THE JSON SURVIVAL KIT

BRIAN HICKS

The JSON Survival Kit

Brian Hicks

Tweet This Book!

Please help Brian Hicks by spreading the word about this book on Twitter!

The suggested hashtag for this book is #jsonsurvivalkit.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#jsonsurvivalkit

Contents

You Are Here

Welcome to *The JSON Survival Kit*! In this book, you'll find real-world problems and their solutions.

About You

This book is for you if you're stuck on JSON Decoding for any reason. If you just want to get past decoders stage and on to the rest of your app, welcome!

On the other hand, this book is *not* for you if you're looking for a general overview of Elm. Pick up Richard Feldman's excellent *Elm in Action*[1] for that.

About Me

I'm Brian Hicks. Hi. I organize elm-conf US, run the State of Elm survey, and blog about it all at brianthicks.com[2].

I discovered Elm by watching Richard Feldman's *Make The Backend Team Jealous*[3]. As a backend person myself, I was *very* jealous of Elm's type safety and runtime guarantees. But as I tried to implement my own Elm apps, I kept butting up against the JSON Decoding syntax. I eventually got past them, but it turns out everyone runs into the same small(ish) set of problems.

Actually, the fact that you're reading this book means that you've probably run into them yourself! So, let's get started!

How To Read This Book

The JSON Survival Kit is structured as a series of situations and their solutions. If you're the type who prefers to jump around, that will work fine; each chapter is more-or-less self contained. That said, if you're more the type who would prefer to read straight through from beginning to end, that will work too.

We'll conclude with several real-world JSON decoder examples. These use as much of the `Json.Decode` API as appropriate, and are broken down so you can see how all the parts work.

Code Blocks

Code blocks look like this:

[1] https://www.manning.com/books/elm-in-action

[2] https://www.brianthicks.com

[3] https://www.youtube.com/watch?v=FV0DXNB94NE

```
1  decodeString bool "true" == Ok True
```

When we're talking about decoding strings directly, we'll follow the pattern from the Elm documentation. Code blocks will have a call followed by == and the result of the call. If we need to use more than a single statement, the result will be in a comment (--) instead.

The code samples assume that you've imported all of `Json.Decode` into the current namespace with `import Json.Decode exposing (..)`. You generally won't want to do this, but referring to the namespace everywhere in the whole book gets clunky. In actual code, I generally import like `import Json.Decode as Decode exposing (Decoder)`. We'll talk more about this in the integrations chapter.

Just Enough to be Dangerous: Decoding From A String

We're going to be doing a lot of decoding in this book, so let's talk about the mechanism we'll be using to do it. `decodeString` takes a decoder and a JSON string, and returns a `Result` type. The result is either an error string, or the JSON value that the decoder decodes.

We'll go over this in more depth in the integrations chapter, but we'll need to use it right away to get to the "real world". For the sake of example, let's try to decode a bool value with the built in `bool` decoder. (Don't worry if you don't get that yet, it's explained in the next chapter.)

When we call `decodeString` with a matching decoder and JSON string, we get an `Ok` result:

```
1  decodeString bool "true" == Ok True
```

We got `Ok True` here, which indicates the decode was successful. If we try the same call with a number in our JSON string instead of a boolean, we'll get an error:

```
1  decodeString bool "42" == Err "Expecting a Bool but instead got: 42"
```

In this case we get an Error value, called `Err`. The error contains a friendly enough error value to figure out what happened.

In short, `decodeString` gives us a `Result`. Specifically, it's a `Result String a`, where `a` is the type of our Decoder. And speaking of, let's see how those decoders work...

When Would I Even Use `int`? (And Other Base Values)

You know that feeling you get when you look at the `Json.Decode` docs? The feeling that says "why are there all these tiny decoders?"

You might have even been tempted, as I was, to use `String.toInt` instead of `decodeString int`. The type signatures are the same, after all. Why bother with all the tiny little decoders?

And this is a fair point! It'll get you pretty far… if you're only using simple values, like ints and strings.

Once you're dealing with bigger values, like lists or objects, you need something more powerful. That's where `Json.Decode` really shines.

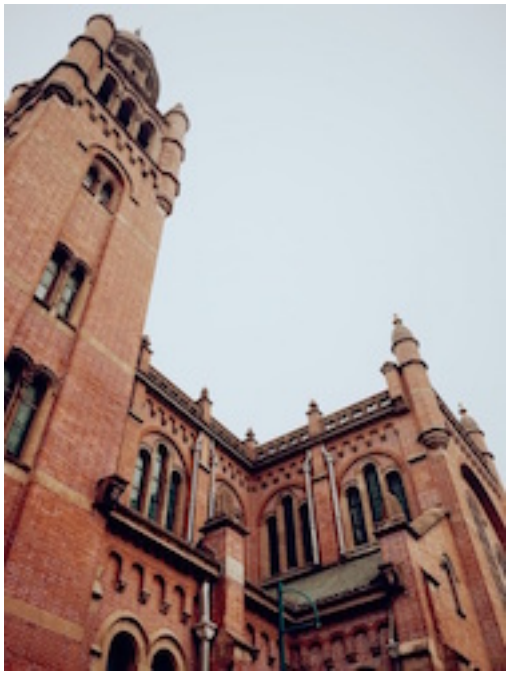
But to get there, you've got to readjust your mindset. You've got to think… with bricks.

Build Your Decoders Brick By Brick

A single brick isn't useful on it's own. I mean, you could put it on the ground and wait, but… meh.

Same with a decoder like `int`. "Oh, a single integer! How wonderful!" Again, meh.

But when you want to actually build something, bricks are exactly what you need. The trick is to know where you're going!



You'll end up with a building!

If you throw out the whole concept because one brick alone isn't interesting then you'll never get your castle, or school, or fancy mansion.

`Json.Decode` is the same way. The little decoders get composed into bigger and bigger decoders, until they're exactly the shape of you need. `int` may not be very useful on it's own, but `list int` is more useful, and `field "ids" (list int)` more useful still. See how they build up?

But before you can build your castle, you've got to adjust how you think about bricks.

The Decoder Type

To see how we can stack our bricks, let's look at the type they all implement: `Decoder`. It looks like this:

```
1  type Decoder a
```

The `a` in `Decoder a` means that we can plug whatever type we like in there. So `int` is a `Decoder Int`, and `string` is a `Decoder String`. Our basic values are `int`, `float`, `string`, `bool`, and `null`, corresponding to the JSON types.[4] Each decoder defines what it parses out of your JSON, and the type that it fulfills in Elm. This means that since `decodeString` takes any `Decoder a`, we can plug in whatever we want. Let's try it with a simple decoder, `int`:

[4] Since the JSON spec only specifies floating point numbers, the `Json.Decode` API lets us tell it which we want with `int` or `float`. Elm will *mostly* do what you want, but `int` won't truncate information past the point. That is, parsing `3.14` as an `Int` results in an error, but `3.0` will work fine.

```
decodeString int "1" == Ok 1
```

In this case, we get an `Int` back, since the `int` says it's a `Decoder Int`. If we pass in a wrong value, like so...

```
decodeString int "true" == Err "Expecting an Int but instead got: true"
```

...we get an error. The decoder we passed in informs the return type of `decodeString`. Since our decoder is a `Decoder Int`, the return type is `Result String Int`.[5]

But we still haven't seen how this is better than `String.toInt`. So let's get composing!

Building Your JSON Castle

A good many of the decoders take *other* decoders as input and use them to create bigger decoders. `list` is a good good example. It looks like this:

```
list : Decoder a -> Decoder (List a)
```

This reads as: `list` takes some `Decoder a` and uses that `a` to inform what type should go in the resulting list. So if we give it the `int` decoder, we get `Decoder (List Int)`. We can use this with `decodeString` just like before:

```
decodeString (list int) "[1]" == Ok [1]
```

This works with `dict` too. `dict` gives you a `Decoder (Dict String a)` for whatever `Decoder a` you pass in:

```
dict : Decoder a -> Decoder (Dict String a)
```

To use it:

```
decodeString (dict int) "{\"x\": 1}" == Ok (Dict.fromList [("x", 1)])
```

Now, since all of the functions in the API take decoders, wrap them, and return them, we can compose however we'd like. For example, if we have a `List` of `Dicts` of `Ints`:

[5] The first type in `Result String Int` is the error, which is always a `String` for `decodeString`. The second type, `Int` is the part that varies.

```
decodeString (list (dict int)) "[{\"x\": 1}]" == Ok [Dict.fromList [("x", 1)]]
```

Since `dict` returns a `Decoder`, we can just pass it to `list`. It works the other way, too. Suppose we had a object whose values were lists of ints:

```
decodeString (dict (list int)) "{\"x\": [1]}" == Ok (Dict.fromList [("x", [1])])
```

As long as you can express the type you want in a `Decoder` you can decode to whatever you like. This basic pattern, repeated, will build us our castle—er, I mean JSON Decoder. We'll cover more of these in the next couple of chapters.

The JSON Survival Kit

Did you enjoy this sample? Then you'll love the full book. Just to whet your appetite, here's the full chapter listing:

- When Would I Even use `int`? (And Other Base Values) *(you just read this)*
- Get What You Need Out of JSON Objects
- Beyond The Basics
- JSON in the Real World

On top of that, the full book includes a reading plan and exercises for each chapter.

When you're ready to get yourself unstuck from your JSON-based quagmire, buy the book on my site[6]. If you're not quite ready, or have more questions, I want to hear from you! Email me[7] and I'll get back to you as soon as I can.

See you soon!

Brian Hicks

[6] https://www.brianthicks.com/json-survival-kit/

[7] brian@brianthicks.com