
MAKE THE
BACK-END TEAM
JEALOUS
Elm in Production

I work for a company called NoRedInk. We make grammar and writing software for English teachers. http://noredink.com

We use Ruby on Rails at NoRedInk.

Rails is built on top of Ruby, a language known for having a nice user experience.

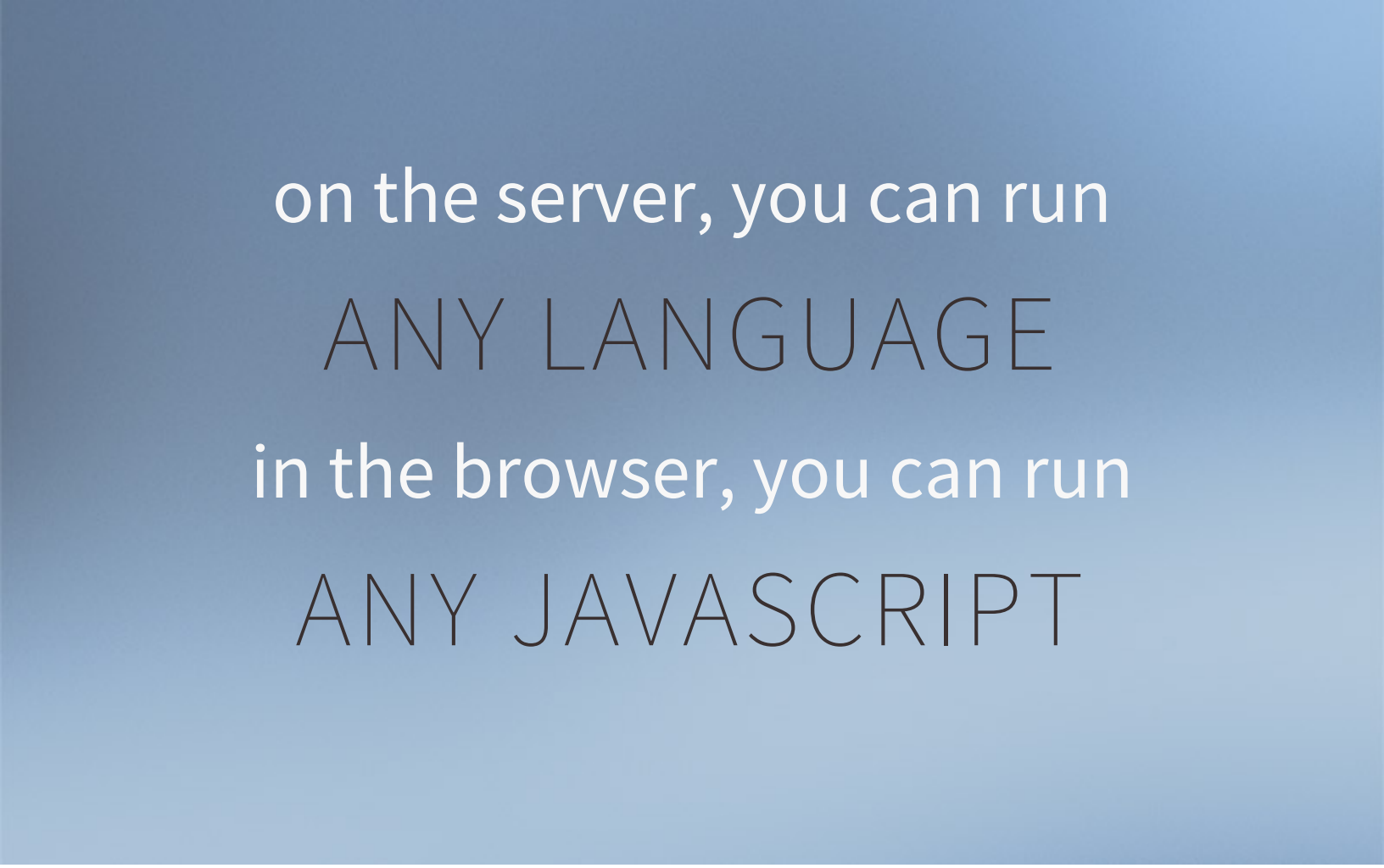
I don't get to deal with Ruby much. As someone who loves building Web UIs, I get to deal with this language a lot instead.

The user experience of using JavaScript...could be better.

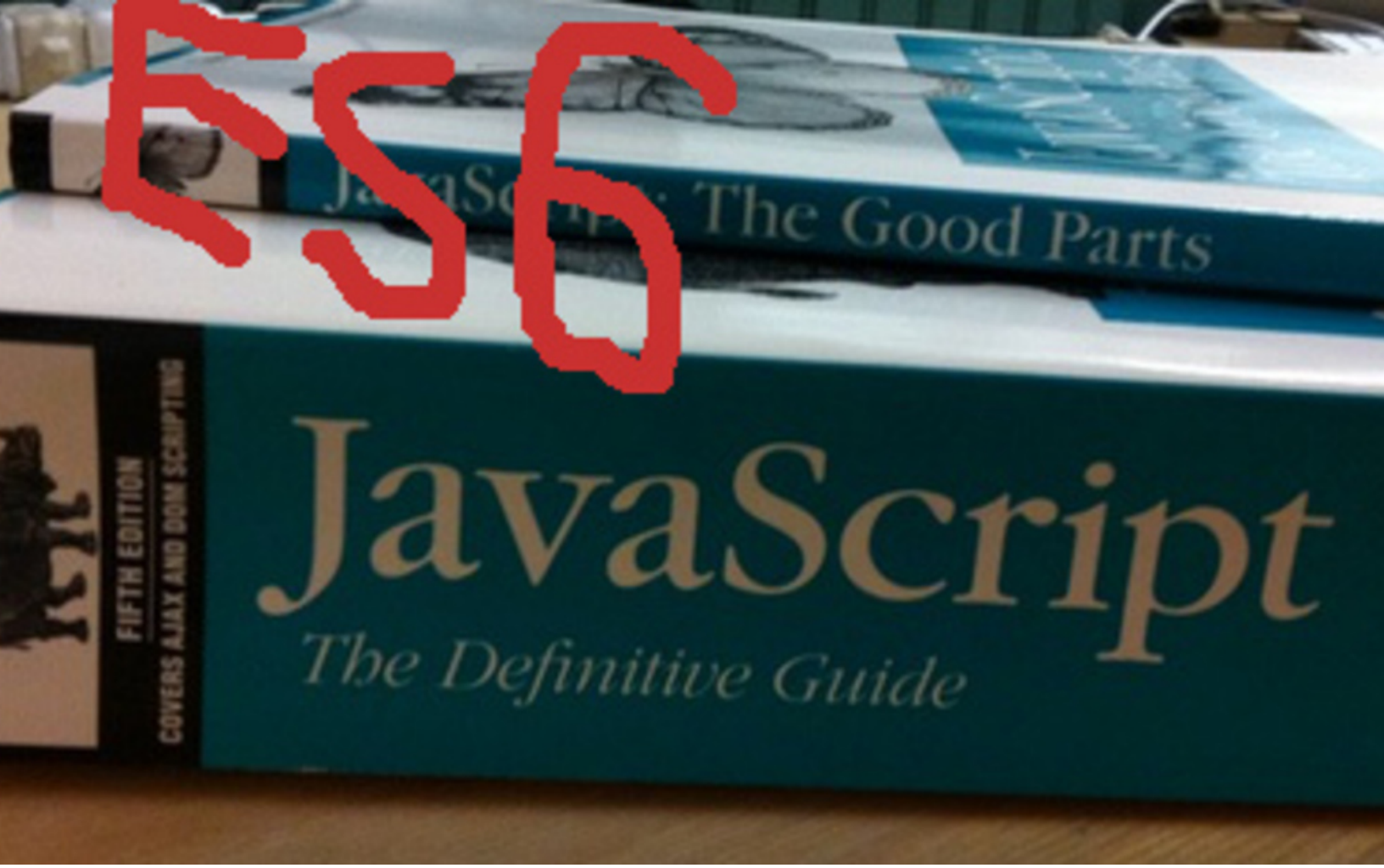
Why can’t JavaScript be

MORE LIKE RUBY?

Here's something I've found myself wondering.



The answer is that we're stuck with it because that's what the browser understands.

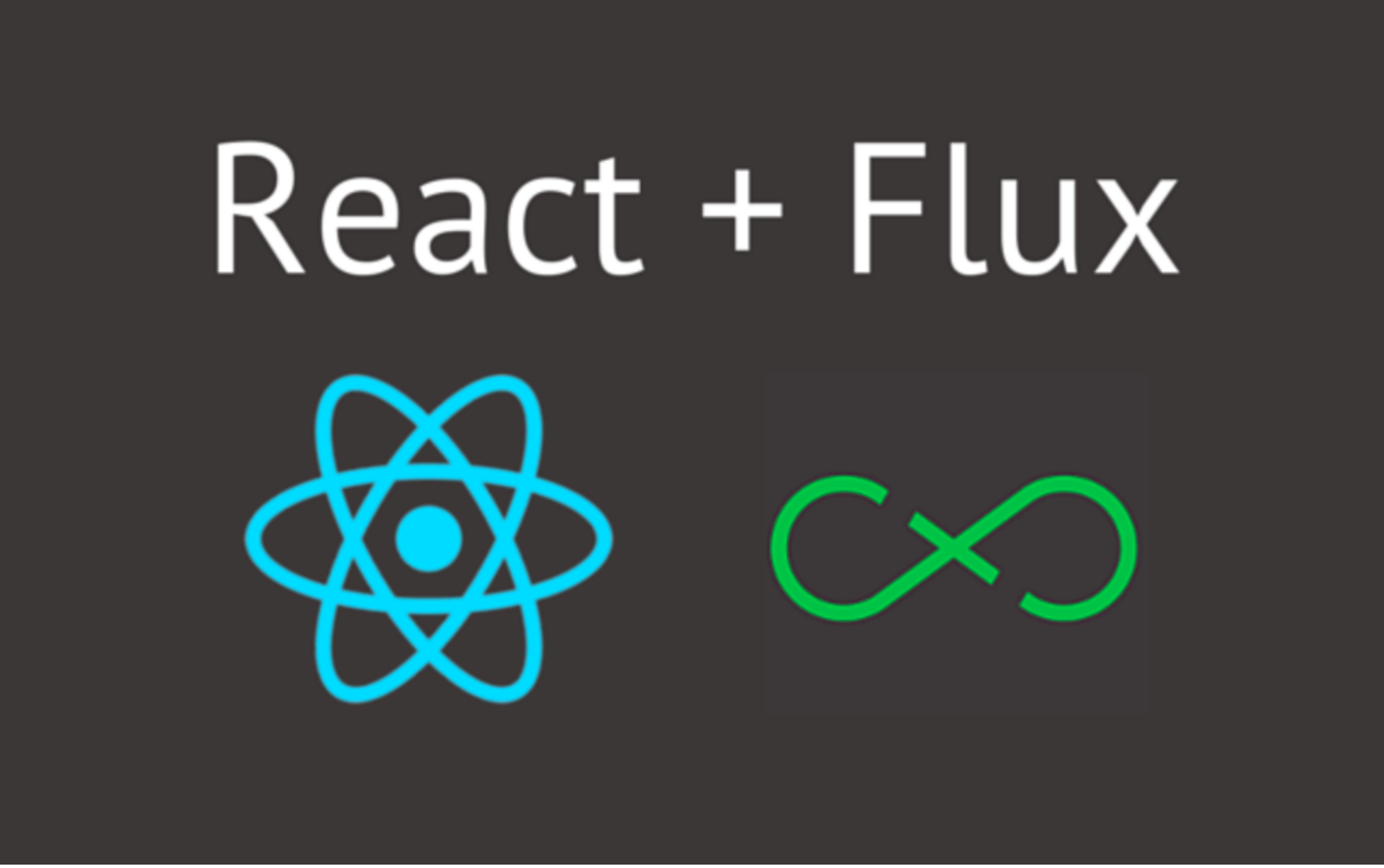


JavaScript is also getting nicer. It's still JavaScript, but it is getting nicer.

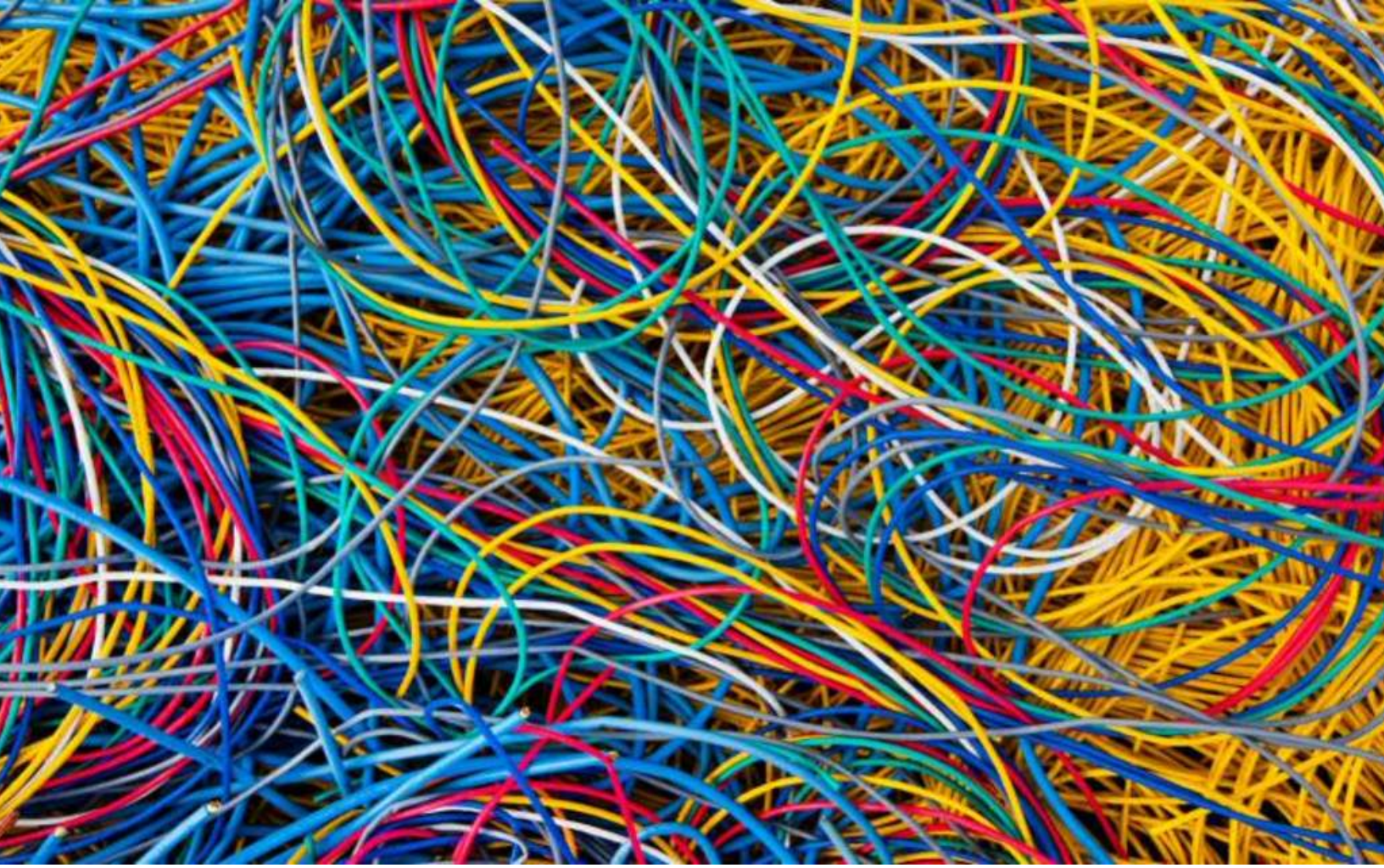
At work we use CoffeeScript, which is basically JavaScript that's more like Ruby: a language with a nicer user experience for programmers using it.

Compiling to JavaScript opens up a world of possibilities.

Separately from the language Ruby, there's also the architecture of Rails. It's more of an idea, an architecture, a way to structure things.



Similarly, React and Flux are a way to organize things for front-end developers.



React and Flux give you a great way to replace a tangled mess of UI dependencies...

...into something pretty clean and well-organized. It's a way to manage complexity, and to make things more testable.

ACTIVE VOICE

vs.

PASSIVE VOICE

We used React and Flux when we needed to make a user interface for active voice versus passive voice.

Our first design did not survive contact with end users.

ITERATE

What do we do when our design is too flawed to ship? We iterate! We did several revisions on this.

To do this, effectively we needed to *rearrange* these libraries.

Suppose we need to swap the orange wires on the left with the blue wires on the right. This is conceptually easy but mechanically error-prone.

CONCEPTUALLY EASY

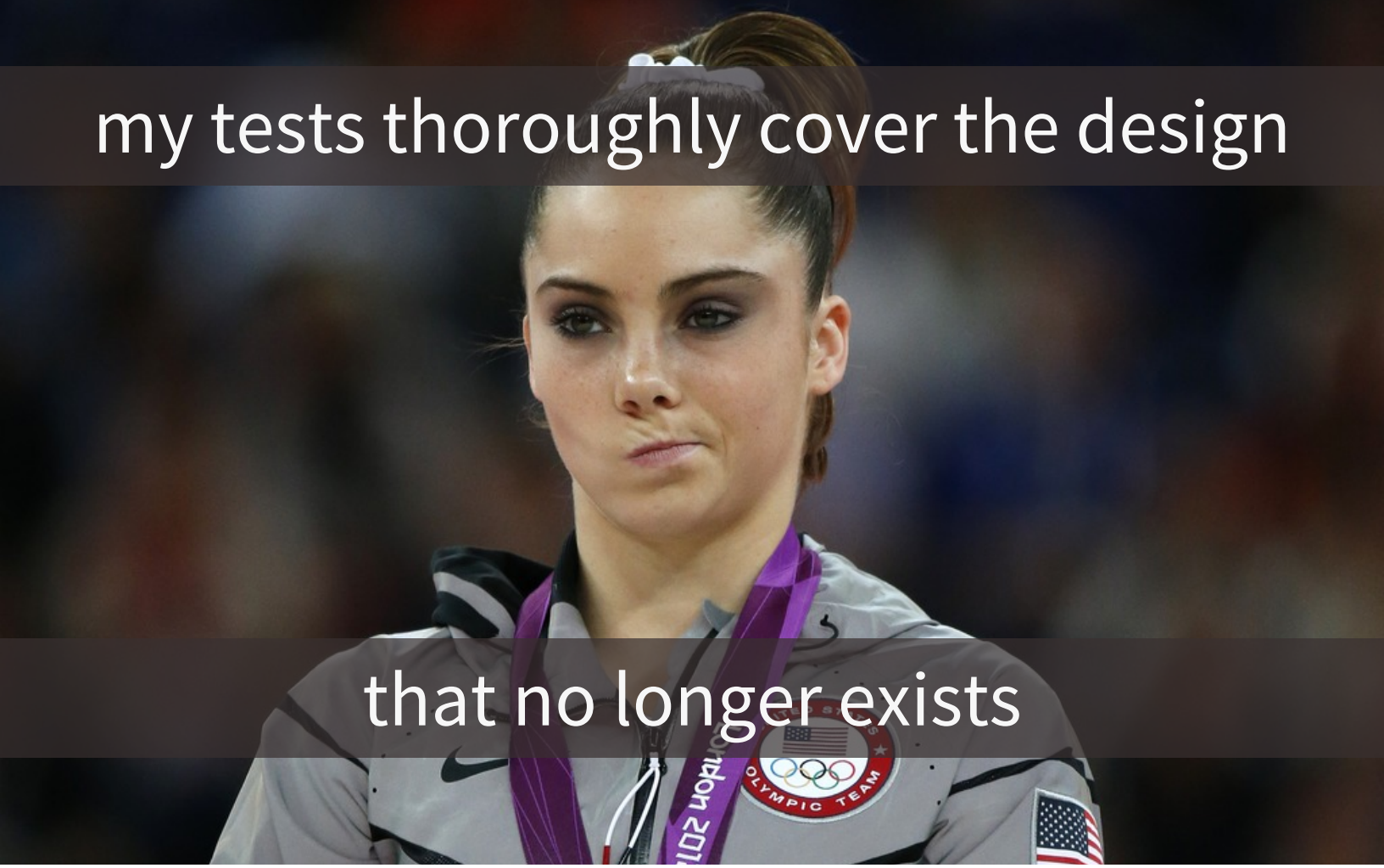
but mechanically

ERROR-PRONE

We can't know if we put all the wires back correctly until we finish and can turn everything on again. So we plug everything back in, turn it on, and....

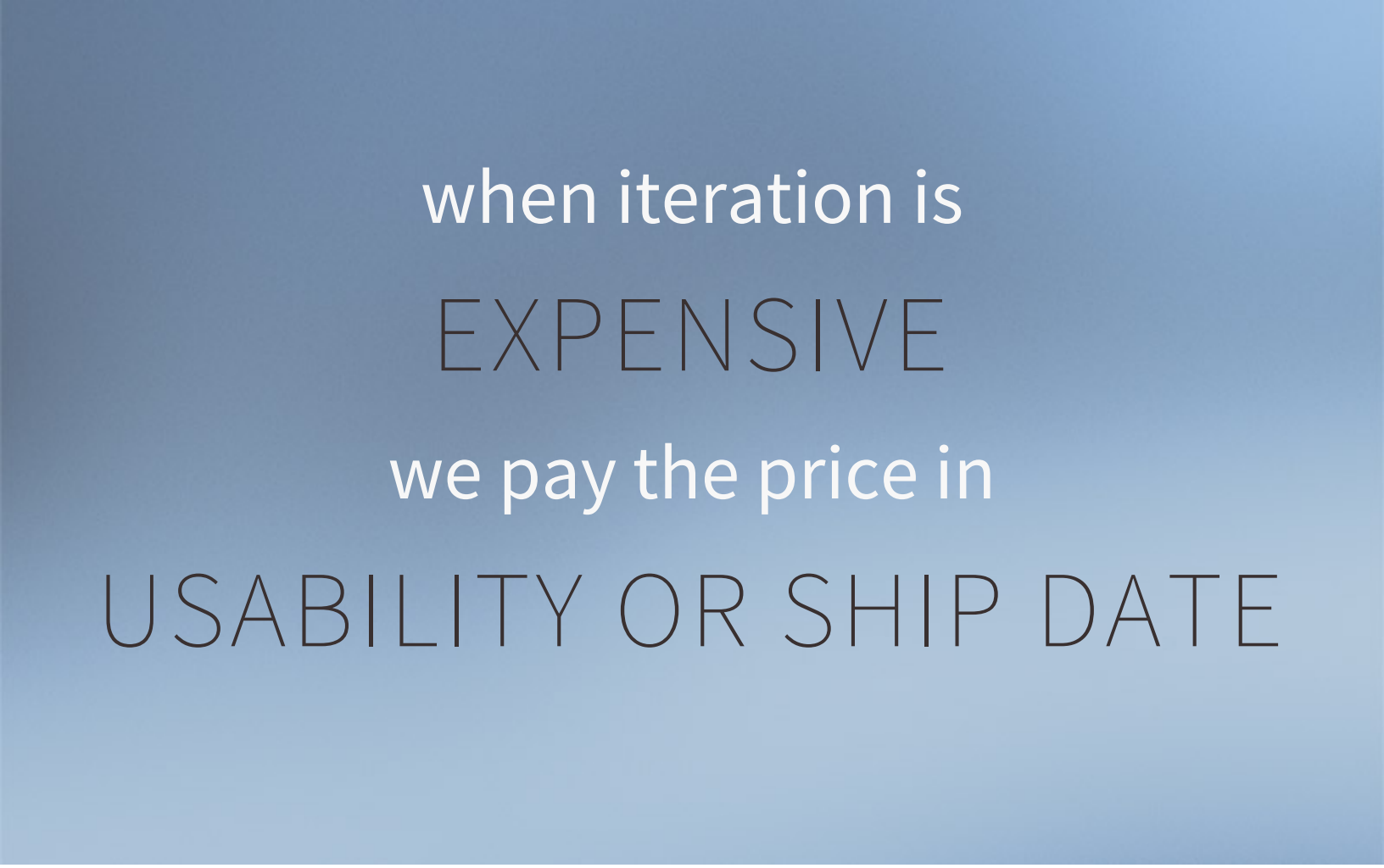
...ouch.

What happens when I need to do a major reorganization here? What can make this easier?



Tests are great for catching regressions, but not when doing major iteration - the kind that invalidates all your tests.

It took *months* to ship this feature, because iterating on the design took so long.



This feature cost a lot of time and money to ship because iteration was expensive.

if we could get

CHEAP ITERATION

we could

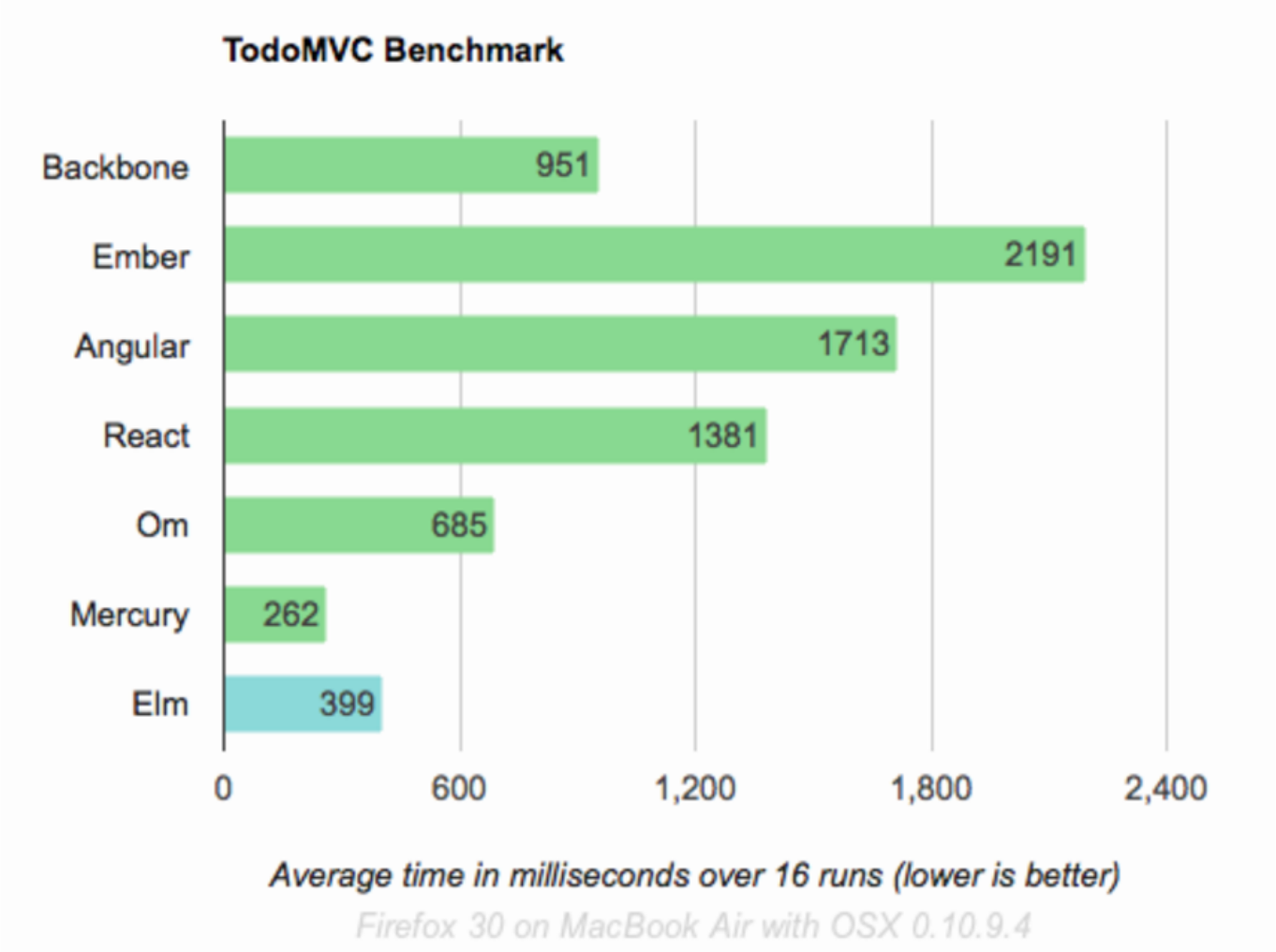
SHIP SOONER & BETTER

Was there a better way? Was there some other way we could have gone with this?

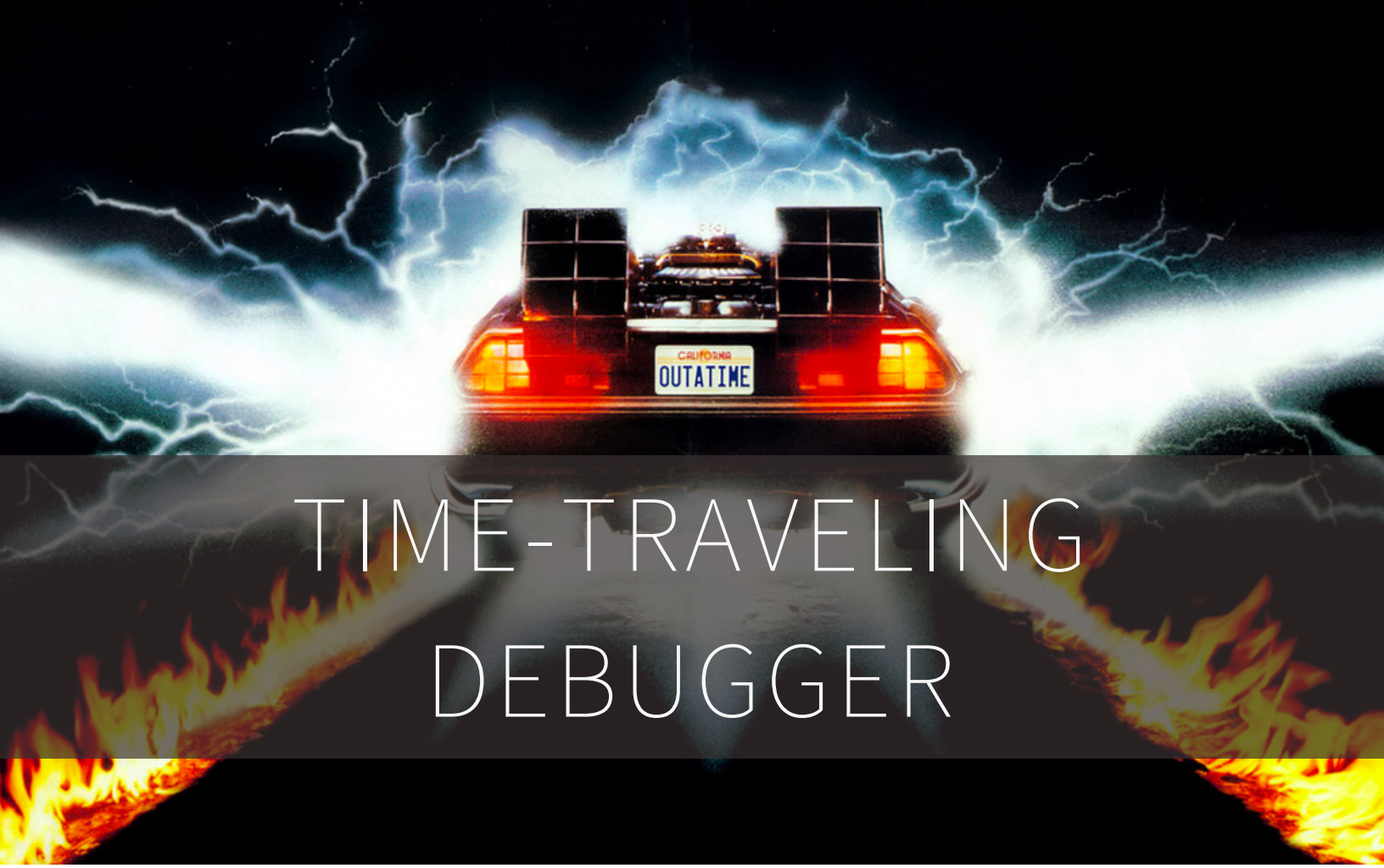
Actually, yes. I had a side project called Dreamwriter, a novel-writing Web App I talked about last year at Strange Loop.

elm

Dreamwriter is a lot more complicated than this one feature, and I rewrote it in Elm.



One reason was this. I knew Elm used the same "Virtual DOM" approach as React, but it outperformed it despite being compiled from a nicer language. Impressive!



TIME-TRAVELING DEBUGGER

Then I saw this: http://debug.elm-lang.org/edit/Mario.elm - really cool!

semantic versioning

GUARANTEED

for every single package

NO EXCEPTIONS

This was another really impressive selling point. If you make a breaking change to a public interface in a library, Elm's package manager won't accept it unless you bump the major version number. Awesome!

These big, flashy things were what drew me to Elm...but what really makes it a pleasure to use is the little details. The nice user experience.

```elm
numbers =
    [ 1, 2, 3 ]

first =
    List.head numbers

doubleFirst =
    first * 2
```

Here is some Elm code. This could have been written in any language. The basic idea is to take a list of numbers, retrieve the first one, and double it.

```
numbers =
    []

first =
    List.head numbers

doubleFirst =
    first * 2
```

The question is: what if the numbers list is empty? What does doubleFirst become?

```
numbers =
    [ 1, 2, 3 ]

first =
    List.head numbers

doubleFirst =
    first * 2
```

JavaScript

```
NaN
```

Here's what you get in JS. Have fun tracking that one down!

```
numbers =
    []

first =
    List.head numbers

doubleFirst =
    first * 2
```

Ruby

```
NoMethodError: undefined method `*' for nil:NilClass
```

This isn't great, but at least it won't insidiously propagate throughout your code base.

```elm
numbers =
    []

first =
    List.head numbers

doubleFirst =
    first * 2
```

What happens in Elm?

```
first =
    List.head numbers
```

```
-- TYPE MISMATCH ---------------------------------------- src/NumberTest.elm

The left argument of (*) is causing a type mismatch.

10|     first * 2
        ^^^^^
As I infer the type of values flowing through your program, I see a conflict
between these two types:

    number

    Maybe number
```

In Elm, this code won't even compile! That's because in Elm, List.head returns a Maybe - which is a container that describes the idea of "this might potentially not exist." This is what Elm does instead of null/nil/undefined - and it saves you from forgetting edge cases like this.

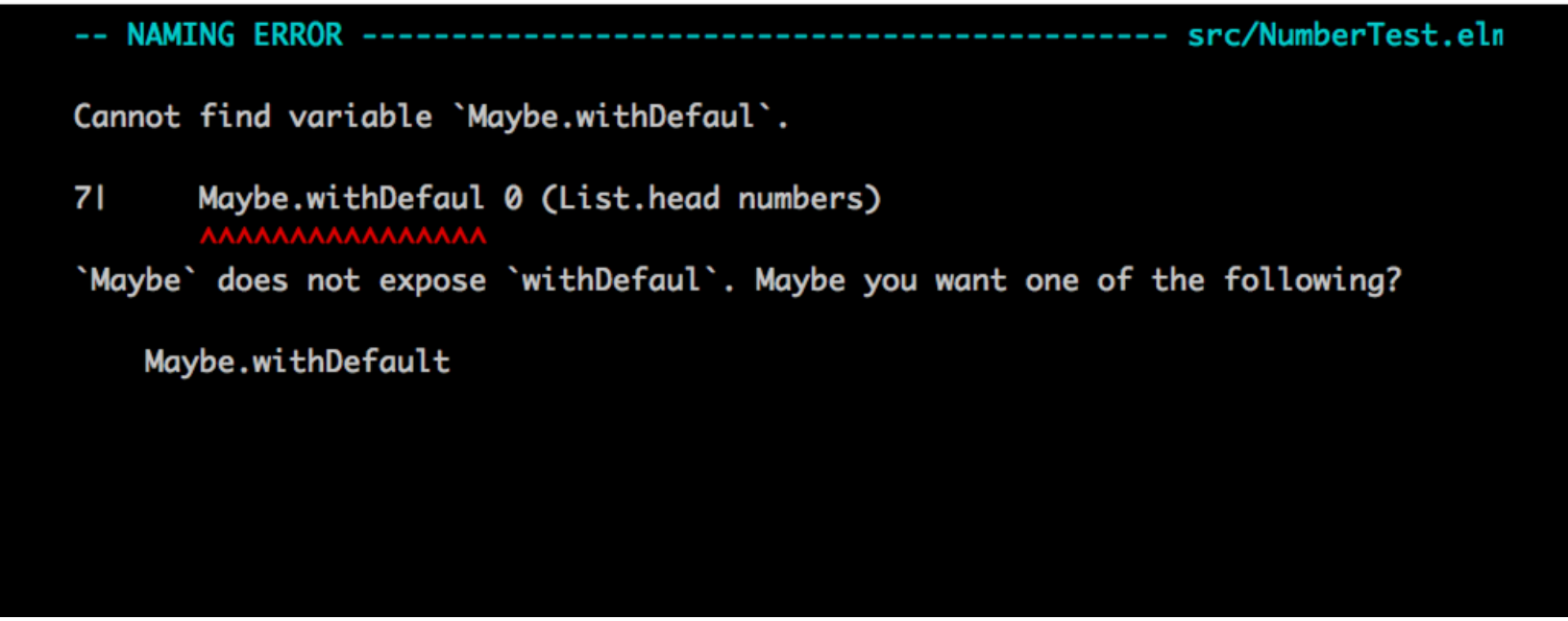
```
numbers =
    []

first =
    Maybe.withDefault 0 (List.head numbers)

doubleFirst =
    first * 2
```

This, on the other hand, will compile. Maybe.withDefault is one way to handle what happens when the list is empty.

```
first =
    Maybe.withDefaul 0 (List.head numbers)
```



If you misspell Maybe.withDefault, not only does it give you a compiler error, it even suggests the correct spelling!

This is how things ought to be!

“Building a Live-Validated Signup Form in Elm"

A COMPLETE TUTORIAL

If you're wondering how we do AJAX and things like that, check this out:
http://noredinktech.tumblr.com/post/129641182738/building-a-live-validated-signup-form-in-elm

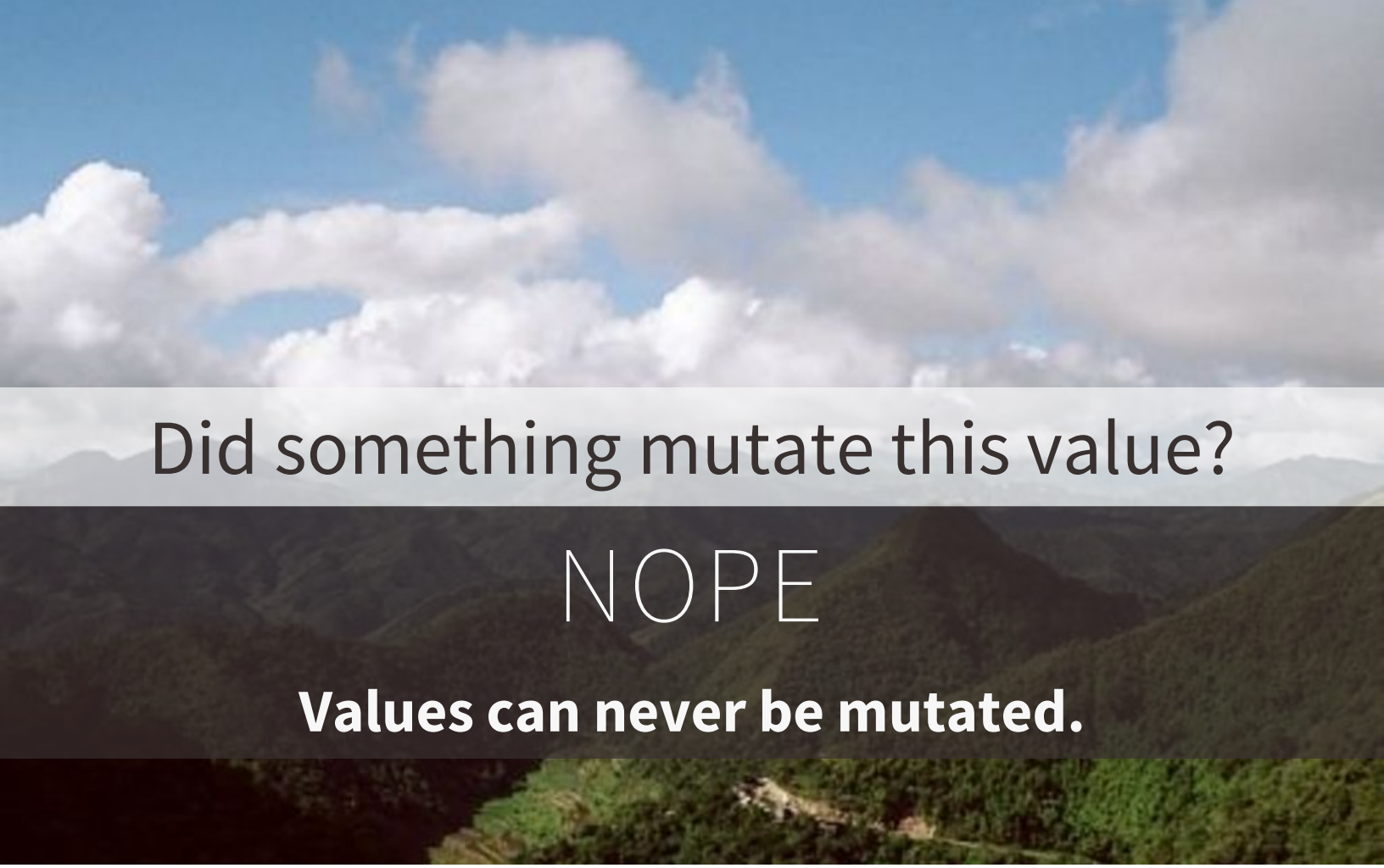
Everything in Elm is either immutable data or a stateless function. A stateless function (aka pure function, aka referential transparency) is one that looks at inputs and returns a value, and does nothing else. No side effects, no looking at the outside world, just a translation from inputs to a single return value.

Instead of side effects, Elm has Tasks. Tasks are just data representing the effect you want done - like writing down a series of instructions, rather than actually *doing* them.

Running this code instantiates the task, but does not run it.

Clearly, these design constraints have a cost. But what do they get you? Guarantees.

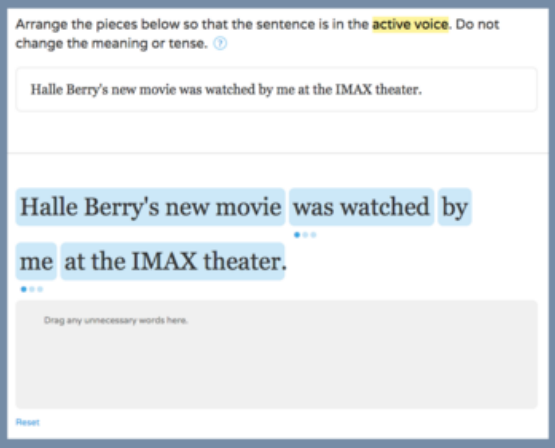
This makes debugging easier. If a function calls five other functions, which in turn call five other functions, did any of them modify localStorage? If they end up returning a String, the answer is definitely no.



You also don't have to worry about unrelated functions impacting one another. The only way one function can have anything to do with another is if one of them explicitly calls the other.

This makes refactoring really, really nice.

In Elm, this is how things typically are. It's great!



If I'd written this in Elm instead, it would have shipped sooner, even counting setup costs.

Looking back on the experience of shipping Active & Passive Voice, I couldn't help but think this.

I knew this was out there, but I didn't do it. Why not?

There's a risk to using any new technology. Even if you know what you're doing, you never quite know you can pull it off until you...well, leap.

risk
setup time
learning curve

I thought back to our experience with React. At the time, React had just come out, so it was a big risk with significant setup time and learning curve.

risk

setup time

learning curve

SO WORTH IT

...and it was so, so worth it!

This Tina Fey quote expresses how I ended up thinking about this. Go down the chute!

This was how we got started: choose a small, unscary project and rewrite it in Elm.

PORTS

talk to JavaScript by passing data

NOT SHARING CODE

We used ports to rewrite one single Flux store in Elm.

“Introducing Elm to a JS Web App"

`http://noredinktech.tumblr.com`

If you'd like a full walkthrough of how we did this, we blogged about it!
http://noredinktech.tumblr.com/post/126978281075/walkthrough-introducing-elm-to-a-js-web-app

risk
setup time
learning curve

So we had the same situation as we did with React: risk, setup time and learning curve....

risk

setup cost

learning curve

OMG SO WORTH IT

...and the result was even *more* worth it!

noredink

fewer bugs

improved team spirit

increase in job applicants

surprisingly low learning curve

Here's a short list of the benefits we've seen since adopting Elm.

By the way, we're hiring! https://www.noredink.com/jobs

clean UI architecture

delightfully helpful compiler

semantic versioning guaranteed

CHANGE IS CHEAP

In summary, this is what it's been like using Elm.

What a change!

Why can’t JavaScript be

MORE LIKE RUBY?

Whereas before I was asking this...

...Why can't Ruby be

MORE LIKE ELM?

...Now I find myself asking this.

Thanks!