Extracted from:

# React for Real

## Front-End Code, Untangled



The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

# React for Real

## Front-End Code, Untangled

Ludovico Fischer

*edited by Brian P. Hogan*

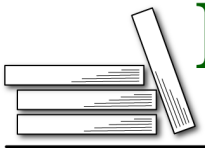
# React for Real

## Front-End Code, Untangled

Ludovico Fischer

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf



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
Development Editor: Brian P. Hogan
Copy Editor: Nicole Abramowtiz
Layout: Gilson Graphics

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: 978-1-68050-263-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2017

CHAPTER 2

# Work with State and Events

Web applications are about dynamic data: you have to update the information you display in response to user actions and HTTP responses. In this chapter, you'll learn how to model data that changes over time, and you'll learn how to respond to user events and HTTP responses. You'll discover how to transform the word counter into a fully reactive interface.

## Get to Know State

We'd like to allow users to edit the word counter text. For this, we need some way to handle values that change over time: a classic book asserts that an object has *state* if its behavior is influenced by its history (*Structure and Interpretation of Computer Programs [AS96]*).

Let's take a little detour to understand state. Open your browser's JavaScript console and type the `add()` function, which just adds its arguments together and returns the result:

uptospeed/src/functions.js

```
function add(x, y) {
  return x + y;
}
```

Then call the `add()` method a few times:

```
add(1,2); // returns 3
add(5, 7); // returns 12
add(1, 2); // returns 3 again
add(5, 7); // returns 12 again
```

The return value of `add()` depends only on its arguments, because it is a pure function.

Now type the `addS()` function, which multiplies the second argument by `n` before adding it to the first argument, and increments `n` each time you call it.

uptospeed/src/functions.js

```
let n = 1;

function addS(x, y) {
  const result = x + n * y;
  n++;
  return result;
}
```

The return value changes each time you call `addS`, even if the arguments stay the same, because the return value depends on how many times you've called `addS`. `addS` has a history: it stores state in the `n` variable.

The potential for bugs grows as the program grows. State complicates applications, as the same function, with the same arguments, can return different results based on when you call it.

We never tried to update a component's props inside the component itself, for a good reason: inside a React component, props are immutable (in practice, you can assign a new value to a prop inside a component, but it won't have any effect). To handle state, you need an additional mechanism. One good option is learning the tools that React itself provides. They get the job done in the majority of situations with little boilerplate.