    return append(obj1[:len(obj1)-1], obj2...) // Then join
}
```

Caveats

There are a few caveats to consider with respect to this technique.

The first, and most obvious, is that it only works with actual objects. Joining arrays, strings, or numbers this way simply won't work.

A related point is that this will not work with null objects, although it should be fairly straight forward to detect and work around this with a simple check, in the cases where it matters. As an example:

```
func joinJSONObjects(obj1, obj2 []byte) []byte {
    if string(obj1) == "null" {
        return obj2
    }
    if string(obj2) == "null" {
        return obj1
    }
    obj2[0] = ','
    return append(obj1[:len(obj1)-1], obj2...)
}
```

One other limitation of this exact approach is that it assumes no extra whitespace before or after the respective objects. This, too, could be accounted for, but unless you're combinging objects from an untrusted source, this should rarely be necessary. As the recipes in this book merge JSON objects fresh out of `json.Marshal`, so this is a safe assumption for this context.

Part III

Recipes

7 Working with Simple Types (TODO)

7.1 Standard Types (TODO)

WIP: How to write a custom encoder and decoder for standard types, using `time.Time` as an example.

7.2 Numbers (TODO)

WIP: Advanced handling of numbers in JSON, using the standard library.

Working with Complex Types 8

WIP: How to write a custom encoder and decoder for standard types, using `time.Time` as an example.

8.1 Wrapping Existing Types

Background

A common request is the ability to extend an existing type for custom marshaling or unmarshaling. Of course, Go does not allow type inheritance, so this traditional Object-Oriented approach is not available to us. But not all hope is lost.

We can embed an existing type in our own type, to allow the application of custom (un)marshaling logic. To illustrate, I will provide a few examples from the standard library.

Regular Expressions

Background

We want the ability to encode and decode regular expressions. Since regular expressions have a standard text representation, it's reasonable to do this with the text encoding interfaces of the `encoding` package.

For illustration purposes, we'll use this JSON array of regular expressions:

```
[ "cow|horse|pig", "(?i)quack", "https?://example\\.(com|net)/" ]
```

The first limitation we're up against is that the standard library's regular expression library, `regexp`, does not implement any sort of JSON Text marshaling. This means we need to define a type that does, which embeds the standard `regexp.Regexp` type, so that we can easily access all of its methods. We may also want to create our own constructor, to simplify use of our custom type:

```go
type MyRegexp struct {
    regexp.Regexp
}

func Compile(expr string) (*MyRegexp, error) {
    re, err := regexp.Compile(expr)
    if err != nil {
        return nil, err
    }
    return &MyRegexp{*re}, nil
}
```

Unmarshaling a regular expression

It is impossible to directly unmarshal into a `regexp.Regexp` value, because the type contains unexported fields[1]:

> Regexp is the representation of a compiled regular expression. A Regexp is safe for concurrent use by multiple goroutines, except for configuration methods, such as Longest.
>
> ```
> type Regexp struct {
>     // contains filtered or unexported fields
> }
> ```

1: https://golang.org/pkg/regexp/#Regexp

But since our `Compile` function already exists, unmarshaling is simply a matter of parsing the text with that function:

```go
func (r *MyRegexp) UnmarshalText(text []byte) error {
    re, err := Compile(string(text))
    if err != nil {
        return err
    }
    *r = *re
    return nil
}
```

Marshaling a regular expression

Marshaling a regular expression is just as straight forward. We're helped by the fact that `regexp.Regexp` is already a `fmt.Stringer`, so all of the hard work is already done for us. All we must do is call the `String` method from within our marshaler, and do the necessary type conversion:

```go
func(r *MyRegexp) MarshalText() ([]byte, error) {
    return []byte(r.String()), nil
}
```

Listing 8.1: Marshal and Unmarshal Regular Expression

```go
import "regexp"

// MyRegexp embeds a regexp.Regexp, and adds Text/JSON
// (un)marshaling.
type MyRegexp struct {
    regexp.Regexp
}

// Compile wraps the result of the standard library's
// regexp.Compile, for easy (un)marshaling.
func Compile(expr string) (*MyRegexp, error) {
    re, err := regexp.Compile(expr)
    if err != nil {
        return nil, err
    }
    return &MyRegexp{*re}, nil
}

// UnmarshalText satisfies the encoding.TextMarshaler interface,
// also used by json.Unmarshal.
func (r *MyRegexp) UnmarshalText(text []byte) error {
    rr, err := Compile(string(text))
    if err != nil {
        return err
    }
    *r = *rr
    return nil
}

// MarshalText satisfies the encoding.TextMarshaler interface,
// also used by json.Marshal.
func (r *MyRegexp) MarshalText() ([]byte, error) {
    return []byte(r.String()), nil
}
```

`time.Time`

Background

The `time.Time` type from the Go standard library already includes `MarshalText`, `UnmarshalText`, `MarshalJSON` and `UnmarshalJSON` functions, which will work for most users when creating a new system. But since these functions only work with RFC 3339[1] formatted time strings, this obviously won't work in every situation.

1: `2020-06-25T11:28:08+02:00` is an example of an RFC 3339 time string. See the full spec at `https://tools.ietf.org/html/rfc3339`

Unmarshaling a custom time format

Fortunately, the basic approach is quite straight forward. We simply need to embed the standard `time.Time` type in a custom type, where we can define our own `UnmarshalText` function.

```go
    type MyTime struct {
        time.Time
    }
```

The main complication here is that for the common case, of working with JSON, it is not enough to define our own `MarshalText` and `UnmarshalText` functions, because the JSON functions are still accessible from the embedded value. You might choose to implement *only* the JSON function, if you know you'll only ever use your custom format in a JSON context, but I prefer to cover my bases by implementing both, to reduce the potential for surprises if my custom type is ever used in new ways.

I should also mention that we could obscure the JSON functions for the embedded type (see Section 6.5 (Embedding fields (TODO))), but I prefer to do this only for unexported types, where the subtlety is less likely to be lost. So my advice is to implement custom `MarshalJSON` and `UnmarshalJSON` methods, as well.

To facilitate both `MarshalText` and `MarshalJSON`, as well as future expansion, I will create a simple wrapper for `time.Parse`, to handle our custom format. For illustration purposes, I have used an American-style date string[1].

1: And as an American myself, I strongly encourage you to avoid such date formats whenever possible! If required, use them only when displaying a date to an actual (American) human! Whenever possible, use a less ambiguous format when communicating with another computer

```go
const myTimeLayout = "3:04PM, Jan 06 2006"

func parseMyTime(input string) (MyTime, error) {
    t, err := time.Parse(myTimeLayout, input)
    return MyTime{t}, err
}
```

With this foundation laid, we can now write our two unmarsaling functions.

```go
func (t *MyTime) UnmarshalText(text []byte) error {
    var err error
    *t, err = parseMyTime(string(text))
    return err
}

func (t *MyTime) UnmarshalJSON(data []byte) error {
    if string(data) == "null" {
```

```
 9          return nil
10      }
11      var err error
12      *t, err = parseMyTime(string(data[1 : len(data)-1]))
13      return err
14  }
```

The `UnmarshalText` implementation should be pretty straight forward: We simply pass the text into our custom parser.

The `UnmarshalJSON` method has a couple additions, though.

First, on line 8, we check for a literal `"null"` input, and return a `nil` value in this case. This consideration does not generally apply to `UnmarshalText` since strings cannot be null, but objects in JavaScript (and thus JSON) can. Although in some applications, it might make sense to treat an empty string the same in `UnmarshalText`.

Second, on line 12, you can see that we're reading all but the first and last characters of the `data` variable, as these characters are the quotation marks, which will confuse the parser.

Marshaling a custom time format

As is often the case, producing a custom output is often simpler. Once again, though, it's not enough to implement just `MarshalText`, since `MarshalJSON` is also implemented on the embedded type.

```
func (t MyTime) MarshalText() ([]byte, error) {
    return []byte(t.Format(myTimeLayout)), nil
}

func (t MyTime) MarshalJSON() ([]byte, error) {
    str := t.Format(myTimeLayout)
    return []byte(`"` + str + `"`), nil
}
```

If performance is critical, some improvements can be made here, particularly by allocating a properly-sized byte slice in `MarshalJSON`, but I have omitted this for clarity in this example. I refer you to the standard library's implementation[2], for an example.

2: https://golang.org/pkg/time/#Time.MarshalJSON

Unmarshaling multiple formats

A very common request is to support multiple input formats. This is also easily accomplished with a custom unmarshaler function.

For the example, let us suppose that we want to support any of a number of common formats: RFC 822, used in email headers, RFC 850, used by Usenet, RFC 3339, and our own custom time format, described above. Any other format should produce an error.

The first thing is to define a list of our acceptable time formats. I do this as an unexported package variable, since Go does not allow slice or array constants:

```go
var supportedLayouts = []string{
    myTimeLayout,
    time.RFC822Z,
    time.RFC850,
    time.RFC3339Nano,
}
```

Then we simply update our custom parsing function to iterate through the supported formats until we find success, or exhaust our options.

```go
func parseMyTime(input string) (MyTime, error) {
    for _, layout := range supportedLayouts {
        t, err := time.Parse(myTimeLayout, input)
        if err == nil {
            return t, nil
        }
    }
    return MyTime{}, errors.New("unsupported time format")
}
```

One thing to note here is that this does discard some error context. When calling the standard `time.Parse` function (or our original implementation of `parseMyTime`), an error would include some descriptive text such as "hour out of range" or "cannot parse "2020-06-25T13:11:56.821960492+02:00" as "Monday"". This is not possible when trying multiple formats, since we expect at most one format to succeed.

Another potential drawback to supporting multiple time formats is the issue of ambiguous formats. While the layout strings supported by the `time` package allow for unambiguous output of times. As an example, suppose we support these two layouts:

- `2006-01-02 15:04:05`
- `2006-02-01 15:04:05`

The only difference is the order of the month and day. For a date input like `2020-12-30 12:15:03`, there is no ambiguity—the input only works for the first format. But for `2020-02-10 19:50:00`, it's unclear whether the date refers to February 10, or October 2.

Of course this problem is not unique to our brand of custom unmarshaler. And unfortunately, there is no good solution to this, other than to know which format to accept.

Marshaling multiple formats

The reverse of reading multiple formats—producing multiple formats—is much less obvious. Fortunately, it should also rarely ever be needed. If we follow the Robustness principle[3], "Be conservative in what you send, be liberal in what you accept", we should always be content producing a single date format. The most obvious exception would be a user interface, which supports different locales, but this falls well beyond the scope of data serialization, so isn't discussed here.

3: Formally **Postel's law**, stated *"Be conservative in what you do, be liberal in what you accept from others"*, `https://en.wikipedia.org/wiki/Robustness_principle`

If you really must support different output formats for the same date string, you might consider applying the approaches discussed in Section 6.3 (Configurable Marshalers (TODO)).

Listing 8.2: Marshal and Unmarshal Custom Time Formats

```go
import (
    "errors"
    "time"
)

// MyTime embeds a time.Time, and adds Text/JSON
// (un)marshaling.
type MyTime struct {
    time.Time
}

// myTimeLayout is a custom time formatting string, which
// resembles a format likely used by an American.
const myTimeLayout = "3:04PM, Jan 06 2006"

// supportedLayouts are the time layouts supported by the MyTime
// custom parser.
var supportedLayouts = []string{
    myTimeLayout,
    time.RFC822Z,
    time.RFC850,
    time.RFC3339Nano,
}

// parseMyTime wraps time.Parse, and cycles through the supported
// layouts until input is successfully parsed.
func parseMyTime(input string) (MyTime, error) {
    for _, layout := range supportedLayouts {
        t, err := time.Parse(layout, input)
        if err == nil {
            return MyTime{t}, nil
        }
    }
    return MyTime{}, errors.New("unsupported time format")
}

// UnmarshalText satisfies the encoding.TextUnmarshaler interface.
func (t *MyTime) UnmarshalText(text []byte) error {
    var err error
    *t, err = parseMyTime(string(text))
    return err
}

// UnmarshalJSON satisfies the json.Unmarshaler interface.
func (t *MyTime) UnmarshalJSON(data []byte) error {
    if string(data) == "null" {
        return nil
    }
    var err error
    *t, err = parseMyTime(string(data[1 : len(data)-1]))
    return err
}

// MarshalText satisfies the encoding.TextMarshaler interface.
func (t MyTime) MarshalText() ([]byte, error) {
    return []byte(t.Format(myTimeLayout)), nil
}
```

```go

// MarshalJSON satisfies the json.Marshaler interface.
func (t MyTime) MarshalJSON() ([]byte, error) {
    str := t.Format(myTimeLayout)
    return []byte(‘"‘ + str + ‘"‘), nil
}
```

8.2 JSON Array as Go Struct

Background

Most JSON you find in the wild uses objects when different types of data are required. A contrived, but believable example:

```
{
    "status": 404,
    "result": "error",
    "reason": "Not Found"
}
```

But sometimes you'll find this exact same data expressed in a different way, particularly if the producer of the JSON is a loosely-typed language such as JavaScript or Python:

```
[ 404, "error", "Not Found" ]
```

I could probably write several pages about why I dislike this approach, but instead I'll focus on how to work with this (backwards) JSON in Go, without going insane.

The challenge

The natural approach to using the first example in Go would be quite straight forward:

```go
type Result struct {
    Status int    `json:"status"`
    Result string `json:"status"`
    Reason string `json:"status"`
}
```

No further explanation needed.

For the second example, though, we're a bit stuck. The naïve solution is to use a slice, but we have different types of data, so we're forced to use a slice of `interface{}`:

```go
type Result []interface{}
```

But this is ugly and annoying for a number of reasons:

1. A type assertion is required to get at the underlying data
2. The number* and position of elements is implicit
3. No type safety

*: We could, of course, use an array instead of a slice, to have a fixed number of elements, but arrays are slightly more cumbersome to use, and still don't solve

Go Proverb

`interface{}` says nothing

```go
var data = []byte(`[ 404, "error", "Not Found" ]`)
var r Result
if err := json.Unmarshal(data, &r); err != nil {
    log.Fatal(err)
}
// Ugh, so ugly and fragile!
fmt.Println("Status code %d, reason: %s\n", r[0].(float64), r[2].(
    string))
```

Custom unmarshaler to the rescue

With a custom unmarshaler, we can solve this problem by unmarshaling to a slice as an intermediate step, on our way to a proper struct.

```go
type Result struct {
    Status int
    Result string
    Reason string
}

func (r *Result) UnmarshalJSON(p []byte) error {
    var tmp []interface{}
    if err := json.Unmarshal(p, &tmp); err != nil {
        return err
    }
    r.Status = int(tmp[0].(float64))
    r.Result = tmp[1].(string)
    r.Reason = tmp[2].(string)
    return r
}
```

With this new function in place, it is now possible to unmarshal the JSON array "directly" into a Go struct:

```go
var data = []byte(`[ 404, "error", "Not Found" ]`)
var r Result
if err := json.Unmarshal(data, &r); err != nil {
    log.Fatal(err)
}
// Not ugly! Not fragile!
fmt.Println("Status code %d, reason: %s\n", r.Status, r.Reason)
```

This is great, but there's still room for improvement. You may have noticed that we've just moved the ugly and fragile code into the custom `UnmarshalJSON` function. This is still a worthy improvement—at least that special knowledge is contained in one place, rather than spread around your code base.

But we can do better.

Eliminating the type assertions

The first thing I'd like to eliminate is the various type assertions. These are fragile, because if we ever receive an unexpected input, our code is liable to panic, rather than return a proper error.

One option would be to use the two-value type assertion format, and this would improve safety of the operation:

```go
    r.Status, ok = int(tmp[0].(float64))
    if !ok {
        return errors.New("not a float64!")
    }
```

To be sure, this is an improvement, but we can still do much better. In a case like this, I would rather return a standard JSON error, rather than a custom error of my own invention. This is easily done with just a couple small tweaks.

First, we'll use the `json.RawMessage` type in our slice, rather than the empty interface. Second, we'll unmarshal each element of the slice directly into our target slice. This will allow the standard `encoding/json` package to do standard error handling for us.

```go
func (r *Result) UnmarshalJSON(p []byte) error {
    var tmp []json.RawMessage
    if err := json.Unmarshal(p, &tmp); err != nil {
        return err
    }
    if err := json.Unmarshal(tmp[0], &r.Status); err != nil {
        return err
    }
    if err := json.Unmarshal(tmp[1], &r.Result); err != nil {
        return err
    }
    if err := json.Unmarshal(tmp[2], &r.Reason); err != nil {
        return err
    }
    return r
}
```

This is a bit longer, and a bit more repetitive, but so much more robust!

Handling inputs of unexpected length

There's still one glaring weakness to this implementation: What if we receive an input of more or fewer than 3 elements?

Suppose we were to receive the following input:

```
[ 404, "error" ]
```

The above implementation would panic with "index out of range [3] with length 2".

So let us add an explicit check the size of the input, and return an error if an unexpected value is received.

```go
type Result struct {
    Status int    ‘json:"status"‘
    Result string ‘json:"status"‘
    Reason string ‘json:"status"‘
}

func (r *Result) UnmarshalJSON(p []byte) error {
    var tmp []json.RawMessage
    if err := json.Unmarshal(p, &tmp); err != nil {
        return err
    }
    if l := len(tmp); l != 3 {
        return fmt.Errorf("Expected 3 elements, got %d", l)
    }
    if err := json.Unmarshal(tmp[0], &r.Status); err != nil {
        return err
    }
    if err := json.Unmarshal(tmp[1], &r.Result); err != nil {
        return err
    }
    if err := json.Unmarshal(tmp[2], &r.Reason); err != nil {
        return err
    }
    return r
}
```

Reversing the process

Reversing the process is somewhat simpler, as is often the case.

```go
func (r *Result) MarshalJSON() ([]byte, error) {
    return json.MarshalJSON([]interface{}{r.Status, r.Result, r.
    Reason})
}
```

Arrays of variable length

Let's suppose that we want to allow varying-length input. For example, perhaps this same API returns only two elements in the case of success:

```
[ 200, "success" ]
```

Here we have a couple of options. One quick-and-dirty fix to avoid a panic would be to use an array instead of a slice:

```go
func (r *Result) UnmarshalJSON(p []byte) error {
    var tmp [3]json.RawMessage
    if err := json.Unmarshal(p, &tmp); err != nil {
        /* ... snip ... */
```

However, this approach suffers from a couple few potential weaknesses. Perhaps most important, it's not very expressive, and the subtle behavior difference can be easily missed when reading the code. It also allocates elements for the entire array, whether they're ultimately needed or not (admittedly not important in this example, but this could matter for larger inputs, or in heavily-used code). And finally, this approach provides no safety at all for long input. From the `encoding/json` documentation[1]:

1: https://golang.org/pkg/encoding/json/#Unmarshal

> To unmarshal a JSON array into a Go array, Unmarshal decodes JSON array elements into corresponding Go array elements. **If the Go array is smaller than the JSON array, the additional JSON array elements are discarded.** If the JSON array is smaller than the Go array, the additional Go array elements are set to zero values.

All things considered, the best approach is usually to be as explicit as possible. With a couple of modifications, we have a working example that allows either 2 or 3 inputs, and returns an error in any other case.

```go
type Result struct {
    Status int    ‘json:"status"‘
    Result string ‘json:"status"‘
    Reason string ‘json:"status"‘
}

func (r *Result) UnmarshalJSON(p []byte) error {
    var tmp []json.RawMessage
    if err := json.Unmarshal(p, &tmp); err != nil {
        return err
    }
    if l := len(tmp); l < 2 || l > 3 {
        return fmt.Errorf("Expected 2-3 elements, got %d", l)
    }
    if err := json.Unmarshal(tmp[0], &r.Status); err != nil {
        return err
    }
    if err := json.Unmarshal(tmp[1], &r.Result); err != nil {
        return err
    }
    if len(tmp) > 2 {
        if err := json.Unmarshal(tmp[2], &r.Reason); err != nil {
            return err
        }
    }
    return r
}
```

And we need to be sure to produce valid outputs, too:

```
func (r *Result) MarshalJSON() ([]byte, error) {
    if r.Reason == {
        return json.MarshalJSON([]interface{}{r.Status, r.Result,
     r.Reason})
    }
    return json.MarshalJSON([]interface{}{r.Status, r.Result})
}
```

Listings

Alphabetical Index