Extracted from:

Programming Elm

Build Safe and Maintainable Front-End Applications

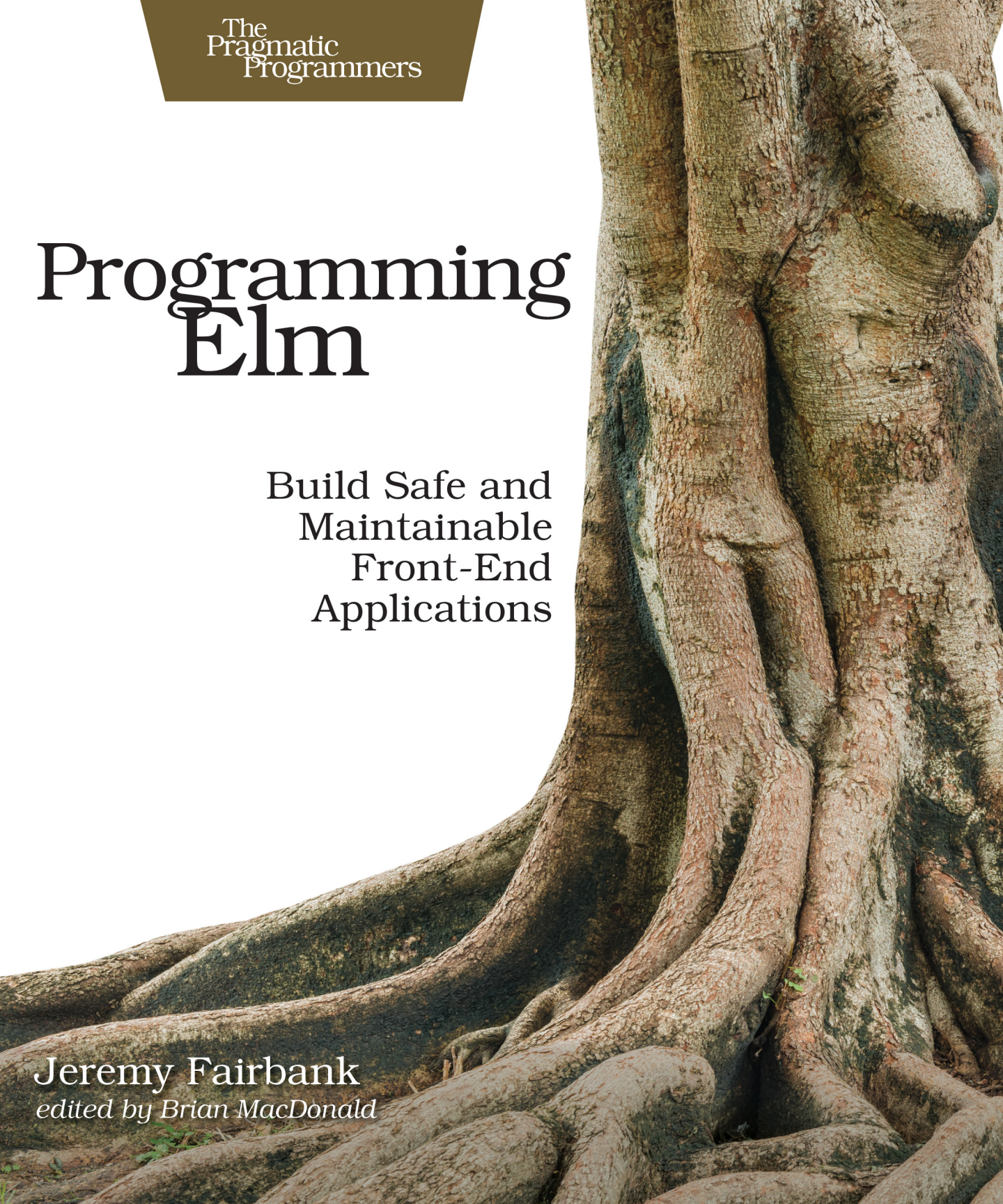
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Programming Elm

Build Safe and Maintainable Front-End Applications



Jeremy Fairbank
edited by Brian MacDonald

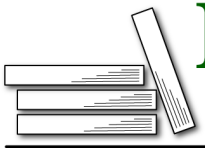
Programming Elm

Build Safe and Maintainable Front-End Applications

Jeremy Fairbank

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Brian MacDonald
Copy Editor: Sean Dennis
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-285-5
Book version: P1.0—June 2019

Safely Decode JSON

Prior to this point, you've been able to stay safely within the confines of Elm's magical world of static types. However, you're going to run into an interesting dilemma if you want to accept an arbitrary JSON payload from a server. Elm doesn't have a `JSON.parse` function like JavaScript because it can't dynamically create records like JavaScript can create objects. In this section, you're going to learn about *JSON decoders*, why they're important, and how to use them to safely convert JSON into a static type Elm can use.

Understand the Problem

To understand why you need JSON decoders, let's look at a couple of example JSON payloads you could use with your application. Visit https://programming-elm.com/feed/1 in your browser. You should see this JSON payload.

```
{
  "id": 1,
  "url": "https://programming-elm.surge.sh/1.jpg",
  "caption": "Surfing",
  "liked": false,
  "comments": ["Cowabunga, dude!"],
  "username": "surfing_usa"
}
```

This JSON closely mimics the photo record type you created in the previous chapter. The only differences are that the JSON payload has an `id` property and a `username` property and lacks a `newComment` property. You could easily fix your static type to include `id` and `username` fields. The `newComment` property also isn't a problem because you only use it locally to temporarily store a typed comment.

Even with those changes, you still can't trust an arbitrary API payload. Elm is pure and safe, and some of its guarantees come from guarding your application from the outside world. If the JSON payload doesn't match what's expected in the record type, you will have a serious problem. For example, let's assume the API returned this JSON payload.

```
{
  "id": 1,
  "src": "https://programming-elm.surge.sh/1.jpg",
  "caption": null,
  "liked": "no"
}
```

This hardly matches your record type. The caption property is null instead of a string, the liked property is a string with the value "no" instead of the boolean false, and the comments property is missing.

Elm is caught in a catch-22. Elm requires the payload to have a specific *shape* but must protect your application from inconsistent, bad data. By shape, I mean a payload that contains specific properties with specific types.

Elm solves this dilemma with JSON decoders. When you create a JSON decoder, you describe the expected shape of the JSON payload and what static type to create from the payload. Elm uses your decoder to attempt to decode the JSON payload into your static type. You will work through creating a JSON decoder for your application over the next few sections.

Initial Setup

Before you create a decoder, let's get a few prerequisite steps out of the way. Later in this chapter, you're going to change the Model type in your Picshare application from a photo to a record that contains the photo. This means you need to create a new type alias to represent a photo. You also need to add the id field to your record type because your application will fetch multiple photos (in the next chapter). You'll handle the username field in Chapter 10, Build Single-Page Applications, on page ?.

Open up your Picshare.elm file. Rename the Model type alias to Photo and then create a new Model type alias to the Photo type. The type alias rabbit hole can go as deep as you want, but be wary, there be dragons down that hole.

While you're at it, create a type called Id that aliases to Int. This will help make your later type annotations more readable when you want to treat an Int argument as an Id. Right underneath your imported modules, you should now have this code:

communicate/Picshare01.elm

```
type alias Id =
    Int

type alias Photo =
    { id : Id
    , url : String
    , caption : String
    , liked : Bool
    , comments : List String
    , newComment : String
    }

type alias Model =
```

```
    Photo
```

Because you added an id field to the Photo type, you'll need to add an initial id to your initialModel to ensure your application can still compile. Add an id of 1 at the start of the initialModel definition:

```
initialModel =
    { id = 1
    -- other fields you already defined
    }
```

The first step is out of the way. The next prerequisite step is to grab a couple of packages.

Elm has its own package manager that you can use to install additional dependencies.

Elm should have previously installed Elm's main JSON package elm/json as an indirect dependency when you ran elm init in the first chapter. An indirect dependency is a dependency of some other dependency in your application. You need to install elm/json as a direct dependency to let your application code use it. Make sure you're in your picshare directory and run this command:

```
elm install elm/json
```

The command should prompt you to move the dependency from indirect to direct dependencies. Accept the prompt.

```
I found it in your elm.json file, but in the "indirect" dependencies.
Should I move it into "direct" dependencies for more general use? [Y/n]:
```

Next, install a really helpful package called NoRedInk/elm-json-decode-pipeline, which has a lot of cool helper functions for building complex JSON object decoders:

```
elm install NoRedInk/elm-json-decode-pipeline
```

The command should prompt you to add the dependency to elm.json. Accept.

Great…you just learned how to install packages. You can browse all available Elm packages at https://package.elm-lang.org. You're also ready to build a photo decoder. But before you jump in, let's get your feet wet in the REPL with some simpler decoders.

Play with Decoders

Writing a full-fledged decoder for the Photo type will be relatively easy and require little code. Understanding the code will be the challenging part. Let's get familiar with decoders by playing with some primitive decoders before you

attempt to decode a photo object. Open up the Elm REPL and import the Json.Decode module.

```
> import Json.Decode exposing (decodeString, bool, int, string)
```

The Json.Decode module comes from the elm/json package and contains a few primitive type decoders as well as helper functions for building complex decoders. The primitive decoders you use here are bool, int, and string. As you might imagine, the bool decoder represents Bool types, the int decoder represents Int types, and the string decoder represents String types. Elm has one more primitive decoder called float.

Each of these primitive decoders has the type Decoder a. The type variable a refers to the static type that the decoder decodes to. For example, string has the type Decoder String, so it would decode to an Elm String.

The decodeString function uses a decoder to decode a raw JSON string into a static type. Let's create an int decoder and try it out with the number 42. Run this in the REPL:

```
> decodeString int "42"
Ok 42 : Result Json.Decode.Error Int
```

The first argument is the int decoder. The next argument is the JSON string "42". The return value is interesting, however. You didn't get back 42. Instead, you received Ok 42 with the type Result Json.Decode.Error Int. Before you investigate that further, run this snippet in the REPL:

```
> decodeString int "\"Elm\""
Err (Failure ("Expecting an INT") <internals>) : Result Json.Decode.Error Int
```

This time you received a value called Err with the same type as before, Result Json.Decode.Error Int. The Err value contains a Failure value with a string message "Expecting an INT". (The <internals> bit refers to the raw JavaScript that Elm parsed. Elm uses JavaScript's JSON.parse underneath the hood to initially parse to JavaScript before decoding to a type in Elm.)

The Result type is how Elm safeguards applications from bad JSON payloads. When you called decodeString, you declared that the payload was an integer, but you passed in the JSON string "\"Elm\"" instead of a number. The decode operation failed and gave back an error from the Result type.

The Result type is a built-in custom type with two constructors called Ok and Err. This is how it's defined in Elm.

```
type Result error value
    = Ok value
    | Err error
```

In the last chapter, you saw how custom type constructors could take arguments when you defined the UpdateComment String constructor. The argument type doesn't have to be set in stone, so you can use a type variable. If you use a type variable, then Elm's type system requires you to declare the type variable on the type itself too. The Result type has two type variables called error and value.

In Elm, you use the Result type to handle an operation that could succeed or fail. If the operation succeeds, you can use the Ok constructor to wrap the successful value. Conversely, if the operation fails, you can use the Err constructor to wrap an error.

The decodeString function returns the Result type to signal that the decoding process could fail, specifically if the JSON payload type doesn't match the decoder type. If the decoding process succeeds, then you get Ok with the actual decoded value. If the decoding process fails, then you get Err with a value explaining the error. You saw both of those scenarios just a moment ago when you tried decodeString with the JSON strings "42" and "\"Elm\"".

The Result type satisfies the type system so you can safely decode without any runtime errors. The Result type's two type variables indicate the static types it can contain. In the REPL example, the returned type was Result Json.Decode.Error Int. The Json.Decode.Error and Int types indicated that the result could contain a decoder error or a successful integer value.

The Json.Decode.Error type is another custom type defined in the Json.Decode module. You will work more with Json.Decode.Error in Chapter 7, Develop, Debug, and Deploy with Powerful Tooling, on page ?. You can also learn more about it from the docs.[1]

I know what you're probably thinking. It's all well and good that my application won't blow up, but I still need to access the successful value. That's what pattern matching is for. In fact, you'll see how to use pattern matching on the Result type later in this chapter when you actually fetch a photo from an API. For now, play with a few more primitive decoders in the REPL before you move on to decoding objects.

```
> import Json.Decode exposing (decodeString, bool, field, int, list, string)

> decodeString bool "true"
Ok True : Result Json.Decode.Error Bool

> decodeString string "\"Elm is Awesome\""
Ok ("Elm is Awesome") : Result Json.Decode.Error String
```

1. https://package.elm-lang.org/packages/elm/json/latest/Json-Decode#Error

```
> decodeString (list int) "[1, 2, 3]"
Ok [1,2,3] : Result Json.Decode.Error (List Int)

> decodeString (field "name" string) """{"name": "Tucker"}"""
Ok "Tucker" : Result Json.Decode.Error String
```

The bool and string decoders are similar to the int decoder you used earlier. You also imported two helper decoders called field and list that build decoders from other decoders.

The list decoder lets you decode a JSON array. It accepts a decoder argument to decode each item in the array. This means every item in the array needs to be the same type, or decoding will fail.

The field decoder lets you decode the value of a property in a JSON object. It takes two arguments, the property name and a decoder for the property. This decoder will fail if the JSON string doesn't contain an object, if the property is missing, or if the property type doesn't match the decoder property type. You generate the JSON string with triple quote """ syntax. This syntax allows you to create special strings that don't require escaping quotes inside the string. It also lets you create multiline strings like so.

```
myElmPoem =
    """
    Roses are red
    Violets are blue
    Elm is awesome
    And so are you
    """
```

Elm has more decoder helpers that you can explore in the docs.[2] For example, the at helper is great for extracting deeply nested object values, and the oneOf helper is great for trying multiple decoders until one succeeds. Try out a few other decoders on your own in the REPL.

Pipe through Functions

Before we go further with decoders, we need to briefly detour to look at Elm's most useful operator, the *pipe operator*. You will need the pipe operator to create object decoders with elm-json-decode-pipeline.

One benefit of functional programming is that you can combine small, specialized functions to create more complex functions. Functional programmers call this *function composition*.

2. https://package.elm-lang.org/packages/elm/json/latest/Json-Decode

Let's say you need to write a function called excitedGreeting that accepts a String name and returns a greeting with the name in uppercase and ends with an exclamation point. You can create this function with smaller functions. Inside the REPL, add the greet and exclaim functions like the following:

```
> greet name = "Hello, " ++ name
<function> : String -> String

> exclaim phrase = phrase ++ "!"
<function> : String -> String
```

The greet function takes a String name and prepends the String "Hello, " to it. The exclaim function takes a String phrase and appends an exclamation point to it.

Now, along with the built-in String.toUpper function, create the excitedGreeting function like so:

```
> excitedGreeting name = \
|   exclaim (greet (String.toUpper name))
<function> : String -> String
```

You compose the three functions together by passing in the result of one function as the argument to the next function. First, you call String.toUpper with name. This returns name in uppercase, which you then pass into greet. Finally, you pass the result of greet into exclaim. Try out excitedGreeting like so:

```
> excitedGreeting "Elm"
"Hello, ELM!" : String
```

Composition lets you build more complex functions but the syntax is awkward right now. Notice that you wrapped function calls in parentheses to enforce the order of operations. If you had left out parentheses, Elm would have thought you wanted to call exclaim with three arguments, greet, String.toUpper, and name.

The pipe operator fixes this problem by giving you more readable composition. Rewrite excitedGreeting in the REPL, like this:

```
> excitedGreeting name = \
|   name |> String.toUpper |> greet |> exclaim
<function> : String -> String
```

The pipe operator |> takes the left operand and passes it in as the last argument to the function operand on the right. In this case, you take name on the left and pass it into String.toUpper on the right. Then, you pass the result of String.toUpper into greet on the right. You repeat this process, passing the next result into exclaim.

This style of composition simulates chaining or piping function calls together. You can think of the pipe operator as a chain link between each function. The pipe operator also points to the right, so you can clearly see the direction of applied functions to the previous result. You can even improve readability by placing each function call on a newline:

```
> excitedGreeting name = \
|   name \
|     |> String.toUpper \
|     |> greet \
|     |> exclaim
<function> : String -> String
```

Now you can scan from top to bottom to see each step you take to transform the name argument into the final result. Call excitedGreeting again with "Elm", and you should see the same return value as before.

Decode an Object

Great. You're now familiar with the concept of decoders, know how to build some simple decoders, and know how to use the pipe operator. You're ready to go a step further and decode an entire JSON object. The elm-json-decode-pipeline package will come in handy here.

Let's revisit the dog record from the previous chapter to build a JSON dog decoder. Once you get a handle on that, you'll be ready to build a decoder for the Photo type in the Picshare application. Make sure your REPL is open and expose these members of the Json.Decode and Json.Decode.Pipeline modules:

```
> import Json.Decode exposing (decodeString, int, string, succeed)
> import Json.Decode.Pipeline exposing (required)
```

You've already seen the Json.Decode module. The Json.Decode.Pipeline module comes from elm-json-decode-pipeline. You expose a helper called required. Next, you need a helper function for creating a dog. Run this in the REPL:

```
> dog name age = { name = name, age = age }
<function> : a -> b -> { age : b, name : a }
```

You will need this function to build a dog decoder. Create the dog decoder by running this code in the REPL:

```
> dogDecoder = \
|   succeed dog \
|       |> required "name" string \
|       |> required "age" int
<internals> : Json.Decode.Decoder { age : Int, name : String }
```

Here's where the fun begins, so let's dissect the dogDecoder piece by piece. On the first line, you call the succeed function from Json.Decode on the dog function. The succeed function creates a decoder literal. For example, if you call succeed on the string "Elm", then you get back a Decoder String. For the dog function, you get back a Decoder (a -> b -> { age : b, name : a }). Essentially, you get back a decoder of whatever you pass in, even if it's a function like dog.

On the next line, you use the pipe operator to feed the decoder into the required function. The required function comes from elm-json-decode-pipeline and resembles the field function you used earlier. It *requires* a property to exist in the JSON object just like field. It's different from field in that it not only extracts the property but also *applies* the value to the function inside the current decoder. Look at the type signature of required to see what I mean.

```
required : String -> Decoder a -> Decoder (a -> b) -> Decoder b
```

The first argument is a String, which is the name of the property. You used "name" for the property name in the dog example. The second argument is a Decoder a that expects the property to have a type of a. Recall that lowercase types such as a are type variables, so this can be a Decoder of anything. You used the string decoder in the dogDecoder example, so the concrete type you pass in will be Decoder String. The third argument is another decoder that contains a function. This inner function must translate the type a to the type b. This translation process allows required to return a Decoder b.

In this example, the third argument is the decoder that contains the dog function. If you had only run the first two lines from the example, your decoder would now have this type.

```
Decoder (a -> { age : a, name : String })
```

Compare that type to what you had previously from executing only the first line of the example.

```
Decoder (a -> b -> { age : b, name : a })
```

Notice that you filled in the first type variable to be a String. That is, you went from a function with two arguments to a function with one argument.

Moving to the third line in the example, you call the required function with the string "age", the int decoder, and the current dog decoder. The dog decoder can now extract the age property and apply it as the second argument to the original dog function, which gives you the following final decoder.

```
Decoder { age : Int, name : String }
```

The elm-json-decode-pipeline package makes decoders easy to read and write. The trick to understanding them is to remember that each pipe operation is applying an extracted value to a function inside a decoder. Once you satisfy all the arguments, you get back a decoder of the record you want to create. Let's try your newly minted dogDecoder on an actual JSON object. Run this code in the REPL:

```
> decodeString dogDecoder """{"name": "Tucker", "age": 11}"""
Ok { age = 11, name = "Tucker" }
    : Result Json.Decode.Error { age : Int, name : String }
```

Good job! You just grasped one of the trickiest concepts in Elm. Decoders are versatile and powerful. You can build some highly complex decoders in Elm.

Create a Photo Decoder

Now that you're familiar with elm-json-decode-pipeline, let's use it to create a photo decoder. Switch back to editing Picshare.elm. First, import Json.Decode and Json.Decode.Pipeline underneath the other imported modules:

communicate/Picshare01.elm

```
import Json.Decode exposing (Decoder, bool, int, list, string, succeed)
import Json.Decode.Pipeline exposing (hardcoded, required)
```

These module imports look similar to what you had in the REPL. You import one additional function from Json.Decode.Pipeline called hardcoded. Next, add the decoder below the Model type alias:

```
photoDecoder : Decoder Photo
photoDecoder =
    succeed Photo
        |> required "id" int
        |> required "url" string
        |> required "caption" string
        |> required "liked" bool
        |> required "comments" (list string)
        |> hardcoded ""
```

This decoder resembles the dogDecoder you wrote in the REPL earlier with a couple of differences. First, you call succeed on Photo, which may seem confusing at first. You're not calling succeed on the Photo type but the Photo *constructor function*. Recall from Chapter 3, Refactor and Enhance Elm Applications, on page ? that a type alias to a record also creates a constructor function for the record.

As you saw in the previous section, you can call succeed on a function and then pipe the decoder through elm-json-decode-pipeline helper functions to extract properties and apply them to the underlying function. Here you're

doing exactly that, only you're capitalizing on the convenient constructor function that Elm creates for record type aliases.

You pipe the constructor function through several calls to required with different decoders. For the "id" property you use the int decoder. For the "url" and "caption" properties you use the string decoder. For the "liked" property you use the bool decoder. Finally, for the "comments" property you use list string. Remember that the list decoder takes another decoder as an argument to decode each item in the JSON array to that inner decoder's type.

At the end, you use the hardcoded function. The Photo record has six fields, which means the Photo constructor function takes six arguments. One of those fields is newComment, which the JSON payload on page 5 lacks. You can use the hardcoded function to tell the decoder to use a static value as an argument to the underlying decoder function instead of extracting a property from the JSON object. In this case, you use hardcoded to provide the empty string as the final newComment argument to the Photo constructor function.

Let's try out photoDecoder in the REPL to confirm it works. Temporarily expose photoDecoder from Picshare.elm:

```
module Picshare exposing (main, photoDecoder)
```

Make sure you're in the same directory as the Picshare.elm file and run this code in a new REPL session:

```
> import Picshare exposing (photoDecoder)
> import Json.Decode exposing (decodeString)

> decodeString photoDecoder """ \
|     { "id": 1 \
|     , "url": "https://programming-elm.surge.sh/1.jpg" \
|     , "caption": "Surfing" \
|     , "liked": false \
|     , "comments": ["Cowabunga, dude!"] \
|     } \
|     """
Ok { caption = "Surfing"
   , comments = ["Cowabunga, dude!"]
   , id = 1
   , liked = False
   , newComment = ""
   , url = "https://programming-elm.surge.sh/1.jpg"
   }
    : Result.Result Json.Decode.Error Picshare.Photo
```

You import photoDecoder from the Picshare module and import decodeString from the Json.Decode module. Then you apply the photoDecoder to a JSON object to get back an instance of the Photo record. Revert the Picshare module to only expose main.

Let's recap what you accomplished. You created a photo decoder by calling the succeed function from Json.Decode with the Photo constructor function and then piping the decoder through the required and hardcoded helper functions from Json.Decode.Pipeline. Each helper function applies the next argument to the Photo constructor function. The required function extracts a property from the JSON object and uses that as the argument to Photo. The hardcoded function uses whatever argument it receives as the argument to Photo. The successive application of each argument eventually builds up an entire Photo record.

One important note to add is that the order of the piping operations matters. The order needs to match the order of the arguments to the constructor function. For example, if you switched the order of the `id` and `url` field decoders, you would get a compiler error. That's because the decoder would think it needs to call the constructor function with a `String` first instead of an `Int`.

OK. You've learned a lot about decoders and why they're important. You've also successfully created a photo decoder. You're now ready to put it to use by fetching an initial photo from an API. Make sure your code matches `code/communicate/Picshare01.elm` at this point, and then let's get to work using HTTP in the application.