Extracted from:

Programming Elm

Build Safe and Maintainable Front-End Applications

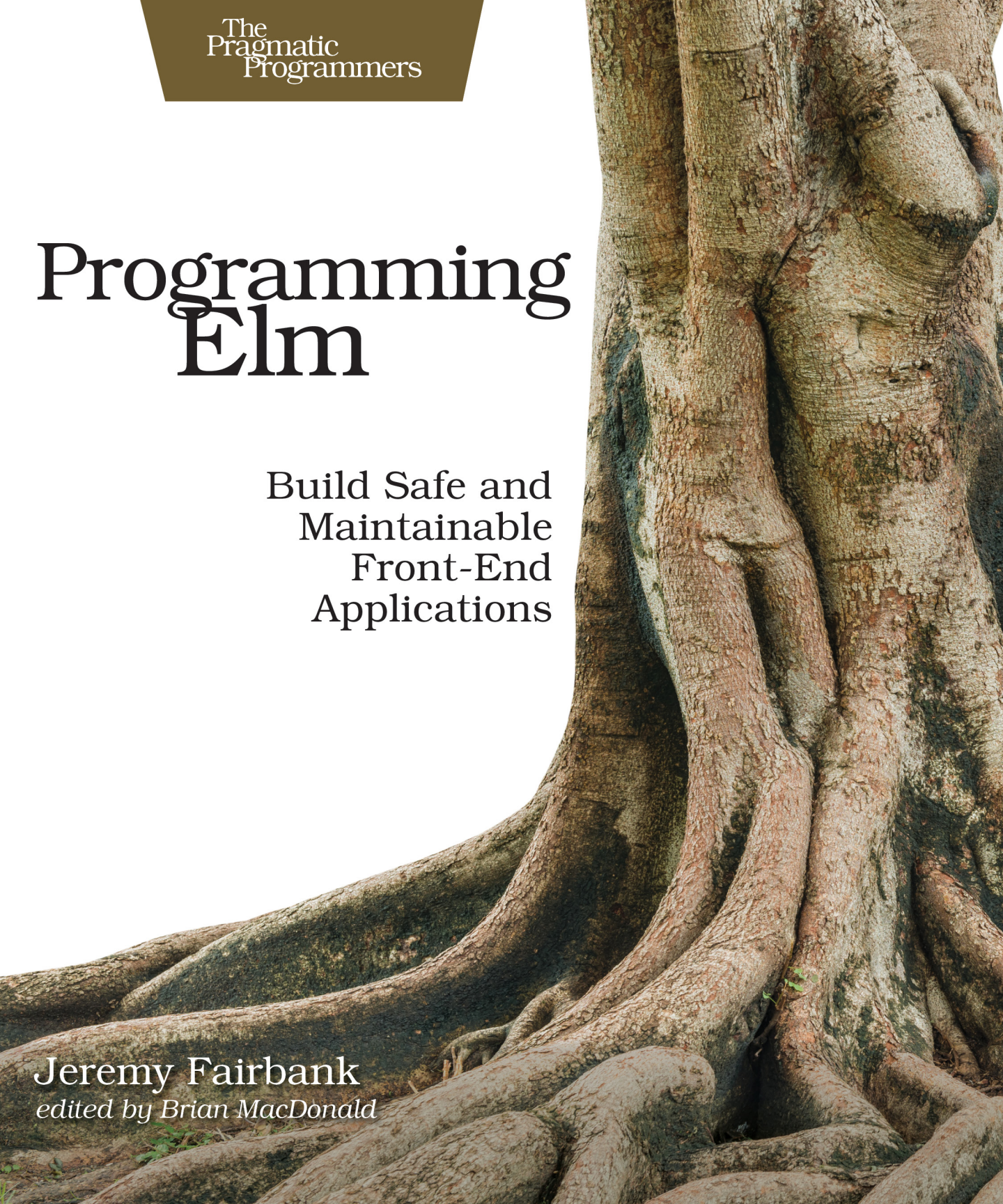
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Programming Elm

Build Safe and Maintainable Front-End Applications



Jeremy Fairbank
edited by Brian MacDonald

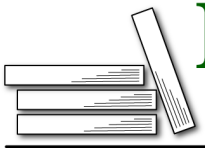
Programming Elm

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*.

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

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

Printed in the United States of America.
ISBN-13: 978-1-68050-285-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—December 20, 2017

Travel through Time

Have you ever wanted to travel through time? Well, we're not exactly hopping in a DeLorean to go ride some hoverboards. However, we will bend the rules of time in Elm applications. I can hardly tell you the number of bugs I've received that I can't reproduce according to the steps in a bug report. It's the classic "works on my machine" scenario.

Elm does better.

Recall that state changes one message at a time in Elm applications as the `update` function returns new state. Therefore, you could capture the lifetime of an Elm application by saving the state returned from `update`. You're safe to hold on to it because it's immutable.

If the QA (quality assurance) team recorded their test runs like this, then you could replay the state changes in development to exactly reproduce bugs. This isn't a fantasy; it's a reality with the Elm *time travel debugger*.

The time travel debugger records and replays state changes in Elm applications. You can effectively rerun the application as another user did. In this section, we will use the time travel debugger to debug our Picshare application from previous chapters. You will learn how to step through state changes to find the source of bugs in Elm applications.

Replay with the Time Travel Debugger

After developing the Picshare application, you've handed it off to QA to test. The QA team finds a few issues and sends back this bug report.

- New comments appear to add in the wrong order.
- After adding comments, I was unable to unlike a photo.
- New photos from the photo stream appear at the bottom of the feed.

This bug report lacks details, which makes it harder to find the source of the bugs in the code. Luckily, the QA team exported a history file from the time travel debugger and attached it to the bug report.

We could attempt to recreate the bugs and manually search code. Instead, let's import the history file into the time travel debugger. Then, we can walk through the same steps as the QA team. The time travel debugger will help us quickly identify where to locate the buggy code.

Look for the buggy version of Picshare inside the `code/develop-debug-deploy` directory from this book's code downloads. Locate the files `Picshare.elm`, `pic-`

share.html, picshare.css, and history.txt and copy them into the debugging directory from the previous section.

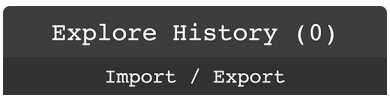
You will need the Http and WebSocket modules. Install them in the debugging directory like so.

```
elm package install -y elm-lang/http
elm package install -y elm-lang/websocket
```

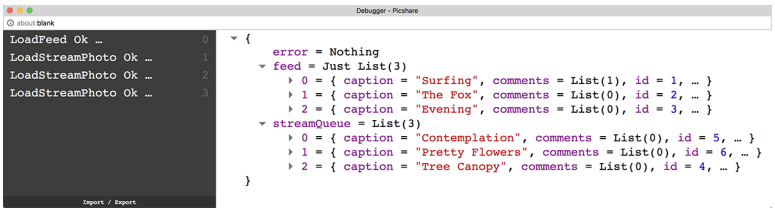
Next, compile Picshare.elm into a JavaScript file, but this time include the --debug option. The --debug option will enable the Elm time travel debugger in the compiled application.

```
elm make Picshare.elm --debug --output picshare.js
```

Open picshare.html in your browser. The application should load as normal, but you should now see the time travel debugger in the bottom right corner of your browser window.



The number next to “Explore History” should begin incrementing too. That number indicates how many messages the update function has processed. Click “Explore History” to make the time travel debugger reveal more details. You should see a popup that resembles the screenshot below.



On the left side of the popup, you will see some of the application’s Msg values in the following order.

```
LoadFeed Ok ...
LoadStreamPhoto Ok ...
LoadStreamPhoto Ok ...
LoadStreamPhoto Ok ...
```

These are the exact Msg values the update function has handled in order from top to bottom. This is the *history* of your application.

Elm only changes state by calling the update function with Msg values. You should see that the application has loaded the initial feed with LoadFeed and

received three new photos from the WebSocket stream with LoadStreamPhoto similar to the listing above.

Not only does the time travel debugger display the current history of dispatched messages, but it also shows the application's Model state. Look at the right side of the popup. You should see an Elm record with populated feed and streamQueue fields.

Try replaying history by clicking on the LoadFeed Msg on the left. The state on the right should change. The streamQueue list should be empty, but the feed list should persist. If you look at the application UI, you should also see the notification banner for the photo stream disappear.

You've essentially rewound your application like a cassette tape. I hope that doesn't make me sound old. The application is now back at the start when Elm processed the LoadFeed message from the fetchFeed command in init.

Because state is immutable, Elm easily accomplishes time travel by keeping a reference to every new model returned from update. Then, Elm "replays" history by swapping the current state with historical state and calling the view function with the historical state.

Click on the next LoadStreamPhoto message on the left side. You should see one photo appear in the streamQueue field on the right and the notification banner reappear in the application UI.

Track Down the Bugs

Now that you're familiar with the time travel debugger, let's actually import the history file from QA and track down the bugs they found. At the bottom of the left side of the debugger, you should see the words "Import / Export". Click on "Import", and your operating system's file dialog should appear. Navigate to your debugging directory and open the history.txt file.

The debugger popup window might disappear behind your browser window, so bring it back to the front. The history should contain a lot of messages now. Let's work through this new history to fix the bugs.

The first bug stated that new comments appear in the wrong order. You should see messages that refer to comments early in the history, so let's walk through messages from the beginning. Click on the first LoadFeed message to reset the application. Then, press your keyboard's ↓ key to move through history one message at a time.

As you progress, the first photo should become liked in the UI thanks to the ToggleLike message. After that, you'll see Elm "retype" the comment "test" into

the first photo's comments. Once you step through the first SaveComment message, you should immediately see the problem. The comment appears *above* the original comment "Cowabunga, dude!" Recall in our original application that comments appear *underneath* the previous comment to preserve chronological order. If you walk through the next series of UpdateComment and SaveComment messages from QA, you'll see the QA tester confirm the buggy behavior by adding another comment above the previous one.

Let's digest what the debugger is telling us. The bug seems to occur when the update function processes the SaveComment message. Open up Picshare.elm in your editor and go to the SaveComment branch inside the update function.

develop-debug-deploy/Picshare.elm

```
SaveComment id ->
    ( { model
        | feed = updateFeed saveNewComment id model.feed
      }
    , Cmd.none
    )
```

This branch calls out to the helper functions updateFeed and saveNewComment. The saveNewComment function sounds like the culprit, so jump to its definition. Look at the bottom of the function definition. You'll see that it *prepends* comments to the photo with the :: operator. That's the source of our bug.

```
{ photo
    | comments = comment :: photo.comments
    , newComment = ""
}
```

Fix the code by *appending* the comment with the ++ operator. Make sure you place comment inside a list and flip the order of comment and photo.comments.

```
photo.comments ++ [ comment ]
```

Not only can you find bugs with the time travel debugger, but you can also confirm bug fixes. Let's replay QA's history with the bug fix in place. Recompile your application with the --debug option and refresh your browser. Import the history.txt and walk through the history through the second SaveComment message. You should see the new comments appear in the correct order underneath the initial comment. That was an easy fix with the time travel debugger.

Let's fix the next bug. Continue walking through the history. You should see QA's attempt to unlike the first photo with two ToggleLike messages. Track down ToggleLike in the update function. The update function calls the toggleLike helper function, so go to its definition. You should immediately see the bug.

It only sets a photo's liked field to True. This was likely left over during some local testing of the love button.

```
toggleLike photo =
    { photo | liked = True }
```

Fix the code by toggling the current photo.liked field with the not function.

```
{ photo | liked = not photo.liked }
```

Recompile the application and import the history file. Walk through the history, and you should now see the photo become liked and unliked correctly with each ToggleLike message.

To fix the final bug, walk through the history to the end. When you reach the FlushStreamQueue message, the stream photos should appear at the end of the feed instead of the beginning. Go to the FlushStreamQueue branch of the update function. This issue is similar to the comments bug. The update function concatenates the model.feed and model.streamQueue values in the wrong order.

```
FlushStreamQueue ->
    ( { model
      | feed = Maybe.map (\feed -> feed ++ model.streamQueue) model.feed
      , streamQueue = []
      }
    , Cmd.none
    )
```

To fix the bug, flip the order of feed and model.streamQueue inside the anonymous function passed into Maybe.map.

```
Maybe.map (\feed -> model.streamQueue ++ feed) model.feed
```

Alternatively, use partial application with the function version of ++ like we did in code on page ?. Partial application will make model.streamQueue the left operand during concatenation, meaning the stream photos will appear at the beginning of the feed.

```
Maybe.map ((++) model.streamQueue) model.feed
```

Compile one last time with the --debug option and refresh your browser. Import the history file and replay the history of changes. Comments should appear in the right order, photos should become liked and unliked correctly, and the photo stream should load at the top of the feed.

The time travel debugger is an invaluable tool for finding bugs and verifying that bug fixes work. OK, I'll admit that I deliberately introduced these bugs so you could easily find them with the time travel debugger. Sometimes, you may not discover bugs so easily. For example, assume the feed photos only

ever had one comment added in the history file. Then, the time travel debugger wouldn't have revealed that the application adds additional comments in the wrong order.

Now that you've fixed Picshare, you can confidently ship a new bug-free version. In fact, you will do that in the next section. You will also learn how to speed up your development cycle in the process.