Table of Contents

Hey there!

This is a sample of *The JSON Survival Kit*. It includes chapter 2, *Get What You Need Out of JSON Objects*. You can get the full book at https://www.brianthicks.com/json-survival-kit.

If you have any questions, please email me at brian@brianthicks.com. I'm more than happy to help if anything is unclear or confusing.

Enjoy!

Brian Hicks

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* 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.

I discovered Elm by watching Richard Feldman's *Make The Backend Team Jealous*. 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.

Formats

Code Blocks

Code blocks look like this:

```
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:

```
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:

```
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…

Get What You Need Out of JSON Objects

We've come to our first example, and so we need some JSON data. Let's use GitHub!

```json
{
  "login": "octocat",
  "id": 583231,
  "avatar_url": "https://avatars.githubusercontent.com/u/583231?v=3",
  "gravatar_id": "",
  "url": "https://api.github.com/users/octocat",
  "html_url": "https://github.com/octocat",
  "followers_url": "https://api.github.com/users/octocat/followers",
  "following_url": "https://api.github.com/users/octocat/following{/other_user}",
  "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}",
  "starred_url": "https://api.github.com/users/octocat/starred{/owner}{/repo}",
  "subscriptions_url": "https://api.github.com/users/octocat/subscriptions",
  "organizations_url": "https://api.github.com/users/octocat/orgs",
  "repos_url": "https://api.github.com/users/octocat/repos",
  "events_url": "https://api.github.com/users/octocat/events{/privacy}",
  "received_events_url": "https://api.github.com/users/octocat/received_events",
  "type": "User",
  "site_admin": false,
  "name": "The Octocat",
  "company": "GitHub",
  "blog": "http://www.github.com/blog",
  "location": "San Francisco",
  "email": "octocat@github.com",
  "hireable": null,
  "bio": null,
  "public_repos": 7,
  "public_gists": 8,
  "followers": 1667,
  "following": 6,
  "created_at": "2011-01-25T18:44:36Z",
  "updated_at": "2016-12-23T05:44:24Z"
}
```

This is representative of a real-world JSON response in that it has a *ton* of fields. We'll be referring to this Github user JSON data as a string named `github`.

Our Types

Now, your first approach to this JSON might be to create a record that encodes *all 30 fields*. That was my first reaction too! But, as it turns out, that's not necessary.

Your application probably doesn't need to use every single field, so why would you decode them all? There's nothing in the `Json.Decode` API that says you have to decode everything, or even stick to the order of the fields in your JSON. Your record type is the most important thing for your `Decoder`s; the JSON itself only matters as a repository to pull data from.

So, all that said, we're going to create a much smaller user record:

```
type alias GithubUser =
    { id : Int
    , login : String
    , name : String
    , gravatarId : Maybe String
    }
```

I've kept the field names almost the same as their JSON counterparts, but that's optional too.

Reach Into JSON Objects with `field` and `at`

Remember how we can compose JSON `Decoder`s together? Before, we had `int : Decoder Int`, and `list : Decoder a -> Decoder (List a)`. It seems reasonable to assume that every new decoder you compose would change the type of `Decoder`, but that's not always the case. In fact, it's handy that we *don't* have to do that.

`field`[1] and `at` are prime examples. Rather than decoding a value, these decoder functions change how Elm finds the value.

`field`

Let's look at `field` first. It takes a string for the field to look at, and a decoder to use on the value it finds. The signature is `field : String -> Decoder a -> Decoder a`.

So if we wanted to get the login name from the Github user data we used before:

```
decodeString (field "login" string) github == Ok "octocat"
```

By the same token, if we ask `field` to get a field that doesn't exist, it gives us an error:

```
Err "Expecting an object with a field named `schadenfreude` but intead got {}"
```

You'll also get the "normal" error from the decoder if the value doesn't match the type of the decoder.

`at`

`at` works almost exactly the same as `field`, except it takes a list of strings. `at` uses these strings, one at a time, to traverse the JSON object to find a value. That means these two calls give exactly the same result:

```
decodeString (field "login" string) github == Ok "octocat"
decodeString (at ["login"] string)  github == Ok "octocat"
```

Where `field` can only handle a single field name, `at` can handle as many as you need. We can use this to reach deep in to JSON objects. So if we have some JSON that looks like this:

```
{
  "x": {
    "y": {
      "z": "Hello, World!"
    }
  }
}
```

We could get the "Hello, World!" string like this:

```
decodeString (at ["x", "y", "z"] string) sample == Ok "Hello, World!"
```

At each level, the decoder looks at the object it got from the previous level and evaluates the fields there. You might think, "hey, I could just do that by chaining `field` calls together!" and you'd be right. That is, in fact, how `at` is implemented. But using it is *much* easier than writing your own folds.

> 1. In Elm 0.17 and prior, `field` was an operator called `(:=)`. ↩

Use The Fields You Have To Get The Records You Want

We have a `gravatarId` in `GithubUser`. In the example above it's empty, but that's not a useful value! If we represent the field as `Maybe String`, it will prevent us from requesting a Gravatar with no ID. When `gravatarId` is an empty string we'll represent it as `Nothing`. Anything else will be `Just value`.

Let's take care of that with `map`. `map` looks like this:

```
map : (a -> value) -> Decoder a -> Decoder value
```

It takes a function to transform what we get into what we want, and a decoder to get that initial value. It only applies the function to the result of the initial decoder if the decoder succeeded.

We can use `map` to define a custom `Decoder` for `gravatarId`:

```
gravatarId : Decoder (Maybe String)
gravatarId =
    map (\id -> if id == "" then Nothing else Just id) string
```

We want to return `Nothing` if the string is empty, so we make that check in a conditional. The compiler keeps us honest here; if we substituted `int` for `string` as our decoder, we would get an error message.

```
Function `map` is expecting the 2nd argument to be:

    Decoder String

But it is:

    Decoder Int

Hint: I always figure out the type of arguments from left to right. If an
argument is acceptable when I check it, I assume it is "correct" in subsequent
checks. So the problem may actually be in how previous arguments interact with
the 2nd.
```

If you get these messages and you're not sure what's going on, try moving your map function to a separate function and adding a type annotation.

```
maybeId : String -> Maybe String
maybeId id =
    if id == "" then
        Nothing
    else
        Just id


gravatarId : Decoder (Maybe String)
gravatarId =
    map maybeId string
```

By specifying your types, you let the compiler (and future you) know what you're expecting to get. This makes it much easier to debug.

Now that we have our decoder, let's apply it decoder to some sample values:

```
decodeString gravatarId "\"\""         == Ok Nothing
decodeString gravatarId "\"id\"" == Ok (Just "id")
```

Of course, we can still pass this bad values so it complains:

```
decodeString gravatarId "1" == Err "Expecting a String but instead got: 1"
```

Note that we got `string` 's error message. Our custom conversion function was never called because the decoder couldn't parse a value for us to use.

We can also combine this with `field` to pull the Gravatar ID out of our user data:

```
decodeString (field "gravatar_id" gravatarId) github == Ok Nothing
```

Stack those bricks!

`map2` and Friends

And speaking of stacking bricks, we've finally arrived at objects. Hooray!

`map2` through `map8` [1] work almost exactly like `map` , but with more arguments and decoders. It may help to look at the type of `map2` :

```
map2 : (a -> b -> value) -> Decoder a -> Decoder b -> Decoder value
```

To determine which arguments will go where, match the `a` and `b` to `Decoder a` and `Decoder b` . The pattern holds with `map3` and `map8` (which has `a` through `h` .)

This means that you can write any function that takes two (through eight) values and use it to combine the results of decoders. As an example, let's get a tuple with the login name and given name for our Github user:

```
loginAndName : Decoder (String, String)
loginAndName =
    map2 (,)
        (field "login" string)
        (field "name" string)


decodeString loginAndName github -- Ok ("octocat", "monalisa octocat")
```

Breaking this down:

1. The comma operator creates a tuple which takes `a` and `b`. We pass this as the function to use for `map2` (remembering that we can get a "regular" function out of an operator by wrapping it in parentheses.)
2. We then provide the `a` and `b` for our tuple by telling `map2` to look at the "login" and "name" fields and give us the strings within.
3. Then we're done, so we pass this whole thing to `decodeString`, where it gets turned into a tuple containing `("octocat", "monalisa octocat")`.

> There were a bunch of new interactions going on above, and we're about to go deeper. Take a minute and make sure you understand what just happened before going on. Open up `elm repl` and play around with the values if you need to.

We'll get records out of these functions by relying on the fact that records come with their own constructors. The usual rules of currying apply here too, as we'll see in a minute.

In our specific case, we have a function `GithubUser` that looks like this:

```
GithubUser : Int -> String -> String -> Maybe String -> GitHubUser
```

Again, we didn't do anything special to get this; Elm makes a constructor like this for every record. The order of arguments is the same as the order the fields appear in the source. To find yours, inspect it in the REPL or just follow the order in your source.

We can use that as the first argument to `map4` (because the function takes four arguments.) We're going to call this `githubUser` (note the lowercase g.) This is conventional for decoders, but you could call it anything you like.

```
githubUser : Decoder GithubUser
githubUser =
    map4 GithubUser
        (field "id" int)
        (field "login" string)
        (field "name" string)
        (field "gravatar_id" gravatarId)
```

Then just use it like any other decoder:

```
decodeString githubUser github
    == Ok { id = 1
          , login = "octocat"
          , name = "monalisa octocat"
          , gravatarId = Nothing
          }
```

While we're composing, Elm will typecheck our decoder in it's entirety. If we violate any of the types, including the mapping of `Decoder a` to `a` in the map function, Elm will tell us with it's famously friendly error messages.

> 1. Before 0.18, these were `object2` through `object8`. ↩

How Can I Decode A JSON Object With More Than 9 Fields?

Now, if you wanted to decode all 30 fields in our object, `map2` through `map8` wouldn't do you any favors. You could try to compose your calls carefully to get exactly as many as you needed, but don't bother. NoRedInk has published a package called `Json.Decode.Pipeline` that will let you grow and shrink your object decoders as much as you need. Add it to your project with `elm package install NoRedInk/elm-decode-pipeline`.

The main benefit of using `Json.Decode.Pipeline` is that you can set up a pipeline with `|>` to create your decoder. This tends to be easier to understand. Here's how our `githubUser` decoder would look:

```
import Json.Decode.Pipeline exposing (decode, required)


githubUser : Decoder GithubUser
githubUser =
    decode GithubUser
        |> required "id" int
        |> required "login" string
        |> required "name" string
        |> required "gravatar_id" gravatarId
```

The API has to change slightly to accommodate the pipeline style. Instead of `mapN`, we use `decode`. This style doesn't have to change for different numbers of fields the way `mapN` does. We can add or remove fields as necessary and Elm will type check for us.

We're using `required` in each pipeline step. `required` is exactly like `field`, but it works with pipelines. The API also provides `requiredAt`, the pipeline version of `at`.

Instead of matching decoders to function arguments all at once as `mapN` does, pipelines deal with one argument at a time. They do this by relying on currying. In the case of a map function with two arguments `a -> b -> c`, applying a single argument `a` will give us a new function `b -> c`. But since the partially applied function is itself a value, the type signatures for `Json.Decode.Pipeline` refer to it as `a -> b` no matter how many arguments are left. If you get a confusing error message, remember that the value of `b` might be a partially applied function!

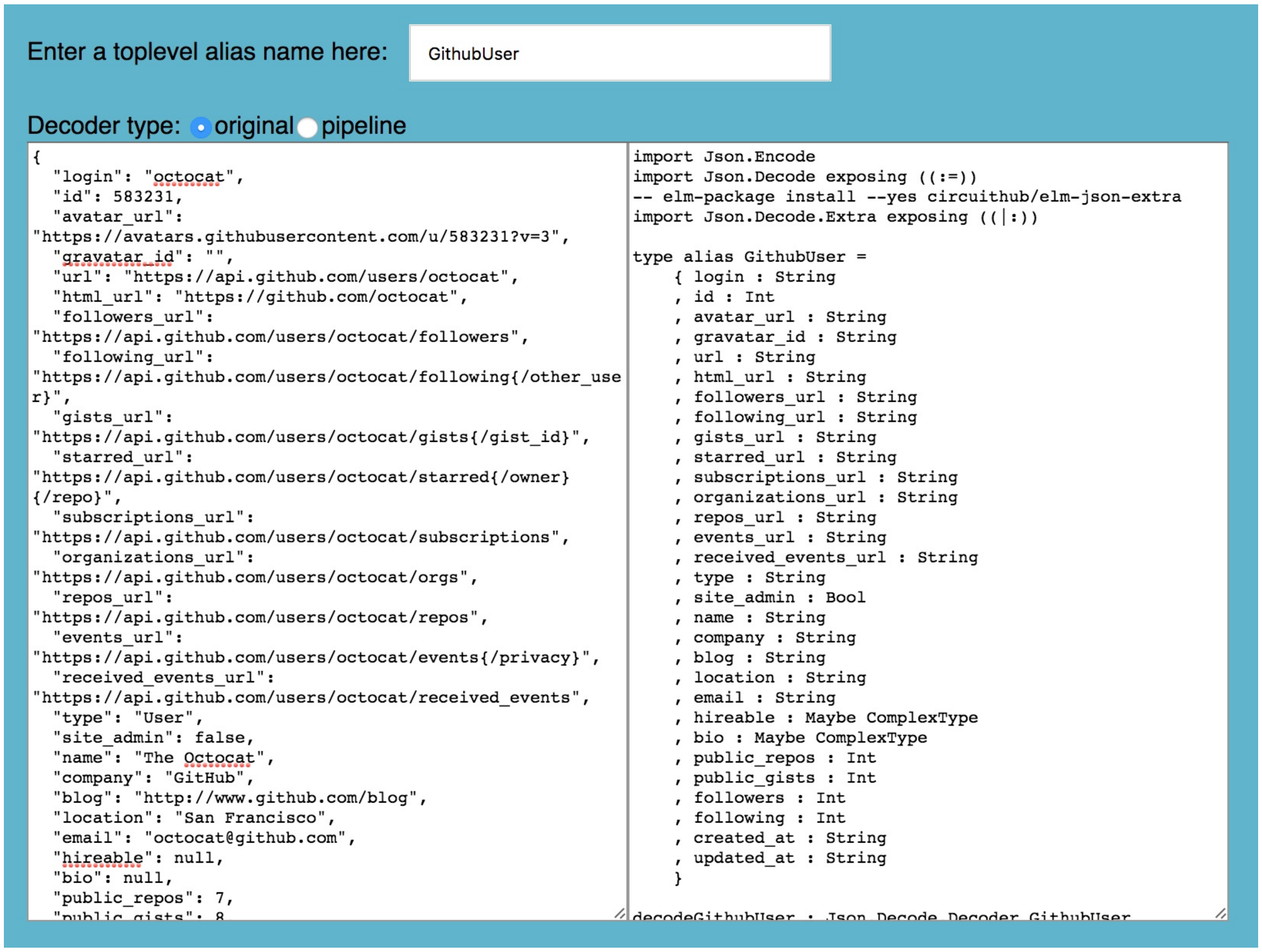
That said, the pipeline style generally is more readable and understandable than `mapN`, even if it is a little harder to explain everything that's going on. Pipelines are easier to change, and you won't have make a change if you need more than 8 fields. You should use them instead over `mapN` whenever you can.

> The object examples for the rest of this book will use `Json.Decode.Pipeline` instead of `mapN`. We'll assume `decode`, `required`, and `requiredAt` are present for any following code samples. If we use any other pipeline functions, I'll call them out as needed. If you're following along in a repl:M
>
> ```
> import Json.Decode.Pipeline exposing (decode, required, requiredAt)
> ```

Yeah, But Writing Object Decoders Is Tedious!

It's useful to look at all this to understand what's going on, but when you're actually decoding simple objects like we're doing here you don't really want to write decoders, objects, and encoders for every single field by hand. It can get super tedious!

Fortunately, JSON to Elm (json2elm.com) can generate a base decoder from a sample JSON object.



Enter a toplevel alias name here: GithubUser

Decoder type: [✓ original] pipeline

```
{
  "login": "octocat",
  "id": 583231,
  "avatar_url":
"https://avatars.githubusercontent.com/u/583231?v=3",
  "gravatar_id": "",
  "url": "https://api.github.com/users/octocat",
  "html_url": "https://github.com/octocat",
  "followers_url":
"https://api.github.com/users/octocat/followers",
  "following_url":
"https://api.github.com/users/octocat/following{/other_use
r}",
  "gists_url":
"https://api.github.com/users/octocat/gists{/gist_id}",
  "starred_url":
"https://api.github.com/users/octocat/starred{/owner}
{/repo}",
  "subscriptions_url":
"https://api.github.com/users/octocat/subscriptions",
  "organizations_url":
"https://api.github.com/users/octocat/orgs",
  "repos_url":
"https://api.github.com/users/octocat/repos",
  "events_url":
"https://api.github.com/users/octocat/events{/privacy}",
  "received_events_url":
"https://api.github.com/users/octocat/received_events",
  "type": "User",
  "site_admin": false,
  "name": "The Octocat",
  "company": "GitHub",
  "blog": "http://www.github.com/blog",
  "location": "San Francisco",
  "email": "octocat@github.com",
  "hireable": null,
  "bio": null,
  "public_repos": 7,
  "public_gists": 8
```

```
import Json.Encode
import Json.Decode exposing ((:=))
-- elm-package install --yes circuithub/elm-json-extra
import Json.Decode.Extra exposing ((|:))

type alias GithubUser =
    { login : String
    , id : Int
    , avatar_url : String
    , gravatar_id : String
    , url : String
    , html_url : String
    , followers_url : String
    , following_url : String
    , gists_url : String
    , starred_url : String
    , subscriptions_url : String
    , organizations_url : String
    , repos_url : String
    , events_url : String
    , received_events_url : String
    , type : String
    , site_admin : Bool
    , name : String
    , company : String
    , blog : String
    , location : String
    , email : String
    , hireable : Maybe ComplexType
    , bio : Maybe ComplexType
    , public_repos : Int
    , public_gists : Int
    , followers : Int
    , following : Int
    , created_at : String
    , updated_at : String
    }

decodeGithubUser : Json.Decode.Decoder GithubUser
```

Once you have your decoder, you can paste it into your project and then customize. It's a great jump start to getting your object Decoders off the ground.

That said, when you're using JSON to Elm you'll need to be aware of a few things:

First, as of the time of this writing JSON to Elm generates 0.17 compatible code. In particular, it uses `(:=)` instead of `field`. There's a switch at the top which lets you turn on `Json.Decode.Pipeline` generation. You almost always want this anyway, so go ahead and flip it on to avoid the 0.17-and-prior API.

The program also only takes a single JSON value. This means that it doesn't know what the proper value of `null` fields is. In these cases, it will provide a placeholder value that you need to fill in.

Last, JSON to Elm also doesn't introspect values inside values, particularly strings. That means that if you want to parse dates or the Gravatar ID we made above, you'll have to substitute in the proper parser yourself.

Please note that these things are not failings of JSON to Elm. Modeling data in JSON is hard, and we have a limited set of values to encode into. That said, JSON to Elm is a super useful tool and can speed up your development workflow significantly. Give it a try!