Extracted from:

Programming Elm

Build Safe and Maintainable Front-End Applications

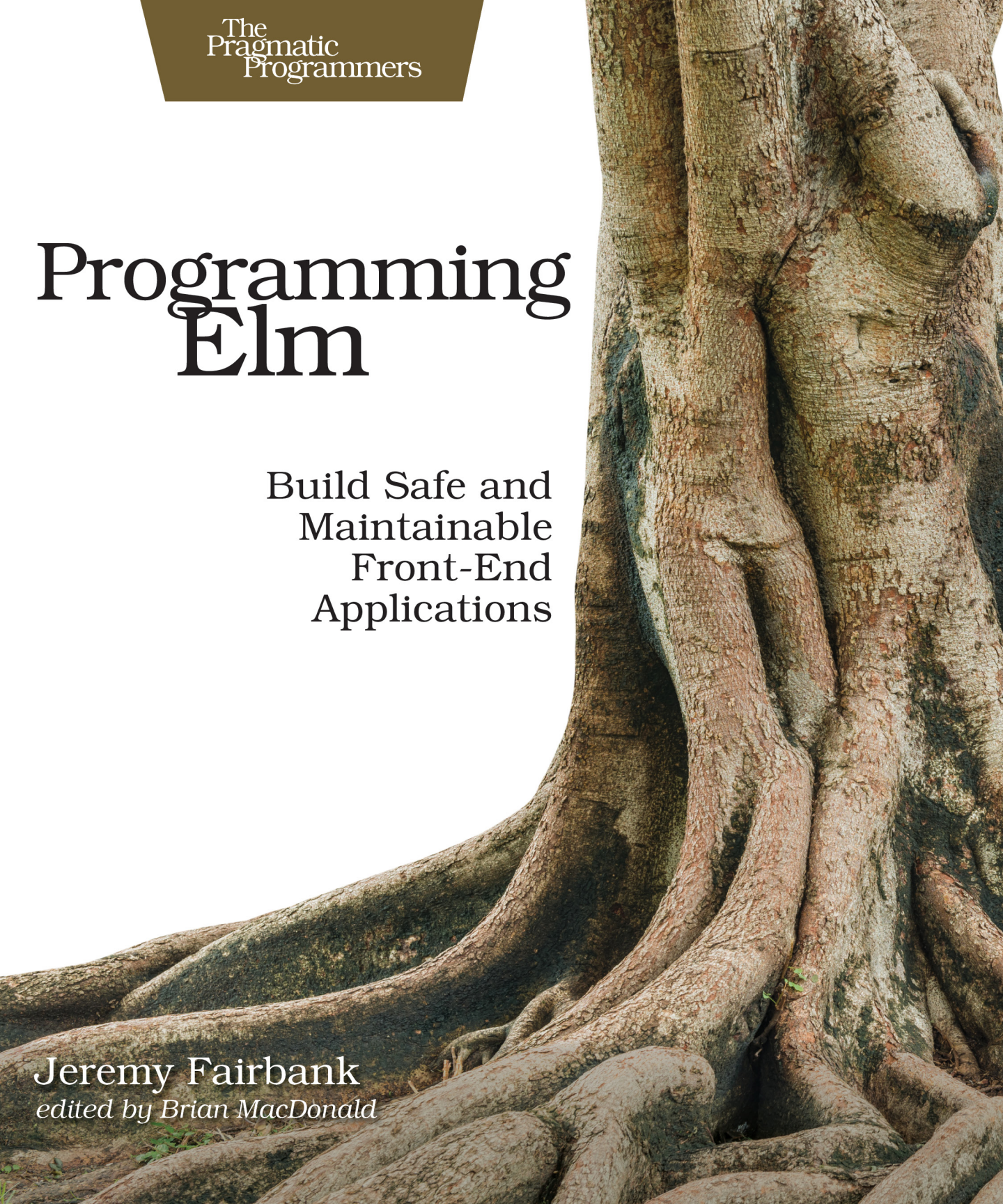
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Programming Elm

Build Safe and Maintainable Front-End Applications



Jeremy Fairbank
edited by Brian MacDonald

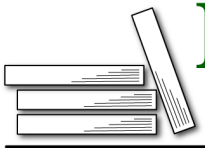
Programming Elm

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

Apply the Elm Architecture

The Elm Architecture is Elm's built-in framework for creating applications. In this section, you will learn about the architecture through adding a new feature to the Picshare application.

The Elm Architecture provides a standard way of building applications known as the *Model-View-Update pattern*. As its name suggests, this pattern has three important parts: a *model*, a *view*, and a method of *updating* the model. In the figure on page 6, you can see an overview of how the Elm Architecture works. You'll revisit later how all the pieces in the figure fit together. For now, we'll gain our understanding of a model by adding one to the application.

Create a Model

In Elm applications, the *model* is responsible for containing all your application state. This is different from other architectures such as MVC (Model-View-Controller) and MVVM (Model-View-ViewModel), or stuffing data in the DOM via data-* attributes. Those approaches encourage spreading your state across multiple models, making it hard to keep track of where state is located and how and when state changes. The Elm Architecture allows you to know where your state is located because it's consolidated in one place.

In Elm, the model can be whatever data type you want, such as a string or an integer. Typically, your model will be a *record* data type, which is what you'll use for the Picshare application.

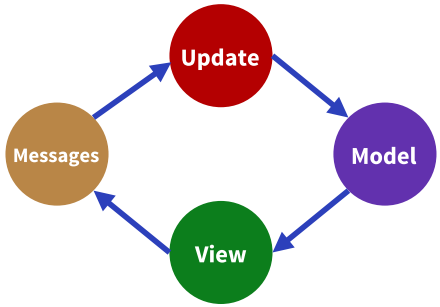
Work with Records

A *record* is similar to a plain old JavaScript object. It groups together related fields into key-value pairs. Elm developers typically refer to entries in a record as *fields*.

Let's learn more about records by creating a simple record to represent everyone's best friend, the dog. Fire up the Elm REPL from the command line with elm repl. Enter the following into the REPL:

```
> dog = { name = "Tucker", age = 11 }
{ age = 11, name = "Tucker" } : { age : number, name : String }
```

Notice that you use {} to create records, similar to creating JavaScript objects. The one difference from JavaScript objects is that you separate fields and their values with the = symbol instead of the : symbol.



After creating the dog constant, you get back a record instance with a record type of { age : number, name : String }. The record type looks similar to record values except it uses the : symbol to separate field names and their types.

You can access individual record fields with the dot operator, the same as with JavaScript objects. Try this out in the REPL:

```
> dog.name
"Tucker" : String

> dog.age
11 : number
```

Similar to lists in the previous chapter, accessing properties via bracket notation will not work, though. Elm will interpret the incorrect code below as a function call with a list argument. You can't access fields dynamically like you can in JavaScript. You'll see why in the next section.

```
> dog["name"]
-- TOO MANY ARGS ---------------------------------------------------------- elm

The `dog` value is not a function, but it was given 1 argument.

5|   dog["name"]
```

Create New Records

One significant difference between JavaScript objects and Elm records is that records are static. When you create a record instance, its type is set in stone. That means you won't be able to add new fields later or change the type of existing fields.

For example, the following code in the REPL will not work.

```
> dog.breed = "Sheltie"
-- PARSE ERROR ------------------------------------------------------------ elm

I was not expecting this equals sign while parsing repl_value_3's definition.

4| repl_value_3 =
```

```
5|   dog.breed = "Sheltie"
               ^
Maybe this is supposed to be a separate definition? If so, it is indented too
far. Spaces are not allowed before top-level definitions.
```

Records are also *immutable*, which is a hallmark of many functional languages. An immutable data type can't change in place. In the case of a record, this means you won't be able to change the value of an existing field, so the following code will not work either.

```
> dog.name = "Rover"
> dog.age = 12
```

Changing values in place as in the previous example is known as *mutation* and would be valid in JavaScript. In Elm, you can't mutate values.

Not being able to mutate fields in a record might seem like a hindrance, but it's actually a great safeguard. You have a guarantee that no code can accidentally or intentionally change your record instance, meaning fewer bugs in your code.

Elm isn't going to leave you high and dry, however. Instead of mutating records, you can create *new* instances of records.

Let's write a function for the dog to have a birthday. You'll want to take a dog record as an argument and return a new dog with its age incremented by 1. Enter this into the REPL:

```
> haveBirthday d = { name = d.name, age = d.age + 1 }
<function>
    : { b | age : number, name : a } -> { age : number, name : a }
```

You get back a function with a very interesting-looking type annotation. It takes a record of type `b` that must have an `age` field of type `number` and a `name` field of type `a`. The types `a` and `b` are type variables similar to what you saw in the previous chapter. The `number` type is a special type variable that can only be an `Int` or `Float` type when filled in. (The "b-type" record is called an extensible record, which you'll learn more about in Chapter 6, Build Larger Applications, on page ?.)

Notice in the implementation that you reuse the `d.name` field and add 1 to the `d.age` field in the new record. You can use the `haveBirthday` function on the original `dog` record to create a new instance of a dog record. Try this in the REPL:

```
> olderDog = haveBirthday dog
{ age = 12, name = "Tucker" } : { age : number, name : String }

> dog
{ age = 11, name = "Tucker" } : { age : number, name : String }
```

You assign the new dog record to a constant called olderDog. If you inspect olderDog, you have a dog with the same name that is one year older. If you inspect the original dog reference, you see that it still has the same age.

Use Record Update Syntax

Creating functions like haveBirthday might seem like a lot of boilerplate, especially when dealing with records with more fields. You have to make sure to copy all existing fields to return the same type. Elm provides some syntactical sugar for simplifying this process. Enter a new version of the haveBirthday function into the REPL like this:

```
> haveBirthday d = { d | age = d.age + 1 }
<function> : { a | age : number } -> { a | age : number }
```

We introduced the | symbol to record syntax. This is sometimes known as *record update* syntax. To the left of the | symbol, you provide an existing record reference, d. To the right, you specify any changes you want to make to values in the record reference. Elm will take all existing fields from the reference on the left and merge in changes from the right to create a new instance of the record with the changes. Try rerunning the examples from the code on page 7 in the REPL. You'll get back the same results from earlier.

One word of caution: the record update syntax might sound similar to the Object.assign function in JavaScript. Object.assign lets you merge together different JavaScript objects. Elm's record update syntax only allows you to create new values for *existing* fields in a record. You can't add new fields to the new record. Trying to add a breed field like this won't work.

```
> { dog | breed = "Sheltie" }
```

Immutability Has Benefits

Creating new instances of data types is common in functional languages like Elm. If this concept still seems foreign or wrong to you, don't worry. It felt like that to me too when I first started with functional programming. Coming from an object-oriented programming (OOP) background, I didn't see how you could accomplish anything if you didn't mutate data.

With more experience, I realized that it's easy to get things done in a functional language and that immutable data has great perks.

1. It makes data flow explicit. If functions want to "change" a record, they have to return a new record instead of mutating an existing one.

2. Instances of data structures can share data internally because there is no risk of code accidentally or intentionally mutating the shared data.
3. In multithreaded languages, there is no risk of threads mutating the shared data.

Create a Record Model

Now that you've learned about records, let's use one as the model for the Picshare application. In the last chapter, you statically displayed three images. To ease into making this a stateful application, you'll simplify the application to use one photo. Then, you can use a record model to represent the single photo.

For now, let's focus on displaying the single photo based on the fields of the model. You can jump into creating a *view* function in a moment. Open up the `Picshare.elm` file that you created in the last chapter and add this below the module imports and `baseUrl` constant:

stateful-applications/Picshare01.elm
```
initialModel : { url : String, caption : String }
initialModel =
    { url = baseUrl ++ "1.jpg"
    , caption = "Surfing"
    }
```

You create an initialModel record with two String fields, url and caption. Notice you also add a type annotation similar to the dog type annotation that the REPL printed earlier.

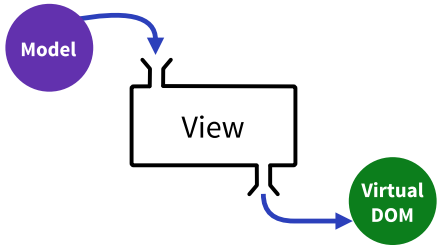
It's important for Elm applications to supply an initial state so there is something to initially display. That is why you named the record model initialModel. Using initialModel as the name for your initial state is common in Elm applications, but not required.

That's it as far as the model goes for right now. Let's turn our attention to displaying that model with a view function.

Create the View

In the Elm Architecture, the *view* is responsible for displaying a model. In many JavaScript frameworks, the view layer not only displays state but can manage state of its own. Unfortunately, this leads to the same problem of spreading out state that I mentioned at the start of the previous section. The Elm Architecture enforces separation of concerns by preventing the view layer from storing state. The view is the visual representation of the model and nothing more.

In Elm, views are implemented as functions. They take a model as an argument and return a virtual DOM tree. Recall from Chapter 1, Get Started with Elm, on page ? how you built a virtual DOM tree with the main constant by using the functions from the Html module. The virtual DOM tree describes what you want your application to display. Elm is responsible for converting the virtual DOM tree into real DOM nodes in the browser. You'll learn more about why and how Elm uses this virtual DOM tree later in this chapter.



Create the view function by reusing the main constant at the bottom of Picshare.elm. Rename main to view and update it to take the model as an argument like so:

```
view : { url : String, caption : String } -> Html msg
view model =
    div []
        [ div [ class "header" ]
            [ h1 [] [ text "Picshare" ] ]
        , div [ class "content-flow" ]
            [ viewDetailedPhoto model ]
        ]
```

The type signature now takes the record type and returns Html msg. The function implementation takes the model and passes it into the viewDetailedPhoto function. You'll need to update the implementation of viewDetailedPhoto next.

Display the Photo

The viewDetailedPhoto function currently takes the String arguments url and caption. You will want to condense those arguments down to just the record model because it contains fields for the url and caption. Update viewDetailedPhoto like this:

```
viewDetailedPhoto : { url : String, caption : String } -> Html msg
viewDetailedPhoto model =
    div [ class "detailed-photo" ]
        [ img [ src model.url ] []
        , div [ class "photo-info" ]
            [ h2 [ class "caption" ] [ text model.caption ] ]
        ]
```

The changes are minimal. You use model.url for the img src attribute and model.caption for the text content of the h2 tag.

Finally, you need to render the application in the browser. Create a new main constant for Elm to use:

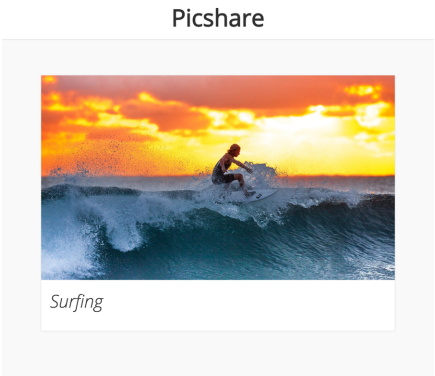
```
main : Html msg
main =
    view initialModel
```

The main constant ties the model and view together by passing in initialModel to the view function. This allows Elm to display your view function in the browser.

Inside your directory with Picshare.elm, make sure you still have the index.html and main.css files from the previous chapter. If you don't, you can grab them from the book's code downloads inside the stateful-applications directory. Compile your application and open up index.html in your browser.

```
elm make src/Picshare.elm --output picshare.js
```

You should see this in your browser.



You now have a minimally stateful application. The difference between the static application and this one is that your view depends on state it receives as an argument instead of hard-coding in photo URLs and captions. State flows top down from `main` to `view` and finally to `viewDetailedPhoto`.

Try changing the caption in `initialModel` to something different or use one of the other images in the url (`2.jpg` or `3.jpg`). After recompiling then refreshing your browser, you should see the changes reflected in what Elm displays.

You might say that you're still technically hard-coding in a photo via the `initialModel`, which is partially true. That is temporary. What you're really doing is setting up the application for later when the initial state can come from other sources like a server. Letting state flow through an application as a function argument is crucial to decoupling state from the view, and is also important when state can change—as you'll see when we introduce the `update` function in a bit.