

**SURVIVE**JS

# React

FROM APPRENTICE TO MASTER



Juho Vepsäläinen

# SurviveJS - React

From apprentice to master

Juho Vepsäläinen

This book is for sale at <http://leanpub.com/survivejs-react>

This version was published on 2016-06-09



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.



This work is licensed under a [Creative Commons Attribution-NonCommercial3.0 3.0 Unported License](#)

# Contents

<b>Introduction</b>	<b>i</b>
What is React?	i
What Will You Learn?	ii
How is This Book Organized?	ii
What is Kanban?	iii
Who is This Book for?	iv
How to Approach the Book?	iv
Book Versioning	iv
Extra Material	v
Getting Support	vi
Announcements	vi
Acknowledgments	vii
 <b>I Getting Started</b>	 <b>1</b>
1. Introduction to React	2
1.1 What is React?	2
1.2 Virtual DOM	3
1.3 React Renderers	4
1.4 <code>React.createElement</code> and JSX	5
1.5 Conclusion	6
2. Setting Up the Project	7
2.1 Setting Up Node.js and Git	7
2.2 Running the Project	8
2.3 Boilerplate npm scripts	9
2.4 Boilerplate Language Features	10
2.5 Conclusion	12
3. Implementing a Note Application	13
3.1 Initial Data Model	13
3.2 Rendering Initial Data	14
3.3 Generating the Ids	15

## CONTENTS

3.4	Adding New Notes to the List . . . . .	16
3.5	Conclusion . . . . .	24
<b>4.</b>	<b>Deleting Notes . . . . .</b>	<b>25</b>
4.1	Separating Note . . . . .	25
4.2	Adding a Stub for onDelete Callback . . . . .	26
4.3	Communicating Deletion to App . . . . .	27
4.4	Conclusion . . . . .	29
<b>5.</b>	<b>Understanding React Components . . . . .</b>	<b>30</b>
5.1	Lifecycle Methods . . . . .	30
5.2	Refs . . . . .	31
5.3	Custom Properties and Methods . . . . .	32
5.4	React Component Conventions . . . . .	33
5.5	Conclusion . . . . .	34
<b>6.</b>	<b>Editing Notes . . . . .</b>	<b>35</b>
6.1	Implementing Editable . . . . .	35
6.2	Extracting Rendering from Note . . . . .	36
6.3	Adding Editable Stub . . . . .	37
6.4	Connecting Editable with Notes . . . . .	38
6.5	Tracking Note editing State . . . . .	39
6.6	Implementing Edit . . . . .	41
6.7	On Namespacing Components . . . . .	43
6.8	Conclusion . . . . .	44
<b>7.</b>	<b>Styling the Notes Application . . . . .</b>	<b>45</b>
7.1	Styling “Add Note” Button . . . . .	45
7.2	Styling Notes . . . . .	46
7.3	Styling Individual Notes . . . . .	47
7.4	Conclusion . . . . .	50

## II Implementing Kanban . . . . . 51

<b>8.</b>	<b>React and Flux . . . . .</b>	<b>52</b>
8.1	Quick Introduction to Redux . . . . .	52
8.2	Quick Introduction to MobX . . . . .	52
8.3	Which Data Management Solution to Use? . . . . .	53
8.4	Introduction to Flux . . . . .	53
8.5	Porting to Alt . . . . .	55
8.6	Understanding connect . . . . .	59
8.7	Dispatching in Alt . . . . .	64
8.8	Conclusion . . . . .	64

## CONTENTS

<b>9. Implementing NoteStore and NoteActions</b>	<b>65</b>
9.1 Setting Up a NoteStore	65
9.2 Understanding Actions	68
9.3 Setting Up NoteActions	69
9.4 Connecting NoteActions with NoteStore	70
9.5 Porting App.addNote to Flux	71
9.6 Porting App.deleteNote to Flux	72
9.7 Porting App.activateNoteEdit to Flux	74
9.8 Porting App.editNote to Flux	75
9.9 Conclusion	77
<b>10. Implementing Persistency over localStorage</b>	<b>78</b>
10.1 Understanding localStorage	78
10.2 Implementing a Wrapper for localStorage	79
10.3 Persisting the Application Using FinalStore	79
10.4 Implementing the Persistency Logic	80
10.5 Connecting Persistency Logic with the Application	80
10.6 Cleaning Up NoteStore	81
10.7 Alternative Implementations	82
10.8 Relay?	83
10.9 Conclusion	83
<b>11. Handling Data Dependencies</b>	<b>84</b>
11.1 Defining Lanes	84
11.2 Connecting Lanes with App	86
11.3 Modeling Lane	87
11.4 Making Lanes Responsible of Notes	89
11.5 Extracting LaneHeader from Lane	96
11.6 Conclusion	99
<b>12. Editing Lanes</b>	<b>100</b>
12.1 Implementing Editing for Lane names	100
12.2 Implementing Lane Deletion	102
12.3 Styling Kanban Board	104
12.4 Conclusion	107
<b>13. Implementing Drag and Drop</b>	<b>108</b>
13.1 Setting Up React DnD	108
13.2 Allowing Notes to Be Dragged	109
13.3 Allowing Notes to Detect Hovered Notes	111
13.4 Developing onMove API for Notes	112
13.5 Adding Action and Store Method for Moving	115
13.6 Implementing Note Drag and Drop Logic	117
13.7 Dragging Notes to Empty Lanes	119

## CONTENTS

13.8	Conclusion . . . . .	123
------	----------------------	-----

## III Advanced Techniques . . . . . 124

14.	Testing React . . . . .	125
14.1	Levels of Testing . . . . .	125
14.2	Writing Your First Test . . . . .	129
14.3	Understanding the Test Setup . . . . .	129
14.4	Testing Kanban Components . . . . .	131
14.5	Testing Editable . . . . .	131
14.6	Testing Note . . . . .	135
14.7	Testing Kanban Stores . . . . .	136
14.8	Conclusion . . . . .	140
15.	Typing with React . . . . .	141
15.1	propTypes and defaultProps . . . . .	141
15.2	Typing Kanban . . . . .	143
15.3	Type Checking with Flow . . . . .	148
15.4	Converting propTypes to Flow Checks . . . . .	151
15.5	Babel Typecheck . . . . .	156
15.6	TypeScript . . . . .	157
15.7	Conclusion . . . . .	158
16.	Styling React . . . . .	159
16.1	Old School Styling . . . . .	159
16.2	CSS Methodologies . . . . .	159
16.3	CSS Processors . . . . .	161
16.4	React Based Approaches . . . . .	162
16.5	CSS Modules . . . . .	168
16.6	Conclusion . . . . .	170
17.	Structuring React Projects . . . . .	171
17.1	Directory per Concept . . . . .	171
17.2	Directory per Component . . . . .	172
17.3	Directory per View . . . . .	173
17.4	Conclusion . . . . .	174

## Appendices . . . . . 176

Language Features . . . . .	177
Modules . . . . .	177
Classes . . . . .	179
Class Properties and Property Initializers . . . . .	181

## CONTENTS

Functions . . . . .	183
String Interpolation . . . . .	186
Destructuring . . . . .	186
Object Initializers . . . . .	187
const, let, var . . . . .	188
Decorators . . . . .	188
Conclusion . . . . .	188
<b>Understanding Decorators . . . . .</b>	<b>189</b>
Implementing a Logging Decorator . . . . .	189
Implementing @connect . . . . .	191
Decorator Ideas . . . . .	192
Conclusion . . . . .	193
<b>Troubleshooting . . . . .</b>	<b>194</b>
EPEERINVALID . . . . .	194
Warning: setState(...): Cannot update during an existing state transition . . . . .	195
Warning: React attempted to reuse markup in a container but the checksum was invalid . . . . .	195
Module parse failed . . . . .	195
Project Fails to Compile . . . . .	196

# Introduction

Front-end development moves forward fast. A good indication of this is the pace at which new technologies appear to the scene. [React](https://facebook.github.io/react/)<sup>1</sup> is one of these recent newcomers. Even though the technology itself is simple, there's a lot going on around it.

The purpose of this book is to help you get started with React and provide understanding of the surrounding ecosystem so you know where to look.

Our development setup is based on Webpack. There's [a separate book](http://survivejs.com/webpack/introduction/)<sup>2</sup> that digs into it, but I don't expect you to understand Webpack well to get through this book.

## What is React?

Facebook's React, a JavaScript library, is a component based view abstraction. A component could be a form input, button, or any other element in your user interface. This provides an interesting contrast to earlier approaches as React isn't bound to the DOM by design. You can use it to implement mobile applications for example.

## React is Only One Part of the Whole

Given React focuses only on the view, you'll likely have to complement it with other libraries to give you the missing bits. This provides an interesting contrast to framework based approaches as they give you a lot more out of the box. Both approaches have their merits. In this book, we will focus on the library oriented approach.

Ideas introduced by React have influenced the development of the frameworks. Most importantly it has helped us to understand how well component based thinking fits web applications.

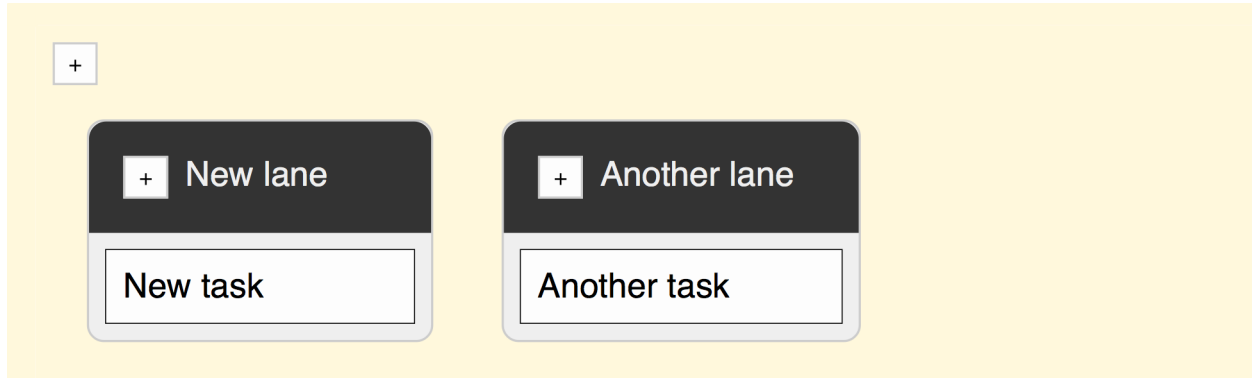
---

<sup>1</sup><https://facebook.github.io/react/>

<sup>2</sup><http://survivejs.com/webpack/introduction/>



## What Will You Learn?



Kanban application

This book teaches you to build a [Kanban](https://en.wikipedia.org/wiki/Kanban)<sup>3</sup> application. Beyond this, more theoretical aspects of web development are discussed. Completing the project gives you a good idea of how to implement something on your own. During the process you will learn why certain libraries are useful and will be able to justify your technology choices better.

## How is This Book Organized?

To get started, we will develop a small clone of a famous [Todo application](http://todomvc.com/)<sup>4</sup>. This leads us to problems of scaling. Sometimes, you need to do things the dumb way to understand why better solutions are needed after all.

We will generalize from there and put [Flux architecture](https://facebook.github.io/flux/docs/overview.html)<sup>5</sup> in place. We will apply some [Drag and Drop \(DnD\) magic](https://gaearon.github.io/react-dnd/)<sup>6</sup> and start dragging things around. Finally, we will get a production grade build done.

The final, theoretical part of the book covers more advanced topics. If you are reading the commercial edition of this book, there's something extra in it for you. I will show you how to deal with typing in React in order to produce higher quality code. You will also learn to test your components and logic. You will learn to style your React application in emerging ways and have a better idea of how to structure your project.

The appendices at the end are meant to give food for thought and explain aspects, such as language features, in greater detail. If there's a bit of syntax that seems weird to you in the book, you'll likely find more information there.

---

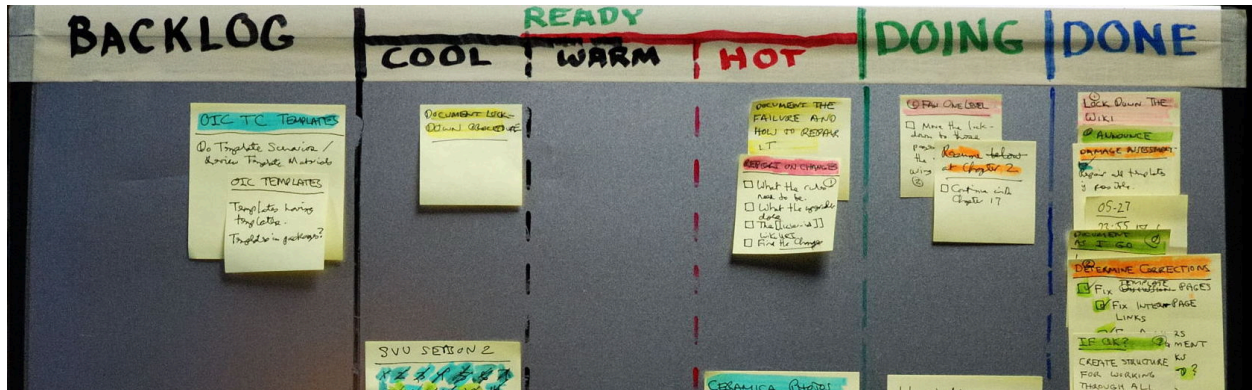
<sup>3</sup><https://en.wikipedia.org/wiki/Kanban>

<sup>4</sup><http://todomvc.com/>

<sup>5</sup><https://facebook.github.io/flux/docs/overview.html>

<sup>6</sup><https://gaearon.github.io/react-dnd/>

## What is Kanban?



Kanban by Dennis Hamilton (CC BY)

Kanban, originally developed at Toyota, allows you to track the status of tasks. It can be modeled in terms of Lanes and Notes. Notes move through Lanes representing stages from left to right as they become completed. Notes themselves can contain information about the task itself, its priority, and so on as required.

The system can be extended in various ways. One simple way is to apply a Work In Progress (WIP) limit per lane. The effect of this is that you are forced to focus on getting tasks done. That is one of the good consequences of using Kanban. Moving those notes around is satisfying. As a bonus you get visibility and know what is yet to be done.

## Where to Use Kanban?

This system can be used for various purposes, including software and life management. You could use it to track your personal projects or life goals for instance. Even though it's a simple tool, it's quite powerful, and you can find use for it in many places.

## How to Build a Kanban?

The simplest way to build a Kanban is to get a bunch of Post-it notes and find a wall. After that, you split it up into columns. These Lanes could consist of the following stages: Todo, Doing, Done. All Notes would go to Todo initially. As you begin working on them, you would move them to Doing, and finally, to Done when completed. This is the simplest way to get started.

This is just one example of a lane configuration. The lanes can be configured to match your process. There can be approval steps for instance. If you are modeling a software development process, you could have separate lanes for testing and deployment for instance.

## Available Kanban Implementations

[Trello](#)<sup>7</sup> is perhaps the most known online implementation of Kanban. Sprintly has open sourced their [React implementation of Kanban](#)<sup>8</sup>. Meteor based [wekan](#)<sup>9</sup> is another good example. Ours won't be as sophisticated as these, but it will be enough to get started.

## Who is This Book for?

I expect that you have a basic knowledge of JavaScript and Node.js. You should be able to use npm on an elementary level. If you know something about React, or ES6, that's great. By reading this book you will deepen your understanding of these technologies.

One of the hardest things about writing a book is to write it on the right level. Given the book covers a lot of ground, there are appendices that cover basic topics, such as language details, with greater detail than the main content does. So if you are feeling unsure of something, check them out.

There's also a [community chat](#)<sup>10</sup> available. If you want to ask something directly, we are there to help. Any comments you might have will go towards improving the book content. The last thing I want is to have someone struggling with the book.

## How to Approach the Book?

Although a natural way to read a book is to start from the first chapter and then read the chapters sequentially, that's not the only way to approach this book. The chapter order is just a reading suggestion. Depending on your background, you could consider skimming through the first part and then digging deeper into the advanced topics.

The book doesn't cover everything you need to know in order to develop front-end applications. That's simply too much for a single book. I do believe, however, that it might be able to push you in the right direction. The ecosystem around React is fairly large and I've done my best to cover a good chunk of it.

Given the book relies on a variety of new language features, I've gathered the most important ones used to a separate *Language Features* appendix that provides a quick look at them. If you want to understand the features in isolation or feel unsure of something, that's a good place to look.

## Book Versioning

As this book receives a fair amount of maintenance and improvements due to the pace of innovation, there's a rough versioning scheme in place. I maintain release notes for each new version at the [book](#)

---

<sup>7</sup><https://trello.com/>

<sup>8</sup><https://github.com/sprintly/sprintly-kanban>

<sup>9</sup><https://github.com/wekan/wekan>

<sup>10</sup><https://gitter.im/survivejs/react>

[blog](#)<sup>11</sup> to describe what has changed between versions. Also examining the GitHub repository may be beneficial. I recommend using the GitHub *compare* tool for this purpose. Example:

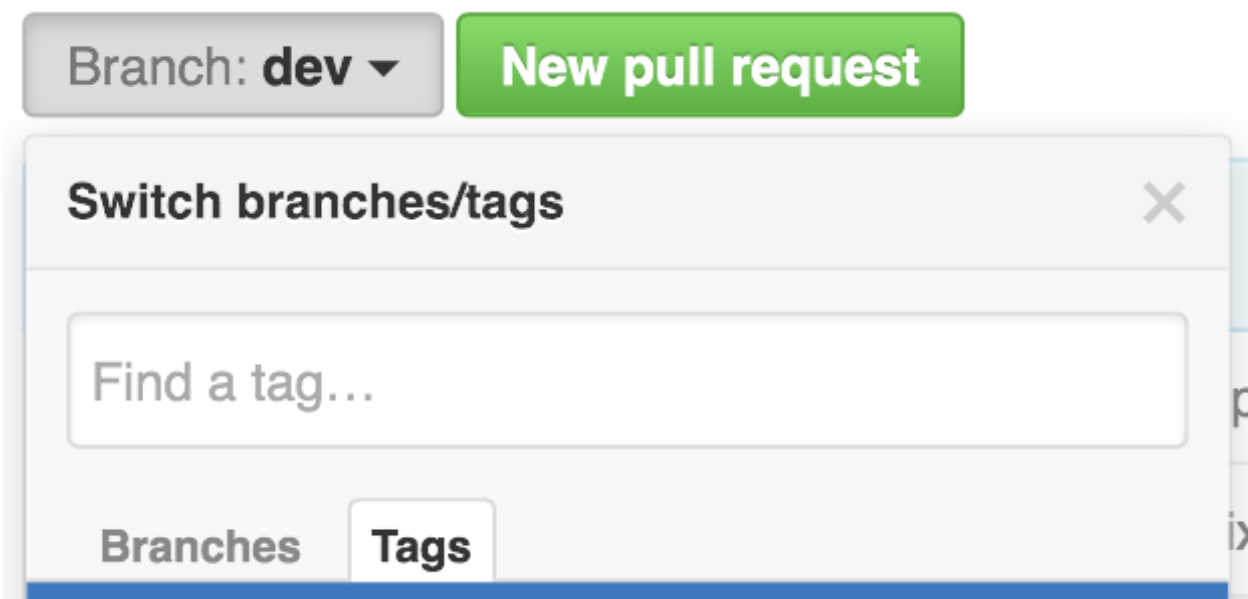
<https://github.com/survivejs/react/compare/v2.1.0...v2.5.6>

The page will show you the individual commits that went to the project between the given version range. You can also see the lines that have changed in the book. This excludes the private chapters, but it's enough to give you a good idea of the major changes made to the book.

The current version of the book is 2.5.6.

## Extra Material

The book content and source are available at the [book's repository at GitHub](#)<sup>12</sup>. Please note that the repository defaults to the dev branch of the project as this makes it convenient to contribute. To find source matching the version of the book you are reading, use the tag selector at GitHub's user interface as in the image below:



GitHub tag selector

The book repository contains code per chapter. This means you can start from anywhere you want without having to type it all through yourself. If you are unsure of something, you can always refer to that.

---

<sup>11</sup><http://survivejs.com/blog/>

<sup>12</sup><https://github.com/survivejs/react>

You can find a lot of complementary material at the [survivejs organization](#)<sup>13</sup>. Examples of this are alternative implementations of the application available written in [MobX](#)<sup>14</sup>, [Redux](#)<sup>15</sup>, and [Cerebral/Baobab](#)<sup>16</sup>. Studying those can give you a good idea of how different architectures work out using the same example.

## Getting Support

As no book is perfect, you will likely come by issues and might have some questions related to the content. There are a couple of options to deal with this:

- Contact me through the [GitHub Issue Tracker](#)<sup>17</sup>
- Join me at the [Gitter Chat](#)<sup>18</sup>
- Follow [@survivejs](#)<sup>19</sup> at Twitter for official news or poke me through [@bebraw](#)<sup>20</sup> directly
- Send me email to [info@survivejs.com](mailto:info@survivejs.com)<sup>21</sup>
- Ask me anything about Webpack or React at [SurviveJS AmA](#)<sup>22</sup>

If you post questions to Stack Overflow, tag them using **survivejs** so I will get notified of them. You can use the hashtag **#survivejs** at Twitter for same effect.

I have tried to cover some common issues at the *Troubleshooting* appendix. That will be expanded as common problems are found.

## Announcements

I announce SurviveJS related news through a couple of channels:

- [Mailing list](#)<sup>23</sup>
- [Twitter](#)<sup>24</sup>
- [Blog RSS](#)<sup>25</sup>

Feel free to subscribe.

---

<sup>13</sup><https://github.com/survivejs/>

<sup>14</sup><https://github.com/survivejs/mobx-demo>

<sup>15</sup><https://github.com/survivejs/redux-demo>

<sup>16</sup><https://github.com/survivejs/cerebral-demo>

<sup>17</sup><https://github.com/survivejs/react/issues>

<sup>18</sup><https://gitter.im/survivejs/react>

<sup>19</sup><https://twitter.com/survivejs>

<sup>20</sup><https://twitter.com/bebraw>

<sup>21</sup><mailto:info@survivejs.com>

<sup>22</sup><https://github.com/survivejs/ama/issues>

<sup>23</sup><http://eepurl.com/bth1v5>

<sup>24</sup><https://twitter.com/survivejs>

<sup>25</sup><http://survivejs.com/atom.xml>

## Acknowledgments

An effort like this wouldn't be possible without community support. There are a lot of people to thank as a result!

Big thanks to [Christian Alfoni](#)<sup>26</sup> for starting the [react-webpack-cookbook](#)<sup>27</sup> with me. That work eventually led to this book and eventually became [a book of its own](#)<sup>28</sup>.

The book wouldn't be half as good as it is without patient editing and feedback by my editor [Jesús Rodríguez Rodríguez](#)<sup>29</sup>. Thank you.

Special thanks to Steve Piercy for numerous contributions. Thanks to [Prospect One](#)<sup>30</sup> and [Dixon & Moe](#)<sup>31</sup> for helping with the logo and graphical outlook. Thanks for proofreading to Ava Mallory and EditorNancy from fiverr.com.

Numerous individuals have provided support and feedback along the way. Thank you in no particular order Vitaliy Kotov, @af7, Dan Abramov, @dnmd, James Cavanaugh, Josh Perez, Nicholas C. Zakas, Ilya Volodin, Jan Nicklas, Daniel de la Cruz, Robert Smith, Andreas Eldh, Brandon Tilley, Braden Evans, Daniele Zannotti, Partick Forringer, Rafael Xavier de Souza, Dennis Bunskoek, Ross Mackay, Jimmy Jia, Michael Bodnarchuk, Ronald Borman, Guy Ellis, Mark Penner, Cory House, Sander Wapstra, Nick Ostrovsky, Oleg Chiruhin, Matt Brookes, Devin Pastoor, Yoni Weisbrod, Guyon Moree, Wilson Mock, Herryanto Siatono, Héctor Cascos, Erick Bazán, Fabio Bedini, Gunnari Auvinen, Aaron McLeod, John Nguyen, Hasitha Liyanage, Mark Holmes, Brandon Dail, Ahmed Kamal, Jordan Harband, Michel Weststrate, Ives van Hoorne, Luca DeCaprio, @dev4Fun, Fernando Montoya, Hu Ming, @mpr0xy, David “@davegomez” Gómez, Aleksey Guryanov, Elio D’antoni, Yosi Taguri, Ed McPadden, Wayne Maurer, Adam Beck, Omid Hezaveh, Connor Lay, Nathan Grey, Avishay Orpaz, Jax Cavallera, Juan Diego Hernández, Peter Poulsen, Harro van der Klauw, Tyler Anton, Michael Kelley, @xuyuanme, @RogerSep, Jonathan Davis, @snowyplover, Tobias Koppers, Diego Toro, George Hilios, Jim Alateras, @atleb, Andy Klimczak, James Anaipakos, Christian Hettlage, Sergey Lukin, Matthew Toledo, Talha Mansoor, Pawel Chojnacki, @eMerzh, Gary Robinson, Omar van Galen, Jan Van Bruggen, Savio van Hoi, Alex Shepard, Derek Smith, Tetsushi Omi, Maria Fisher, Rory Hunter, Dario Carella, Toni Laukka, Blake Dietz, Felipe Almeida, Greg Kedge, Deepak Kannan, Jake Peyser, Alfred Lau, Tom Byrer, Stefanos Grammenos, Lionel Ringenbach, Hamilton Greene, Daniel Robinson, and many others. If I’m missing your name, I might have forgotten to add it.

---

<sup>26</sup><http://www.christianalfoni.com/>

<sup>27</sup><https://github.com/christianalfoni/react-webpack-cookbook>

<sup>28</sup><http://survivejs.com/webpack/introduction>

<sup>29</sup><https://github.com/Foxandxss>

<sup>30</sup><http://prospectone.pl/>

<sup>31</sup><http://dixonandmoe.com/>

# I Getting Started

React, despite being a young library, has had a significant impact on the front-end development community. It introduced concepts, such as the virtual DOM, and made the community understand the power of components. Its component oriented design approach works well for the web. But React isn't limited to the web. You can use it to develop mobile and even terminal user interfaces.

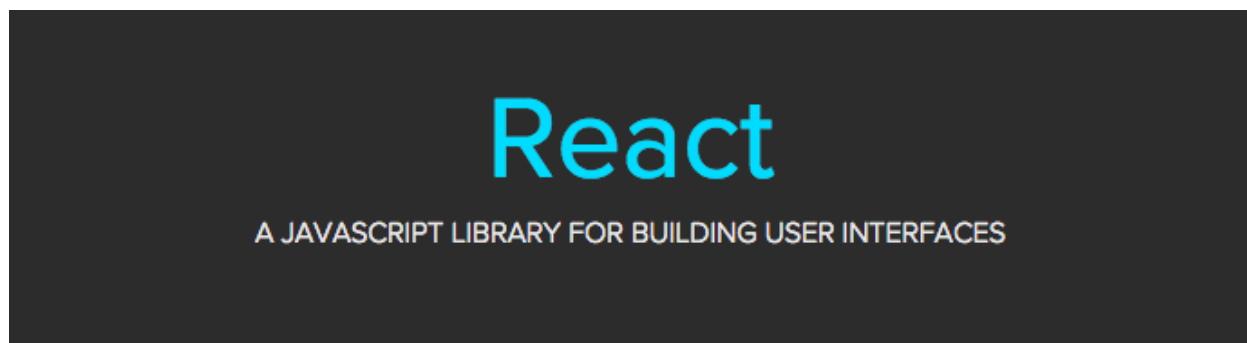
In this part, we will start digging into React and implement a small Notes application. That is something that will eventually become a Kanban. You will learn the basics of React and get used to working with it.

# 1. Introduction to React

Facebook's [React](https://facebook.github.io/react/)<sup>1</sup> has changed the way we think about web applications and user interface development. Due to its design, you can use it beyond web. A feature known as the **Virtual DOM** enables this.

In this chapter we'll go through some of the basic ideas behind the library so you understand React a little better before moving on.

## 1.1 What is React?



React

React is a JavaScript library that forces you to think in terms of components. This model of thinking fits user interfaces well. Depending on your background it might feel alien at first. You will have to think very carefully about the concept of state and where it belongs.

Because **state management** is a difficult problem, a variety of solutions have appeared. In this book, we'll start by managing state ourselves and then push it to a Flux implementation known as Alt. There are also implementations available for several other alternatives, such as Redux, MobX, and Cerebral.

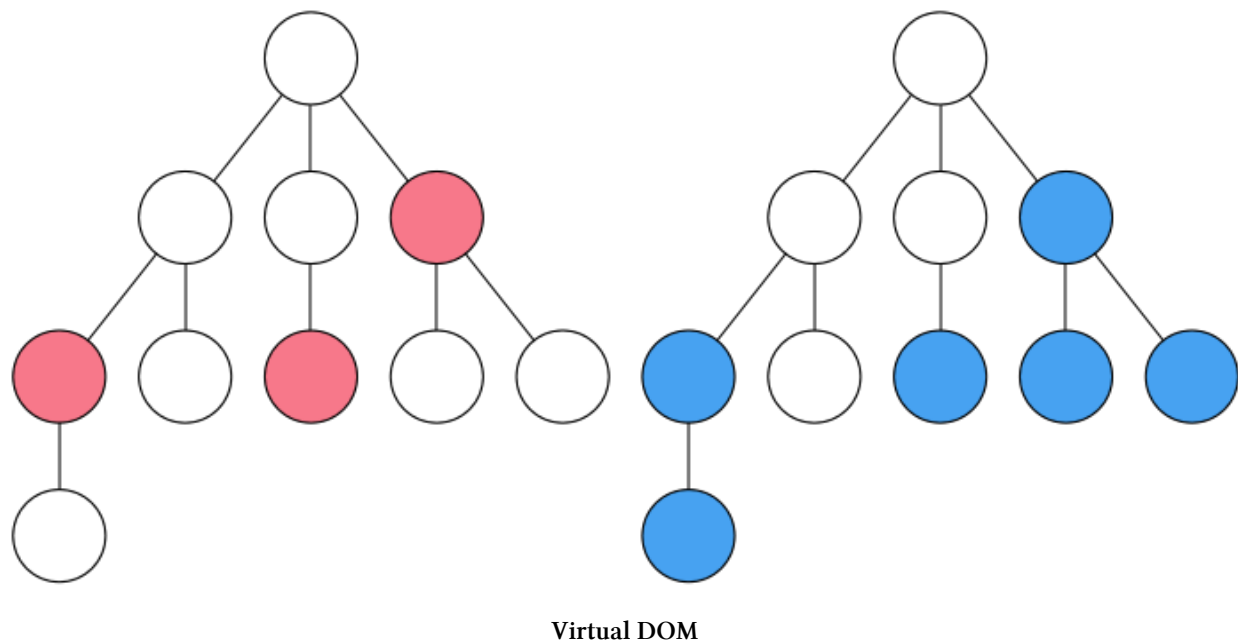
React is pragmatic in the sense that it contains a set of escape hatches. If the React model doesn't work for you, it is still possible to revert back to something lower level. For instance, there are hooks that can be used to wrap older logic that relies on the DOM. This breaks the abstraction and ties your code to a specific environment, but sometimes that's the pragmatic thing to do.

---

<sup>1</sup><https://facebook.github.io/react/>



## 1.2 Virtual DOM



One of the fundamental problems of programming is how to deal with state. Suppose you are developing a user interface and want to show the same data in multiple places. How do you make sure the data is consistent?

Historically we have mixed the concerns of the DOM and state and tried to manage it there. React solves this problem in a different way. It introduced the concept of the **Virtual DOM** to the masses.

Virtual DOM exists on top of the actual DOM, or some other render target. It solves the state manipulation problem in its own way. Whenever changes are made to it, it figures out the best way to batch the changes to the underlying DOM structure. It is able to propagate changes across its virtual tree as in the image above.

## Virtual DOM Performance

Handling the DOM manipulation this way can lead to increased performance. Manipulating the DOM by hand tends to be inefficient and is hard to optimize. By leaving the problem of DOM manipulation to a good implementation, you can save a lot of time and effort.

React allows you to tune performance further by implementing hooks to adjust the way the virtual tree is updated. Though this is often an optional step.

The biggest cost of Virtual DOM is that the implementation makes React quite big. You can expect the bundle sizes of small applications to be around 150-200 kB minified, React included. gzipping will help, but it's still big.



Solutions such as [preact](#)<sup>2</sup> and [react-lite](#)<sup>3</sup> allow you to reach far smaller bundle sizes while sacrificing some functionality. If you are size conscious, consider checking out these solutions.



Libraries, such as [Matt-Esch/virtual-dom](#)<sup>4</sup> or [paldepind/snabbdom](#)<sup>5</sup>, focus entirely on Virtual DOM. If you are interested in the theory and want to understand it further, check these out.

## 1.3 React Renderers

As mentioned, React's approach decouples it from the web. You can use it to implement interfaces across multiple platforms. In this case we'll be using a renderer known as [react-dom](#)<sup>6</sup>. It supports both client and server side rendering.

### Universal Rendering

We could use react-dom to implement so called *universal* rendering. The idea is that the server renders the initial markup and passes the initial data to the client. This improves performance by avoiding unnecessary round trips as each request comes with an overhead. It is also useful for search engine optimization (SEO) purposes.

Even though the technique sounds simple, it can be difficult to implement for larger scale applications. But it's still something worth knowing about.

Sometimes using the server side part of react-dom is enough. You can use it to [generate invoices](#)<sup>7</sup> for example. That's one way to use React in a flexible manner. Generating reports is a common need after all.

### Available React Renderers

Even though react-dom is the most used renderer, there are a few others you might want to be aware of. I've listed some of the well known alternatives below:

- [React Native](#)<sup>8</sup> - React Native is a framework and renderer for mobile platforms including iOS and Android. You can also run [React Native applications on the web](#)<sup>9</sup>.

---

<sup>2</sup><https://developit.github.io/preact/>

<sup>3</sup><https://github.com/Lucifier129/react-lite>

<sup>4</sup><https://github.com/Matt-Esch/virtual-dom>

<sup>5</sup><https://github.com/paldepind/snabbdom>

<sup>6</sup><https://www.npmjs.com/package/react-dom>

<sup>7</sup><https://github.com/bebraw/generate-invoice>

<sup>8</sup><https://facebook.github.io/react-native/>

<sup>9</sup><https://github.com/necolas/react-native-web>

- [react-blessed<sup>10</sup>](https://github.com/Yomguithereal/react-blessed) - react-blessed allows you to write terminal applications using React. It's even possible to animate them.
- [gl-react<sup>11</sup>](https://projectseptemberinc.gitbooks.io/gl-react/content/) - gl-react provides WebGL bindings for React. You can write shaders this way for example.
- [react-canvas<sup>12</sup>](https://github.com/Flipboard/react-canvas) - react-canvas provides React bindings for the Canvas element.

## 1.4 React.createElement and JSX

Given we are operating with virtual DOM, there's a [high level API<sup>13</sup>](https://facebook.github.io/react/docs/top-level-api.html) for handling it. A naïve React component written using the JavaScript API could look like this:

```
const Names = () => {  
  const names = ['John', 'Jill', 'Jack'];  
  
  return React.createElement(  
    'div',  
    null,  
    React.createElement('h2', null, 'Names'),  
    React.createElement(  
      'ul',  
      { className: 'names' },  
      names.map(name => {  
        return React.createElement(  
          'li',  
          { className: 'name' },  
          name  
        );  
      })  
    )  
  );  
};
```

As it is verbose to write components this way and the code is quite hard to read, often people prefer to use a language known as [JSX<sup>14</sup>](https://facebook.github.io/jsx/) instead. Consider the same component written using JSX below:

---

<sup>10</sup><https://github.com/Yomguithereal/react-blessed>

<sup>11</sup><https://projectseptemberinc.gitbooks.io/gl-react/content/>

<sup>12</sup><https://github.com/Flipboard/react-canvas>

<sup>13</sup><https://facebook.github.io/react/docs/top-level-api.html>

<sup>14</sup><https://facebook.github.io/jsx/>

```
const Names = () => {  
  const names = ['John', 'Jill', 'Jack'];  
  
  return (  
    <div>  
      <h2>Names</h2>  
  
      {/* This is a list of names */}  
      <ul className="names">{  
        names.map(name =>  
          <li className="name">{name}</li>  
        )  
      }</ul>  
    </div>  
  );  
};
```

Now we can see the component renders a set of names within a HTML list. It might not be the most useful component, but it's enough to illustrate the basic idea of JSX. It provides us a syntax that resembles HTML. It also provides a way to write JavaScript within it by using braces ({}).

Compared to vanilla HTML, we are using `className` instead of `class`. This is because the API has been modeled after the DOM naming. It takes some getting used to and you might experience a [JSX shock](#)<sup>15</sup> until you begin to appreciate the approach. It gives us an additional level of validation.



[HyperScript](#)<sup>16</sup> is an interesting alternative to JSX. It provides a JavaScript based API and as such is closer to the metal. You can use the syntax with React through [hyperscript-helpers](#)<sup>17</sup>.



There is a semantic difference between React components and React elements. In the example each of those JSX nodes would be converted into an element. In short, components can have state whereas elements are simpler by nature. They are just pure objects. Dan Abramov goes into further detail in a [blog post](#)<sup>18</sup> of his.

## 1.5 Conclusion

Now that we have a rough understanding of what React is, we can move onto something more technical. It's time to get a small project up and running.

---

<sup>15</sup><https://medium.com/@housecor/react-s-jsx-the-other-side-of-the-coin-2ace7ab62b98>

<sup>16</sup><https://github.com/dominictarr/hyperscript>

<sup>17</sup><https://www.npmjs.com/package/hyperscript-helpers>

<sup>18</sup><https://facebook.github.io/react/blog/2015/12/18/react-components-elements-and-instances.html>

## 2. Setting Up the Project

To make it easier to get started, I've set up a simple Webpack based boilerplate that allows us to dig into React straight away. The boilerplate includes a development mode with a feature known as *hot loading* enabled.

Hot loading allows Webpack to patch the code running in the browser without a full refresh. It works the best especially with styling although React supports it fairly well too.

Unfortunately it's not a fool proof technology and it won't be able to detect all changes made to the code. This means there will be times when you need to force a hard refresh to make the browser to catch the recent changes.



Common editors (Sublime Text, Visual Studio Code, vim, emacs, Atom and such) have good support for React. Even IDEs, such as [WebStorm](https://www.jetbrains.com/webstorm/)<sup>1</sup>, support it up to an extent. [Nuclide](https://nuclide.io/)<sup>2</sup>, an Atom based IDE, has been developed with React in mind. Make sure you have React related plugins installed and enabled.



If you use an IDE, disable a feature known as **safe write**. It is known to cause issues with the setup we'll be using in this book.

### 2.1 Setting Up Node.js and Git

To get started, make sure you have fresh versions of [Node.js](https://nodejs.org/)<sup>3</sup> and [Git](https://git-scm.com/)<sup>4</sup> installed. I recommend using at least the LTS version of Node.js. You might run into hard to debug issues with older versions. Same can apply to versions newer than LTS because of their bleeding edge status.



One interesting option is to manage your environment through [Vagrant](https://www.vagrantup.com/)<sup>5</sup> or a tool like [nvm](https://www.npmjs.com/package/nvm)<sup>6</sup>.

### Downloading the Boilerplate

In order to fetch the boilerplate our project needs, clone it through Git as follows at your terminal:

---

<sup>1</sup><https://www.jetbrains.com/webstorm/>

<sup>2</sup><http://nuclide.io/>

<sup>3</sup><https://nodejs.org>

<sup>4</sup><https://git-scm.com/>

<sup>5</sup><https://www.vagrantup.com/>

<sup>6</sup><https://www.npmjs.com/package/nvm>

```
git clone https://github.com/survivejs/react-boilerplate.git kanban-app
```

This will create a new directory, *kanban-app*. Inside it you can find everything we need to get ahead. As the boilerplate can change between book versions, I recommend you check out the version specific of it:

```
cd kanban-app
git checkout v2.5.6
rm -rf .git
git init
git add .
git commit -am "Initial commit"
```

After this process you have a fresh project to work against.

The repository contains a small seed application that shows Hello World! and basic Webpack configuration. To get the seed application dependencies installed, execute

```
npm install
```

Upon completion you should see a `node_modules/` directory containing the project dependencies.

## 2.2 Running the Project

To get the project running, execute `npm start`. You should see something like this at the terminal if everything went right:

```
> webpack-dev-server

http://localhost:8080/
webpack result is served from /
content is served from /Users/juhovepsalainen/Projects/tmp/kanban-app
404s will fallback to /index.html
Child html-webpack-plugin for "index.html":

webpack: bundle is now VALID.
```

In case you received an error, make sure there isn't something else running in the same port. You can run the application through some other port easily using an invocation such as `PORT=3000 npm start` (Unix only). The configuration will pick up the new port from the environment. If you want to fix the port to something specific, adjust its value at *webpack.config.js*.

Assuming everything went fine, you should see something like this at the browser:

# Hello world

Hello world

You can try modifying the source to see how hot loading works.

I'll discuss the boilerplate in greater detail next so you know how it works. I'll also cover the language features we are going to use briefly.



In case you want to start with a fresh Git history, this would be a good point to remove `.git` directory (`rm -rf .git`) and initialize the project again (`git init && git add . && git commit -am "Initial commit"`).



The techniques used by the boilerplate are covered in greater detail at [SurviveJS - Webpack](#)<sup>7</sup>.

## 2.3 Boilerplate npm scripts

Our boilerplate is able to generate a production grade build with hashing. There's also a deployment related target so that you can show your project to other people through [GitHub Pages](#)<sup>8</sup>. I've listed all of the scripts below:

- `npm run start` (or `npm start`) - Starts the project in the development mode. Surf to `localhost:8080` in your browser to see it running.
- `npm run build` - Generates a production build below `build/`. You can open the generated *index.html* through the browser to examine the result.
- `npm run deploy` - Deploys the contents of `build/` to the *gh-pages* branch of your project and pushes it to GitHub. You can access the project below `<user>.github.io/<project>` after that. Before this can work correctly, you should set `publicName` at *webpack.config.js* to match your project name on GitHub.

---

<sup>7</sup><http://survivejs.com/webpack/introduction/>

<sup>8</sup><https://pages.github.com/>

- `npm run stats` - Generates statistics (*stats.json*) about the project. You can [analyze the build output](#)<sup>9</sup> further.
- `npm run test` (or `npm test`) - Executes project tests. The *Testing React* chapter digs deeper into the topic. In fact, writing tests against your components can be a good way to learn to understand React better.
- `npm run test:tdd` - Executes project tests in TDD mode. This means it will watch for changes and run the tests when changes are detected allowing you to develop fast without having to run the tests manually.
- `npm run test:lint` - Executes [ESLint](#)<sup>10</sup> against the code. ESLint is able to catch smaller issues. You can even configure your development environment to work with it. This allows you to catch potential mistakes as you make them. Our setup lints even during development so you rarely need to execute this command yourself.

Study the "scripts" section of *package.json* to understand better how each of these works. There is quite a bit configuration. See [SurviveJS - Webpack](#)<sup>11</sup> to dig deeper into the topic.

## 2.4 Boilerplate Language Features



### Babel

The boilerplate relies on a transpiler known as [Babel](#)<sup>12</sup>. It allows us to use features from the future of JavaScript. It transforms your code to a format understandable by the browsers. You can even use it to develop your own language features. It supports JSX through a plugin.

Babel provides support for certain [experimental features](#)<sup>13</sup> from ES7 beyond standard ES6. Some of these might make it to the core language while some might be dropped altogether. The language proposals have been categorized within stages:

- **Stage 0** - Strawman
- **Stage 1** - Proposal

---

<sup>9</sup><http://survivejs.com/webpack/building-with-webpack/analyzing-build-statistics/>

<sup>10</sup><http://eslint.org/>

<sup>11</sup><http://survivejs.com/webpack/introduction/>

<sup>12</sup><https://babeljs.io/>

<sup>13</sup><https://babeljs.io/docs/plugins/#stage-x-experimental-presets->



- **Stage 2** - Draft
- **Stage 3** - Candidate
- **Stage 4** - Finished

I would be very careful with **stage 0** features. The problem is that if the feature changes or gets removed you will end up with broken code and will need to rewrite it. In smaller experimental projects it may be worth the risk, though.

In addition to standard ES2015 and JSX, we'll be using a few custom features in this project. I've listed them below. See the *Language Features* appendix to learn more of each.

- **Property initializers**<sup>14</sup> - Example: `addNote = (e) => {`. This binds the `addNote` method to an instance automatically. The feature makes more sense as we get to use it.
- **Decorators**<sup>15</sup> - Example: `@DragDropContext(HTML5Backend)`. These annotations allow us to attach functionality to classes and their methods.
- **Object rest/spread**<sup>16</sup> - Example: `const {a, b, ...props} = this.props`. This syntax allows us to easily extract specific properties from an object.

In order to make it easier to set up the features, I created [a specific preset](#)<sup>17</sup>. It also contains [babel-plugin-transform-object-assign](#)<sup>18</sup> and [babel-plugin-array-includes](#)<sup>19</sup> plugins. The former allows us to use `Object.assign` while the latter provides `Array.includes` without having to worry about shimming these for older environments.

A preset is simply a npm module exporting Babel configuration. Maintaining presets like this can be useful especially if you want to share the same set of functionality across multiple projects.



You can [try out Babel online](#)<sup>20</sup> to see what kind of code it generates.



If you are interested in a lighter alternative, check out [Buble](#)<sup>21</sup>.

---

<sup>14</sup><https://github.com/jeffmo/es-class-static-properties-and-fields>

<sup>15</sup><https://github.com/wycats/javascript-decorators>

<sup>16</sup><https://github.com/sebmarkbage/ecmascript-rest-spread>

<sup>17</sup><https://github.com/survivejs/babel-preset-survivejs-kanban>

<sup>18</sup><https://www.npmjs.com/package/babel-plugin-transform-object-assign>

<sup>19</sup><https://www.npmjs.com/package/babel-plugin-array-includes>

<sup>20</sup><https://babeljs.io/repl/>

<sup>21</sup><https://gitlab.com/Rich-Harris/buble>

## 2.5 Conclusion

Now that we have a simple “Hello World!” application running, we can focus on development. Developing and getting into trouble is a good way to learn after all.

## 3. Implementing a Note Application

Now that we have a nice development setup, we can actually get some work done. Our goal here is to end up with a crude note-taking application. It will have basic manipulation operations. We will grow our application from scratch and get into some trouble. This way you will understand why architectures, such as Flux, are needed.

### 3.1 Initial Data Model

Often a good way to begin designing an application is to start with the data. We can model a list of notes as follows:

```
[
  {
    id: '4e81fc6e-bfb6-419b-93e5-0242fb6f3f6a',
    task: 'Learn React'
  },
  {
    id: '11bbffc8-5891-4b45-b9ea-5c99aadf870f',
    task: 'Do laundry'
  }
];
```

Each note is an object which will contain the data we need, including an `id` and a `task` we want to perform. Later on it is possible to extend this data definition to include things like the note color or the owner.

We could have skipped `ids` in our definition. This would become problematic as we grow the application and add the concept of references to it. Each Kanban lane needs to be able to refer to some notes after all. By adopting proper indexing early on, we save ourselves some effort later.



Another interesting way to approach data would be to normalize it. In this case we would end up with a [`<id>` -> { `id`: '...', `task`: '...' }] kind of structure. Even though there's some redundancy, it is convenient to operate using the structure as it gives us easy access by index. The structure becomes even more useful once we start getting references between data entities.

## 3.2 Rendering Initial Data

Now that we have a rough data model together, we can try rendering it through React. We are going to need a component to hold the data. Let's call it `Notes` for now. We can grow from that as we want more functionality. Set up a file with a small dummy component as follows:

**app/components/Notes.jsx**

```
import React from 'react';

const notes = [
  {
    id: '4e81fc6e-bfb6-419b-93e5-0242fb6f3f6a',
    task: 'Learn React'
  },
  {
    id: '11bbffc8-5891-4b45-b9ea-5c99aadf870f',
    task: 'Do laundry'
  }
];

export default () => (
  <ul>{notes.map(note =>
    <li key={note.id}>{note.task}</li>
  )}</ul>
)
```

We are using various important features of JSX in the snippet above. I have annotated the difficult parts below:

- `<ul>{notes.map(note => ...}</ul> - {}`'s allow us to mix JavaScript syntax within JSX. `map` returns a list of `li` elements for React to render.
- `<li key={note.id}>{note.task}</li>` - In order to tell React in which order to render the elements, we use the `key` property. It is important that this is unique or else React won't be able to figure out the correct order in which to render. If not set, React will give a warning. See [Multiple Components<sup>1</sup>](https://facebook.github.io/react/docs/multiple-components.html) for more information.

We also need to refer to the component from the entry point of our application:

**app/index.jsx**

---

<sup>1</sup><https://facebook.github.io/react/docs/multiple-components.html>

```
import React from 'react';
import ReactDOM from 'react-dom';
import Notes from './components/Notes';

if(process.env.NODE_ENV !== 'production') {
  React.Perf = require('react-addons-perf');
}

ReactDOM.render(
  <div>Hello world</div>,
  <Notes />,
  document.getElementById('app')
);
```

If you run the application now, you should see a list of notes. It's not particularly pretty or useful yet, but it's a start:

- Learn React
- Do laundry

A list of notes



We need to import React to *Notes.jsx* given there's that JSX to JavaScript transformation going on. Without it the resulting code would fail.

### 3.3 Generating the Ids

Normally the problem of generating the ids is solved by a back-end. As we don't have one yet, we'll use a standard known as [RFC4122](https://www.ietf.org/rfc/rfc4122.txt)<sup>2</sup> instead. It allows us to generate unique ids. We'll be using a Node.js implementation known as *uuid* and its *uuid.v4* variant. It will give us ids, such as `1c8e7a12-0b4c-4f23-938c-00d7161f94fc` and they are guaranteed to be unique with a very high probability.

To connect the generator with our application, modify it as follows:

`app/components/Notes.jsx`

---

<sup>2</sup><https://www.ietf.org/rfc/rfc4122.txt>

```
import React from 'react';
import uuid from 'uuid';

const notes = [
  {
id: '4e81fc6e-bfb6-419b-93e5-0242fb6f3f6a',
    id: uuid.v4(),
    task: 'Learn React'
  },
  {
id: '11bbffe8-5891-4b45-b9ea-5e99aadf870f',
    id: uuid.v4(),
    task: 'Do laundry'
  }
];

...
```

The development setup will install the `uuid` dependency automatically. Once that has happened and the application has refreshed, everything should still look the same. If you try debugging it, you can see the ids should change if you refresh. You can verify this easily either by inserting a `console.log(notes);` line or a [debugger](#);<sup>3</sup> statement within the component function.

The `debugger;` statement is particularly useful as it tells the browser to break execution. This way you can examine the current call stack and examine the available variables. If you are unsure of something, this is a great way to debug and figure out what's going on.

`console.log` is a lighter alternative. You can even design a logging system around it and use the techniques together. See [MDN](#)<sup>4</sup> and [Chrome documentation](#)<sup>5</sup> for the full API.



If you are interested in the math behind id generation, check out [the calculations at Wikipedia](#)<sup>6</sup> for details. You'll see that the possibility for collisions is somewhat miniscule and something we don't have to worry about.

## 3.4 Adding New Notes to the List

Even though we can display individual notes now, we are still missing a lot of logic to make our application useful. A logical way to start would be to implement adding new notes to the list. To achieve this, we need to expand the application a little.

---

<sup>3</sup><https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/debugger>

<sup>4</sup><https://developer.mozilla.org/en-US/docs/Web/API/Console>

<sup>5</sup><https://developers.google.com/web/tools/chrome-devtools/debug/console/console-reference>

<sup>6</sup>[https://en.wikipedia.org/wiki/Universally\\_unique\\_identifier#Random\\_UUID\\_probability\\_of\\_duplicates](https://en.wikipedia.org/wiki/Universally_unique_identifier#Random_UUID_probability_of_duplicates)

## Defining a Stub for App

To enable adding new notes, we should have a button for that somewhere. Currently our Notes component does only one thing, display notes. That's perfectly fine. To make room for more functionality, we could add a concept known as App on top of that. This component will orchestrate the execution of our application. We can add the button we want there and manage state as well as we add notes. At a basic level App could look like this:

**app/components/App.jsx**

```
import React from 'react';
import Notes from './Notes';

export default () => <Notes />;
```

All it does now is render Notes, so it's going to take more work to make it useful. To glue App to our application, we still need to tweak the entry point as follows:

**app/index.jsx**

```
import React from 'react';
import ReactDOM from 'react-dom';
import Notes from './components/Notes';
import App from './components/App';

if(process.env.NODE_ENV !== 'production') {
  React.Perf = require('react-addons-perf');
}

ReactDOM.render(
  <Notes />,
  <App />,
  document.getElementById('app')
);
```

If you run the application now, it should look exactly the same as before. But now we have room to grow.

## Adding a Stub for Add Button

A good step towards something more functional is to add a stub for an *add* button. To achieve this, App needs to evolve:

**app/components/App.jsx**

```
import React from 'react';
import Notes from './Notes';

export default () => <Notes />;
export default () => (
  <div>
    <button onClick={() => console.log('add note')}></button>
    <Notes />
  </div>
);
```

If you press the button we added, you should see an “add note” message at the browser console. We still have to connect the button with our data somehow. Currently the data is trapped within the Notes component so before we can do that, we need to extract it to the App level.



Given React components have to return a single element, we had to wrap our application within a div.

## Pushing Data to App

To push the data to App we need to make two modifications. First we need to literally move it there and pass the data through a prop to Notes. After that we need to tweak Notes to operate based on the new logic. Once we have achieved this, we can start thinking about adding new notes.

The App side is simple:

**app/components/App.jsx**

```
import React from 'react';
import uuid from 'uuid';
import Notes from './Notes';

const notes = [
  {
    id: uuid.v4(),
    task: 'Learn React'
  },
  {
    id: uuid.v4(),
    task: 'Do laundry'
  }
];
```



```
export default () => (
  <div>
    <button onClick={() => console.log('add note')}>+</button>
    <Notes />
    <Notes notes={notes} />
  </div>
);
```

This won't do much until we tweak Notes as well:

**app/components/Notes.jsx**

```
import React from 'react';
import uuid from 'uuid';

const notes = [
  {
    id: uuid.v4(),
    task: 'Learn React'
  },
  {
    id: uuid.v4(),
    task: 'Do laundry'
  }
];

export default () => {
export default ({notes}) => (
  <ul>{notes.map(note =>
    <li key={note.id}>{note.task}</li>
  )}</ul>
);
```

Our application should look exactly the same as it did before these changes. Now we are ready to add some logic to it.



The way we extract notes from props (the first parameter) is a standard trick you see with React. If you want to access the remaining props, you can use `{notes, ...props}` kind of syntax. We'll use this later so you can get a better feel for how this works and why you might use it.

## Pushing State to App

Now that we have everything in the right place, we can start to worry about modifying the data. If you have used JavaScript before, the intuitive way to handle it would be to set up an event handler like `() => notes.push({id: uuid.v4(), task: 'New task'})`. If you try this, you'll see that nothing happens.

The reason why is simple. React cannot notice the structure has changed and won't react accordingly (that is, trigger `render()`). To overcome this issue, we can implement our modification through React's own API. This makes it notice that the structure has changed. As a result it is able to `render()` as we might expect.

As of the time of writing, the function based component definition doesn't support the concept of state. The problem is that these components don't have a backing instance. It is something in which you would attach state. We might see a way to solve this through functions only in the future but for now we have to use a heavy duty alternative.

In addition to functions, you can create React components through `React.createClass` or a class based component definition. In this book we'll use function based components whenever possible. If there's a good reason why those can't work, then we'll use the class based definition instead.

In order to convert our App to a class based component, adjust it as follows to push the state within:

**app/components/App.jsx**

```
import React from 'react';
import uuid from 'uuid';
import Notes from '../Notes';

const notes = [
  {
    id: uuid.v4(),
    task: 'Learn React'
  },
  {
    id: uuid.v4(),
    task: 'Do laundry'
  }
];

export default () => (
  <div>
    <button onClick={() => console.log('add note')}>+</button>
    <Notes notes={notes} />
  </div>
```

);

```
export default class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      notes: [
        {
          id: uuid.v4(),
          task: 'Learn React'
        },
        {
          id: uuid.v4(),
          task: 'Do laundry'
        }
      ]
    };
  }
  render() {
    const {notes} = this.state;

    return (
      <div>
        <button onClick={() => console.log('add note')}>+</button>
        <Notes notes={notes} />
      </div>
    );
  }
}
```

After this change App owns the state even though the application still should look the same as before. We can begin to use React's API to modify the state.



Data management solutions, such as [MobX<sup>7</sup>](https://mobxjs.github.io/mobx/), solve this problem in their own way. Using them you annotate your data structures and React components and leave the updating problem to them. We'll get back to the topic of data management later in this book in detail.

---

<sup>7</sup><https://mobxjs.github.io/mobx/>



We're passing props to super by convention. If you don't pass it, `this.props` won't get set! Calling `super` invokes the same method of the parent class and you see this kind of usage in object oriented programming often.

## Implementing Note Adding Logic

All the effort will pay off soon. We have just one step left. We will need to use React's API to manipulate the state and to finish our feature. React provides a method known as `setState` for this purpose. In this case we will need to call it like this: `this.setState({... new state goes here ...}, () => ...)`.

The callback is optional. React will call it when the state has been set and often you don't need to care about it at all. Once `setState` has gone through, React will call `render()`. The asynchronous API might feel a little strange at first but it will allow React to optimize its performance by using techniques such as batching updates. This all ties back to the concept of Virtual DOM.

One way to trigger `setState` would be to push the associated logic to a method of its own and then call it when a new note is added. The class based component definition doesn't bind custom methods like this by default so we will need to handle the binding somehow. It would be possible to do that at the constructor, `render()`, or by using a specific syntax. I'm opting for the syntax option in this book. Read the *Language Features* appendix to learn more.

To tie the logic with the button, adjust `App` as follows:

**app/components/App.jsx**

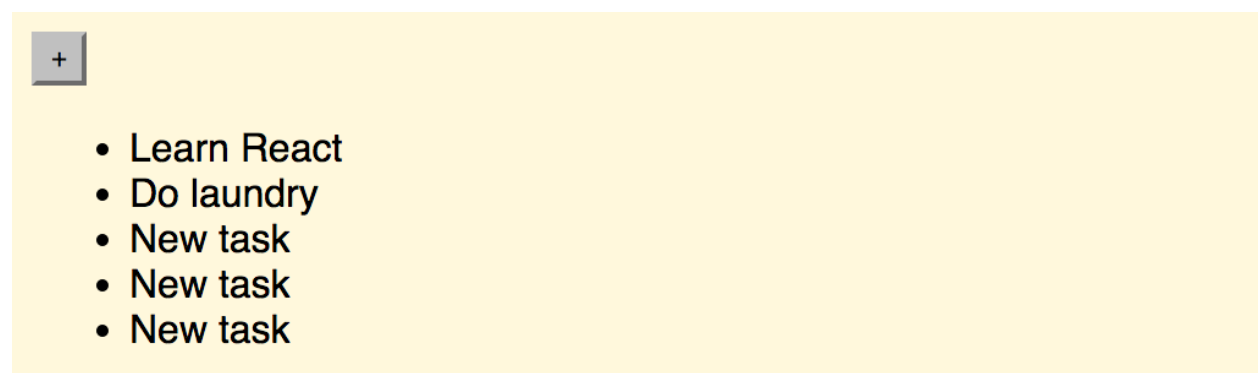
```
import React from 'react';
import uuid from 'uuid';
import Notes from '../Notes';

export default class App extends React.Component {
  constructor(props) {
    ...
  }
  render() {
    const {notes} = this.state;

    return (
      <div>
        <button onClick={() => console.log('add note')}></button>
        <button onClick={this.addNote}></button>
        <Notes notes={notes} />
      </div>
    );
  }
}
```

```
    );  
  }  
  addNote = () => {  
    // It would be possible to write this in an imperative style.  
    // I.e., through `this.state.notes.push` and then  
    // `this.setState({notes: this.state.notes})` to commit.  
    //  
    // I tend to favor functional style whenever that makes sense.  
    // Even though it might take more code sometimes, I feel  
    // the benefits (easy to reason about, no side effects)  
    // more than make up for it.  
    //  
    // Libraries, such as Immutable.js, go a notch further.  
    this.setState({  
      notes: this.state.notes.concat([{  
        id: uuid.v4(),  
        task: 'New task'  
      }])  
    });  
  }  
}
```

Given we are binding to an instance here, the hot loading setup cannot pick up the change. To try out the new feature, refresh the browser and try clicking the + button. You should see something:



Notes with a plus



If we were operating with a back-end, we would trigger a query here and capture the id from the response. For now it's enough to just generate an entry and a custom id.



You could use `this.setState({notes: [...this.state.notes, {id: uuid.v4(), task: 'New task'}]})` to achieve the same result. This [spread operator](#)<sup>8</sup> can be used with function parameters as well. See the *Language Features* appendix for more information.



Using [autobind-decorator](#)<sup>9</sup> would be a valid alternative for property initializers. In this case we would use `@autobind` annotation either on class or method level. To learn more about decorators, read the *Understanding Decorators* appendix.

## 3.5 Conclusion

Even though we have a rough application together now, we are still missing two crucial features: editing and deleting notes. It's a good time to focus on those next. Let's do deletion first and handle editing after that.

---

<sup>8</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread\\_operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator)

<sup>9</sup><https://www.npmjs.com/package/autobind-decorator>

## 4. Deleting Notes

One easy way to handle deleting notes is to render a “x” button for each Note. When it’s clicked we will simply delete the note in question from our data structure. As before, we can start by adding stubs in place. This might be a good place to separate the concept of a Note from the current Notes component.

Often you work this way with React. You set up components only to realize they are composed of smaller components that can be extracted. This process of separation is cheap. Sometimes it can even improve the performance of your application as you can optimize the rendering of smaller parts.

### 4.1 Separating Note

To keep the list formatting aspect separate from a Note we can model it using a div like this:

**app/components/Note.jsx**

```
import React from 'react';

export default ({task}) => <div>{task}</div>;
```

Remember that this declaration is equivalent to:

```
import React from 'react';

export default (props) => <div>{props.task}</div>;
```

As you can see, destructuring removes some noise from the code and keeps our implementation simple.

To make our application use the new component, we need to patch Notes as well:

**app/components/Notes.jsx**

```
import React from 'react';
import Note from './Note';

export default ({notes}) => (
  <ul>{notes.map(note =>
    <li key={note.id}>{note.task}</li>
    <li key={note.id}><Note task={note.task} /></li>
  )}</ul>
)
```

The application should look exactly the same after these changes. Now we have room to expand it further.

## 4.2 Adding a Stub for onDelete Callback

To capture the intent to delete a Note, we'll need to extend it to include a button that triggers a onDelete callback. We can connect our logic with that after this step is complete. Consider the code below:

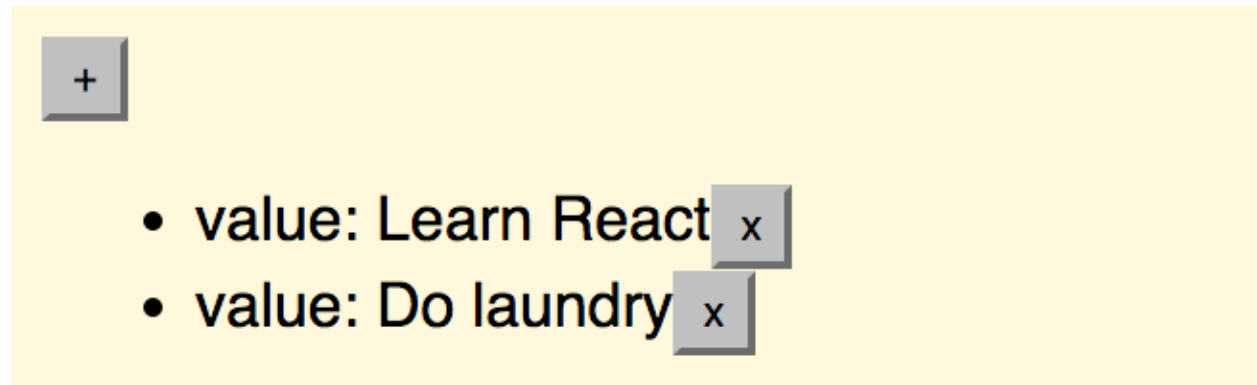
**app/components/Note.jsx**

```
import React from 'react';

export default ({task}) => <div>{task}</div>;
export default ({task, onDelete}) => (
  <div>
    <span>{task}</span>
    <button onClick={onDelete}>x</button>
  </div>
);
```

You should see small “x”s next to each Note:



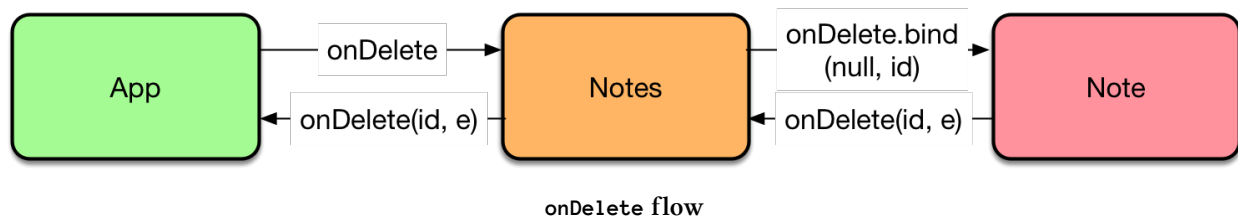


Notes with delete controls

They won't do anything yet. Fixing that is the next step.

### 4.3 Communicating Deletion to App

Now that we have the controls we need, we can start thinking about how to connect them with the data at App. In order to delete a Note, we'll need to know its id. After that we can implement the logic based on that at App. To illustrate the idea, we'll want to end up with a situation like this:



**Key** That `e` represents a DOM event you might be used to. We can do things like stop event propagation through it. This will come in handy as we want more control over the application behavior.

**Key** `bind`<sup>1</sup> allows us to set the function context (first parameter) and arguments (following parameters). This gives us a technique known as **partial application**.

To achieve the scheme, we are going to need a new prop at Notes. We will also need to bind the id of each note to the `onDelete` callback to match the logic. Here's the full implementation of Notes:

`app/components/Notes.jsx`

<sup>1</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Function/bind](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind)

```
import React from 'react';
import Note from './Note';

export default ({notes}) => (
  <ul>{notes.map(note =>
    <li key={note.id}><Note task={note.task} /></li>
  )}</ul>
)
export default ({notes, onDelete=() => {}}) => (
  <ul>{notes.map(({id, task}) =>
    <li key={id}>
      <Note
        onDelete={onDelete.bind(null, id)}
        task={task} />
      </li>
    )}</ul>
)
```

To keep our code from crashing if `onDelete` is not provided, I defined a dummy callback for it. Another good way to handle this would have been to go through `propTypes` as discussed in the *Typing with React* chapter.

Now that we have the hooks in place, we can use them at App:

### app/components/App.jsx

```
import React from 'react';
import uuid from 'uuid';
import Notes from './Notes';

export default class App extends React.Component {
  constructor(props) {
    ...
  }
  render() {
    const {notes} = this.state;

    return (
      <div>
        <button onClick={this.addNote}>+</button>
        <Notes notes={notes} />
        <Notes notes={notes} onDelete={this.deleteNote} />
      </div>
    )
  }
}
```

```
    );  
  },  
  addNote = () => {  
    ...  
  }  
  deleteNote = (id, e) => {  
    // Avoid bubbling to edit  
    e.stopPropagation();  
  
    this.setState({  
      notes: this.state.notes.filter(note => note.id !== id)  
    });  
  }  
}
```

After refreshing the browser, you should be able to delete notes. To prepare for the future I added an extra line in form of `e.stopPropagation()`. The idea of this is to tell the DOM to stop bubbling events. In short, we'll avoid triggering possible other events elsewhere in the structure if we delete a note.

## 4.4 Conclusion

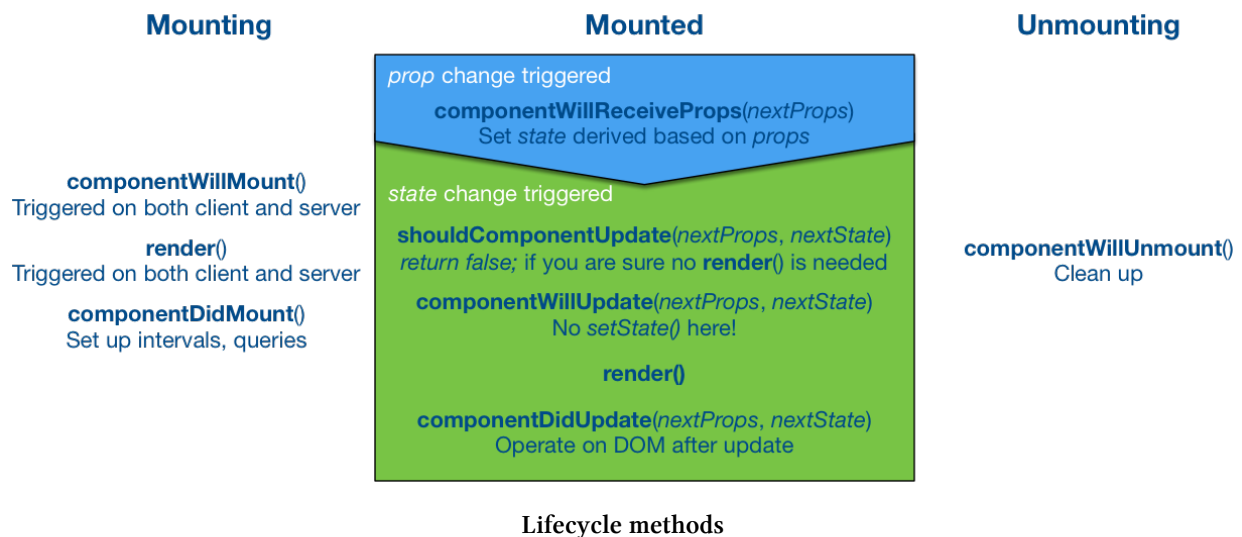
Working with React is often like this. You will identify components and flows based on your needs. Here we needed model a `Note` and then design a data flow so that we have enough control at the right place.

We are missing one more feature to call the first part of Kanban done. Editing is hardest of them all. One way to implement it is to solve it through an *inline editor*. By implementing a proper component now, we'll save time later when we have to edit something else. Before continuing with the implementation, though, we'll take a better look at React components to understand what kind of functionality they provide.

## 5. Understanding React Components

As we have seen so far, React components are fairly simple. They can have internal state. They can also accept props. Beyond this React provides escape hatches that allow you to do handle advanced use cases. These include lifecycle methods and refs. There are also a set of custom properties and methods you may want to be aware of.

### 5.1 Lifecycle Methods



From the image above we can see that a React component has three phases during its lifecycle. It can be **mounting**, **mounted**, and **unmounting**. Each of these phases comes with related methods.

During the mounting phase you have access to the following:

- **componentWillMount()** gets triggered once before any rendering. One way to use it would be to load data asynchronously there and force rendering through **setState**. **render()** will see the updated state and will be executed only once despite the state change. This will get triggered when rendering on a server.
- **componentDidMount()** gets triggered after initial rendering. You have access to the DOM here. You could use this hook to wrap a jQuery plugin within a component, for instance. This **won't** get triggered when rendering on a server.

After a component has been mounted and it's running, you can operate through the following methods:

- `componentWillReceiveProps(object nextProps)` triggers when the component receives new props. You could, for instance, modify your component state based on the received props.
- `shouldComponentUpdate(object nextProps, object nextState)` allows you to optimize the rendering. If you check the props and state and see that there's no need to update, return `false`.
- `componentWillUpdate(object nextProps, object nextState)` gets triggered after `shouldComponentUpdate` and before `render()`. It is not possible to use `setState` here, but you can set class properties, for instance. This is where [Immutable.js](https://facebook.github.io/immutable-js/)<sup>1</sup> and similar libraries come in handy thanks to their easy equality checks. [The official documentation](#)<sup>2</sup> goes to greater detail.
- `componentDidUpdate()` is triggered after rendering. You can modify the DOM here. This can be useful for adapting other code to work with React.

Finally, when a component is unmounting, there's one more hook you can use:

- `componentWillUnmount()` is triggered just before a component is unmounted from the DOM. This is the ideal place to perform cleanup (e.g., remove running timers, custom DOM elements, and so on).

Often `componentDidMount` and `componentWillUnmount` come as a pair. If you set up something DOM related or a listener at `componentDidMount`, you also have to remember to clean it up at `componentWillUnmount`.

## 5.2 Refs

React's [refs](#)<sup>3</sup> allow you to access the underlying DOM structure easily. Using them will bind your code to the web, but sometimes there's no way around this if you are measuring components for instance.

Refs need a backing instance. This means they will work only with `React.createClass` or class based component definitions. The basic idea goes as follows:

```
<input type="text" ref="input" />
```

```
...
```

```
// Access somewhere  
this.refs.input
```

In addition to strings, refs support a callback that gets called right after the component is mounted. You can do some initialization here or capture the reference:

---

<sup>1</sup><https://facebook.github.io/immutable-js/>

<sup>2</sup><https://facebook.github.io/react/docs/advanced-performance.html#shouldcomponentupdate-in-action>

<sup>3</sup><https://facebook.github.io/react/docs/more-about-refs.html>

```
<input type="text" ref={element => element.focus()} />
```

## 5.3 Custom Properties and Methods

Beyond the lifecycle methods and refs, there are a variety of [properties and methods](#)<sup>4</sup> you should be aware of especially if you are going to use `React.createClass`:

- `displayName` - It is preferable to set `displayName` as that will improve debug information. For ES6 classes this is derived automatically based on the class name. You can attach `displayName` to an anonymous function based component as well.
- `getInitialState()` - In class based approach the same can be achieved through constructor.
- `getDefaultProps()` - In classes you can set these in constructor.
- `render()` - This is the workhorse of React. It [must return a single node](#)<sup>5</sup> as returning multiple won't work!
- `mixins` - `mixins` contains an array of mixins to apply to components.
- `statics` - `statics` contains static properties and method for a component. In ES6 you can assign them to the class as below:

```
class Note {  
  render() {  
    ...  
  }  
}  
  
Note.willTransitionTo = () => {...};  
  
export default Note;
```

This could also be written as:

---

<sup>4</sup><https://facebook.github.io/react/docs/component-specs.html>

<sup>5</sup><https://facebook.github.io/react/tips/maximum-number-of-jsx-root-nodes.html>

```
class Note {  
  static willTransitionTo() {...}  
  render() {  
    ...  
  }  
}  
  
export default Note;
```

Some libraries, such as React DnD, rely on static methods to provide transition hooks. They allow you to control what happens when a component is shown or hidden. By definition statics are available through the class itself.

React components allow you to document the interface of your component using `propTypes` as below.

```
const Note = ({task}) => <div>{task}</div>;  
Note.propTypes = {  
  task: React.PropTypes.string.isRequired  
}
```

To understand `propTypes` better, read the *Typing with React* chapter.

## 5.4 React Component Conventions

I prefer to have the constructor first, followed by lifecycle methods, `render()`, and finally, methods used by `render()`. This top-down approach makes it straightforward to follow code. There is also an inverse convention that leaves `render()` as the last method. Naming conventions vary as well. You will have to find conventions which work the best for you.

You can enforce a convention by using a linter such as [ESLint](http://eslint.org/)<sup>6</sup>. Using a linter decreases the amount of friction when working on code written by others. Even on personal projects, using tools to verify syntax and standards for you can be useful. It lessens the amount and severity of mistakes and allows you to spot them early.

By setting up a continuous integration system you can test against multiple platforms and catch possible regressions early. This is particularly important if you are using lenient version ranges. Sometimes dependencies might have problems and it's good to catch those.

---

<sup>6</sup><http://eslint.org/>

## 5.5 Conclusion

Even though React's component definition is fairly simple, it's also powerful and pragmatic. Especially the advanced parts can take a while to master, but it's good to know they are there.

We'll continue the implementation in the next chapter as we allow the user to edit individual notes.



## 6. Editing Notes

Editing notes is a similar problem as deleting them. The data flow is exactly the same. We'll need to define an `onEdit` callback and bind an id of the note being edited at `Notes`.

What makes this scenario difficult is the user interface requirement. It's not enough just to have a button. We'll need some way to allow the user to input a new value which we can then commit to the data model.

One way to achieve this is to implement so called **inline editing**. The idea is that when a user click a note, we'll show an input. After the user has finished editing and signals that to use either by hitting *enter*, we'll capture the value and update.

### 6.1 Implementing `Editable`

To keep the application clean, I'll wrap this behavior into a component known as `Editable`. It will give us an API like this:

```
<Editable
  editing={editing}
  value={task}
  onEdit={onEdit.bind(null, id)} />
```

This is an example of a **controlled** component. We'll control the editing state explicitly from outside of the component. This gives us more power, but it also makes `Editable` more involved to use.



It can be a good idea to name your callbacks using `on` prefix. This will allow you to distinguish them from other props and keep your code a little tidier.

### Controlled vs. Uncontrolled Design

An alternative way to handle this would have been to leave the control over the editing state to `Editable`. This **uncontrolled** way of designing can be valid if you don't want to do anything with the state outside of the component.

It is possible to use both of these designs together. You can even have a controlled component that has uncontrolled elements inside. In this case we'll end up using an uncontrolled design for the

input that `Editable` will contain for example. Even that could be turned into something controlled should we want to.

Logically `Editable` consists of two separate portions. We'll need to display the default value while we are not editing. In case we are editing, we'll want to show an `Edit` control instead. In this case we'll settle for a simple input as that will do the trick.

Before digging into the details, we can implement a little stub and connect that to the application. This will give us the basic structure we need to grow the rest. To get started, we'll adjust the component hierarchy a notch to make it easier to implement the stub.



The official documentation of React discusses [controlled components](https://facebook.github.io/react/docs/forms.html)<sup>1</sup> in greater detail.

## 6.2 Extracting Rendering from Note

Currently `Note` controls what is rendered inside it. It renders the passed task and connects a deletion button. We could push `Editable` inside it and handle the wiring through `Note` interface. Even though that might be one valid way to do it, we can push the rendering concern on a higher level.

Having the concept of `Note` is useful especially when we'll expand the application further so there's no need to remove it. Instead, we can give the control over its rendering behavior to `Notes` and wire it there.

React provides a prop known as `children` for this purpose. Adjust `Note` and `Notes` as follows to push the control over `Note` rendering to `Notes`:

**app/components/Note.jsx**

```
import React from 'react';

export default ({task, onDelete}) => (
  <div>
    <span>{task}</span>
    <button onClick={onDelete}>x</button>
  </div>
);

export default ({children, ...props}) => (
  <div {...props}>
    {children}
  </div>
);
```

**app/components/Notes.jsx**

---

<sup>1</sup><https://facebook.github.io/react/docs/forms.html>

```
import React from 'react';
import Note from './Note';

export default ({notes, onDelete=() => {}}) => (
  <ul>{notes.map(({id, task}) =>
    <li key={id}>
      <Note
        onDelete={onDelete.bind(null, id)}
        task={task} />
      <Note>
        <span>{task}</span>
        <button onClick={onDelete.bind(null, id)}>x</button>
      </Note>
    </li>
  )}</ul>
)
```

Now that we have room to work, we can set up a stub for Editable.

## 6.3 Adding Editable Stub

We can model a rough starting point based on our specification as below. The idea is that we'll branch based on the editing prop and attach the props needed for implementing our logic:

app/components/Editable.jsx

```
import React from 'react';

export default ({editing, value, onEdit, ...props}) => {
  if(editing) {
    return <Edit value={value} onEdit={onEdit} {...props} />;
  }

  return <span {...props}>value: {value}</span>;
}

const Edit = ({onEdit = () => {}, value, ...props}) => (
  <div onClick={onEdit} {...props}>
    <span>edit: {value}</span>
  </div>
);
```

To see our stub in action we still need to connect it with our application.

## 6.4 Connecting Editable with Notes

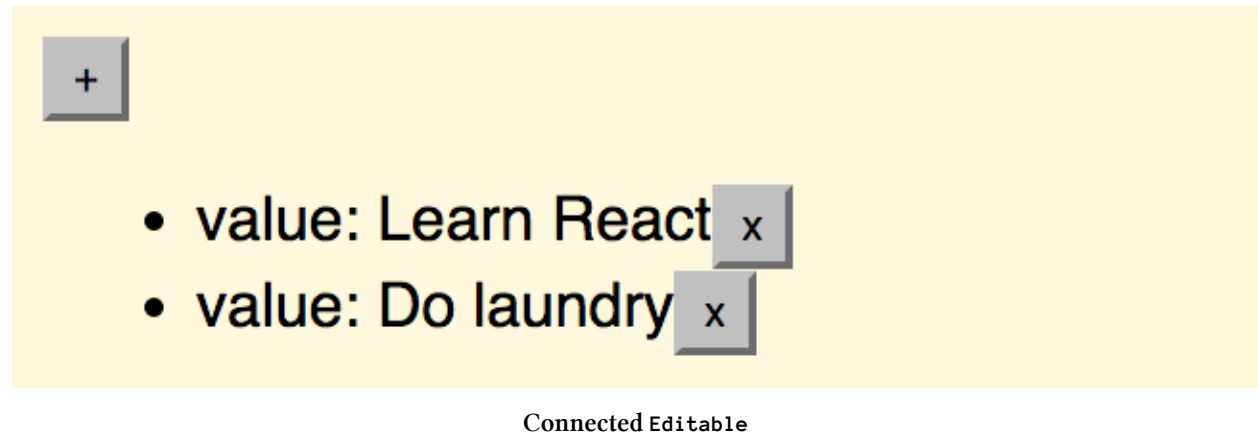
We still need to replace the relevant portions of the code to point at `Editable`. There are more props to track and to connect:

`app/components/Notes.jsx`

```
import React from 'react';
import Note from './Note';
import Editable from './Editable';

export default ({notes, onDelete=() => {}}) => (
export default ({
  notes,
  onNoteClick=() => {}, onEdit=() => {}, onDelete=() => {}
}) => (
  <ul>{notes.map(({id, task}) =>
    <li key={id}>
      <Note>
        <span>{task}</span>
        <button onClick={onDelete.bind(null, id)}>x</button>
      </Note>
    </li>
  )}</ul>
  <ul>{notes.map(({id, editing, task}) =>
    <li key={id}>
      <Note onClick={onNoteClick.bind(null, id)}>
        <Editable
          editing={editing}
          value={task}
          onEdit={onEdit.bind(null, id)} />
        <button onClick={onDelete.bind(null, id)}>x</button>
      </Note>
    </li>
  )}</ul>
)
```

If everything went right, you should see something like this:



## 6.5 Tracking Note editing State

We are still missing logic needed to control the Editable. Given the state of our application is maintained at App, we'll need to deal with it there. It should set the editable flag of the edited note to true when we begin to edit and set it back to false when we complete the editing process. We should also adjust its task using the new value. For now we are interested in just getting the editable flag to work, though. Modify as follows:

**app/components/App.jsx**

```
...

export default class App extends React.Component {
  constructor(props) {
    ...
  }
  render() {
    const {notes} = this.state;

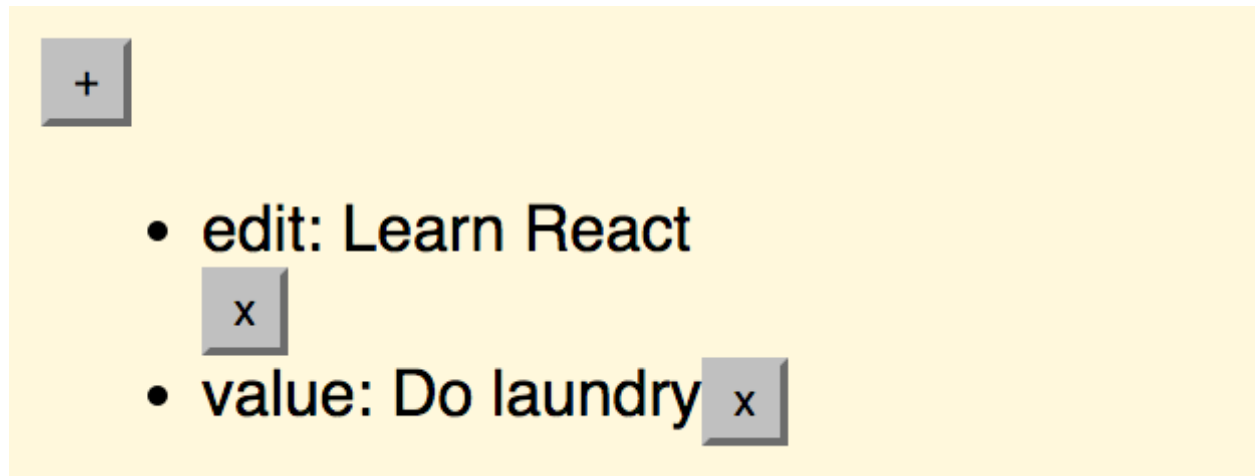
    return (
      <div>
        <button onClick={this.addNote}>>+</button>
        <Notes notes={notes} onDelete={this.deleteNote} />
        <Notes
          notes={notes}
          onNoteClick={this.activateNoteEdit}
          onEdit={this.editNote}
          onDelete={this.deleteNote}
        />
      </div>
    )
  }
}
```

```
    );
  }
  addNote = () => {
    ...
  }
  deleteNote = (id, e) => {
    ...
  }
  activateNoteEdit = (id) => {
    this.setState({
      notes: this.state.notes.map(note => {
        if(note.id === id) {
          note.editing = true;
        }

        return note;
      })
    });
  }
  editNote = (id, task) => {
    this.setState({
      notes: this.state.notes.map(note => {
        if(note.id === id) {
          note.editing = false;
          note.task = task;
        }

        return note;
      })
    });
  }
}
```

If you try to edit a Note now, you should see something like this:



### Tracking editing state

If you click a Note twice to confirm the edit, you should see an `Uncaught Invariant Violation` error at the browser console. This happens because we don't deal with `task` correctly yet. We have bound only `id` and `task` will actually point to an event object provided by React. This is something we should fix next.



If we used a normalized data structure (i.e., `{<id>: {id: <id>, task: <str>}}`), it would be possible to write the operations using `Object.assign` and avoid mutation.



In order to clean up the code, you could extract a method to contain the logic shared by `activateNoteEdit` and `editNote`.

## 6.6 Implementing `Edit`

We are missing one more part to make this work. Even though we can manage the editing state per Note now, we still can't actually edit them. For this purpose we need to expand `Edit` and make it render a text input for us.

In this case we'll be using **uncontrolled** design and extract the value of the input from the DOM only when we need it. We don't need more control than that here.

Consider the code below for the full implementation. Note how we are handling finishing the editing. We capture `onKeyPress` and check for `Enter` to confirm editing. We also run the finish logic `onBlur` so that we can end the editing when the input loses focus:

`app/components/Editable.jsx`

...

```
export default ({editing, value, onEdit, ...props}) => {
  if(editing) {
    return <Edit value={value} onEdit={onEdit} {...props} />;
  }
```

```
return <span {...props}>value: {value}</span>;
  return <span {...props}>{value}</span>;
}
```

```
const Edit = ({onEdit = () => {}, value, ...props}) => (
<div onClick={onEdit} {...props}>
<span>edit: {value}</span>
</div>
);
```

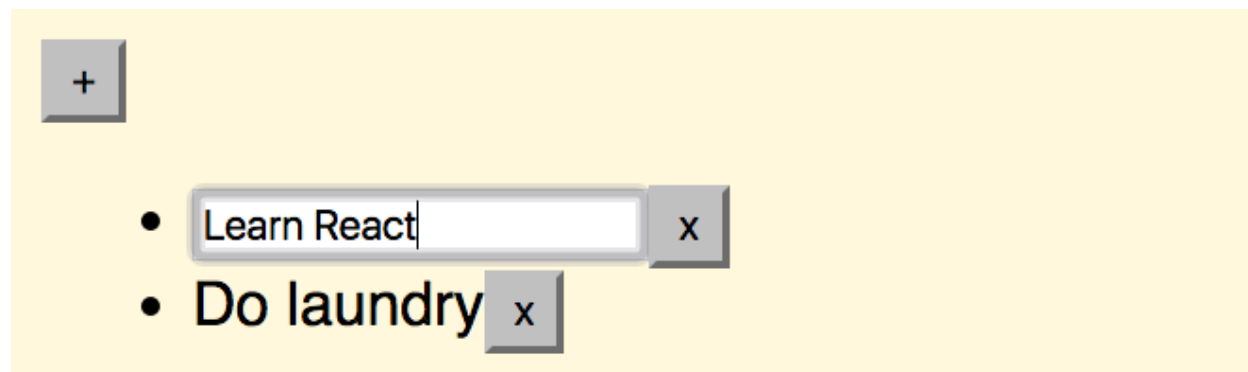
```
class Edit extends React.Component {
  render() {
    const {value, ...props} = this.props;

    return <input
      type="text"
      autoFocus={true}
      defaultValue={value}
      onBlur={this.finishEdit}
      onKeyPress={this.checkEnter}
      {...props} />;
  }
  checkEnter = (e) => {
    if(e.key === 'Enter') {
      this.finishEdit(e);
    }
  }
  finishEdit = (e) => {
    const value = e.target.value;

    if(this.props.onEdit) {
      this.props.onEdit(value);
    }
  }
}
```



If you refresh and edit a note, the commits should go through:



Editing a Note

## 6.7 On Namespacing Components

We could have approached `Editable` in a different way. In an earlier edition of this book I ended up developing it as a single component. I handled rendering the value and the edit control through methods (i.e., `renderValue`). Often method naming like that is a clue that it's possible to refactor your code and extract separate components like we did here.

You can go one step further and [namespace<sup>2</sup>](https://facebook.github.io/react/docs/jsx-in-depth.html#namespaced-components) your component parts. It would have been possible to define `Editable.Value` and `Editable.Edit` components. Better yet, we could have allowed the user to swap those components through props. As long as the interface is the same, the components should work. This would give an extra dimension of customizability.

Implementation-wise we would have had to do something like this in case we had gone with namespacing:

`app/components/Editable.jsx`

```
import React from 'react';

// We could allow edit/value to be swapped here through props
const Editable = ({editing, value, onEdit}) => {
  if(editing) {
    return <Editable.Edit value={value} onEdit={onEdit} />;
  }

  return <Editable.Value value={value} />;
};
```

<sup>2</sup><https://facebook.github.io/react/docs/jsx-in-depth.html#namespaced-components>

```
Editable.Value = ({value, ...props}) => <span {...props}>{value}</span>

class Edit extends React.Component {
  ...
}
Editable.Edit = Edit;

// We could export individual components too to allow modification
export default Editable;
```

You can use a similar approach for more generic components as well. Consider something like `Form`. You could easily have `Form.Label`, `Form.Input`, `Form.Textarea` and so on. Each would contain your custom formatting and logic as needed. This is one way to make your designs more flexible.

## 6.8 Conclusion

It took quite a few steps, but we can edit our notes now. Best of all, `Editable` should be useful whenever we need to edit some property. We could have extracted the logic later on as we see duplication, but this is one way to do it.

Even though the application kind of works, it is still quite ugly. We'll do something about that in the next chapter as we add basic styling to it.

## 7. Styling the Notes Application

Aesthetically, our current application is very barebones. As pretty applications are more fun to use, we can do a little something about that. In this case we'll be sticking to an old skool way of styling.

In other words, we'll sprinkle some CSS classes around and then apply CSS selectors based on those. The *Styling React* chapter discusses various other approaches in greater detail.

### 7.1 Styling “Add Note” Button

To style the “Add Note” button we'll need to attach a class to it first:

`app/components/App.jsx`

```
...

export default class App extends React.Component {
  constructor(props) {
    ...
  }
  render() {
    const {notes} = this.state;

    return (
      <div>
        <button onClick={this.addNote}>>+</button>
        <button className="add-note" onClick={this.addNote}>>+</button>
        <Notes
          notes={notes}
          onNoteClick={this.activateNoteEdit}
          onEdit={this.editNote}
          onDelete={this.deleteNote}
        />
      </div>
    );
  }
  ...
}
```

We also need to add corresponding styling:

**app/main.css**

```
...

.add-note {
  background-color: #fdfdfd;

  border: 1px solid #ccc;
}
```

A more general way to handle this would be to set up a Button component and style it. That would give us nicely styled buttons across the application.

## 7.2 Styling Notes

Currently the Notes list looks a little rough. We can improve that by hiding the list specific styling. We can also fix Notes width so if the user enter a long task, our user interface still remains fixed to some maximum width. A good first step is to attach some classes to Notes so it's easier to style:

**app/components/Notes.jsx**

```
import React from 'react';
import Note from './Note';
import Editable from './Editable';

export default ({
  notes,
  onNoteClick=() => {}, onEdit=() => {}, onDelete=() => {}
}) => (
  <ul>{notes.map(({id, editing, task}) =>
    <ul className="notes">{notes.map(({id, editing, task}) =>
      <li key={id}>
        <Note onClick={onNoteClick.bind(null, id)}>
          <Note className="note" onClick={onNoteClick.bind(null, id)}>
            <Editable
              className="editable"
              editing={editing}
              value={task}
              onEdit={onEdit.bind(null, id)} />
          <button onClick={onDelete.bind(null, id)}>x</button>
```

```

      <button
        className="delete"
        onClick={onDelete.bind(null, id)}>x</button>
    </Note>
  </li>
)}</ul>
)

```

In order to eliminate the list specific styling, we can apply rules like these:

app/main.css

```

...

.notes {
  margin: 0.5em;
  padding-left: 0;

  max-width: 10em;
  list-style: none;
}

```

## 7.3 Styling Individual Notes

There is still Note related portions left to style. Before attaching any rules, we should make sure we have good styling hooks on Editable:

app/components/Editable.jsx

```

import React from 'react';
import classNames from 'classnames';

export default ({editing, value, onEdit, ...props}) => {
export default ({editing, value, onEdit, className, ...props}) => {
  if(editing) {
    return <Edit value={value} onEdit={onEdit} {...props} />;
    return <Edit
      className={className}
      value={value}
      onEdit={onEdit}
      {...props} />;
  }
}

```

```

return <span {...props}>{value}</span>;
return <span className={classnames('value', className)} {...props}>
  {value}
</span>;
}

class Edit extends React.Component {
  render() {
const {value, ...props} = this.props;
    const {className, value, ...props} = this.props;

    return <input
      type="text"
      className={classnames('edit', className)}
      autoFocus={true}
      defaultValue={value}
      onBlur={this.finishEdit}
      onKeyPress={this.checkEnter}
      {...props} />;
  }
  ...
}

```

Given `className` accepts only a string, it can be difficult to work with when you have multiple classes depending on some logic. This is where a package known as [classnames<sup>1</sup>](https://www.npmjs.org/package/classnames) can be useful. It accepts almost arbitrary input and converts that to a string solving the problem.

There are enough classes to style the remainder now. We can show a shadow below the hovered note. It's also a good touch to show the delete control on hover as well. Unfortunately this won't work on touch based interfaces, but it's good enough for this demo:

**app/main.css**

---

<sup>1</sup><https://www.npmjs.org/package/classnames>

```
...

.note {
  overflow: auto;

  margin-bottom: 0.5em;
  padding: 0.5em;

  background-color: #fdfdfd;
  box-shadow: 0 0 0.3em .03em rgba(0,0,0,.3);
}
.note:hover {
  box-shadow: 0 0 0.3em .03em rgba(0,0,0,.7);

  transition: .6s;
}

.note .value {
  /* force to use inline-block so that it gets minimum height */
  display: inline-block;
}

.note .editable {
  float: left;
}

.note .delete {
  float: right;

  padding: 0;

  background-color: #fdfdfd;
  border: none;

  cursor: pointer;

  visibility: hidden;
}
.note:hover .delete {
  visibility: visible;
}
```

Assuming everything went fine, your application should look roughly like this now:



Styled Notes Application

## 7.4 Conclusion

This is only one way to style a React application. Relying on classes like this will become problematic as the scale of your application grows. That is why there are alternative ways to style that address this particular problem. The *Styling React* chapter touches a lot of those techniques.

It can be a good idea to try out a couple of alternative ways to find something you are comfortable with. Particularly **CSS Modules** are promising as they solve the fundamental problem of CSS - the problem of globals. The technique allows styling locally per component. That happens to fit React very well since we are dealing with components by default.

Now that we have a simple Notes application up and running, we can begin to generalize it into a full blown Kanban. It will take some patience as we'll need to improve the way we are dealing with the application state. We also need to add some missing structure and make sure it's possible to drag and drop things around. Those are good goals for the next part of the book.



## II Implementing Kanban

In this part, we will turn our Notes application into a Kanban application. During the process, you will learn the basics of React. As React is just a view library we will also discuss supporting technology. We will set up a state management solution to our application. We will also see how to use React DnD to add drag and drop functionality to the Kanban board.

## 8. React and Flux

You can get pretty far by keeping everything in components. That's an entirely valid way to get started. The problems begin as you add state to your application and need to share it across different parts. This is the reason why various state management solutions have emerged. Each one of those tries to solve the problem in its own way.

The [Flux application architecture](https://facebook.github.io/flux/docs/overview.html)<sup>1</sup> was the first proper solution to the problem. It allows you to model your application in terms of **Actions**, **Stores**, and **Views**. It also has a part known as **Dispatcher** to manage actions and allow you to model dependencies between different calls.

This separation is particularly useful when you are working with large teams. The unidirectional flow makes it easy to tell what's going on. That's a common theme in various data management solutions available for React.

### 8.1 Quick Introduction to Redux

A solution known as [Redux](http://redux.js.org/)<sup>2</sup> took the core ideas of Flux and pushed them to a certain form. Redux is more of a guideline, though a powerful one, that gives your application certain structure and pushes you to model data related concerns in a certain way. You will maintain the state of your application in a single tree which you then alter using pure functions (no side effects) through reducers.

This might sound a little complex but in practice Redux makes your data flow very explicit. Standard Flux isn't as opinionated in certain parts. I believe understanding basic Flux before delving into Redux is a good move as you can see shared themes in both.

### 8.2 Quick Introduction to MobX

[MobX](https://mobxjs.github.io/mobx/)<sup>3</sup> takes an entirely different view on data management. If Redux helps you to model data flow explicitly, MobX makes a large part of that implicit. It doesn't force you to any certain structure. Instead you will annotate your data structures as **observable** and let MobX handle updating your views.

Whereas Redux embraces the concept of immutability through its idea of reducers, MobX does something opposite and relies on mutation. This means aspects like reference handling can be surprisingly simple in MobX while in Redux you will most likely be forced to normalize your data so that it is easy to manipulate through reducers.

---

<sup>1</sup><https://facebook.github.io/flux/docs/overview.html>

<sup>2</sup><http://redux.js.org/>

<sup>3</sup><https://mobxjs.github.io/mobx/>

Both Redux and MobX are valuable in their own ways. There's no one right solution when it comes to data management. I'm sure more alternatives will appear as time goes by. Each solution comes with its pros/cons. By understanding the alternatives you have a better chance of picking a solution that fits your purposes at a given time.

## 8.3 Which Data Management Solution to Use?

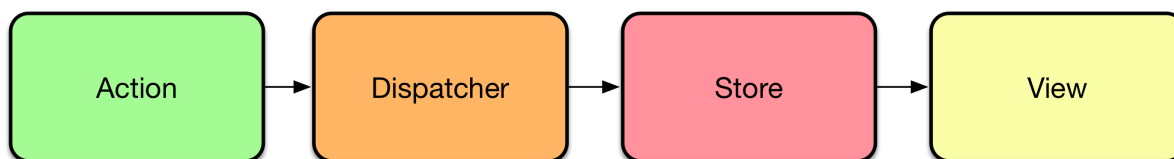
The data management situation is changing constantly. At the moment [Redux](http://rackt.org/redux/)<sup>4</sup> is very strong, but there are good alternatives in sight. [voronianski/flux-comparison](https://github.com/voronianski/flux-comparison)<sup>5</sup> provides a nice comparison between some of the more popular ones.

When choosing a library, it comes down to your own personal preferences. You will have to consider factors, such as API, features, documentation, and support. Starting with one of the more popular alternatives can be a good idea. As you begin to understand the architecture, you are able to make choices that serve you better.

In this application we'll use a Flux implementation known as [Alt](http://alt.js.org/)<sup>6</sup>. The API is neat and enough for our purposes. As a bonus, Alt has been designed universal (isomorphic) rendering in mind. If you understand Flux, you have a good starting point for understanding the alternatives.

The book doesn't cover the alternative solutions in detail yet, but we'll design our application architecture so that it's possible to plug in alternatives at a later time. The idea is that we isolate our view portion from the data management so that we can swap parts without tweaking our React code. It's one way to design for change.

## 8.4 Introduction to Flux



Unidirectional Flux dataflow

So far, we've been dealing only with views. Flux architecture introduces a couple of new concepts to the mix. These are actions, dispatcher, and stores. Flux implements unidirectional flow in contrast to popular frameworks, such as Angular or Ember. Even though two-directional bindings can be convenient, they come with a cost. It can be hard to deduce what's going on and why.

---

<sup>4</sup><http://rackt.org/redux/>

<sup>5</sup><https://github.com/voronianski/flux-comparison>

<sup>6</sup><http://alt.js.org/>

## Actions and Stores

Flux isn't entirely simple to understand as there are many concepts to worry about. In our case, we will model `NoteActions` and `NoteStore`. `NoteActions` provide concrete operations we can perform over our data. For instance, we can have `NoteActions.create({task: 'Learn React'})`.

## Dispatcher

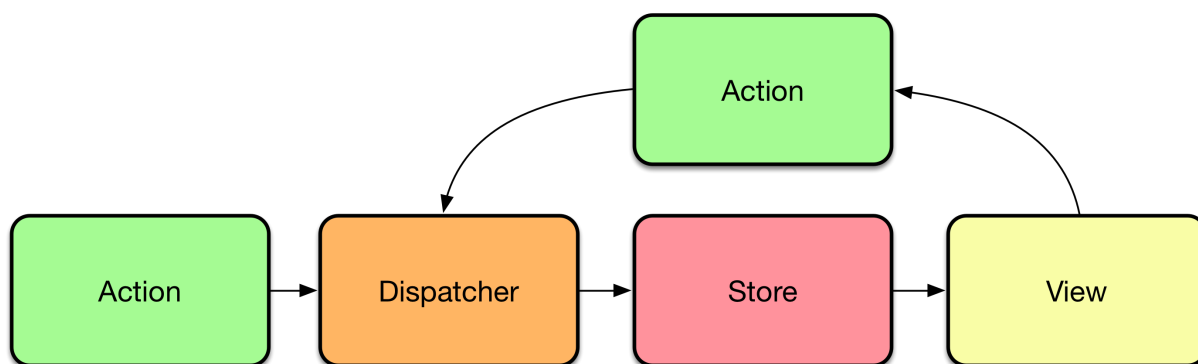
When we trigger an action, the dispatcher will get notified. The dispatcher will be able to deal with possible dependencies between stores. It is possible that a certain action needs to happen before another. The dispatcher allows us to achieve this.

At the simplest level, actions can just pass the message to the dispatcher as is. They can also trigger asynchronous queries and hit the dispatcher based on the result eventually. This allows us to deal with received data and possible errors.

Once the dispatcher has dealt with an action, the stores listening to it get triggered. In our case, `NoteStore` gets notified. As a result, it will be able to update its internal state. After doing this, it will notify possible listeners of the new state.

## Flux Dataflow

This completes the basic unidirectional, yet linear, process flow of Flux. Usually, though, the unidirectional process has a cyclical flow and it doesn't necessarily end. The following diagram illustrates a more common flow. It is the same idea again, but with the addition of a returning cycle. Eventually, the components depending on our store data become refreshed through this looping process.



Cyclical Flux dataflow

This sounds like a lot of steps for achieving something simple as creating a new `Note`. The approach does come with its benefits. Given the flow is always in a single direction, it is easy to trace and debug. If there's something wrong, it's somewhere within the cycle.

Better yet, we can consume the same data across our application. You will just connect your view to a store and that's it. This is one of the great benefits of using a state management solution.

## Advantages of Flux

Even though this sounds a little complicated, the arrangement gives our application flexibility. We can, for instance, implement API communication, caching, and *idn* outside of our views. This way they stay clean of logic while keeping the application easier to understand.

Implementing Flux architecture in your application will actually increase the amount of code somewhat. It is important to understand that minimizing the amount of code written isn't the goal of Flux. It has been designed to allow productivity across larger teams. You could say that explicit is better than implicit.

## 8.5 Porting to Alt



### Alt

In Alt, you'll deal with actions and stores. The dispatcher is hidden, but you will still have access to it if needed. Compared to other implementations, Alt hides a lot of boilerplate. There are special features to allow you to save and restore the application state. This is handy for implementing persistency and universal rendering.

There are a couple of steps we must take to push our application state to Alt:

1. Set up an Alt instance to keep track of actions and stores and to coordinate communication.
2. Connect Alt with views.
3. Push our data to a store.
4. Define actions to manipulate the store.

We'll do this gradually. The Alt specific portions will go behind adapters. The adapter approach allows us to change our mind later easier so it's worth implementing.

## Setting Up an Alt Instance

Everything in Alt begins from an Alt instance. It keeps track of actions and stores and keeps communication going on. To keep things simple, we'll be treating all Alt components as a [singleton](https://en.wikipedia.org/wiki/Singleton_pattern)<sup>7</sup>. With this pattern, we reuse the same instance within the whole application.

To achieve this we can push it to a module of its own and then refer to that from everywhere. Configure it as follows:

app/libs/alt.js

```
import Alt from 'alt';

const alt = new Alt();

export default alt;
```

Webpack caches the modules so the next time you import Alt from somewhere, it will return the same instance again.

## Connecting Alt with Views

Normally state management solutions provide two parts you can use to connect them with a React application. These are a `Provider` component and a `connect` higher order function (function returning function generating a component). The `Provider` sets up a React [context](https://facebook.github.io/react/docs/context.html)<sup>8</sup>.

Context is an advanced feature that can be used to pass data through a component hierarchy implicitly without going through props. The `connect` function uses the context to dig the data we want and then passes it to a component.

It is possible to use a `connect` through function invocation or a decorator as we'll see soon. The *Understanding Decorators* appendix digs deeper into the pattern.

To keep our application architecture easy to modify, we'll need to set up two adapters. One for `Provider` and one for `connect`. We'll deal with Alt specific details in both places.

## Setting Up a Provider

In order to keep our `Provider` flexible, I'm going to use special configuration. We'll wrap it within a module that will choose a `Provider` depending on our environment. This enables us to use development tooling without including it to the production bundle. There's some additional setup involved, but it's worth it given you end up with a cleaner result.

---

<sup>7</sup>[https://en.wikipedia.org/wiki/Singleton\\_pattern](https://en.wikipedia.org/wiki/Singleton_pattern)

<sup>8</sup><https://facebook.github.io/react/docs/context.html>

The core of this arrangement is the index of the module. CommonJS picks up the **index.js** of a directory by default when we perform an import against the directory. Given the behavior we want is dynamic, we cannot rely on ES6 modules here.

The idea is that our tooling will rewrite the code depending on `process.env.NODE_ENV` and choose the actual module to include based on that. Here's the entry point of our Provider:

#### **app/components/Provider/index.js**

```
if(process.env.NODE_ENV === 'production') {
  module.exports = require('./Provider.prod');
}
else {
  module.exports = require('./Provider.dev');
}
```

We also need the files the index is pointing at. The first part is easy. We'll need to point to our Alt instance there, connect it with a component known as `AltContainer`, and then render out application within it. That's where `props.children` comes in. It's the same idea as before.

`AltContainer` will enable us to connect the data of our application at component level when we implement `connect`. To get to the point, here's the production level implementation:

#### **app/components/Provider/Provider.prod.jsx**

```
import React from 'react';
import AltContainer from 'alt-container';
import alt from '../../libs/alt';
import setup from './setup';

setup(alt);

export default ({children}) =>
  <AltContainer flux={alt}>
    {children}
  </AltContainer>
```

The implementation of `Provider` can change based on which state management solution we are using. It is possible it ends up doing nothing, but that's acceptable. The idea is that we have an extension point where to alter our application if needed.

We are still missing one part, the development related setup. It is like the production one except this time we can enable development specific tooling. This is a good chance to move the *react-addons-perf* setup here from the *app/index.jsx* of the application. I'm also enabling [Alt's Chrome debug utilities](#)<sup>9</sup>. You'll need to install the Chrome portion separately if you want to use those.

---

<sup>9</sup><https://github.com/goatslacker/alt-devtool>

Here's the full code of the development provider:

**app/components/Provider/Provider.dev.jsx**

```
import React from 'react';
import AltContainer from 'alt-container';
import chromeDebug from 'alt-utils/lib/chromeDebug';
import alt from '../../libs/alt';
import setup from './setup';

setup(alt);

chromeDebug(alt);

React.Perf = require('react-addons-perf');

export default ({children}) =>
  <AltContainer flux={alt}>
    {children}
  </AltContainer>
```

That setup module allows us to perform Alt related setup that's common for both production and development environment. For now it's enough to do nothing there like this:

**app/components/Provider/setup.js**

```
export default alt => {}
```

We still need to connect the Provider with our application by tweaking *app/index.jsx*. Perform the following changes to hook it up:

**app/index.jsx**

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/App';
import Provider from './components/Provider';

if(process.env.NODE_ENV !== 'production') {
  React.Perf = require('react-addons-perf');
}

ReactDOM.render(
```



```

—<App />,
  <Provider><App /></Provider>,
  document.getElementById('app')
);

```

If you check out Webpack output, you'll likely see it is installing new dependencies to the project. That's expected given the changes. The process might take a while to complete. Once completed, refresh the browser.

Given we didn't change the application logic in any way, everything should still look the same. A good next step is to implement an adapter for connecting data to our views.



You can see a similar idea in [react-redux](https://react-redux.js.org/)<sup>10</sup>. MobX won't need a Provider at all. In that case our implementation would simply return `children`.

## 8.6 Understanding connect

The idea of `connect` is to allow us to attach specific data and actions to components. I've modeled the API after `react-redux`. Fortunately we can adapt various data management systems to work against it. Here's how you would connect lane data and actions with `App`:

```

@connect(({lanes}) => ({lanes}), {
  laneActions: LaneActions
})
export default class App extends React.Component {
  render() {
    return (
      <div>
        <button className="add-lane" onClick={this.addLane}></button>
        <Lanes lanes={this.props.lanes} />
      </div>
    );
  }
  addLane = () => {
    this.props.laneActions.create({name: 'New lane'});
  }
}

```

The same could be written without decorators. This is the syntax we'll be using in our application:

---

<sup>10</sup><https://www.npmjs.com/package/react-redux>

```
class App extends React.Component {  
  ...  
}  
  
export default connect(({lanes}) => ({lanes}), {  
  LaneActions  
})(App)
```

In case you need to apply multiple higher order functions against a component, you could use an utility like `compose` and end up with `compose(a, b)(App)`. This would be equal to `a(b(App))` and it would read a little better.

As the examples show, `compose` is a function returning a function. That's why we call it a higher order function. In the end we get a component out of it. This wrapping allows us to handle our data connection concern.

We could use a higher order function to annotate our components to give them other special properties as well. We will see the idea again when we implement drag and drop later in this part. Decorators provide a nicer way to attach these types of annotations. The *Understanding Decorators* appendix delves deeper into the topic.

Now that we have a basic understanding of how `connect` should work, we can implement it.

## Setting Up `connect`

In this case I'm going to plug in a custom `connect` to highlight a couple of key ideas. The implementation isn't optimal when it comes to performance. It is enough for this application, though.

It would be possible to optimize the behavior with further effort. You could use one of the established connectors instead or develop your own here. That's one reason why having control over `Provider` and `connect` is useful. It allows further customization and understanding of how the process works.

In case we have a custom state transformation defined, we'll dig the data we need from the `Provider`, apply it over our data as we defined, and then pass the resulting data to the component through props:

**app/libs/connect.jsx**

```

import React from 'react';

export default (state, actions) => {
  if(typeof state === 'function' ||
    (typeof state === 'object' && Object.keys(state).length)) {
    return target => connect(state, actions, target);
  }

  return target => props => (
    <target {...Object.assign({}, props, actions)} />
  );
}

// Connect to Alt through context. This hasn't been optimized
// at all. If Alt store changes, it will force render.
//
// See *AltContainer* and *connect-alt* for optimized solutions.
function connect(state = () => {}, actions = {}, target) {
  class Connect extends React.Component {
    componentDidMount() {
      const {flux} = this.context;

      flux.FinalStore.listen(this.handleChange);
    }
    componentWillUnmount() {
      const {flux} = this.context;

      flux.FinalStore.unlisten(this.handleChange);
    }
    render() {
      const {flux} = this.context;
      const stores = flux.stores;
      const composedStores = composeStores(stores);

      return React.createElement(target,
        {...Object.assign(
          {}, this.props, state(composedStores), actions
        )}
      );
    }
  }
  handleChange = () => {
    this.forceUpdate();
  }
}

```

```

    }
  }
  Connect.contextTypes = {
    flux: React.PropTypes.object.isRequired
  }

  return Connect;
}

// Transform {store: <AltStore>} to {<store>: store.getState()}
function composeStores(stores) {
  let ret = {};

  Object.keys(stores).forEach(k => {
    const store = stores[k];

    // Combine store state
    ret = Object.assign({}, ret, store.getState());
  });

  return ret;
}

```

As `flux.FinalStore` won't be available by default, we'll need to alter our `Alt` instance to contain it. After that we can access it whenever we happen to need it:

**app/libs/alt.js**

```

import Alt from 'alt';
import makeFinalStore from 'alt-utils/lib/makeFinalStore';

const alt = new Alt();

export default alt;

class Flux extends Alt {
  constructor(config) {
    super(config);

    this.FinalStore = makeFinalStore(this);
  }
}

const flux = new Flux();

```

```
export default flux;
```

In order to see connect in action, we could use it to attach some dummy data to App and then render it. Adjust it as follows to pass data test to App and then show it in the user interface:

**app/components/App.jsx**

```
import React from 'react';
import uuid from 'uuid';
import Notes from '../Notes';
import connect from '../libs/connect';

export default class App extends React.Component {
class App extends React.Component {
  constructor(props) {
    ...
  }
  render() {
    const {notes} = this.state;

    return (
      <div>
        {this.props.test}
        <button className="add-note" onClick={this.addNote}></button>
        <Notes
          notes={notes}
          onNoteClick={this.activateNoteEdit}
          onEdit={this.editNote}
          onDelete={this.deleteNote}
        />
      </div>
    );
  }
  ...
}

export default connect(() => ({
  test: 'test'
}))(App)
```

To make the text show up, refresh the browser. You should see the text that we connected to App now.

## 8.7 Dispatching in Alt

Even though you can get far without ever using Flux dispatcher, it can be useful to know something about it. Alt provides two ways to use it. If you want to log everything that goes through your alt instance, you can use a snippet, such as `alt.dispatcher.register(console.log.bind(console))`. Alternatively, you could trigger `this.dispatcher.register(...)` at a store constructor. These mechanisms allow you to implement effective logging.

Other state management systems provide similar hooks. It is possible to intercept the data flow in many ways and even build custom logic on top of that.

## 8.8 Conclusion

In this chapter we discussed the basic idea of the Flux architecture and started porting our application to it. We pushed the state management related concerns behind an adapter to allow altering the underlying system without having to change the view related code. The next step is to implement a store for our application data and define actions to manipulate it.

## 9. Implementing NoteStore and NoteActions

Now that we have pushed data management related concerns in the right places, we can focus on implementing the remaining portions - NoteStore and NoteActions. These will encapsulate the application data and logic.

No matter what state management solution you end up using, there is usually something equivalent around. In Redux you would end up using actions that then trigger a state change through a reducer. In MobX you could model action API within an ES6 class that then manipulates the data causing your views to refresh as needed.

The idea is similar here. We will set up actions that will end up triggering our store methods that modify the state. As the state changes, our views will update. To get started, we can implement a NoteStore and then define logic to manipulate it. Once we have done that, we have completed porting our application to the Flux architecture.

### 9.1 Setting Up a NoteStore

Currently we maintain the application state at App. The first step towards pushing it to Alt is to define a store and then consume it from there. This will break the logic of our application temporarily as that needs to be pushed to Alt as well. Setting up an initial store is a good step towards this overall goal, though.

To set up a store we need to perform three steps. We'll need to set it up, then connect it with Alt at Provider, and finally connect it with App.

In Alt we model stores using ES6 classes. Here's a minimal implementation modeled after our current state:

`app/stores/NoteStore.js`

```
import uuid from 'uuid';

export default class NoteStore {
  constructor() {
    this.notes = [
      {
        id: uuid.v4(),
        task: 'Learn React'
      },
      {
        id: uuid.v4(),
        task: 'Do laundry'
      }
    ];
  }
}
```

The next step is connecting the store with Provider. This is where that setup module comes in handy:

**app/components/Provider/setup.js**

```
export default alt => {}
import NoteStore from '../../stores/NoteStore';

export default alt => {
  alt.addStore('NoteStore', NoteStore);
}
```

To prove that our setup works, we can adjust App to consume its data from the store. This will break the logic since we don't have any way to adjust the store data yet, but that's something we'll fix in the next section. Tweak App as follows to make notes available there:

**app/components/App.jsx**



```

...

class App extends React.Component {
  constructor(props) {
  super(props);

  this.state = {
    notes: [
      {
        id: uuid.v4(),
        task: 'Learn React'
      },
      {
        id: uuid.v4(),
        task: 'Do laundry'
      }
    ]
  }
  render() {
    const {notes} = this.state;
    const {notes} = this.props;

  return (
    <div>
    {this.props.test}
    <button className="add-note" onClick={this.addNote}></button>
    <Notes
      notes={notes}
      onNoteClick={this.activateNoteEdit}
      onEdit={this.editNote}
      onDelete={this.deleteNote}
    />
    </div>
  );
}
...
}

export default connect(() => ({
  test: 'test'
})))(App)

```

```
export default connect(({notes}) => ({
  notes
}))(App)
```

If you refresh the application now, you should see exactly the same data as before. This time, however, we are consuming the data from our store. As a result our logic is broken. That's something we'll need to fix next as we define NoteActions and push our state manipulation to the NoteStore.



Given App doesn't depend on state anymore, it would be possible to port it as a function based component. Often most of your components will be based on functions just for this reason. If you aren't using state or refs, then it's safe to default to functions.

## 9.2 Understanding Actions

Actions are one of the core concepts of the Flux architecture. To be exact, it is a good idea to separate **actions** from **action creators**. Often the terms might be used interchangeably, but there's a considerable difference.

Action creators are literally functions that *dispatch* actions. The payload of the action will then be delivered to the interested stores. It can be useful to think them as messages wrapped into an envelope and then delivered.

This split is useful when you have to perform asynchronous actions. You might for example want to fetch the initial data of your Kanban board. The operation might then either succeed or fail. This gives you three separate actions to dispatch. You could dispatch when starting to query and when you receive some response.

All of this data is valuable as it allows you to control the user interface. You could display a progress widget while a query is being performed and then update the application state once it has been fetched from the server. If the query fails, you can then let the user know about that.

You can see this theme across different state management solutions. Often you model an action as a function that returns a function (a *thunk*) that then dispatches individual actions as the asynchronous query progresses. In a naïve synchronous case it's enough to return the action payload directly.



The official documentation of Alt covers [asynchronous actions](http://alt.js.org/docs/createActions/)<sup>1</sup> in greater detail.

---

<sup>1</sup><http://alt.js.org/docs/createActions/>

## 9.3 Setting Up NoteActions

Alt provides a little helper method known as `alt.generateActions` that can generate simple action creators for us. They will simply dispatch the data passed to them. We'll then connect these actions at the relevant stores. In this case that will be the `NoteStore` we defined earlier.

When it comes to the application, it is enough if we model basic CRUD (Create, Read, Update, Delete) operations. Given Read is implicit, we can skip that. But having the rest available as actions is useful. Set up `NoteActions` using the `alt.generateActions` shorthand like this:

**app/actions/NoteActions.js**

```
import alt from '../libs/alt';

export default alt.generateActions('create', 'update', 'delete');
```

This doesn't do much by itself. Given we need to connect the actions with App to actually trigger them, this would be a good place to do that. We can start worrying about individual actions after that as we expand our store. To connect the actions, tweak App like this:

**app/components/App.jsx**

```
import React from 'react';
import uuid from 'uuid';
import Notes from './Notes';
import connect from '../libs/connect';
import NoteActions from '../actions/NoteActions';

class App extends React.Component {
  ...
}

export default connect(({notes}) => ({
  —notes
}))(App)
export default connect(({notes}) => ({
  notes
}), {
  NoteActions
})(App)
```

This gives us `this.props.NoteActions.create` kind of API for triggering various actions. That's good for expanding the implementation further.

## 9.4 Connecting NoteActions with NoteStore

Alt provides a couple of convenient ways to connect actions to a store:

- `this.bindAction(NoteActions.CREATE, this.create)` - Bind a specific action to a specific method.
- `this.bindActions(NoteActions)` - Bind all actions to methods by convention. I.e., create action would map to a method named create.
- `reduce(state, { action, data })` - It is possible to implement a custom method known as reduce. This mimics the way Redux reducers work. The idea is that you'll return a new state based on the given state and payload.

We'll use `this.bindActions` in this case as it's enough to rely on convention. Tweak the store as follows to connect the actions and to add initial stubs for the logic:

**app/stores/NoteStore.js**

```
import uuid from 'uuid';
import NoteActions from '../actions/NoteActions';

export default class NoteStore {
  constructor() {
    this.bindActions(NoteActions);

    this.notes = [
      {
        id: uuid.v4(),
        task: 'Learn React'
      },
      {
        id: uuid.v4(),
        task: 'Do laundry'
      }
    ];
  }

  create(note) {
    console.log('create note', note);
  }

  update(updatedNote) {
    console.log('update note', updatedNote);
  }

  delete(id) {
```

```

    console.log('delete note', id);
  }
}

```

To actually see it working, we'll need to start connecting our actions at App and then start porting the logic over.

## 9.5 Porting App.addNote to Flux

App.addNote is a good starting point. The first step is to trigger the associated action (NoteActions.create) from the method and see if we see something at the browser console. If we do, then we can manipulate the state. Trigger the action like this:

app/components/App.jsx

```

...

class App extends React.Component {
  render() {
    ...
  }
  addNote = () => {
    // It would be possible to write this in an imperative style.
    // I.e., through `this.state.notes.push` and then
    // `this.setState({notes: this.state.notes})` to commit.
    //
    // I tend to favor functional style whenever that makes sense.
    // Even though it might take more code sometimes, I feel
    // the benefits (easy to reason about, no side effects)
    // more than make up for it.
    //
    // Libraries, such as Immutable.js, go a notch further.
    this.setState({
      notes: this.state.notes.concat([
        id: uuid.v4(),
        task: 'New task'
      ])
    });
    this.props.NoteActions.create({
      id: uuid.v4(),
      task: 'New task'
    });
  }
}

```

```

    }
    ...
  }

  ...

```

If you refresh and click the “add note” button now, you should see messages like this at the browser console:

```
create note Object {id: "62098959-6289-4894-9bf1-82e983356375", task: "New task"}
```

This means we have the data we need at the `NoteStore` `create` method. We still need to manipulate the data. After that we have completed the loop and we should see new notes through the user interface. Alt follows a similar API as React here. Consider the implementation below:

**app/stores/NoteStore.js**

```

import uuid from 'uuid';
import NoteActions from '../actions/NoteActions';

export default class NoteStore {
  constructor() {
    ...
  }
  create(note) {
    console.log('create note', note);
    this.setState({
      notes: this.notes.concat(note)
    });
  }
  ...
}

```

If you try adding a note now, the update should go through. Alt maintains the state now and the edit goes through thanks to the architecture we set up. We still have to repeat the process for the remaining methods to complete the work.

## 9.6 Porting `App.deleteNote` to Flux

The process exactly the same for `App.deleteNote`. We’ll need to connect it with our action and then port it over. Here’s the App portion:

**app/components/App.jsx**

```

...

class App extends React.Component {
  ...
  deleteNote = (id, e) => {
    // Avoid bubbling to edit
    e.stopPropagation();

    this.setState({
    notes: this.state.notes.filter(note => note.id !== id)
    });
    this.props.NoteActions.delete(id);
  }
  ...
}

...

```

If you refresh and try to delete a note now, you should see a message like this at the browser console:

```
delete note 501c13e0-40cb-47a3-b69a-b1f2f69c4c55
```

To finalize the porting, we'll need to move the `setState` logic to the `delete` method. Remember to drop `this.state.notes` and replace that with just `this.notes`:

### app/stores/NoteStore.js

```

import uuid from 'uuid';
import NoteActions from '../actions/NoteActions';

export default class NoteStore {
  ...
  delete(id) {
    console.log('delete note', id);
    this.setState({
      notes: this.notes.filter(note => note.id !== id)
    });
  }
}

```

After this change you should be able to delete notes just like before. There are still a couple of methods to port.

## 9.7 Porting App.activateNoteEdit to Flux

App.activateNoteEdit is essentially an update operation. We'll need to change the editing flag of the given note as true. That will initiate the editing process. As usual, we can port App to the scheme first:

app/components/App.jsx

```
...

class App extends React.Component {
  ...
  activateNoteEdit = (id) => {
    this.setState({
      notes: this.state.notes.map(note => {
        if(note.id === id){
          note.editing = true;
        }
      }
    });
    return note;
  }}
});
    this.props.NoteActions.update({id, editing: true});
  }
  ...
}

...
```

If you refresh and try to edit now, you should see messages like this at the browser console:

```
update note Object {id: "2c91ba0f-12f5-4203-8d60-ea673ee00e03", editing: true}
```

We still need to commit the change to make this work. The logic is the same as in App before except we have generalized it further using Object.assign:

app/stores/NoteStore.js



```
import uuid from 'uuid';
import NoteActions from '../actions/NoteActions';

export default class NoteStore {
  ...
  update(updatedNote) {
    console.log('update note', updatedNote);
    this.setState({
      notes: this.notes.map(note => {
        if(note.id === updatedNote.id) {
          return Object.assign({}, note, updatedNote);
        }

        return note;
      })
    });
  }
  ...
}
```

It should be possible to start editing a note now. If you try to finish editing, you should get an error like `Uncaught TypeError: Cannot read property 'notes' of null`. This is because we are missing one final portion of the porting effort, `App.editNote`.

## 9.8 Porting `App.editNote` to Flux

This final part is easy. We have already the logic we need. Now it's just a matter of connecting `App.editNote` to it in a correct way. We'll need to call our update method the correct way:

`app/components/App.jsx`

```
...

class App extends React.Component {
  ...
  editNote = (id, task) => {
    this.setState({
      notes: this.state.notes.map(note => {
        if(note.id === id) {
          note.editing = false;
          note.task = task;
        }
      }

```

```

return note;
}}
});
    const {NoteActions} = this.props;

    NoteActions.update({id, task, editing: false});
  }
}

...

```

After refreshing you should be able to modify tasks again and the application should work just like before now. As we alter NoteStore through actions, this leads to a cascade that causes our App state to update through `setState`. This in turn will cause the component to render. That's Flux's unidirectional flow in practice.

We actually have more code now than before, but that's okay. App is a little neater and it's going to be easier to develop as we'll soon see. Most importantly we have managed to implement the Flux architecture for our application.



The current implementation is naïve in that it doesn't validate parameters in any way. It would be a very good idea to validate the object shape to avoid incidents during development. [Flow<sup>2</sup>](https://flowtype.org/) based gradual typing provides one way to do this. In addition you could write tests to support the system.

## What's the Point?

Even though integrating a state management system took a lot of effort, it was not all in vain. Consider the following questions:

1. Suppose we wanted to persist the notes within `localStorage`. Where would you implement that? One approach would be to handle that at the `Provider` setup.
2. What if we had many components relying on the data? We would just consume the data through `connect` and display it, however we want.
3. What if we had many, separate Note lists for different types of tasks? We could set up another store for tracking these lists. That store could refer to actual Notes by id. We'll do something like this in the next chapter, as we generalize the approach.

---

<sup>2</sup><http://flowtype.org/>

Adopting a state management system can be useful as the scale of your React application grows. The abstraction comes with some cost as you end up with more code. But on the other hand if you do it right, you'll end up with something that's easy to reason and develop further. Especially the unidirectional flow embraced by these systems helps when it comes to debugging and testing.

## 9.9 Conclusion

In this chapter, you saw how to port our simple application to use Flux architecture. In the process we learned more about **actions** and **stores** of Flux. Now we are ready to start adding more functionality to our application. We'll add `localStorage` based persistency to the application next and perform a little clean up while at it.

# 10. Implementing Persistency over `localStorage`

Currently our application cannot retain its state if refreshed. One neat way to get around this problem is to store the application state to `localStorage`<sup>1</sup> and then restore it when we run the application again.

If you were working against a back-end, this wouldn't be a problem. Even then having a temporary cache in `localStorage` could be handy. Just make sure you don't store anything sensitive there as it is easy to access.

## 10.1 Understanding `localStorage`

`localStorage` is a part of the Web Storage API. The other half, `sessionStorage`, exists as long as the browser is open while `localStorage` persists even in this case. They both share [the same API](#)<sup>2</sup> as discussed below:

- `storage.getItem(k)` - Returns the stored string value for the given key.
- `storage.removeItem(k)` - Removes the data matching the key.
- `storage.setItem(k, v)` - Stores the given value using the given key.
- `storage.clear()` - Empties the storage contents.

It is convenient to operate on the API using your browser developer tools. In Chrome especially the *Resources* tab is useful as it allows you to inspect the data and perform direct operations on it. You can even use `storage.key` and `storage.key = 'value'` shorthands in the console for quick tweaks.

`localStorage` and `sessionStorage` can use up to 10 MB of data combined. Even though they are well supported, there are certain corner cases that can fail. These include running out of memory in Internet Explorer (fails silently) and failing altogether in Safari's private mode. It is possible to work around these glitches, though.



You can support Safari in private mode by trying to write into `localStorage` first. If that fails, you can use Safari's in-memory store instead, or just let the user know about the situation. See [Stack Overflow](#)<sup>3</sup> for details.

---

<sup>1</sup><https://developer.mozilla.org/en/docs/Web/API/Window/localStorage>

<sup>2</sup>[https://developer.mozilla.org/en-US/docs/Web/API/Web\\_Storage\\_API/Using\\_the\\_Web\\_Storage\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API/Using_the_Web_Storage_API)

<sup>3</sup><https://stackoverflow.com/questions/14555347/html5-localstorage-error-with-safari-quota-exceeded-err-dom-exception-22-an>

## 10.2 Implementing a Wrapper for localStorage

To keep things simple and manageable, we will implement a little wrapper for storage to wrap the complexity. The API will consist of `get(k)` to fetch items from the storage and `set(k, v)` to set them. Given the underlying API is string based, we'll use `JSON.parse` and `JSON.stringify` for serialization. Since `JSON.parse` can fail, that's something we need to take into account. Consider the implementation below:

`app/libs/storage.js`

```
export default storage => ({
  get(k) {
    try {
      return JSON.parse(storage.getItem(k));
    }
    catch(e) {
      return null;
    }
  },
  set(k, v) {
    storage.setItem(k, JSON.stringify(v));
  }
})
```

The implementation is enough for our purposes. It's not foolproof and it will fail if we put too much data into a storage. To overcome these problems without having to solve them yourself, it would be possible to use a wrapper such as [localForage](https://github.com/mozilla/localForage)<sup>4</sup> to hide the complexity.

## 10.3 Persisting the Application Using FinalStore

Just having means to write and read from the `localStorage` won't do. We still need to connect our application to it somehow. State management solutions provide hooks for this purpose. Often you'll find a way to intercept them somehow. In Alt's case that happens through a built-in store known as `FinalStore`.

We have already set it up at our Alt instance. What remains is writing the application state to the `localStorage` when it changes. We also need to load the state when we start running it. In Alt terms these processes are known as **snapshotting** and **bootstrapping**.



An alternative way to handle storing the data would be to take a snapshot only when the window gets closed. There's a `Window` level `beforeunload` hook that could be used. This approach is brittle, though. What if something unexpected happens and the hook doesn't get triggered for some reason? You'll lose data.

---

<sup>4</sup><https://github.com/mozilla/localForage>

## 10.4 Implementing the Persistency Logic

We can handle the persistency logic at a separate module dedicated to it. We will hook it up at the application setup and off we go.

Given it can be useful to be able to disable snapshotting temporarily, it can be a good idea to implement a debug flag. The idea is that if the flag is set, we'll skip storing the data.

This is particularly useful if we manage to break the application state dramatically during development somehow as it allows us to restore it to a blank slate easily through `localStorage.setItem('debug', 'true')` (`localStorage.debug = true`), `localStorage.clear()`, and finally a refresh.

Given bootstrapping could fail for an unknown reason, we catch a possible error. It can still be a good idea to proceed with starting the application even if something horrible happens at this point. The snapshot portion is easier as we just need to check for the debug flag there and then set data if the flag is not active.

The implementation below illustrates the ideas:

**app/libs/persist.js**

```
export default function(alt, storage, storageName) {
  try {
    alt.bootstrap(storage.get(storageName));
  }
  catch(e) {
    console.error('Failed to bootstrap data', e);
  }

  alt.FinalStore.listen(() => {
    if(!storage.get('debug')) {
      storage.set(storageName, alt.takeSnapshot());
    }
  });
}
```

You would end up with something similar in other state management systems. You'll need to find equivalent hooks to initialize the system with data loaded from the `localStorage` and write the state there when it happens to change.

## 10.5 Connecting Persistency Logic with the Application

We are still missing one part to make this work. We'll need to connect the logic with our application. Fortunately there's a suitable place for this, the setup. Tweak as follows:

### app/components/Provider/setup.js

```
import storage from '../../libs/storage';
import persist from '../../libs/persist';
import NoteStore from '../../stores/NoteStore';

export default alt => {
  alt.addStore('NoteStore', NoteStore);

  persist(alt, storage(localStorage), 'app');
}
```

If you try refreshing the browser now, the application should retain its state. Given the solution is generic, adding more state to the system shouldn't be a problem. We could also integrate a proper back-end through the same hooks if we wanted.

If we had a real back-end, we could pass the initial payload as a part of the HTML and load it from there. This would avoid a round trip. If we rendered the initial markup of the application as well, we would end up implementing basic **universal rendering** approach. Universal rendering is a powerful technique that allows you to use React to improve the performance of your application while gaining SEO benefits.



Our persist implementation isn't without its flaws. It is easy to end up in a situation where localStorage contains invalid data due to changes made to the data model. This brings you to the world of database schemas and migrations. The lesson here is that the more you inject state and logic to your application, the more complicated it gets to handle.

## 10.6 Cleaning Up NoteStore

Before moving on, it would be a good idea to clean up NoteStore. There's still some code hanging around from our earlier experiments. Given persistency works now, we might as well start from a blank slate. Even if we wanted some initial data, it would be better to handle that at a higher level, such as application initialization. Adjust NoteStore as follows:

app/stores/NoteStore.js

```

import uuid from 'uuid';
import NoteActions from '../actions/NoteActions';

export default class NoteStore {
  constructor() {
    this.bindActions(NoteActions);

    this.notes = [
      {
        id: uuid.v4(),
        task: 'Learn React'
      },
      {
        id: uuid.v4(),
        task: 'Do laundry'
      }
    ];
    this.notes = [];
  }
  ...
}

```

This is enough for now. Now our application should start from a blank slate.

## 10.7 Alternative Implementations

Even though we ended up using Alt in this initial implementation, it's not the only option. In order to benchmark various architectures, I've implemented the same application using different techniques. I've compared them briefly below:

- [Redux](http://rackt.org/redux/)<sup>5</sup> is a Flux inspired architecture that was designed with hot loading as its primary constraint. Redux operates based on a single state tree. The state of the tree is manipulated using *pure functions* known as reducers. Even though there's some boilerplate code, Redux forces you to dig into functional programming. The implementation is quite close to the Alt based one. - [Redux demo](https://github.com/survivejs/redux-demo)<sup>6</sup>
- Compared to Redux, [Cerebral](http://www.cerebraljs.com/)<sup>7</sup> had a different starting point. It was developed to provide insight on *how* the application changes its state. Cerebral provides more opinionated way to develop, and as a result, comes with more batteries included. - [Cerebral demo](https://github.com/survivejs/cerebral-demo)<sup>8</sup>

---

<sup>5</sup><http://rackt.org/redux/>

<sup>6</sup><https://github.com/survivejs/redux-demo>

<sup>7</sup><http://www.cerebraljs.com/>

<sup>8</sup><https://github.com/survivejs/cerebral-demo>



- [MobX](#)<sup>9</sup> allows you to make your data structures observable. The structures can then be connected with React components so that whenever the structures update, so do the React components. Given real references between structures can be used, the Kanban implementation is surprisingly simple. - [MobX demo](#)<sup>10</sup>

## 10.8 Relay?

Compared to Flux, Facebook's [Relay](#)<sup>11</sup> improves on the data fetching department. It allows you to push data requirements to the view level. It can be used standalone or with Flux depending on your needs.

Given it's still largely untested technology, we won't be covering it in this book yet. Relay comes with special requirements of its own (GraphQL compatible API). Only time will tell how it gets adopted by the community.

## 10.9 Conclusion

In this chapter, you saw how to set up `localStorage` for persisting the application state. It is a useful little technique to know. Now that we have persistency sorted out, we are ready to start generalizing towards a full blown Kanban board.

---

<sup>9</sup><https://mobxjs.github.io/mobx/>

<sup>10</sup><https://github.com/survivejs/mobx-demo>

<sup>11</sup><https://facebook.github.io/react/blog/2015/02/20/introducing-relay-and-graphql.html>

# 11. Handling Data Dependencies

So far we have developed an application for keeping track of notes in `localStorage`. To get closer to Kanban, we need to model the concept of `Lane`. A `Lane` is something that should be able to contain many `Notes` within itself and track their order. One way to model this is simply to make a `Lane` point at `Notes` through an array of `Note` ids.

This relation could be reversed. A `Note` could point at a `Lane` using an id and maintain information about its position within a `Lane`. In this case, we are going to stick with the former design as that works well with re-ordering later on.

## 11.1 Defining Lanes

As earlier, we can use the same idea of two components here. There will be a component for the higher level (i.e., `Lanes`) and for the lower level (i.e., `Lane`). The higher level component will deal with lane ordering. A `Lane` will render itself (i.e., name and `Notes`) and have basic manipulation operations.

Just as with `Notes`, we are going to need a set of actions. For now it is enough if we can just create new lanes so we can create a corresponding action for that as below:

`app/actions/LaneActions.js`

```
import alt from '../libs/alt';

export default alt.generateActions('create');
```

In addition, we are going to need a `LaneStore` and a method matching to create. The idea is pretty much the same as for `NoteStore` earlier. `create` will concatenate a new lane to the list of lanes. After that, the change will propagate to the listeners (i.e., `FinalStore` and components).

`app/stores/LaneStore.js`

```
import LaneActions from '../actions/LaneActions';

export default class LaneStore {
  constructor() {
    this.bindActions(LaneActions);

    this.lanes = [];
  }
  create(lane) {
    // If `notes` aren't provided for some reason,
    // default to an empty array.
    lane.notes = lane.notes || [];

    this.setState({
      lanes: this.lanes.concat(lane)
    });
  }
}
```

To connect LaneStore with our application, we need to connect it to it through setup:

#### app/components/Provider/setup.js

```
import storage from '../../libs/storage';
import persist from '../../libs/persist';
import NoteStore from '../../stores/NoteStore';
import LaneStore from '../../stores/LaneStore';

export default alt => {
  alt.addStore('NoteStore', NoteStore);
  alt.addStore('LaneStore', LaneStore);

  persist(alt, storage(localStorage), 'app');
}
```

We are also going to need a Lanes container to display our lanes:

#### app/components/Lanes.jsx

```
import React from 'react';
import Lane from './Lane';

export default ({lanes}) => (
  <div className="lanes">{lanes.map(lane =>
    <Lane className="lane" key={lane.id} lane={lane} />
  )}</div>
)
```

And finally we can add a little stub for Lane to make sure our application doesn't crash when we connect Lanes with it. A lot of the current App logic will move here eventually:

**app/components/Lane.jsx**

```
import React from 'react';

export default ({lane, ...props}) => (
  <div {...props}>{lane.name}</div>
)
```

## 11.2 Connecting Lanes with App

Next, we need to make room for Lanes at App. We will simply replace Notes references with Lanes, set up lane actions, and store. This means a lot of the old code can disappear. Replace App with the following code:

**app/components/App.jsx**

```
import React from 'react';
import uuid from 'uuid';
import connect from '../libs/connect';
import Lanes from './Lanes';
import LaneActions from '../actions/LaneActions';

const App = ({LaneActions, lanes}) => {
  const addLane = () => {
    LaneActions.create({
      id: uuid.v4(),
      name: 'New lane'
    });
  };
};
```

```

    return (
      <div>
        <button className="add-lane" onClick={addLane}>></button>
        <Lanes lanes={lanes} />
      </div>
    );
  };

export default connect(({lanes}) => ({
  lanes
}), {
  LaneActions
})(App)

```

If you check out the implementation at the browser, you can see that the current implementation doesn't do much. You should be able to add new lanes to the Kanban and see "New lane" text per each but that's about it. To restore the note related functionality, we need to focus on modeling Lane further.

## 11.3 Modeling Lane

Lane will render a name and associated Notes. The example below has been modeled largely after our earlier implementation of App. Replace the file contents entirely as follows:

**app/components/Lane.jsx**

```

import React from 'react';
import uuid from 'uuid';
import connect from '../libs/connect';
import NoteActions from '../actions/NoteActions';
import Notes from './Notes';

const Lane = ({
  lane, notes, NoteActions, ...props
}) => {
  const editNote = (id, task) => {
    NoteActions.update({id, task, editing: false});
  };
  const addNote = e => {
    e.stopPropagation();

    const noteId = uuid.v4();

```

```

    NoteActions.create({
      id: noteId,
      task: 'New task'
    });
  };
  const deleteNote = (noteId, e) => {
    e.stopPropagation();

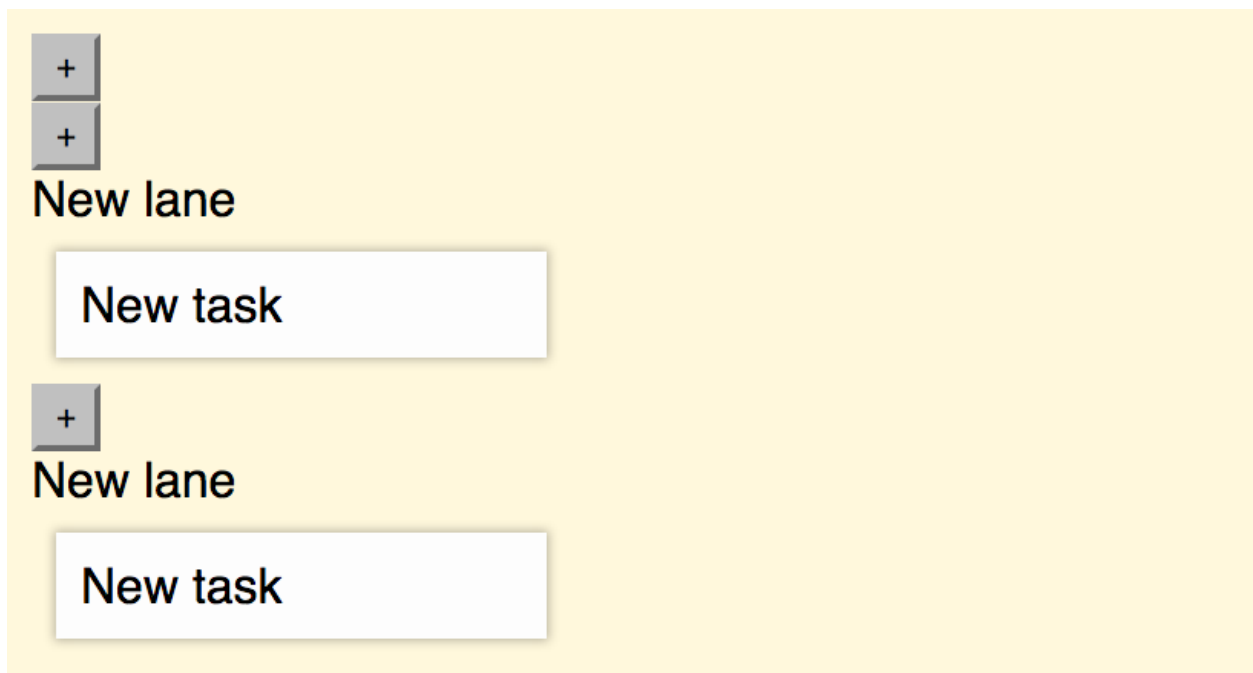
    NoteActions.delete(noteId);
  };
  const activateNoteEdit = id => {
    NoteActions.update({id, editing: true});
  };

  return (
    <div {...props}>
      <div className="lane-header">
        <div className="lane-add-note">
          <button onClick={addNote}></button>
        </div>
        <div className="lane-name">{lane.name}</div>
      </div>
      <Notes
        notes={notes}
        onNoteClick={activateNoteEdit}
        onEdit={editNote}
        onDelete={deleteNote} />
    </div>
  );
};

export default connect(
  ({notes}) => ({
    notes
  }), {
    NoteActions
  }
)(Lane)

```

If you run the application and try adding new notes, you can see there's something wrong. Every note you add is shared by all lanes. If a note is modified, other lanes update too.



Duplicate notes

The reason why this happens is simple. Our `NoteStore` is a singleton. This means every component that is listening to `NoteStore` will receive the same data. We will need to resolve this problem somehow.

## 11.4 Making Lanes Responsible of Notes

Currently, our `Lane` contains just an array of objects. Each of the objects knows its *id* and *name*. We'll need something more sophisticated.

Each `Lane` needs to know which `Notes` belong to it. If a `Lane` contained an array of `Note` ids, it could then filter and display the `Notes` belonging to it. We'll implement a scheme to achieve this next.

### Understanding `attachToLane`

When we add a new `Note` to the system using `addNote`, we need to make sure it's associated to some `Lane`. This association can be modeled using a method, such as `LaneActions.attachToLane({laneId: <id>, noteId: <id>})`. Here's an example of how it could work:

```

const addNote = e => {
  e.stopPropagation();

  const noteId = uuid.v4();

  NoteActions.create({
    id: noteId,
    task: 'New task'
  });
  LaneActions.attachToLane({
    laneId: lane.id,
    noteId
  });
}

```

This is just one way to handle `noteId`. We could push the generation logic within `NoteActions.create` and then return the generated id from it. We could also handle it through a [Promise<sup>1</sup>](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise). This would be very useful if we added a back-end to our implementation. Here's how it would look like then:

```

const addNote = e => {
  e.stopPropagation();

  NoteActions.create({
    task: 'New task'
  }).then(noteId => {
    LaneActions.attachToLane({
      laneId: lane.id,
      noteId: noteId
    });
  })
}

```

Now we have declared a clear dependency between `NoteActions.create` and `LaneActions.attachToLane`. This would be one valid alternative especially if you need to go further with the implementation.



You could model the API using positional parameters and end up with `LaneActions.attachToLane(laneId, note.id)`. I prefer the object form as it reads well and you don't have to care about the order.

<sup>1</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise)





Another way to handle the dependency problem would be to use Flux dispatcher related feature known as [waitFor](http://alt.js.org/guide/wait-for/)<sup>2</sup>. It allows us to state dependencies on store level. It is better to avoid that if you can, though, as data management solutions like Redux make it redundant. Using Promises as above can help as well.

## Setting Up `attachToLane`

To get started we should add `attachToLane` to actions as before:

`app/actions/LaneActions.js`

```
import alt from '../libs/alt';

export default alt.generateActions(
  'create', 'attachToLane'
);
```

In order to implement `attachToLane`, we need to find a lane matching to the given lane id and then attach note id to it. Furthermore, each note should belong only to one lane at a time. We can perform a rough check against that:

`app/stores/LaneStore.js`

```
import LaneActions from '../actions/LaneActions';

export default class LaneStore {
  ...
  attachToLane({laneId, noteId}) {
    this.setState({
      lanes: this.lanes.map(lane => {
        if(lane.notes.includes(noteId)) {
          lane.notes = lane.notes.filter(note => note !== noteId);
        }

        if(lane.id === laneId) {
          lane.notes = lane.notes.concat([noteId]);
        }

        return lane;
      })
    });
  }
}
```

---

<sup>2</sup><http://alt.js.org/guide/wait-for/>

```

    });
  }
}

```

Just being able to attach notes to a lane isn't enough. We are also going to need some way to detach them. This comes up when we are removing notes.



We could give a warning here in case you are trying to attach a note to a lane that doesn't exist. `console.warn` would work nicely for that.

## Setting Up `detachFromLane`

We can model a similar counter-operation `detachFromLane` using an API like this:

```

LaneActions.detachFromLane({noteId, laneId});
NoteActions.delete(noteId);

```



Just like with `attachToLane`, you could model the API using positional parameters and end up with `LaneActions.detachFromLane(laneId, noteId)`.

Again, we should set up an action:

**app/actions/LaneActions.js**

```

import alt from '../libs/alt';

export default alt.generateActions(
  'create', 'attachToLane', 'detachFromLane'
);

```

The implementation will resemble `attachToLane`. In this case, we'll remove the possibly found `Note` instead:

**app/stores/LaneStore.js**

```
import LaneActions from '../actions/LaneActions';

export default class LaneStore {
  ...
  detachFromLane({laneId, noteId}) {
    this.setState({
      lanes: this.lanes.map(lane => {
        if(lane.id === laneId) {
          lane.notes = lane.notes.filter(note => note !== noteId);
        }

        return lane;
      })
    });
  }
}
```

Given we have enough logic in place now, we can start connecting it with the user interface.



It is possible `detachFromLane` doesn't detach anything. If this case is detected, it would be nice to use `console.warn` to let the developer know about the situation.

## Connecting Lane with the Logic

To make this work, there are a couple of places to tweak at Lane:

- When adding a note, we need to **attach** it to the current lane.
- When deleting a note, we need to **detach** it from the current lane.
- When rendering a lane, we need to **select** notes belonging to it. It is important to render then notes in the order they belong to a lane. This needs extra logic.

These changes map to Lane as follows:

`app/components/Lane.jsx`

```

import React from 'react';
import uuid from 'uuid';
import connect from '../libs/connect';
import NoteActions from '../actions/NoteActions';
import LaneActions from '../actions/LaneActions';
import Notes from './Notes';

const Lane = ({
lane, notes, NoteActions, ...props
lane, notes, LaneActions, NoteActions, ...props
}) => {
  const editNote = (id, task) => {
    ...
  };
  const addNote = e => {
    e.stopPropagation();

    const noteId = uuid.v4();

    NoteActions.create({
      id: noteId,
      task: 'New task'
    });
    LaneActions.attachToLane({
      laneId: lane.id,
      noteId
    });
  };
  const deleteNote = (noteId, e) => {
    e.stopPropagation();

    LaneActions.detachFromLane({
      laneId: lane.id,
      noteId
    });
    NoteActions.delete(noteId);
  };
  const activateNoteEdit = id => {
    NoteActions.update({id, editing: true});
  };

  return (

```

```

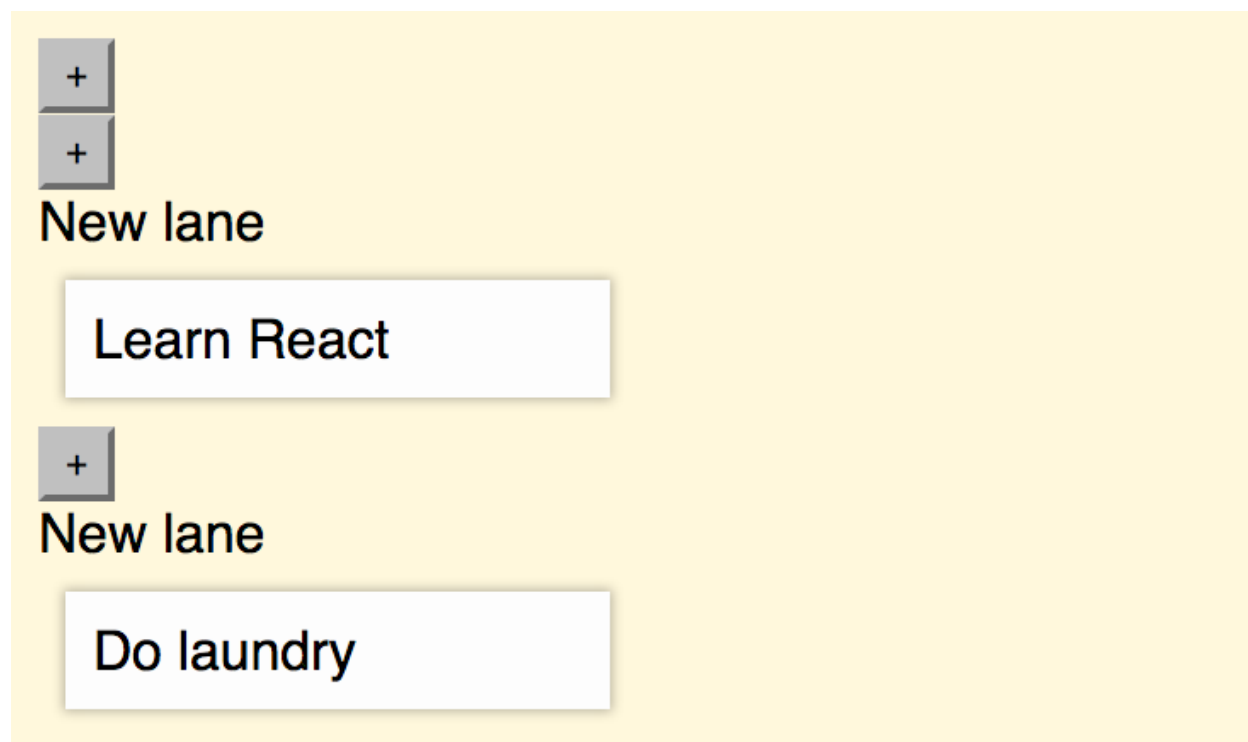
    <div {...props}>
      <div className="lane-header">
        <div className="lane-add-note">
          <button onClick={addNote}>+</button>
        </div>
        <div className="lane-name">{lane.name}</div>
      </div>
      <Notes
        notes={notes}
        notes={selectNotesByIds(notes, lane.notes)}
        onNoteClick={activateNoteEdit}
        onEdit={editNote}
        onDelete={deleteNote} />
    </div>
  );
};

function selectNotesByIds(allNotes, noteIds = []) {
  // `reduce` is a powerful method that allows us to
  // fold data. You can implement `filter` and `map`
  // through it. Here we are using it to concatenate
  // notes matching to the ids.
  return noteIds.reduce((notes, id) =>
    // Concatenate possible matching ids to the result
    notes.concat(
      allNotes.filter(note => note.id === id)
    ), []);
}

export default connect(
  ({notes}) => ({
    notes
  }), {
    NoteActions
    NoteActions,
    LaneActions
  }
)(Lane)

```

If you try using the application now, you should see that each lane is able to maintain its own notes:



#### Separate notes

The current structure allows us to keep singleton stores and a flat data structure. Dealing with references is a little awkward, but that's consistent with the Flux architecture. You can see the same theme in the [Redux implementation](#)<sup>3</sup>. The [MobX one](#)<sup>4</sup> avoids the problem altogether given we can use proper references there.



`selectNotesByIds` could have been written in terms of `map` and `find`. In that case you would have ended up with `noteIds.map(id => allNotes.find(note => note.id === id))`. You would need to polyfill `find` in this case to support older browsers, though.



Normalizing the data would have made `selectNotesByIds` trivial. If you are using a solution like Redux, normalization can make operations like this easy.

## 11.5 Extracting LaneHeader from Lane

Lane is starting to feel like a big component. This is a good chance to split it up to keep our application easier to maintain. Especially the lane header feels like a component of its own. To get started, define `LaneHeader` based on the current code like this:

<sup>3</sup><https://github.com/survivejs-demos/redux-demo>

<sup>4</sup><https://github.com/survivejs-demos/mobx-demo>

**app/components/LaneHeader.jsx**

```

import React from 'react';
import uuid from 'uuid';
import connect from '../libs/connect';
import NoteActions from '../actions/NoteActions';
import LaneActions from '../actions/LaneActions';

export default connect(() => ({}), {
  NoteActions,
  LaneActions
})(({lane, LaneActions, NoteActions, ...props}) => {
  const addNote = e => {
    e.stopPropagation();

    const noteId = uuid.v4();

    NoteActions.create({
      id: noteId,
      task: 'New task'
    });
    LaneActions.attachToLane({
      laneId: lane.id,
      noteId
    });
  };

  return (
    <div className="lane-header" {...props}>
      <div className="lane-add-note">
        <button onClick={addNote}></button>
      </div>
      <div className="lane-name">{lane.name}</div>
    </div>
  );
})

```

We also need to connect the extracted component with Lane:

**app/components/Lane.jsx**

```

import React from 'react';
import uuid from 'uuid';
import connect from '../libs/connect';
import NoteActions from '../actions/NoteActions';
import LaneActions from '../actions/LaneActions';
import Notes from './Notes';
import LaneHeader from './LaneHeader';

const Lane = ({
  lane, notes, LaneActions, NoteActions, ...props
}) => {
  const editNote = (id, task) => {
    NoteActions.update({id, task, editing: false});
  };
const addNote = e => {
  e.stopPropagation();

const noteId = uuid.v4();

NoteActions.create({
  id: noteId,
  task: 'New task'
});
LaneActions.attachToLane({
  laneId: lane.id,
  noteId
});
};
  const deleteNote = (noteId, e) => {
    e.stopPropagation();

    LaneActions.detachFromLane({
      laneId: lane.id,
      noteId
    });
    NoteActions.delete(noteId);
  };
  const activateNoteEdit = id => {
    NoteActions.update({id, editing: true});
  };

  return (

```



```

    <div {...props}>
      <div className="lane-header">
        <div className="lane-add-note">
          <button onClick={addNote}>+</button>
        </div>
      <div className="lane-name">{lane.name}</div>
    </div>
    <LaneHeader lane={lane} />
    <Notes
      notes={selectNotesByIds(notes, lane.notes)}
      onNoteClick={activateNoteEdit}
      onEdit={editNote}
      onDelete={deleteNote} />
  </div>
);
};

...

```

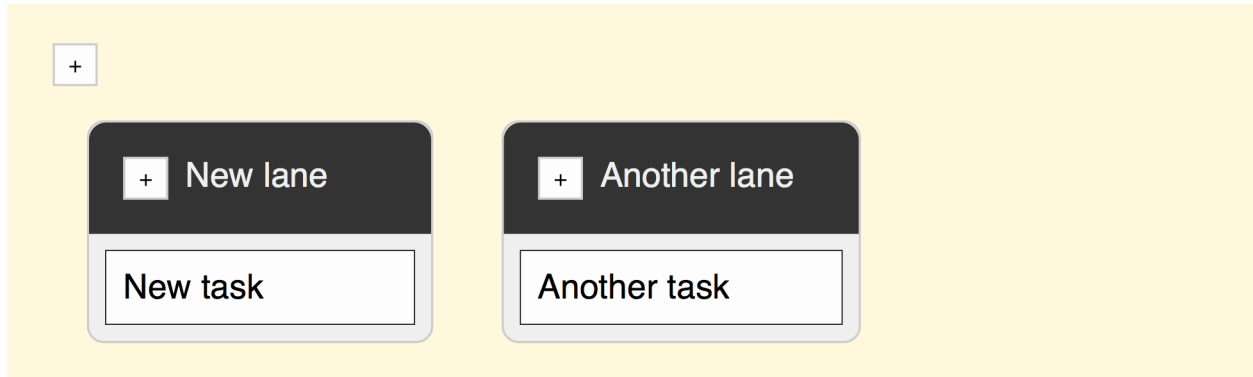
After these changes we have something that's a little easier to work with. It would have been possible to maintain all the related code in a single component. Often these are judgment calls as you realize there are nicer ways to split up your components. Sometimes the need for reuse or performance will force you to split.

## 11.6 Conclusion

We managed to solve the problem of handling data dependencies in this chapter. It is a problem that comes up often. Each data management solution provides a way of its own to deal with it. Flux based alternatives and Redux expect you to manage the references. Solutions like MobX have reference handling integrated. Data normalization can make these type of operations easier.

In the next chapter we will focus on adding missing functionality to the application. This means tackling editing lanes. We can also make the application look a little nicer again. Fortunately a lot of the logic has been developed already.

## 12. Editing Lanes



Kanban board

We still have work to do to turn this into a real Kanban as pictured above. The application is still missing some logic and styling. That's what we'll focus on here.

The `Editable` component we implemented earlier will come in handy. We can use it to make it possible to alter `Lane` names. The idea is exactly the same as for notes.

We should also make it possible to remove lanes. For that to work we'll need to add an UI control and attach logic to it. Again, it's a similar idea as earlier.

### 12.1 Implementing Editing for `Lane` names

To edit a `Lane` name, we need a little bit of logic and UI hooks. `Editable` can handle the UI part. Logic will take more work. To get started, tweak `LaneHeader` as follows:

**app/components/LaneHeader.jsx**

```
import React from 'react';
import uuid from 'uuid';
import connect from '../libs/connect';
import NoteActions from '../actions/NoteActions';
import LaneActions from '../actions/LaneActions';
import Editable from './Editable';

export default connect(() => ({}), {
  NoteActions,
```

```

    LaneActions
  })(({lane, LaneActions, NoteActions, ...props}) => {
    const addNote = e => {
      ...
    };
    const activateLaneEdit = () => {
      LaneActions.update({
        id: lane.id,
        editing: true
      });
    };
    const editName = name => {
      LaneActions.update({
        id: lane.id,
        name,
        editing: false
      });
    };
    return (
      <div className="lane-header" {...props}>
        <div className="lane-header" onClick={activateLaneEdit} {...props}>
          <div className="lane-add-note">
            <button onClick={addNote}>+</button>
          </div>
          <div className="lane-name">{lane.name}</div>
          <Editable className="lane-name" editing={lane.editing}
            value={lane.name} onEdit={editName} />
        </div>
      </div>
    );
  })

```

The user interface should look exactly the same after this change. We still need to implement `LaneActions.update` to make our setup work.

Just like before, we have to tweak two places, the action definition and `LaneStore`. Here's the action part:

**app/actions/LaneActions.js**

```
import alt from '../libs/alt';

export default alt.generateActions(
  'create', 'update', 'attachToLane', 'detachFromLane'
);
```

To add the missing logic, tweak LaneStore like this. It's the same idea as for NoteStore:

**app/stores/LaneStore.js**

```
import LaneActions from '../actions/LaneActions';

export default class LaneStore {
  constructor() {
    this.bindActions(LaneActions);

    this.lanes = [];
  }
  create(lane) {
    ...
  }
  update(updatedLane) {
    this.setState({
      lanes: this.lanes.map(lane => {
        if(lane.id === updatedLane.id) {
          return Object.assign({}, lane, updatedLane);
        }

        return lane;
      })
    });
  }
  ...
}
```

After these changes you should be able to edit lane names. Lane deletion is a good feature to sort out next.

## 12.2 Implementing Lane Deletion

Deleting lanes is a similar problem. We need to extend the user interface, add an action, and attach logic associated to it.

The user interface is a natural place to start. Often it's a good idea to add some `console.logs` in place to make sure your event handlers get triggered as you expect. It would be even better to write tests for those. That way you'll end up with a runnable specification. Here's how to add a stub for deleting lanes:

#### **app/components/LaneHeader.jsx**

```
...

export default connect(() => ({}), {
  NoteActions,
  LaneActions
})(({lane, LaneActions, NoteActions, ...props}) => {
  ...
  const deleteLane = e => {
    // Avoid bubbling to edit
    e.stopPropagation();

    LaneActions.delete(lane.id);
  };

  return (
    <div className="lane-header" onClick={activateLaneEdit} {...props}>
      <div className="lane-add-note">
        <button onClick={addNote}>+</button>
      </div>
      <Editable className="lane-name" editing={lane.editing}
        value={lane.name} onEdit={editName} />
      <div className="lane-delete">
        <button onClick={deleteLane}>x</button>
      </div>
    </div>
  );
});
```

Again, we need to expand our action definition:

#### **app/actions/LaneActions.js**

```
import alt from '../libs/alt';

export default alt.generateActions(
  'create', 'update', 'delete', 'attachToLane', 'detachFromLane'
);
```

And to finalize the implementation, let's add logic:

**app/stores/LaneStore.js**

```
import LaneActions from '../actions/LaneActions';

export default class LaneStore {
  constructor() {
    this.bindActions(LaneActions);

    this.lanes = [];
  }
  create(lane) {
    ...
  }
  update(updatedLane) {
    ...
  }
  delete(id) {
    this.setState({
      lanes: this.lanes.filter(lane => lane.id !== id)
    });
  }
  ...
}
```

Assuming everything went correctly, you should be able to delete entire lanes now.

The current implementation contains one gotcha. Even though we are removing references to lanes, the notes they point remain. This is something that could be turned into a rubbish bin feature. Or we could perform cleanup as well. For the purposes of this application, we can leave the situation as is. It is something good to be aware of, though.

## 12.3 Styling Kanban Board

As we added Lanes to the application, the styling went a bit off. Adjust as follows to make it a little nicer:

**app/main.css**

```
body {
  background-color: cornsilk;

  font-family: sans-serif;
}

.add-note {
  background-color: #fdfdfd;

  border: 1px solid #ccc;
}

.lane {
  display: inline-block;

  margin: 1em;

  background-color: #efefef;
  border: 1px solid #ccc;
  border-radius: 0.5em;

  min-width: 10em;
  vertical-align: top;
}

.lane-header {
  overflow: auto;

  padding: 1em;

  color: #efefef;
  background-color: #333;

  border-top-left-radius: 0.5em;
  border-top-right-radius: 0.5em;
}

.lane-name {
  float: left;
}

.lane-add-note {
```

```
float: left;

margin-right: 0.5em;
}

.lane-delete {
  float: right;

  margin-left: 0.5em;

  visibility: hidden;
}
.lane-header:hover .lane-delete {
  visibility: visible;
}

.add-lane, .lane-add-note button {
  cursor: pointer;

  background-color: #fdfdfd;
  border: 1px solid #ccc;
}

.lane-delete button {
  padding: 0;

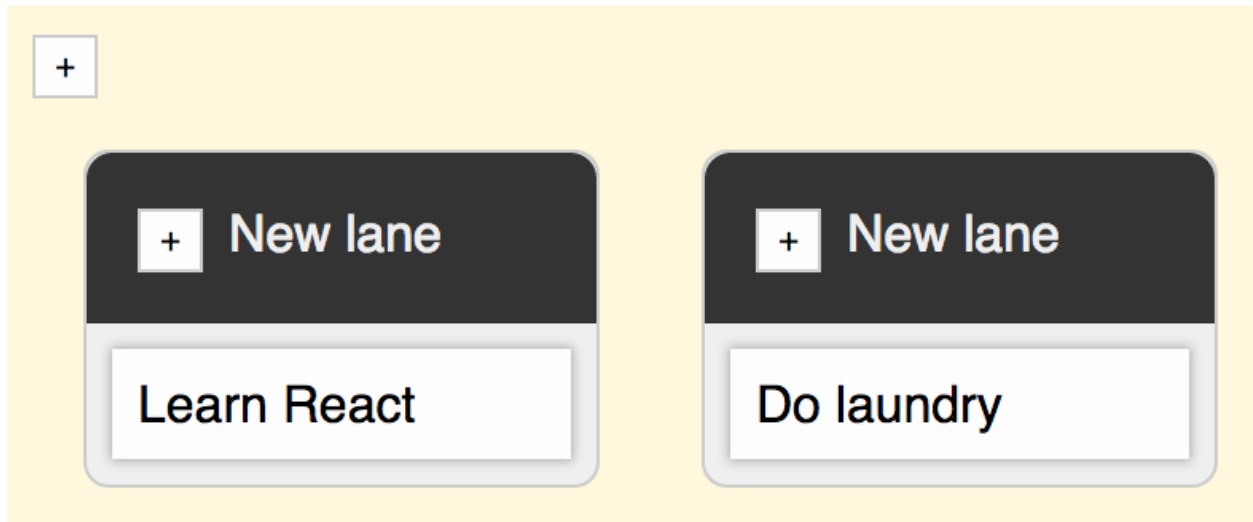
  cursor: pointer;

  color: white;
  background-color: rgba(0, 0, 0, 0);
  border: 0;
}

...
```

You should end up with a result like this:





Styled Kanban

As this is a small project, we can leave the CSS in a single file like this. In case it starts growing, consider separating it to multiple files. One way to do this is to extract CSS per component and then refer to it there (e.g., `require( './lane.css' )` at `Lane.jsx`). You could even consider using **CSS Modules** to make your CSS default to local scope. See the *Styling React* chapter for further ideas.

## 12.4 Conclusion

Even though our application is starting to look good and has basic functionality in it, it's still missing perhaps the most vital feature. We still cannot move notes between lanes. This is something we will resolve in the next chapter as we implement drag and drop.

# 13. Implementing Drag and Drop

Our Kanban application is almost usable now. It looks alright and there's basic functionality in place. In this chapter, we will integrate drag and drop functionality to it as we set up [React DnD](https://gaearon.github.io/react-dnd/)<sup>1</sup>.

After this chapter, you should be able to sort notes within a lane and drag them from one lane to another. Although this sounds simple, there is quite a bit of work to do as we need to annotate our components the right way and develop the logic needed.

## 13.1 Setting Up React DnD

As the first step, we need to connect React DnD with our project. We are going to use the HTML5 Drag and Drop based back-end. There are specific back-ends for testing and [touch](https://github.com/yahoo/react-dnd-touch-backend)<sup>2</sup>.

In order to set it up, we need to use the `DragDropContext` decorator and provide the HTML5 back-end to it. To avoid unnecessary wrapping, I'll use `Redux compose` to keep the code neater and more readable:

**app/components/App.jsx**

```
import React from 'react';
import uuid from 'uuid';
import {compose} from 'redux';
import {DragDropContext} from 'react-dnd';
import HTML5Backend from 'react-dnd-html5-backend';
import connect from '../libs/connect';
import Lanes from './Lanes';
import LaneActions from '../actions/LaneActions';

const App = ({LaneActions, lanes}) => {
  const addLane = () => {
    LaneActions.create({
      id: uuid.v4(),
      name: 'New lane'
    });
  };
};
```

---

<sup>1</sup><https://gaearon.github.io/react-dnd/>

<sup>2</sup><https://github.com/yahoo/react-dnd-touch-backend>

```

    return (
      <div>
        <button className="add-lane" onClick={addLane}>></button>
        <Lanes lanes={lanes} />
      </div>
    );
  };

export default connect(({lanes}) => ({
  lanes
}), {
  LaneActions
})(App)
export default compose(
  DragDropContext(HTML5Backend),
  connect(({lanes}) => ({
    lanes
  })), {
    LaneActions
  })
)(App)

```

After this change, the application should look exactly the same as before. We are ready to add some sweet functionality to it now.

## 13.2 Allowing Notes to Be Dragged

Allowing notes to be dragged is a good first step. Before that, we need to set up a constant so that React DnD can tell different kind of draggables apart. Set up a file for tracking Note as follows:

**app/constants/itemTypes.js**

```

export default {
  NOTE: 'note'
};

```

This definition can be expanded later as we add new types, such as LANE, to the system.

Next, we need to tell our Note that it's possible to drag it. This can be achieved using the DragSource annotation. Replace Note with the following implementation:

**app/components/Note.jsx**

```
import React from 'react';
import {DragSource} from 'react-dnd';
import ItemTypes from '../constants/itemTypes';

const Note = ({
  connectDragSource, children, ...props
}) => {
  return connectDragSource(
    <div {...props}>
      {children}
    </div>
  );
};

const noteSource = {
  beginDrag(props) {
    console.log('begin dragging note', props);

    return {};
  }
};

export default DragSource(ItemTypes.NOTE, noteSource, connect => ({
  connectDragSource: connect.dragSource()
}))(Note)
```

If you try to drag a Note now, you should see something like this at the browser console:

```
begin dragging note Object {className: "note", children: Array[2]}
```

Just being able to drag notes isn't enough. We need to annotate them so that they can accept dropping. Eventually this will allow us to swap them as we can trigger logic when we are trying to drop a note on top of another.



In case we wanted to implement dragging based on a handle, we could apply `connectDragSource` only to a specific part of a Note.



Note that React DnD doesn't support hot loading perfectly. You may need to refresh the browser to see the log messages you expect!

## 13.3 Allowing Notes to Detect Hovered Notes

Annotating notes so that they can notice that another note is being hovered on top of them is a similar process. In this case we'll have to use a `DropTarget` annotation:

**app/components/Note.jsx**

```
import React from 'react';
import {DragSource} from 'react-dnd';
import {compose} from 'redux';
import {DragSource, DropTarget} from 'react-dnd';
import ItemTypes from '../constants/itemTypes';

const Note = ({
  connectDragSource, children, ...props
  connectDragSource, connectDropTarget,
  children, ...props
}) => {
  return connectDragSource(
    return compose(connectDragSource, connectDropTarget)(
      <div {...props}>
        {children}
      </div>
    );
  };
};

const noteSource = {
  beginDrag(props) {
    console.log('begin dragging note', props);

    return {};
  }
};

const noteTarget = {
  hover(targetProps, monitor) {
    const sourceProps = monitor.getItem();

    console.log('dragging note', sourceProps, targetProps);
  }
};

export default DragSource(ItemTypes.NOTE, noteSource, connect => ({
```

```

—connectDragSource:—connect.dragSource({
  })(Note)
export default compose(
  DragSource(ItemTypes.NOTE, noteSource, connect => ({
    connectDragSource: connect.dragSource()
  })),
  DropTarget(ItemTypes.NOTE, noteTarget, connect => ({
    connectDropTarget: connect.dropTarget()
  }))
)(Note)

```

If you try hovering a dragged note on top of another now, you should see messages like this at the console:

```
dragging note Object {} Object {className: "note", children: Array[2]}
```

Both decorators give us access to the Note props. In this case, we are using `monitor.getItem()` to access them at `noteTarget`. This is the key to making this to work properly.

## 13.4 Developing onMove API for Notes

Now, that we can move notes around, we can start to define logic. The following steps are needed:

1. Capture Note id on `beginDrag`.
2. Capture target Note id on `hover`.
3. Trigger `onMove` callback on `hover` so that we can deal with the logic elsewhere. `LaneStore` would be the ideal place for that.

Based on the idea above we can see we should pass `id` to a Note through a prop. We also need to set up a `onMove` callback, define `LaneActions.move`, and `LaneStore.move` stub.

### Accepting id and onMove at Note

We can accept `id` and `onMove` props at Note like below. There is an extra check at `noteTarget` as we don't need `trigger hover` in case we are hovering on top of the Note itself:

`app/components/Note.jsx`

```

...

const Note = ({
  connectDragSource, connectDropTarget,
children, ...props
  oMove, id, children, ...props
}) => {
  return compose(connectDragSource, connectDropTarget)(
    <div {...props}>
      {children}
    </div>
  );
};

const noteSource = {
  beginDrag(props) {
    console.log('begin dragging note', props);

    return {};
  }
};
const noteSource = {
  beginDrag(props) {
    return {
      id: props.id
    };
  }
};

const noteTarget = {
  hover(targetProps, monitor) {
    const sourceProps = monitor.getItem();

    console.log('dragging note', sourceProps, targetProps);
  }
};
const noteTarget = {
  hover(targetProps, monitor) {
    const targetId = targetProps.id;
    const sourceProps = monitor.getItem();
    const sourceId = sourceProps.id;

```

```

    if(sourceId !== targetId) {
      targetProps.onMove({sourceId, targetId});
    }
  }
};

...

```

Having these props isn't useful if we don't pass anything to them at Notes. That's our next step.

## Passing id and onMove from Notes

Passing a note id and onMove is simple enough:

**app/components/Notes.jsx**

```

import React from 'react';
import Note from './Note';
import Editable from './Editable';

export default ({
  notes,
  onNoteClick=() => {}, onEdit=() => {}, onDelete=() => {}
}) => (
  <ul className="notes">{notes.map(({id, editing, task}) =>
    <li key={id}>
      <Note className="note" onClick={onNoteClick.bind(null, id)}>
      <Note className="note" id={id}
        onClick={onNoteClick.bind(null, id)}
        onMove={({sourceId, targetId}) =>
          console.log('moving from', sourceId, 'to', targetId)}>
        <Editable
          className="editable"
          editing={editing}
          value={task}
          onEdit={onEdit.bind(null, id)} />
        <button
          className="delete"
          onClick={onDelete.bind(null, id)}>x</button>
      </Note>
    </li>
  )}</ul>
)

```



If you hover a note on top of another, you should see console messages like this:

```
moving from 3310916b-5b59-40e6-8a98-370f9c194e16 to 939fb627-1d56-4b57-89ea-0420\
7dbfb405
```

## 13.5 Adding Action and Store Method for Moving

The logic of drag and drop goes as follows. Suppose we have a lane containing notes A, B, C. In case we move A below C we should end up with B, C, A. In case we have another list, say D, E, F, and move A to the beginning of it, we should end up with B, C and A, D, E, F.

In our case, we'll get some extra complexity due to lane to lane dragging. When we move a Note, we know its original position and the intended target position. Lane knows what Notes belong to it by id. We are going to need some way to tell LaneStore that it should perform the logic over the given notes. A good starting point is to define `LaneActions.move`:

**app/actions/LaneActions.js**

```
import alt from '../libs/alt';

export default alt.generateActions(
  'create', 'update', 'delete',
  'attachToLane', 'detachFromLane',
  'move'
);
```

We should connect this action with the `onMove` hook we just defined:

**app/components/Notes.jsx**

```
import React from 'react';
import Note from './Note';
import Editable from './Editable';
import LaneActions from '../actions/LaneActions';

export default ({
  notes,
  onNoteClick=() => {}, onEdit=() => {}, onDelete=() => {}
}) => (
  <ul className="notes">{notes.map(({id, editing, task}) =>
    <li key={id}>
      <Note className="note" id={id}>
```

```

      onClick={onNoteClick.bind(null, id)}
      onMove={({sourceId, targetId}) =>
      console.log('moving from', sourceId, 'to', targetId)}>
      onMove={LaneActions.move}>
      <Editable
        className="editable"
        editing={editing}
        value={task}
        onEdit={onEdit.bind(null, id)} />
      <button
        className="delete"
        onClick={onDelete.bind(null, id)}>x</button>
    </Note>
  </li>
)}</ul>
)

```



It could be a good idea to refactor `onMove` as a prop to make the system more flexible. In our implementation the `Notes` component is coupled with `LaneActions`. This isn't particularly nice if you want to use it in some other context.

We should also define a stub at `LaneStore` to see that we wired it up correctly:

**app/stores/LaneStore.js**

```

import LaneActions from '../actions/LaneActions';

export default class LaneStore {
  ...
  detachFromLane({laneId, noteId}) {
    ...
  }
  move({sourceId, targetId}) {
    console.log(`source: ${sourceId}, target: ${targetId}`);
  }
}

```

You should see the same log messages as earlier.

Next, we'll need to add some logic to make this work. We can use the logic outlined above here. We have two cases to worry about: moving within a lane itself and moving from lane to another.

## 13.6 Implementing Note Drag and Drop Logic

Moving within a lane itself is complicated. When you are operating based on ids and perform operations one at a time, you'll need to take possible index alterations into account. As a result, I'm using update [immutability helper](https://facebook.github.io/react/docs/update.html)<sup>3</sup> from React as that solves the problem in one pass.

It is possible to solve the lane to lane case using [splice](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/splice)<sup>4</sup>. First, we splice out the source note, and then we splice it to the target lane. Again, update could work here, but I didn't see much point in that given splice is nice and simple. The code below illustrates a mutation based solution:

app/stores/LaneStore.js

```
import update from 'react-addons-update';
import LaneActions from '../actions/LaneActions';

export default class LaneStore {
  ...
  move({sourceId, targetId}) {
    console.log(`source: ${sourceId}, target: ${targetId}`);
  }
  move({sourceId, targetId}) {
    const lanes = this.lanes;
    const sourceLane = lanes.filter(lane => lane.notes.includes(sourceId))[0];
    const targetLane = lanes.filter(lane => lane.notes.includes(targetId))[0];
    const sourceNoteIndex = sourceLane.notes.indexOf(sourceId);
    const targetNoteIndex = targetLane.notes.indexOf(targetId);

    if(sourceLane === targetLane) {
      // move at once to avoid complications
      sourceLane.notes = update(sourceLane.notes, {
        $splice: [
          [sourceNoteIndex, 1],
          [targetNoteIndex, 0, sourceId]
        ]
      });
    }
    else {
      // get rid of the source
      sourceLane.notes.splice(sourceNoteIndex, 1);

      // and move it to target
```

<sup>3</sup><https://facebook.github.io/react/docs/update.html>

<sup>4</sup>[https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global\\_Objects/Array/splice](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/splice)

```

        targetLane.notes.splice(targetNoteIndex, 0, sourceId);
    }

    this.setState({lanes});
  }
}

```

If you try out the application now, you can actually drag notes around and it should behave as you expect. Dragging to empty lanes doesn't work, though, and the presentation could be better.

It would be nicer if we indicated the dragged note's location more clearly. We can do this by hiding the dragged note from the list. React DnD provides us the hooks we need for this purpose.

## Indicating Where to Move

React DnD provides a feature known as state monitors. Through it we can use `monitor.isDragging()` and `monitor.isOver()` to detect which Note we are currently dragging. It can be set up as follows:

**app/components/Note.jsx**

```

import React from 'react';
import {compose} from 'redux';
import {DragSource, DropTarget} from 'react-dnd';
import ItemTypes from '../constants/itemTypes';

const Note = ({
  connectDragSource, connectDropTarget,
  onMove, id, children, ...props
  connectDragSource, connectDropTarget, isDragging,
  isOver, onMove, id, children, ...props
}) => {
  return compose(connectDragSource, connectDropTarget)(
    <div {...props}>
    {children}
    </div>
    <div style={{
      opacity: isDragging || isOver ? 0 : 1
    }} {...props}>{children}</div>
  );
};

...

```

```
export default compose(
  DragSource(ItemTypes.NOTE, noteSource, connect => ({
  connectDragSource: connect.dragSource(),
  })),
  DropTarget(ItemTypes.NOTE, noteTarget, connect => ({
  connectDropTarget: connect.dropTarget(),
  })),
  DragSource(ItemTypes.NOTE, noteSource, (connect, monitor) => ({
    connectDragSource: connect.dragSource(),
    isDragging: monitor.isDragging()
  })),
  DropTarget(ItemTypes.NOTE, noteTarget, (connect, monitor) => ({
    connectDropTarget: connect.dropTarget(),
    isOver: monitor.isOver()
  }))
)(Note)
```

If you drag a note within a lane, the dragged note should be shown as blank.

There is one little problem in our system. We cannot drag notes to an empty lane yet.

## 13.7 Dragging Notes to Empty Lanes

To drag notes to empty lanes, we should allow them to receive notes. Just as above, we can set up DropTarget based logic for this. First, we need to capture the drag on Lane:

**app/components/Lane.jsx**

```
import React from 'react';
import {compose} from 'redux';
import {DropTarget} from 'react-dnd';
import ItemTypes from '../constants/itemTypes';
import connect from '../libs/connect';
import NoteActions from '../actions/NoteActions';
import LaneActions from '../actions/LaneActions';
import Notes from './Notes';
import LaneHeader from './LaneHeader';

const Lane = ({
  lane, notes, LaneActions, NoteActions, ...props
  connectDropTarget, lane, notes, LaneActions, NoteActions, ...props
}) => {
```

```

...

return (
  return connectDropTarget(
    ...
  );
};

function selectNotesByIds(allNotes, noteIds = []) {
  ...
}

const noteTarget = {
  hover(targetProps, monitor) {
    const sourceProps = monitor.getItem();
    const sourceId = sourceProps.id;

    // If the target lane doesn't have notes,
    // attach the note to it.
    //
    // `attachToLane` performs necessarily
    // cleanup by default and it guarantees
    // a note can belong only to a single lane
    // at a time.
    if(!targetProps.lane.notes.length) {
      LaneActions.attachToLane({
        laneId: targetProps.lane.id,
        noteId: sourceId
      });
    }
  }
};

export default connect(
  ({notes}) => ({
    notes
  }), {
    NoteActions,
    LaneActions
  }
)(Lane)
export default compose(

```

```

DropTarget(ItemTypes.NOTE, noteTarget, connect => ({
  connectDropTarget: connect.dropTarget()
})),
connect(({notes}) => ({
  notes
})), {
  NoteActions,
  LaneActions
})
)(Lane)

```

After attaching this logic, you should be able to drag notes to empty lanes.

Our current implementation of `attachToLane` does a lot of the hard work for us. If it didn't guarantee that a note can belong only to a single lane at a time, we would need to adjust our logic. It's good to have these sort of invariants within the state management system.

## Fixing Editing Behavior During Dragging

The current implementation has a small glitch. If you edit a note, you can still drag it around while it's being edited. This isn't ideal as it overrides the default behavior most people are used to. You cannot for instance double-click on an input to select all the text.

Fortunately, this is simple to fix. We'll need to use the editing state per each Note to adjust its behavior. First we need to pass editing state to an individual Note:

**app/components/Notes.jsx**

```

import React from 'react';
import Note from './Note';
import Editable from './Editable';
import LaneActions from '../actions/LaneActions';

export default ({
  notes,
  onNoteClick=() => {}, onEdit=() => {}, onDelete=() => {}
}) => (
  <ul className="notes">{notes.map(({id, editing, task}) =>
    <li key={id}>
      <Note className="note" id={id}
        editing={editing}
        onClick={onNoteClick.bind(null, id)}
        onMove={LaneActions.move}>
        <Editable

```

```

        className="editable"
        editing={editing}
        value={task}
        onEdit={onEdit.bind(null, id)} />
      <button
        className="delete"
        onClick={onDelete.bind(null, id)}>x</button>
    </Note>
  </li>
}</ul>
)

```

Next we need to take this into account while rendering:

#### app/components/Note.jsx

```

import React from 'react';
import {compose} from 'redux';
import {DragSource, DropTarget} from 'react-dnd';
import ItemTypes from '../constants/itemTypes';

const Note = ({
  connectDragSource, connectDropTarget, isDragging,
isOver, onMove, id, children, ...props
  isOver, onMove, id, editing, children, ...props
}) => {
  // Pass through if we are editing
  const dragSource = editing ? a => a : connectDragSource;

return compose(connectDragSource, connectDropTarget)(
  return compose(dragSource, connectDropTarget)(
    <div style={{
      opacity: isDragging || isOver ? 0 : 1
    }} {...props}>{children}</div>
  );
};

...

```

This small change gives us the behavior we want. If you try to edit a note now, the input should work as you might expect it to behave normally.



Design-wise it was a good idea to keep editing state outside of `Editable`. If we hadn't done that, implementing this change would have been a lot harder as we would have had to extract the state outside of the component.

Now we have a Kanban table that is actually useful! We can create new lanes and notes, and edit and remove them. In addition we can move notes around. Mission accomplished!

## 13.8 Conclusion

In this chapter, you saw how to implement drag and drop for our little application. You can model sorting for lanes using the same technique. First, you mark the lanes to be draggable and droppable, then you sort out their ids, and finally, you'll add some logic to make it all work together. It should be considerably simpler than what we did with notes.

I encourage you to expand the application. The current implementation should work just as a starting point for something greater. Besides extending the DnD implementation, you can try adding more data to the system. You could also do something to the visual outlook. One option would be to try out various styling approaches discussed at the *Styling React* chapter.

To make it harder to break the application during development, you can also implement tests as discussed at *Testing React*. *Typing with React* discussed yet more ways to harden your code. Learning these approaches can be worthwhile. Sometimes it may be worth your while to design your applications test first. It is a valuable approach as it allows you to document your assumptions as you go.

## III Advanced Techniques

There are a variety of advanced React techniques that are good to be aware of. By testing and typing your code well, you can make it more robust against change. It will be easier to develop if you have the right scaffolding in place supporting your application.

Styling React is a complicated topic itself. There are multiple ways to achieve that and there's no clear consensus on what is the correct way in the context of React. I will provide you a good idea of the current situation.

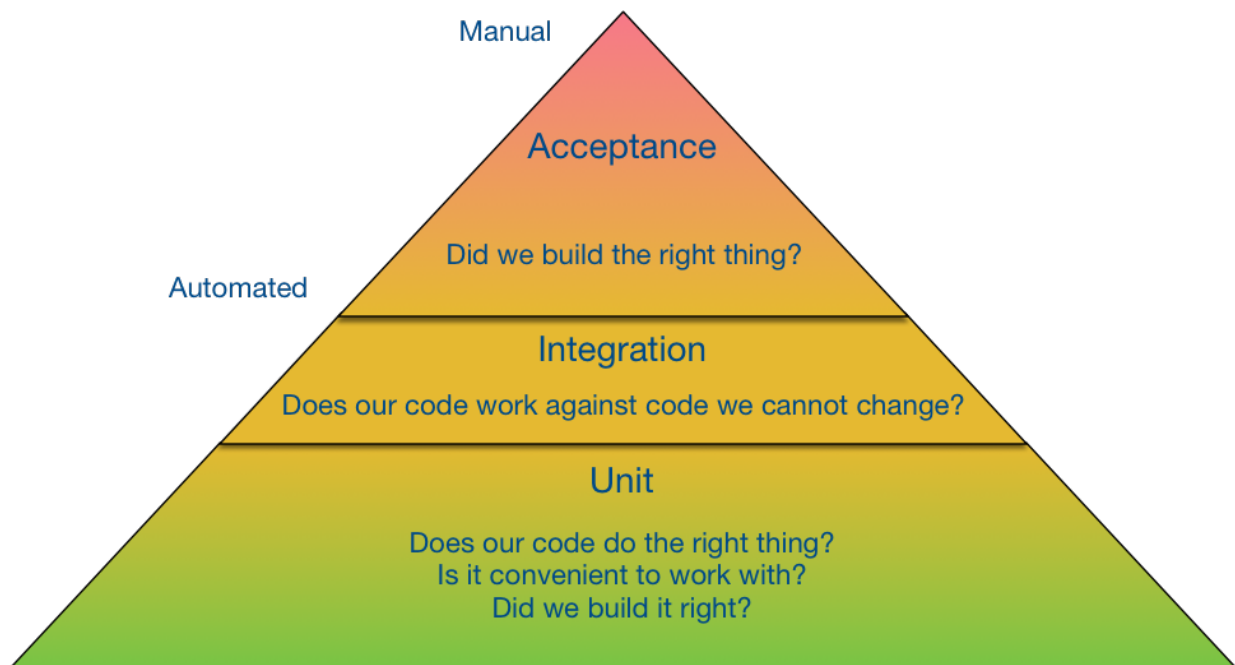
As there's no one right way to structure React projects, I'll also provide some idea on how to do that. It's better to be pragmatic and figure out a structure that's the right one for you.

# 14. Testing React

Testing allows us to make sure everything works as we expect. It provides reassurance when making changes to our code. Even a small change could break something crucial, and without tests we might not realize the mistake until it has been deployed into production.

It is still possible to break things, but tests allow us to catch mistakes early in the development cycle. Especially as you codebase grows, the value of tests goes up. You can consider them as a safety net against regressions. Even if regressions are found, you can develop tests to avoid them again. Test everything you don't want to break.

## 14.1 Levels of Testing



Testing pyramid

Levels of testing can be characterized using the concept of the “testing pyramid” popularized by Mike Cohn. He splits testing into three levels: unit, service, and user interface. He posits that you should have unit test the most, service level (integration) the next, and on top of it all there should be user interface level tests.

Each of these levels provides us information about the way the system behaves. They provide us confidence in that the code works the way we imagine it should. On a high level, we can assert that

the application follows its specification. On a low level, we can assert that some particular function operates the way we mean it to operate.

When studying a new system, testing can be used in an exploratory manner. The same idea is useful for debugging. We'll make a series of assertions to help us isolate the problem and fix it.

There are various techniques we can use to cover the testing pyramid. This is in no way an exhaustive list. It's more about giving you some idea of what is out there. We'll use a couple of these against our project later in this chapter.

## Unit Testing

**Unit testing** is all about testing a piece of code — a unit — in isolation. We can, for instance, perform unit tests against a single function to assert the way it behaves. The definition of a unit can be larger than this, though. At some point we'll arrive to the realm of *integration testing*. By doing that, we want to assert that parts of a system work together as they should.

## Code Coverage

In order to know what portion of the code is covered by unit tests, there are **code coverage** tools for figuring this out. [Istanbul](https://gotwarlost.github.io/istanbul/)<sup>1</sup> is a good alternative for JavaScript. It provides an HTML report to study.

Even though code coverage helps us to understand which portions of the code we are testing, it doesn't tell us anything about the quality of the tests. It just tells that we have hit some particular branches of the code with them.



[Blanket](http://blanketjs.org/)<sup>2</sup> is a valid alternative for Istanbul.



There are plugins that allow you to get coverage information within your editor. Examples: [Atom lcov-info](https://atom.io/packages/lcov-info)<sup>3</sup>, [SublimeJSCoverage](https://github.com/genbit/SublimeJSCoverage)<sup>4</sup>.

## TDD and BDD

Sometimes, it is handy to develop the tests first or in tandem with the code. This sort of **Test Driven Development** (TDD) is a popular technique in the industry. On a higher level, you can use **Behavior Driven Development** (BDD).

---

<sup>1</sup><https://gotwarlost.github.io/istanbul/>

<sup>2</sup><http://blanketjs.org/>

<sup>3</sup><https://atom.io/packages/lcov-info>

<sup>4</sup><https://github.com/genbit/SublimeJSCoverage>

Sometimes, I use **README Driven Development** and write the project README first to guide the component development. This forces me to think about the component from the user point of view and can help me reach a better API with less iteration. Ideally it would be possible to run the code at the README to assert that the code works.

BDD focuses on describing specifications on the level of the business without much technical knowledge. They provide a way for non-technical people to describe application behavior in a fluent syntax. Programmers can then perform lower level testing based on this functional specification.

TDD is a technique that can be described in three simple steps:

1. Write a failing test.
2. Create minimal implementation.
3. Repeat.

Once you have gone far enough, you can *refactor* your code with confidence. This is particularly useful if you are working tired or happen not to know the codebase well.

Testing doesn't come without its cost. Now you have two codebases to maintain. What if your tests aren't maintained well, or worse, are faulty? Even though testing comes with some cost, it is extremely valuable especially as your project grows. It keeps the complexity maintainable and allows you to proceed with confidence.

## Acceptance Testing

Unit level tests look at the system from a technical perspective. *Acceptance tests* are at the other end of the spectrum. They are more concerned about how does the system look for the user. Here we are exercising every piece that lies below the user interface. Integration tests fit between these two ends.

Acceptance tests allow us to measure qualitative concerns. We can, for example, assert that certain elements are visible to the user. We can also have performance requirements and test against those.

Tools, such as [Selenium](http://www.seleniumhq.org/)<sup>5</sup>, allow us to automate this process and perform acceptance testing across various browsers. This can help us to discover user interface level issues our tests might miss otherwise. Sometimes, browsers behave in wildly different manners, and this in turn can cause interesting yet undesirable behavior.



A tool known as [CodeceptJS](http://codecept.io/)<sup>6</sup> allows you to write acceptance testing using JavaScript while targeting Selenium and other testing backends. There are a lot of tools like this. The choice doesn't depend on React.

---

<sup>5</sup><http://www.seleniumhq.org/>

<sup>6</sup><http://codecept.io/>

## Property Based Testing

Beyond these there are techniques that are more specialized. For instance, **property based testing** allows us to check that certain invariants hold against the code. For example, when implementing a sorting algorithm for numbers, we know for sure that the result should be numerically ascending.

Property based testing tools generate tests based on these invariants. For instance, we could end up with automatic tests like this:

```
sort([-12324234242, 231, -0.43429, 1.7976931348623157e+302])
sort([1.72e+32])
sort([0, 0, 0])
sort([])
sort([1])
```

There can be as many of these tests as we want. We can generate them in a pseudo-random way. This means we'll be able to replay the same tests if we want. This allows us to reproduce possible bugs we might find.

The biggest advantage of the approach is that it allows us to test against values and ranges we might not test otherwise. Computers are good at generating tests. The problem lies in figuring out good invariants to test against.



This type of testing is very popular in Haskell. Particularly, [QuickCheck](https://hackage.haskell.org/package/QuickCheck)<sup>7</sup> made the approach well-known and there are implementations for other languages as well. In JavaScript environment [JSVerify](https://jsverify.github.io/)<sup>8</sup> can be a good starting point.



If you want to check invariants during runtime while developing, a package known as [invariant](https://www.npmjs.com/package/invariant)<sup>9</sup> can come in handy. Facebook uses it for some extra safety with React and Flux.

## Mutation Testing

**Mutation testing** allows you to test your tests. Mutation testing frameworks will mutate your source code in various ways and see how your tests behave. It may reveal parts of code that you might be able to remove or simplify based on your tests and their coverage. It's not a particularly popular technique but it's good to be aware of it.

---

<sup>7</sup><https://hackage.haskell.org/package/QuickCheck>

<sup>8</sup><https://jsverify.github.io/>

<sup>9</sup><https://www.npmjs.com/package/invariant>

## 14.2 Writing Your First Test

The testing setup that comes with the project boilerplate has been adapted based on Cesar Andreu's [web-app](#)<sup>10</sup>. It relies on [Karma](#)<sup>11</sup> test runner and [Mocha](#)<sup>12</sup> test framework.

Karma will execute the tests against a browser. In this case it uses [PhantomJS](#)<sup>13</sup>, a popular Webkit based headless browser. You can also tell Karma to run against an actual browser available on your system. It is possible to write tests that don't depend on the DOM without a heavy setup like this.

You can use whatever assertion library you want with Mocha. [Chai](#)<sup>14</sup> is a popular alternative. In this case I'll use [Node.js assert](#)<sup>15</sup> given it's simple and familiar to many.



[Testem](#)<sup>16</sup> is a valid alternative to Karma. You'll likely find a few others as there's no lack of testing tools for JavaScript.



Facebook's [Jest](#)<sup>17</sup> is a popular alternative to Mocha. It is based on [Jasmine](#)<sup>18</sup>, another popular tool, and takes less setup than Mocha.



[rewire-webpack](#)<sup>19</sup> allows you to manipulate your module behavior to make unit testing easier. It uses [rewire](#)<sup>20</sup> internally. If you need to mock dependencies, this is a good way to go. An alternative way to use rewire is to go through [babel-plugin-rewire](#)<sup>21</sup>.

## 14.3 Understanding the Test Setup

The boilerplate includes a sample test and basic commands for executing them. It expects to find the tests below `tests/`. This isn't the only way to structure your tests but it's enough for a simple project like this. The sample test suite with a single unit test looks like this:

`tests/demo_test.js`

---

<sup>10</sup><https://github.com/cesarandreu/web-app>

<sup>11</sup><https://karma-runner.github.io>

<sup>12</sup><https://mochajs.org/>

<sup>13</sup><http://phantomjs.org/>

<sup>14</sup><http://chaijs.com/>

<sup>15</sup><https://nodejs.org/api/assert.html>

<sup>16</sup><https://github.com/airportyh/testem>

<sup>17</sup><https://facebook.github.io/jest/>

<sup>18</sup><https://jasmine.github.io/>

<sup>19</sup><https://github.com/jhnns/rewire-webpack>

<sup>20</sup><https://github.com/jhnns/rewire>

<sup>21</sup><https://www.npmjs.com/package/babel-plugin-rewire>

```
import assert from 'assert';

describe('add', function() {
  it('adds', function() {
    assert.equal(1 + 1, 2);
  });
});
```

## Running Once

If you trigger `npm test` now, you should see something like this:

```
> karma start
```

```
30 05 2016 15:06:50.521:INFO [karma]: Karma v0.13.22 server started at http://localhost:9876/
30 05 2016 15:06:50.531:INFO [launcher]: Starting browser PhantomJS
30 05 2016 15:06:51.360:INFO [PhantomJS 2.1.1 (Mac OS X 0.0.0)]: Connected on socket /#z7ED-FWAP81K9-IIAAAA with id 21009715
```

```
add
  ✓ adds
```

```
PhantomJS 2.1.1 (Mac OS X 0.0.0): Executed 1 of 1 SUCCESS (0.008 secs / 0.001 secs)
TOTAL: 1 SUCCESS
```

You should also find a coverage report below the coverage directory. You can check that out through the browser to see how well your application is covered by tests.

## Running in TDD Mode

The test setup includes a TDD mode. This means it's able to run the tests as you develop your code. You can enable it by using `npm run test:tdd`. Try making the test fail somehow to see it in action.

## Linting the Code

There's also a separate command for linting under `npm run test:lint`. The Webpack setup lints the code automatically for you, though, so you need to trigger it rarely yourself.



## 14.4 Testing Kanban Components

By looking at the components of our Kanban application, we can see that `Editable` has plenty of complexity. It would be a good idea to test that to lock down this logic.

`Note` is another interesting one. Even though it's a trivial component, it's not entirely trivial to test given it depends on React DnD. That takes some additional thought.

### The Testing Process

I'll be covering these two cases as understanding them will help you to implement tests for the remaining components should you want to. The idea is always the same:

1. Figure out what you want to test.
2. Render the component to test through React's `renderIntoDocument`.
3. Optionally manipulate the component somehow. You can, for instance, simulate user operations through React's API.
4. Assert some truth.

Ideally, your unit tests should test only one thing at a time. Keeping them simple is a good idea as that will help when you are debugging your code. As discussed earlier, unit tests won't prove absence of bugs. Instead, they prove that the code is correct for that specific test. This is what makes unit tests useful for debugging. You can use them to prove your assumptions.

To get started, we should make a test plan for `Editable` and get some testing done. Note that you can implement these tests using `npm run tdd` and type them as you go. Feel free to try to break things to get a better feel for it.

## 14.5 Testing `Editable`

### Test Plan for `Editable`

There are a couple of things to test in `Editable`:

- Does it render the given value correctly?
- Does it trigger `onEdit` callback on edit?

I am sure there are a couple of extra cases that would be good to test, but these will get us started and help us to understand how to write unit tests for React components.

## Editable Renders Value

In order to check that `Editable` renders a value, we'll need to:

1. Render the component while passing some value to it.
2. Find the value.
3. Check that the value is what we expect.

In terms of code it would look like this:

`tests/editable_test.jsx`

```
import React from 'react';
import assert from 'assert';
import Editable from '../app/components/Editable';

describe('Editable', function() {
  it('renders value', function() {
    const value = 'value';
    const component = new Editable({value});

    assert.equal(component.props.children, value);
  });
});
```

Given we are using a function based component, we can cheat a little here. Instead of going through React's test utilities, we can instantiate a component and then have a look at the resulting `props.children`. If that is something else than `value`, our test will fail.

As we can be sure that `Editable` can render a value passed to it, we can try something more complicated, namely entering the edit mode.

## Editable Triggers `onEdit`

As per our component definition, `onEdit` should get triggered after the user triggers `blur` event somehow. We can assert that it receives the input value it expects. This probably could be split up into two separate tests but this will do just fine.

The challenge here is that now we need to test against the DOM somehow. This is where the [Test Utilities](https://facebook.github.io/react/docs/test-utils.html)<sup>22</sup> come in. It's not an easy test to write. I've included it below in its entirety:

`tests/editable_test.jsx`

---

<sup>22</sup><https://facebook.github.io/react/docs/test-utils.html>

```
import React from 'react';
import {
  renderIntoDocument,
  findRenderedDOMComponentWithTag,
  Simulate
} from 'react-addons-test-utils';
import assert from 'assert';
import Editable from '../app/components/Editable';

describe('Editable', function() {
  it('renders value', function() {
    const value = 'value';
    const component = new Editable({value});

    assert.equal(component.props.children, value);
  });

  it('triggers onEdit through the DOM', function() {
    let triggered = false;
    const newValue = 'value';
    const onEdit = (val) => {
      triggered = true;
      assert.equal(val, newValue);
    };
    const component = renderIntoDocument(
      <Wrapper>
        <Editable editing={true} value={'value'} onEdit={onEdit} />
      </Wrapper>
    );

    const input = findRenderedDOMComponentWithTag(component, 'input');
    input.value = newValue;

    Simulate.blur(input);

    assert.equal(triggered, true);
  });
});

class Wrapper extends React.Component {
  render() {
    return <div>{this.props.children}</div>;
  }
}
```

```
    }  
  }  
});
```

There are a couple of important points:

- `findRenderedDOMComponentWithTag` is used to match against the input tag. It's the same idea as with classes. There's also a `scry` variant that works in a similar way but against tag names and returns an array of matches.
- Given `renderIntoDocument` won't work with function based components (no backing instance), we need to implement a small `Wrapper` to get around the issue.
- `Simulate.blur` simulates the user blurring (input loses focus). We expect it to trigger out logic.
- We use `triggered` flag and a callback to assert. An alternative would be to use a technique known as spying. A library known as [Sinon.js](http://sinonjs.org/)<sup>23</sup> contains a good implementation.

Compared to the previous test, this one was far harder to write. Dealing with the DOM lead to some complexity.



Given React test API can be somewhat verbose, people have developed lighter alternatives to it. See [jqunse/teaspoon](https://github.com/jquense/teaspoon)<sup>24</sup>, [Legitcode/tests](https://github.com/Legitcode/tests)<sup>25</sup>, and [enzyme](https://github.com/airbnb/enzyme)<sup>26</sup>, for example.



React provides a lighter way to assert component behavior without the DOM. [Shallow rendering](https://facebook.github.io/react/docs/test-utils.html#shallow-rendering)<sup>27</sup> is still missing some functionality, but it provides another way to test React components. It wouldn't help here given we rely on a `ref`, though.

## Checking Test Coverage

We have some basic tests in place now, but what about test coverage? Serve `build/coverage/PhantomJS . . . /` and inspect it in your browser. You should see something like this:

---

<sup>23</sup><http://sinonjs.org/>

<sup>24</sup><https://github.com/jquense/teaspoon>

<sup>25</sup><https://github.com/Legitcode/tests>

<sup>26</sup><https://github.com/airbnb/enzyme>

<sup>27</sup><https://facebook.github.io/react/docs/test-utils.html#shallow-rendering>

**all files components/**

**97.53%** Statements **79/81** **93.33%** Branches **42/45** **93.33%** Functions **14/15** **85.71%** Lines **12/14**

**5 statements, 13 branches** Ignored

File ▾		Statements ▾		Branches ▾		Functions ▾		Lines ▾	
<a href="#">Editable.jsx</a>	<div><div></div></div>	97.53%	79/81	93.33%	42/45	93.33%	14/15	85.71%	12/14

**Istanbul coverage**

If you examine the report further, you can see that the `checkEnter` portion of the code isn't covered. It would be a good idea to implement a unit test for that. Test coverage can reveal weak points such as this easily.

## 14.6 Testing Note

Even though `Note` is a trivial wrapper component, it is useful to test it as this will help us understand how to deal with React DnD. It is a testable library by design. Its [testing documentation](#)<sup>28</sup> goes into great detail.

Instead of `HTML5Backend`, we can rely on `TestBackend` in this case. The hard part is in building the context we need for testing that `Note` does indeed render its contents. Execute

```
npm i react-dnd-test-backend --save-dev
```

to get the backend installed. The test below illustrates the basic idea:

`tests/note_test.jsx`

```
import React from 'react';
import {
  renderIntoDocument
} from 'react-addons-test-utils';
import TestBackend from 'react-dnd-test-backend';
import {DragDropContext} from 'react-dnd';
import assert from 'assert';
import Note from '../app/components/Note';
```

```
describe('Note', function() {
  it('renders children', function() {
```

<sup>28</sup><https://gaearon.github.io/react-dnd/docs-testing.html>

```

    const test = 'test';
    const NoteContent = wrapInTestContext(Note);
    const component = renderIntoDocument(
      <NoteContent id="demo">{test}</NoteContent>
    );

    assert.equal(component.props.children, test);
  });
});

// https://gaearon.github.io/react-dnd/docs-testing.html
function wrapInTestContext(DecoratedComponent) {
  @DragDropContext(TestBackend)
  class TestContextContainer extends React.Component {
    render() {
      return <DecoratedComponent {...this.props} />;
    }
  }

  return TestContextContainer;
}

```

The test itself is easy. We just check that the children prop was set as we expect. The test could be improved by checking the rendered output through DOM.

## 14.7 Testing Kanban Stores

Alt provides a nice means for testing both [actions](http://alt.js.org/docs/testing/actions/)<sup>29</sup> and [stores](http://alt.js.org/docs/testing/stores/)<sup>30</sup>. Given our actions are so simple, it makes sense to focus on stores. To show you the basic idea, I'll show you how to test NoteStore. The same idea can be applied for LaneStore.

### Test Plan for NoteStore

In order to cover NoteStore, we should assert the following facts:

- Does it create notes correctly?
- Does it allow editing notes correctly?
- Does it allow deleting notes by id?

---

<sup>29</sup><http://alt.js.org/docs/testing/actions/>

<sup>30</sup><http://alt.js.org/docs/testing/stores/>

- Does it allow filtering notes by a given array of ids?

In addition, we could test against special cases and try to see how NoteStore behaves with various types of input. This is the useful minimum and will allow us to cover the common paths well.

## NoteStore **Allows** create

Creating new notes is simple. We just need to hit NoteActions.create and see that NoteStore.getState results contain the newly created Note:

tests/note\_store\_test.js

```
import assert from 'assert';
import NoteActions from '../app/actions/NoteActions';
import NoteStore from '../app/stores/NoteStore';
import alt from '../app/libs/alt';

alt.addStore('NoteStore', NoteStore);

describe('NoteStore', function() {
  it('creates notes', function() {
    const task = 'test';

    NoteActions.create({task});

    const state = alt.stores.NoteStore.getState();

    assert.equal(state.notes.length, 1);
    assert.equal(state.notes[0].task, task);
  });
});
```

Apart from the imports needed, this is simpler than our React tests. The test logic is easy to follow. We could even test the store methods directly if we wanted lower level tests.

## NoteStore **Allows** update

In order to update, we'll need to create a Note first. After that, we can change its content somehow. Finally, we can assert that the state changed:

tests/note\_store\_test.js

```
...

describe('NoteStore', function() {
  ...

  it('updates notes', function() {
    const NoteStore = alt.stores.NoteStore;
    const task = 'test';
    const updatedTask = 'test 2';

    NoteActions.create({id: 123, task});

    const note = NoteStore.getState().notes[0];

    NoteActions.update({...note, task: updatedTask});

    const state = NoteStore.getState();

    assert.equal(state.notes.length, 1);
    assert.equal(state.notes[0].task, updatedTask);
  });
});
```

The problem is that `assert.equal(state.notes.length, 1);` will fail. This is because our NoteStore is a singleton. Our first test already created a Note to it. There are two ways to solve this:

1. Push `alt.CreateStore` to a higher level. Now we create the association at the module level and this is causing issues now.
2. `flush` the contents of Alt store before each test.

I'm going to opt for 2. in this case:

`tests/note_store_test.js`



```
...

describe('NoteStore', function() {
  beforeEach(function() {
    alt.flush();
  });

  ...
});
```

After this little tweak, our test behaves the way we expect them to. This just shows that sometimes, we can make mistakes even in our tests. It is a good idea to understand what they are doing under the hood.

## NoteStore **Allows** delete

Testing delete is straight-forward as well. We'll need to create a Note. After that, we can try to delete it by id and assert that there are no notes left:

**tests/note\_store\_test.js**

```
describe('NoteStore', function() {
  ...

  it('deletes notes', function() {
    const NoteStore = alt.stores.NoteStore;

    NoteActions.create({id: 123, task: 'test'});

    const note = NoteStore.getState().notes[0];

    NoteActions.delete(note.id);

    const state = NoteStore.getState();

    assert.equal(state.notes.length, 0);
  });
});
```

It would be a good idea to start pushing some of the common bits to shared functions now. At least this way the tests will remain self-contained even if there's more code.



[Legitcode/tests](https://github.com/Legitcode/tests)<sup>31</sup> provides handy shortcuts for testing Alt stores.

## 14.8 Conclusion

We have some basic unit tests in place for our Kanban application now. It's far from being tested completely. Nonetheless, we've managed to cover some crucial parts of it. As a result, we can have more confidence in that it operates correctly. It would be a nice idea to test the remainder, though, and perhaps refactor those tests a little. There are also special cases that we haven't given a lot of thought to.

We are also missing acceptance tests completely. Fortunately, that's a topic that can be solved outside of React, Alt, and such. [Nightwatch](http://nightwatchjs.org/)<sup>32</sup> is a tool that runs on top of Selenium server and allows you to write these kind of tests. It will take some effort to pick up a tool like this. It will allow you to test more qualitative aspects of your application, though.

---

<sup>31</sup><https://github.com/Legitcode/tests#testing-alt-stores>

<sup>32</sup><http://nightwatchjs.org/>

# 15. Typing with React

Just like testing, typing is another feature that can make our lives easier especially when working with larger codebases. Some languages are very strict about this, but as you know JavaScript is very flexible.

Flexibility is useful during prototyping. Unfortunately, this means it's going to be easy to make mistakes and not notice them until it's too late. This is why testing and typing are so important. Typing is a good way to strengthen your code and make it harder to break. It also serves as a form of documentation for other developers.

In React, you document the expectations of your components using `propTypes`. It is possible to go beyond this by using Flow, a syntax for gradual typing. There are also [TypeScript type definitions](#)<sup>1</sup> for React, but we won't go into that.

## 15.1 `propTypes` and `defaultProps`

`propTypes` allow you to document what kind of data your component expects. `defaultProps` allow you to set default values for `propTypes`. This can cut down the amount of code you need to write as you don't need to worry about special cases so much.

The annotation data is used during development. If you break a type contract, React will let you know. As a result, you'll be able to fix potential problems before they do any harm. The checks will be disabled in production mode (`NODE_ENV=production`) in order to improve performance.

### Annotation Styles

The way you annotate your components depends on the way you declare them. I've given simplified examples using various syntaxes below:

ES5

---

<sup>1</sup><https://github.com/borisyankov/DefinitelyTyped/tree/master/react>

```

module.exports = React.createClass({
  displayName: 'Editable',
  propTypes: {
    value: React.PropTypes.string
  },
  defaultProps: {
    value: ''
  },
  ...
});

```

## ES6

```

class Editable extends React.Component {...}

Editable.propTypes = {
  value: React.PropTypes.string
};
Editable.defaultProps = {
  value: ''
};

export default Editable;

```

## ES7 (proposed property initializer)

```

export default class Editable extends React.Component {
  static propTypes = {
    value: React.PropTypes.string
  }
  static defaultProps = {
    value: ''
  }
}

```

Props are optional by default. Annotation, such as `React.PropTypes.string.isRequired`, can be used to force the prop to be passed. If not passed, you will get a warning.

## Annotation Types

`propTypes` support basic types as follows: `React.PropTypes.[array, bool, func, number, object, string]`. In addition, there's a special node type that refers to anything that can be rendered

by `React.any` includes literally anything. `element` maps to a React element. Furthermore there are functions as follows:

- `React.PropTypes.instanceOf(class)` - Checks using JavaScript `instanceof`.
- `React.PropTypes.oneOf(['cat', 'dog', 'lion'])` - Checks that one of the values is provided.
- `React.PropTypes.oneOfType([<propType>, ...])` - Same for `propTypes`. You can use basic type definitions here (i.e., `React.PropTypes.array`).
- `React.PropTypes.arrayOf(<propType>)` - Checks that a given array contains items of the given type.
- `React.PropTypes.objectOf(<propType>)` - Same idea as for arrays.
- `React.PropTypes.shape({<name>: <propType>})` - Checks that given object is in a particular object shape with certain `propTypes`.

It's also possible to implement custom validators by passing a function using the following signature to a prop type: `function(props, propName, componentName)`. If the custom validation fails, you should return an error (i.e., `return new Error('Not a number!')`).

The [documentation](#)<sup>2</sup> goes into further detail.

## 15.2 Typing Kanban

To give you a better idea of how `propTypes` work, we can type our Kanban application. There are only a few components to annotate. We can skip annotating `App` as that's the root component of our application. The rest can use some typing.

### Annotating Lanes

`Lanes` provide a good starting point. It expects an array of `lanes`. We can make it optional and default to an empty list. In terms of `propTypes`, our annotation looks like this:

`app/components/Lanes.jsx`

---

<sup>2</sup><https://facebook.github.io/react/docs/reusable-components.html>

```
import React from 'react';
import Lane from './Lane';

export default ({lanes}) => (
const Lanes = ({lanes}) => (
  ...
);
Lanes.propTypes = {
  lanes: React.PropTypes.array
};
Lanes.defaultProps = {
  lanes: []
};

export default Lanes;
```

The current annotation is better than nothing. We can still break our application easily by passing invalid lanes around. We know it's going to be an array but not much beyond that. We should annotate Lane and its children to guard against this case.

## Annotating Lane

As per our implicit definition, Lane expects a lane, notes, LaneActions, NoteActions, and connectDropTarget. lane should be required, as a lane without any data doesn't make any sense. We can document that using React.PropTypes.shape. The rest can remain optional. Translated to propTypes we would end up with this:

**app/components/Lane.jsx**

```
...

const Lane = ({
  connectDropTarget, lane, notes, LaneActions, NoteActions, ...props
}) => {
  ...
};
Lane.propTypes = {
  lane: React.PropTypes.shape({
    id: React.PropTypes.string.isRequired,
    editing: React.PropTypes.bool,
    name: React.PropTypes.string,
    notes: React.PropTypes.array
```

```

    }).isRequired,
    LaneActions: React.PropTypes.object,
    NoteActions: React.PropTypes.object,
    connectDropTarget: React.PropTypes.func
  };
Lane.defaultProps = {
  name: '',
  notes: []
};

...

```

If our basic data model is wrong somehow now, we'll know about it during development. To harden our system further, we should annotate notes contained by the lanes.

## Annotating Notes

As you might remember from the implementation, `Notes` accepts `notes`, `onNoteClick`, `onEdit`, and `onDelete` handlers. We can apply the same logic to `notes` as for `Lanes`. If the array isn't provided, we can default to an empty one. We can use empty functions as default handlers if they aren't provided. The idea would translate to code as follows:

`app/components/Notes.jsx`

```

...

export default ({
  — notes,
  — onNoteClick=() => {}, onEdit=() => {}, onDelete=() => {}
}) => {
const Notes = ({notes, onNoteClick, onEdit, onDelete}) => (
  ...
)
);
Notes.propTypes = {
  notes: React.PropTypes.array,
  onEdit: React.PropTypes.func,
  onDelete: React.PropTypes.func,
  onNoteClick: React.PropTypes.func
};
Notes.defaultProps = {
  notes: [],

```

```

    onEdit: () => {},
    onDelete: () => {},
    onNoteClick: () => {}
  };

```

```
export default Notes;
```

Even though useful, this doesn't give any guarantees about the shape of the individual items. We could document it here to get a warning earlier, but it feels like a better idea to push that to Note level. After all, that's what we did with Lanes and Lane earlier.

## Annotating Note

In our implementation, Note works as a wrapper component that renders its content. Its primary purpose is to provide drag and drop related hooks. As per our implementation, it requires an id prop. You can also pass an optional onMove handler to it. It receives connectDragSource and connectDropSource through React DnD. In annotation format we get:

**app/components/Note.jsx**

```

...

const Note = ({
  connectDragSource, connectDropTarget, isDragging,
  isOver, onMove, id, editing, children, ...props
}) => {
  ...
};

Note.propTypes = {
  id: React.PropTypes.string.isRequired,
  editing: React.PropTypes.bool,
  connectDragSource: React.PropTypes.func,
  connectDropTarget: React.PropTypes.func,
  isDragging: React.PropTypes.bool,
  onMove: React.PropTypes.func,
  children: React.PropTypes.node
};

Note.defaultProps = {
  onMove: () => {}
};

...

```

We've annotated almost everything we need. There's just one bit remaining, namely Editable.



## Annotating Editable

In our system, `Editable` takes care of some of the heavy lifting. It is able to render an optional `value`. It should receive the `onEdit` hook. Using the annotation syntax we get the following:

`app/components/Editable.jsx`

```
import React from 'react';
import classNames from 'classnames';

export default ({editing, value, onEdit, className, ...props}) => {
const Editable = ({editing, value, onEdit, className, ...props}) => {
  ...
}
};

Editable.propTypes = {
  value: React.PropTypes.string,
  editing: React.PropTypes.bool,
  onEdit: React.PropTypes.func.isRequired
};
Editable.defaultProps = {
  value: '',
  editing: false,
  onEdit: () => {}
};

export default Editable;

...
```

We have annotated our system now. In case we manage to break our data model somehow, we'll know about it during development. This is very nice considering future efforts. The earlier you catch and fix problems like these, the easier it is to build on top of it.

Even though `propTypes` are nice, they are also a little verbose. Flow typing can help us in that regard.

## 15.3 Type Checking with Flow



Flow

Facebook's [Flow](https://flowtype.org/)<sup>3</sup> provides gradual typing for JavaScript. This means you can add types to your code as you need them. We can achieve similar results as with `propTypes` and we can add additional invariants to our code as needed. To give you an idea, consider the following trivial example:

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

The definition states that `add` should receive two numbers and return one as a result. This is the way it's typically done in statically typed languages. Now we can benefit from the same idea in JavaScript.



See [Try Flow](https://tryflow.org/)<sup>4</sup> for more concrete examples.

Flow relies on a static type checker that has to be installed separately. As you run the tool, it will evaluate your code and provide recommendations. To ease development, there's a way to evaluate Flow types during runtime. This can be achieved through a Babel plugin.

---

<sup>3</sup><http://flowtype.org/>

<sup>4</sup><https://tryflow.org/>



Babel plugin known as [babel-plugin-flow-react-proptypes](https://www.npmjs.com/package/babel-plugin-flow-react-proptypes)<sup>5</sup> allows you to generate propTypes based on Flow definitions. This brings the two ideas together in an interesting manner.



At the time of this writing, major editors and IDEs have poor support for Flow annotations. This may change in the future.

## Setting Up Flow

There are [pre-built binaries](#)<sup>6</sup> for common platforms. You can also install it through Homebrew on Mac OS X (`brew install flow`).

As Flow relies on configuration and won't run without it, we should generate some. Execute `flow init`. That will generate a `.flowconfig` file that can be used for [advanced configuration](#)<sup>7</sup>.

We are going to need some further tweaks to adapt it to our environment. We'll want to make sure it skips `./node_modules` while parses through the `./app` directory. In addition we need to set up an entry for Flow interfaces and make Flow ignore certain language features.

Since we want avoid parsing `node_modules`, we should tweak the configuration as follows:

### `.flowconfig`

```
[ignore]
.*/*node_modules

[include]
./app

[libs]
./interfaces

[options]
esproposal.decorators=ignore
esproposal.class_instance_fields=ignore
```



As of writing, `esproposal.decorators=ignore` skips only method decorators. This means some extra effort is needed to port our codebase to Flow.

---

<sup>5</sup><https://www.npmjs.com/package/babel-plugin-flow-react-proptypes>

<sup>6</sup><http://flowtype.org/docs/getting-started.html>

<sup>7</sup><http://flowtype.org/docs/advanced-configuration.html>

## Running Flow

Running Flow is simple, just execute `flow check`. You will likely see something like this:

```
$ flow check
find: ../../interfaces: No such file or directory
```

```
Found 0 errors
```

We'll fix that warning in a bit. Before that, we should make it easy to trigger it through npm. Add the following bit to your *package.json*:

**package.json**

```
{
  ...
  "scripts": {
    ...
    "test:flow": "flow check"
  },
  ...
}
```

After this, we can execute `npm run test:flow`, and we don't have to care about the exact details of how to call it. Note that if the process fails, it will give a nasty looking npm error (multiple lines of `npm ERR!`). If you want to disable that, you can run `npm run test:flow --silent` instead to chomp it.



To gain some extra performance, Flow can be run in a daemon mode. Simply execute `flow` to start it. If you execute `flow` again, it you should get instant results. This process may be closed using `flow stop`.

## Setting Up a Demo

Flow expects that you annotate the files in which you use it using the declaration `/* @flow */` at the beginning of the file. Alternatively, you can try running `flow check --all`. Keep in mind that it can be slow, as it will process each file you have included, regardless of whether it has been annotated with `@flow`! We will stick to the former in this project.

To get a better idea of what Flow output looks like, we can try a little demo. Set it up as follows:

**demo.js**

```

/* @flow */
function add(x: number, y: number): number {
  return x + y;
}

add(2, 4);

/* this shouldn't be valid as per definition! */
add('foo', 'bar');

```

Run `npm run test:flow --silent` now. If this worked correctly, you should see something like this:

```

find: .../kanban-app/./interfaces: No such file or directory
demo.js:9
  9: add('foo', 'bar');
      ^^^^^^^^^^^^^^^^^ function call
  9: add('foo', 'bar');
      ^^^^^ string. This type is incompatible with
  2: function add(x: number, y: number): number {
      ^^^^^ number

demo.js:9
  9: add('foo', 'bar');
      ^^^^^^^^^^^^^^^^^ function call
  9: add('foo', 'bar');
      ^^^^^ string. This type is incompatible with
  2: function add(x: number, y: number): number {
      ^^^^^ number

```

Found 2 errors

This means everything is working as it should and Flow caught a nasty programming error for us. Someone was trying to pass values of incompatible type to `add`. That's good to know before going to production.

Given we know that Flow can catch issues for us, get rid of the demo file before moving on.

## 15.4 Converting propTypes to Flow Checks

To give you a better idea of what it would take to port our application to Flow, I will show you next how to convert a couple of our components to the format.

## Porting Editable to Flow

Editable is a good starting point. In this case all of our props except `onEdit` are optional. We could easily make them mandatory if we wanted to, though. I've converted the `propTypes` definition to ES7 style as that's what Flow expects:

`app/components/Editable.jsx`

```

/* @flow */
...

const Editable = ({editing, value, onEdit, className, ...props}) => {
  ...
};

const Editable = (props: {
  editing?: boolean,
  value?: string,
  onEdit: Function,
  className?: string
}) => {
  const {editing, className, value, onEdit} = props;

  if(editing) {
    return <Edit
      className={className}
      value={value}
      onEdit={onEdit} />;
  }

  return <span className={classnames('value', className)}>
    {value}
  </span>;
};

Editable.propTypes = {
  value: React.PropTypes.string,
  editing: React.PropTypes.bool,
  onEdit: React.PropTypes.func.isRequired
};

Editable.defaultProps = {
  value: '',
  editing: false,
  onEdit: () => {}
};

```

...

It is important to note that we had to give up on our rest parameter at props (i.e., `...props`). Flow expects stronger typing by default. You could argue doing this makes it harder to make mistakes as you cannot pass anything as props and instead have to document what you are going to pass.

Executing `npm run test:flow --silent` should yield a lot of errors like this:

```
$ npm run test:flow --silent
find: ../kanban-app/interfaces: No such file or directory
app/components/Editable.jsx:3
  3: import classNames from 'classnames';
                                ^^^^^^^^^^^^^^^^ classNames. Required module not found
```

...

Found 7 errors

The problem is that Flow expects to find an interface for `classnames` but fails to find it. Each dependency needs some kind of an interface definition for the system to work. To fix the issue, we have to implement a Flow interface for `classnames`:

**interfaces/classnames.js**

```
declare module 'classnames' {
  declare function exports(): Object;
}
```

A simple component, such as `Note`, is another good candidate.



If you want to display Flow errors through Webpack, consider using [flow-status-webpack-plugin](https://www.npmjs.com/package/flow-status-webpack-plugin)<sup>8</sup>.

## Porting `Note` to Flow

Converting `Note` is straight-forward as well:

**app/components/Note.jsx**

---

<sup>8</sup><https://www.npmjs.com/package/flow-status-webpack-plugin>

```

/* @flow */
...

const Note = ({
  connectDragSource, connectDropTarget, isDragging,
  isOver, onMove, id, editing, children, ...props
}) => {
  ...
};

const Note = (props: {
  connectDragSource?: Function,
  connectDropTarget?: Function,
  isDragging?: boolean,
  isOver?: boolean,
  onMove?: Function,
  id: string,
  editing?: boolean,
  children?: any,
  className?: string,
  onClick?: Function
}) => {
  const {
    connectDragSource, connectDropTarget, isDragging, isOver,
    onMove, id, editing, children, className, onClick
  } = props;

  // Pass through if we are editing
  const dragSource = editing ? a => a : connectDragSource;

  return compose(dragSource, connectDropTarget)(
    <div style={{
      opacity: isDragging || isOver ? 0 : 1
    }} className={className} onClick={onClick}>{children}</div>
  );
};

Note.propTypes = {
  id: React.PropTypes.string.isRequired,
  editing: React.PropTypes.bool,
  connectDragSource: React.PropTypes.func,
  connectDropTarget: React.PropTypes.func,
  isDragging: React.PropTypes.bool,
  onMove: React.PropTypes.func,

```



```

—children: React.PropTypes.node
};
Note.defaultProps = {
—onMove: () => {}
};

...

```

Executing `npm run test:flow --silent` should yield errors like this:

```

$ npm run test:flow --silent
app/components/Note.jsx:3
  3: import {compose} from 'redux';
                                ^^^^^^^^ redux. Required module not found

app/components/Note.jsx:4
  4: import {DragSource, DropTarget} from 'react-dnd';
                                         ^^^^^^^^^^^^^^ react-dnd. Required module\
not found

```

Found 2 errors

We are missing interface definitions again. Better add those in. When it comes to Redux, we are just interested in `compose`. We can add a stub definition for that:

#### **interfaces/redux.js**

```

declare module 'redux' {
  declare function compose(): any;
}

```

React DnD is a similar case. A couple of stubs will work.

#### **interfaces/react-dnd.js**

```

declare module 'react-dnd' {
  declare function DragSource(): any;
  declare function DropTarget(): any;
}

```

This states that `react-dnd` is a module that exports two functions that can return any type. Both definitions could be made stronger. That would help to catch issues earlier. You can start with something loose with Flow and add stricter typing as you want.



[Flow documentation](#)<sup>9</sup> goes into greater detail about declarations.

If you execute `npm run test:flow --silent` now, you shouldn't get any errors or warnings at all:

```
$ npm run test:flow --silent
```

```
Found 0 errors
```

There are still some rough areas, but when it works, it can help to find potential problems sooner. The same annotations can be useful for runtime checking during development. Babel plugin known as [babel-plugin-typecheck](#)<sup>10</sup> can achieve that.

## 15.5 Babel Typecheck

*babel-plugin-typecheck* is able to take your Flow annotations and turn them into runtime checks. To get started, execute

```
npm i babel-plugin-typecheck --save-dev
```

To make Babel aware of it, we'll need to tweak `.babelrc`:

**.babelrc**

```
{
  ...
  "env": {
    "start": {
      "presets": [
        "react-hmre"
      ]
    },
    "plugins": [

```

<sup>9</sup><http://flowtype.org/docs/declarations.html>

<sup>10</sup><https://www.npmjs.com/package/babel-plugin-typecheck>

```
      "typecheck"  
    ]  
  ]  
}  
}
```

After this change, our Flow checks will get executed during development. Flow static checker will be able to catch more errors. Runtime checks have their benefits, though, and it's far better than nothing.

You can test it by breaking `Editable` by purpose. Try returning a string from there and see how it blows up. Flow would catch it, but it's nice to get feedback like this during the development as well. You can also try introducing other subtle bugs to the system and see how it reacts.

## 15.6 TypeScript

Microsoft's [TypeScript](http://www.typescriptlang.org/)<sup>11</sup> is a good alternative to Flow. It has supported React officially since version 1.6 as it introduced JSX support. It is a more established solution compared to Flow. As a result you can find a large amount of [type definitions for popular libraries](https://github.com/DefinitelyTyped/DefinitelyTyped)<sup>12</sup>, React included.

I won't be covering TypeScript in detail as the project would have to change considerably in order to introduce it. Instead, I encourage you to study available Webpack loaders:

- [ts-loader](https://www.npmjs.com/package/ts-loader)<sup>13</sup>
- [awesome-typescript-loader](https://www.npmjs.com/package/awesome-typescript-loader)<sup>14</sup>
- [typescript-loader](https://www.npmjs.com/package/typescript-loader)<sup>15</sup>

This section may be expanded later depending on the adoption of TypeScript within the React community.



If you want to lint your TypeScript code, consider [TSLint](https://palantir.github.io/tslint/)<sup>16</sup>.

---

<sup>11</sup><http://www.typescriptlang.org/>

<sup>12</sup><https://github.com/DefinitelyTyped/DefinitelyTyped>

<sup>13</sup><https://www.npmjs.com/package/ts-loader>

<sup>14</sup><https://www.npmjs.com/package/awesome-typescript-loader>

<sup>15</sup><https://www.npmjs.com/package/typescript-loader>

<sup>16</sup><https://palantir.github.io/tslint/>

## 15.7 Conclusion

Currently, the state of type checking in React is still in bit of a flux. `propTypes` are the most stable solution. Even if a little verbose, they are highly useful for documenting what your components expect. This can save your nerves during development.

More advanced solutions, such as Flow and TypeScript, are still in a growing stage. There are still some sore points, but both have a great promise. Typing is invaluable especially as your codebase grows. Early on, flexibility has more value, but as you develop and understand your problems better, solidifying your design may be worth your while.

# 16. Styling React

Traditionally, web pages have been split up into markup (HTML), styling (CSS), and logic (JavaScript). Thanks to React and similar approaches, we've begun to question this split. We still may want to separate our concerns somehow. But the split can be on different axes.

This change in the mindset has led to new ways to think about styling. With React, we're still figuring out the best practices. Some early patterns have begun to emerge, however. As a result it is difficult to provide any definite recommendations at the moment. Instead, I will go through various approaches so you can make up your mind based on your exact needs.

## 16.1 Old School Styling

The old school approach to styling is to sprinkle some *ids* and *classes* around, set up CSS rules, and hope for the best. In CSS everything is global by default. Nesting definitions(e.g., `.main .sidebar .button`) creates implicit logic to your styling. Both features lead to a lot of complexity as your project grows. This approach can be acceptable when starting out, but as you develop, you most likely want to migrate away from it.

## 16.2 CSS Methodologies

What happens when your application starts to expand and new concepts get added? Broad CSS selectors are like globals. The problem gets even worse if you have to deal with loading order. If selectors end up in a tie, the last declaration wins, unless there's `!important` somewhere. It gets complex very fast.

We could battle this problem by making the selectors more specific, using some naming rules, and so on. That just delays the inevitable. As people have battled with this problem for a while, various methodologies have emerged.

Particularly, [OOCSS](http://oocss.org/)<sup>1</sup> (Object-Oriented CSS), [SMACSS](https://smacss.com/)<sup>2</sup> (Scalable and Modular Approach for CSS), and [BEM](https://en.bem.info/method/)<sup>3</sup> (Block Element Modifier) are well known. Each of them solves problems of vanilla CSS in their own way.

---

<sup>1</sup><http://oocss.org/>

<sup>2</sup><https://smacss.com/>

<sup>3</sup><https://en.bem.info/method/>

## BEM

BEM originates from Yandex. The goal of BEM is to allow reusable components and code sharing. Sites, such as [Get BEM](http://getbem.com/)<sup>4</sup> help you to understand the methodology in more detail.

Maintaining long class names which BEM requires can be arduous. Thus various libraries have appeared to make this easier. For React, examples of these are [react-bem-helper](https://www.npmjs.com/package/react-bem-helper)<sup>5</sup>, [react-bem-render](https://www.npmjs.com/package/react-bem-render)<sup>6</sup>, and [bem-react](https://www.npmjs.com/package/bem-react)<sup>7</sup>.

Note that [postcss-bem-linter](https://www.npmjs.com/package/postcss-bem-linter)<sup>8</sup> allows you to lint your CSS for BEM conformance.

## OOCSS and SMACSS

Just like BEM, both OOCSS and SMACSS come with their own conventions and methodologies. As of this writing, no React specific helper libraries exist for OOCSS and SMACSS.

## Pros and Cons

The primary benefit of adopting a methodology is that it brings structure to your project. Rather than writing ad hoc rules and hoping everything works, you will have something stronger to fall back onto. The methodologies overcome some of the basic issues and help you develop good software over the long term. The conventions they bring to a project help with maintenance and are less prone to lead to a mess.

On the downside, once you adopt one, you are pretty much stuck with that and it's going to be difficult to migrate. But if you are willing to commit, there are benefits to gain.

The methodologies also bring their own quirks (e.g., complex naming schemes). This may make certain things more complicated than they have to be. They don't necessarily solve any of the bigger underlying issues. They rather provide patches around them.

There are various approaches that go deeper and solve some of these fundamental problems. That said, it's not an either-or proposition. You may adopt a methodology even if you use some CSS processor.

---

<sup>4</sup><http://getbem.com/>

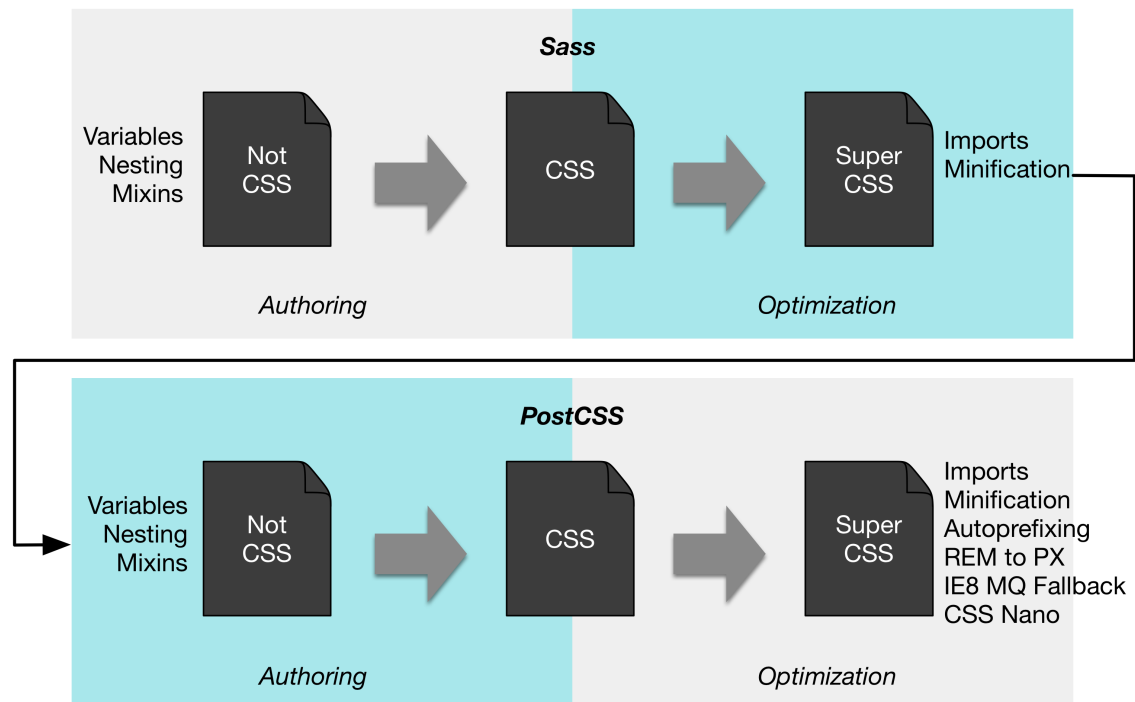
<sup>5</sup><https://www.npmjs.com/package/react-bem-helper>

<sup>6</sup><https://www.npmjs.com/package/react-bem-render>

<sup>7</sup><https://www.npmjs.com/package/bem-react>

<sup>8</sup><https://www.npmjs.com/package/postcss-bem-linter>

## 16.3 CSS Processors



### CSS Processors

Vanilla CSS is missing some functionality that would make maintenance work easier. Consider something basic like variables, nesting, mixins, math or color functions. It would also be nice to be able to forget about browser specific prefixes. These are small things that add up quite fast and make it annoying to write vanilla CSS.

Sometimes, you may see terms *preprocessor* or *postprocessor*. [Stefan Baumgartner](https://medium.com/@ddprtt/deconfusing-pre-and-post-processing-d68e3bd078a3)<sup>9</sup> calls these tools simply *CSS processors*. The image above adapted based on Stefan's work gets to the point. The tooling operates both on authoring and optimization level. By authoring we mean features that make it easier to write CSS. Optimization features operate based on vanilla CSS and convert it into something more optimal for browsers to consume.

The interesting thing is that you may actually want to use multiple CSS processors. Stefan's image illustrates how you can author your code using Sass and still benefit from processing done through PostCSS. For example, it can *autoprefix* your CSS code so that you don't have to worry about prefixing per browser anymore.

<sup>9</sup><https://medium.com/@ddprtt/deconfusing-pre-and-post-processing-d68e3bd078a3>

You can use common processors, such as [Less](#)<sup>10</sup>, [Sass](#)<sup>11</sup>, [Stylus](#)<sup>12</sup>, or [PostCSS](#)<sup>13</sup> with React.

[cssnext](#)<sup>14</sup> is a PostCSS plugin that allows us to experience the future now. There are some restrictions, but it may be worth a go. The advantage of PostCSS and cssnext is that you will literally be coding in the future. As browsers get better and adopt the standards, you don't have to worry about porting.

## Pros and Cons

Compared to vanilla CSS, processors bring a lot to the table. They deal with certain annoyances (e.g., autoprefixing) while improving your productivity. PostCSS is more granular by definition and allows you to use just the features you want. Processors, such as Less or Sass, are more involved. These approaches can be used together, though, so you could, for instance, author your styling in Sass and then apply some PostCSS plugins to it as you see necessary.

In our project, we could benefit from cssnext even if we didn't make any changes to our CSS. Thanks to autoprefixing, rounded corners of our lanes would look good even in legacy browsers. In addition, we could parameterize styling thanks to variables.

## 16.4 React Based Approaches

With React we have some additional alternatives. What if the way we've been thinking about styling has been misguided? CSS is powerful, but it can become an unmaintainable mess without some discipline. Where do we draw the line between CSS and JavaScript?

There are various approaches for React that allow us to push styling to the component level. It may sound heretical. React, being an iconoclast, may lead the way here.

### Inline Styles to Rescue

Ironically, the way solutions based on React solve this is through inline styles. Getting rid of inline styles was one of the main reasons for using separate CSS files in the first place. Now we are back there. This means that instead of something like this:

---

<sup>10</sup><http://lesscss.org/>

<sup>11</sup><http://sass-lang.com/>

<sup>12</sup><https://learnboost.github.io/stylus/>

<sup>13</sup><http://postcss.org/>

<sup>14</sup><https://cssnext.github.io/>



```
render(props, context) {  
  const notes = this.props.notes;  
  
  return <ul className='notes'>{notes.map(this.renderNote)}</ul>;  
}
```

and accompanying CSS, we'll do something like this:

```
render(props, context) {  
  const notes = this.props.notes;  
  const style = {  
    margin: '0.5em',  
    paddingLeft: 0,  
    listStyle: 'none'  
  };  
  
  return <ul style={style}>{notes.map(this.renderNote)}</ul>;  
}
```

Like with HTML attribute names, we are using the same camelcase convention for CSS properties.

Now that we have styling at the component level, we can implement logic that also alters the styles easily. One classic way to do this has been to alter class names based on the outlook we want. Now we can adjust the properties we want directly.

We have lost something in process, though. Now all of our styling is tied to our JavaScript code. It is going to be difficult to perform large, sweeping changes to our codebase as we need to tweak a lot of components to achieve that.

We can try to work against this by injecting a part of styling through props. A component could patch its style based on a provided one. This can be improved further by coming up with conventions that allow parts of style configuration to be mapped to some specific part. We just reinvented selectors on a small scale.

How about things like media queries? This naïve approach won't quite cut it. Fortunately, people have come up with libraries to solve these tough problems for us.

According to Michele Bertoli basic features of these libraries are

- Autoprefixing - e.g., for border, animation, flex.
- Pseudo classes - e.g., :hover, :active.
- Media queries - e.g., @media (max-width: 200px).
- Styles as Object Literals - See the example above.
- CSS style extraction - It is useful to be able to extract separate CSS files as that helps with the initial loading of the page. This will avoid a flash of unstyled content (FOUC).

I will cover some of the available libraries to give you a better idea how they work. See [Michele's list<sup>15</sup>](#) for a more a comprehensive outlook of the situation.

## Radium

[Radium<sup>16</sup>](#) has certain valuable ideas that are worth highlighting. Most importantly it provides abstractions required to deal with media queries and pseudo classes (e.g., `:hover`). It expands the basic syntax as follows:

```
const styles = {
  button: {
    padding: '1em',

    ':hover': {
      border: '1px solid black'
    },

    '@media (max-width: 200px)': {
      width: '100%',

      ':hover': {
        background: 'white',
      }
    }
  },
  primary: {
    background: 'green'
  },
  warning: {
    background: 'yellow'
  },
};

...

<button style={[styles.button, styles.primary]}>Confirm</button>
```

For style prop to work, you'll need to annotate your classes using `@Radium` decorator.

---

<sup>15</sup><https://github.com/MicheleBertoli/css-in-js>

<sup>16</sup><http://projects.formidablelabs.com/radium/>

## React Style

[React Style](#)<sup>17</sup> uses the same syntax as React Native [StyleSheet](#)<sup>18</sup>. It expands the basic definition by introducing additional keys for fragments.

```
import StyleSheet from 'react-style';

const styles = StyleSheet.create({
  primary: {
    background: 'green'
  },
  warning: {
    background: 'yellow'
  },
  button: {
    padding: '1em'
  },
  // media queries
  '@media (max-width: 200px)': {
    button: {
      width: '100%'
    }
  }
});

...

<button styles={[styles.button, styles.primary]}>Confirm</button>
```

As you can see, we can use individual fragments to get the same effect as Radium modifiers. Also media queries are supported. React Style expects that you manipulate browser states (e.g., `:hover`) through JavaScript. Also CSS animations won't work. Instead, it's preferred to use some other solution for that.



[React Style plugin for Webpack](#)<sup>19</sup> can extract CSS declarations into a separate bundle. Now we are closer to the world we're used to, but without cascades. We also have our style declarations on the component level.

---

<sup>17</sup><https://github.com/js-next/react-style>

<sup>18</sup><https://facebook.github.io/react-native/docs/stylesheet.html#content>

<sup>19</sup><https://github.com/js-next/react-style-webpack-plugin>

## JSS

[JSS<sup>20</sup>](https://github.com/jsstyles/jss) is a JSON to StyleSheet compiler. It can be convenient to represent styling using JSON structures as this gives us easy namespacing. Furthermore it is possible to perform transformations over the JSON to gain features, such as autoprefixing. JSS provides a plugin interface just for this.

JSS can be used with React through [react-jss<sup>21</sup>](https://www.npmjs.com/package/react-jss). You can use JSS through *react-jss* like this:

```
...
import classNames from 'classnames';
import useSheet from 'react-jss';

const styles = {
  button: {
    padding: '1em'
  },
  'media (max-width: 200px)': {
    button: {
      width: '100%'
    }
  },
  primary: {
    background: 'green'
  },
  warning: {
    background: 'yellow'
  }
};

@useSheet(styles)
export default class ConfirmButton extends React.Component {
  render() {
    const {classes} = this.props.sheet;

    return <button
      className={classNames(classes.button, classes.primary)}>
      Confirm
    </button>;
  }
}
```

---

<sup>20</sup><https://github.com/jsstyles/jss>

<sup>21</sup><https://www.npmjs.com/package/react-jss>

The approach supports pseudoselectors, i.e., you could define a selector within, such as `&:hover`, within a definition and it would just work.



There's a [jss-loader<sup>22</sup>](#) for Webpack.

## React Inline

[React Inline<sup>23</sup>](#) is an interesting twist on StyleSheet. It generates CSS based on `className` prop of elements where it is used. The example above could be adapted to React Inline like this:

```
import cx from 'classnames';
...

class ConfirmButton extends React.Component {
  render() {
    const {className} = this.props;
    const classes = cx(styles.button, styles.primary, className);

    return <button className={classes}>Confirm</button>;
  }
}
```

Unlike React Style, the approach supports browser states (e.g., `:hover`). Unfortunately, it relies on its own custom tooling to generate React code and CSS which it needs to work.

## jsxstyle

Pete Hunt's [jsxstyle<sup>24</sup>](#) aims to mitigate some problems of React Style's approach. As you saw in previous examples, we still have style definitions separate from the component markup. `jsxstyle` merges these two concepts. Consider the following example:

---

<sup>22</sup><https://www.npmjs.com/package/jss-loader>

<sup>23</sup><https://github.com/martinandert/react-inline>

<sup>24</sup><https://github.com/petehunt/jsxstyle>

```
// PrimaryButton component
<button
  padding='1em'
  background='green'
>Confirm</button>
```

The approach is still in its early days. For instance, support for media queries is missing. Instead of defining modifiers as above, you'll end up defining more components to support your use cases.



Just like React Style, `jsxstyle` comes with a Webpack loader that can extract CSS into a separate file.

## 16.5 CSS Modules

As if there weren't enough styling options for React, there's one more that's worth mentioning. [CSS Modules](#)<sup>25</sup> starts from the premise that CSS rules should be local by default. Globals should be treated as a special case. Mark Dalgleish's post [The End of Global CSS](#)<sup>26</sup> goes into more detail about this.

In short, if you make it difficult to use globals, you manage to solve the biggest problem of CSS. The approach still allows us to develop CSS as we've been used to. This time we're operating in a safer, local context by default.

This itself solves a large amount of problems libraries above try to solve in their own ways. If we need global styles, we can still get them. We still might want to have some around for some higher level styling after all. This time we're being explicit about it.

To give you a better idea, consider the example below:

**style.css**

```
.primary {
  background: 'green';
}

.warning {
  background: 'yellow';
}

.button {
```

---

<sup>25</sup><https://github.com/css-modules/css-modules>

<sup>26</sup><https://medium.com/seek-ui-engineering/the-end-of-global-css-90d2a4a06284>

```
padding: 1em;
}

.primaryButton {
  composes: primary button;
}

@media (max-width: 200px) {
  .primaryButton {
    composes: primary button;

    width: 100%;
  }
}
```


### button.jsx


```
import styles from './style.css';


...

<button className={`${styles.primaryButton}`}>Confirm</button>
```

As you can see, this approach provides a balance between what people are familiar with and what React specific libraries do. It would not surprise me a lot if this approach gained popularity even though it's still in its early days. See [CSS Modules Webpack Demo](#)<sup>27</sup> for more examples.

 You can use other processors, such as Sass, in front of CSS Modules, in case you want more functionality.

 [gajus/react-css-modules](#)<sup>28</sup> makes it even more convenient to use CSS Modules with React. Using it, you don't need to refer to the `styles` object anymore, and you are not forced to use camelCase for naming.

 Glen Maddern discusses the topic in greater detail in his article named [CSS Modules - Welcome to the Future](#)<sup>29</sup>.

---

<sup>27</sup><https://css-modules.github.io/webpack-demo/>

<sup>28</sup><https://github.com/gajus/react-css-modules>

<sup>29</sup><http://glenmaddern.com/articles/css-modules>

## 16.6 Conclusion

It is simple to try out various styling approaches with React. You can do it all, ranging from vanilla CSS to more complex setups. React specific tooling even comes with loaders of their own. This makes it easy to try out different alternatives.

React based styling approaches allow us to push styles to the component level. This provides an interesting contrast to conventional approaches where CSS is kept separate. Dealing with component specific logic becomes easier. You will lose some power provided by CSS. In return you gain something that is simpler to understand. It is also harder to break.

CSS Modules strike a balance between a conventional approach and React specific approaches. Even though it's a newcomer, it shows a lot of promise. The biggest benefit seems to be that it doesn't lose too much in the process. It's a nice step forward from what has been commonly used.

There are no best practices yet, and we are still figuring out the best ways to do this in React. You will likely have to do some experimentation of your own to figure out what ways fit your use case the best.



# 17. Structuring React Projects

React doesn't enforce any particular project structure. The good thing about this is that it allows you to make up a structure to suit your needs. The bad thing is that it is not possible to provide you an ideal structure that would work for every project. Instead, I'm going to give you some inspiration you can use to think about structure.

## 17.1 Directory per Concept

Our Kanban application has a somewhat flat structure:

```
├─ actions
|   └─ LaneActions.js
|   └─ NoteActions.js
├─ components
|   └─ App.jsx
|   └─ Editable.jsx
|   └─ Lane.jsx
|   └─ Lanes.jsx
|   └─ Note.jsx
|   └─ Notes.jsx
├─ constants
|   └─ itemTypes.js
├─ index.jsx
├─ libs
|   └─ alt.js
|   └─ persist.js
|   └─ storage.js
├─ main.css
└─ stores
    └─ LaneStore.js
    └─ NoteStore.js
```

It's enough for this purpose, but there are some interesting alternatives around:

- File per concept - Perfect for small prototypes. You can split this up as you get more serious with your application.

- Directory per component - It is possible to push components to directories of their own. Even though this is a heavier approach, there are some interesting advantages as we'll see soon.
- Directory per view - This approach becomes relevant once you want to introduce routing to your application.

There are more alternatives but these cover some of the common cases. There is always room for adjustment based on the needs of your application.

## 17.2 Directory per Component

If we split our components to directories of their own, we could end up with something like this:

```
├─ actions
│   └─ LaneActions.js
│   └─ NoteActions.js
├─ components
│   └─ App
│       └─ App.jsx
│       └─ app.css
│       └─ app_test.jsx
│       └─ index.js
│   └─ Editable
│       └─ Editable.jsx
│       └─ editable.css
│       └─ editable_test.jsx
│       └─ index.js
...
└─ index.js
├─ constants
│   └─ itemTypes.js
├─ index.jsx
├─ libs
│   └─ alt.js
│   └─ persist.js
│   └─ storage.js
├─ main.css
└─ stores
    └─ LaneStore.js
    └─ NoteStore.js
```

Compared to our current solution, this would be heavier. The *index.js* files are there to provide easy entry points for components. Even though they add noise, they simplify imports.

There are some interesting benefits in this approach, though:

- We can leverage technology, such as CSS Modules, for styling each component separately.
- Given each component is a little “package” of its own now, it would be easier to extract them from the project. You could push generic components elsewhere and consume them across multiple applications.
- We can define unit tests at component level. The approach encourages you to test. We can still have higher level tests around at the root level of the application just like earlier.

It could be interesting to try to push actions and stores to components as well. Or they could follow a similar directory scheme. The benefit of this is that it would allow you to define unit tests in a similar manner.

This setup isn’t enough when you want to add multiple views to the application. Something else is needed to support that.



[gajus/create-index](https://github.com/gajus/create-index)<sup>1</sup> is able to generate the *index.js* files automatically as you develop.

## 17.3 Directory per View

Multiple views bring challenges of their own. First of all, you’ll need to define a routing scheme. [react-router](https://github.com/reactjs/react-router)<sup>2</sup> is a popular alternative for this purpose. In addition to a routing scheme, you’ll need to define what to display on each view. You could have separate views for the home page of the application, registration, Kanban board, and so on, matching each route.

These requirements mean new concepts need to be introduced to the structure. One way to deal with routing is to push it to a Routes component that coordinates which view is displayed at any given time based on the current route. Instead of App we would have just multiple views instead. Here’s what a possible structure could look like:

---

<sup>1</sup><https://github.com/gajus/create-index>

<sup>2</sup><https://github.com/reactjs/react-router>

```
├── components
│   ├── Note
│   │   ├── Note.jsx
│   │   ├── index.js
│   │   ├── note.css
│   │   └── note_test.jsx
│   ├── Routes
│   │   ├── Routes.jsx
│   │   ├── index.js
│   │   └── routes_test.jsx
│   └── index.js
...
├── index.jsx
├── main.css
└── views
    ├── Home
    │   ├── Home.jsx
    │   ├── home.css
    │   ├── home_test.jsx
    │   └── index.js
    ├── Register
    │   ├── Register.jsx
    │   ├── index.js
    │   ├── register.css
    │   └── register_test.jsx
    └── index.js
```

The idea is the same as earlier. This time around we have more parts to coordinate. The application starts from `index.jsx` which will trigger `Routes` that in turn chooses some view to display. After that it's the flow we've gotten used to.

This structure can scale further, but even it has its limits. Once your project begins to grow, you might want to introduce new concepts to it. It could be natural to introduce a concept, such as “feature”, between the views and the components.

For example, you might have a fancy `LoginModal` that is displayed on certain views if the session of the user has timed out. It would be composed of lower level components. Again, common features could be pushed out of the project itself into packages of their own as you see potential for reuse.

## 17.4 Conclusion

There is no single right way to structure your project with React. That said, it is one of those aspects that is worth thinking about. Figuring out a structure that serves you well is worth it. A clear structure helps in the maintenance effort and makes your project more understandable to others.

You can evolve the structure as you go. Too heavy structure early on might just slow you down. As the project evolves, so should its structure. It's one of those things that's worth thinking about given it affects development so much.

# Appendices

As not everything that's worth discussing fits a book like this, I've compiled related material into brief appendices. These support the main material and explain certain topics, such as language features, in greater detail. There are also troubleshooting tips in the end.

# Language Features

ES6 (or ES2015) was arguably the biggest change to JavaScript in a long time. As a result, we received a wide variety of new functionality. The purpose of this appendix is to illustrate the features used in the book in isolation to make it clearer to understand how they work. Rather than going through [the entire specification](http://www.ecma-international.org/ecma-262/6.0/index.html)<sup>3</sup>, I will just focus on the subset of features used in the book.

## Modules

ES6 introduced proper module declarations. Earlier, this was somewhat ad hoc and we used formats, such as AMD or CommonJS. ES6 module declarations are statically analyzable. This is highly useful for tool authors. Effectively, this means we can gain features like *tree shaking*. This allows the tooling to skip unused code easily simply by analyzing the import structure.

### import and export for Single

To give you an example of exporting directly through a module, consider below:

**persist.js**

```
import makeFinalStore from 'alt-utils/lib/makeFinalStore';

export default function(alt, storage, storeName) {
  ...
}
```

**index.js**

```
import persist from './persist';

...
```

### import and export for Multiple

Sometimes it can be useful to use modules as a namespace for multiple functions:

**math.js**

---

<sup>3</sup><http://www.ecma-international.org/ecma-262/6.0/index.html>

```
export function add(a, b) {  
  return a + b;  
}  
  
export function multiply(a, b) {  
  return a * b;  
}  
  
export function square(a) {  
  return a * a;  
}
```

Alternatively we could write the module in a form like this:

**math.js**

```
const add = (a, b) => a + b;  
const multiple = (a, b) => a * b;  
  
// You can omit ()'s with a single parameter if you want.  
const square = a => a * a;  
  
export {add, multiple};  
  
// Equivalent to  
//export {add: add, multiple: multiple};
```

The example leverages the *fat arrow syntax* and the *property value shorthand*.

This definition can be consumed through an import like this:

**index.js**

```
import {add} from './math';  
  
// Alternatively we could bind the math methods to a key  
// import * as math from './math';  
// math.add, math.multiply, ...  
  
...
```

Especially `export default` is useful if you prefer to keep your modules focused. The `persist` function is an example of such. Regular `export` is useful for collecting multiple functions below the same umbrella.





Given the ES6 module syntax is statically analyzable, it enables tooling such as [analyze-es6-modules](https://www.npmjs.com/package/analyze-es6-modules)<sup>4</sup>.

## Aliasing Imports

Sometimes it can be handy to alias imports. Example:

```
import {actions as TodoActions} from '../actions/todo'
```

...

as allows you to avoid naming conflicts.

## Webpack `resolve.alias`

Bundlers, such as Webpack, can provide some features beyond this. You could define a `resolve.alias` for some of your module directories for example. This would allow you to use an import, such as `import persist from 'libs/persist'`, regardless of where you import. A simple `resolve.alias` could look like this:

```
...
resolve: {
  alias: {
    libs: path.join(__dirname, 'libs')
  }
}
```

The official documentation describes [possible variants](https://webpack.github.io/docs/configuration.html#resolve-alias)<sup>5</sup> in fuller detail.

## Classes

Unlike many other languages out there, JavaScript uses prototype based inheritance instead of class based one. Both approaches have their merits. In fact, you can mimic a class based model through a prototype based one. ES6 classes are about providing syntactical sugar above the basic mechanisms of JavaScript. Internally it still uses the same old system. It just looks a little different to the programmer.

These days React supports class based component definitions. Not all agree that it's a good thing. That said, the definition can be quite neat as long as you don't abuse it. To give you a simple example, consider the code below:

---

<sup>4</sup><https://www.npmjs.com/package/analyze-es6-modules>

<sup>5</sup><https://webpack.github.io/docs/configuration.html#resolve-alias>

```

import React from 'react';

export default class App extends React.Component {
  constructor(props) {
    super(props);

    // This is a regular property outside of React's machinery.
    // If you don't need to trigger render() when it's changed,
    // this can work.
    this.privateProperty = 'private';

    // React specific state. Alter this through `this.setState`. That
    // will call `render()` eventually.
    this.state = {
      name: 'Class demo'
    };
  }
  render() {
    // Use the properties somehow.
    const privateProperty = this.privateProperty;
    const name = this.state.name
    const notes = this.props.notes;

    ...
  }
}

```

Perhaps the biggest advantage of the class based approach is the fact that it cuts down some complexity, especially when it comes to React lifecycle methods. It is important to note that class methods won't get by default, though! This is why the book relies on an experimental feature known as property initializers.

## Classes and Modules

As stated above, the ES6 modules allow export and import single and multiple objects, functions, or even classes. In the latter, you can use `export default class` to export an anonymous class or export multiple classes from the same module using `export class className`.

To export and import a single class you can use `export default class` to export an anonymous class and call it whatever you want at import time:

**Note.jsx**

```
export default class extends React.Component { ... };
```

### Notes.jsx

```
import Note from './Note.jsx';  
...
```

Or use `export class className` to export several named classes from a single module:

### Components.jsx

```
export class Note extends React.Component { ... };  
  
export class Notes extends React.Component { ... };
```

### App.jsx

```
import Notes from './Components.jsx';  
import Note from './Components.jsx';  
  
...
```

It is recommended to keep your classes separated in different modules.

## Class Properties and Property Initializers

ES6 classes won't bind their methods by default. This can be problematic sometimes, as you still may want to be able to access the instance properties. Experimental features known as [class properties and property initializers](#)<sup>6</sup> solve this problem. Without them, we might write something like this:

---

<sup>6</sup><https://github.com/jeffmo/es-class-static-properties-and-fields>

```
import React from 'react';

class App extends React.Component {
  constructor(props) {
    super(props);

    this.renderNote = this.renderNote.bind(this);
  }
  render() {
    // Use `renderNote` here somehow.
    ...

    return this.renderNote();
  }
  renderNote() {
    // Given renderNote was bound, we can access `this` as expected
    return <div>{this.props.note}</div>;
  }
}
App.propTypes = {
  value: React.PropTypes.string
};
App.defaultProps = {
  value: ''
};

export default App;
```

Using class properties and property initializers we could write something tidier instead:

```
import React from 'react';

export default class App extends React.Component {
  // propTypes definition through static class properties
  static propTypes = {
    value: React.PropTypes.string
  }
  static defaultProps = {
    value: ''
  }
  render() {
    // Use `renderNote` here somehow.
  }
}
```

```

    ...

    return this.renderNote();
  }
  // Property initializer gets rid of the `bind`
  renderNote = () => {
    // Given renderNote was bound, we can access `this` as expected
    return <div>{this.props.note}</div>;
  }
}

```

Now that we've pushed the declaration to method level, the code reads better. I decided to use the feature in this book primarily for this reason. There is simply less to worry about.

## Functions

Traditionally, JavaScript has been very flexible with its functions. To give you a better idea, see the implementation of `map` below:

```

function map(cb, values) {
  var ret = [];
  var i, len;

  for(i = 0, len = values.length; i < len; i++) {
    ret.push(cb(values[i]));
  }

  return ret;
}

map(function(v) {
  return v * 2;
}, [34, 2, 5]); // yields [68, 4, 10]

```

In ES6 we could write it as follows:

```
function map(cb, values) {
  const ret = [];
  const i, len;

  for(i = 0, len = values.length; i < len; i++) {
    ret.push(cb(values[i]));
  }

  return ret;
}

map((v) => v * 2, [34, 2, 5]); // yields [68, 4, 10]
```

The implementation of `map` is more or less the same still. The interesting bit is at the way we call it. Especially that `(v) => v * 2` part is intriguing. Rather than having to write `function` everywhere, the fat arrow syntax provides us a handy little shorthand. To give you further examples of usage, consider below:

```
// These are the same
v => v * 2;
(v) => v * 2; // I prefer this variant for short functions
(v) => { // Use this if you need multiple statements
  return v * 2;
}

// We can bind these to a variable
const double = (v) => v * 2;

console.log(double(2));

// If you want to use a shorthand and return an object,
// you need to wrap the object.
v => ({
  foo: 'bar'
});
```

## Arrow Function Context

Arrow functions are special in that they don't have `this` at all. Rather, `this` will point at the caller object scope. Consider the example below:

```
var obj = {
  context: function() {
    return this;
  },
  name: 'demo object 1'
};

var obj2 = {
  context: () => this,
  name: 'demo object 2'
};

console.log(obj.context()); // { context: [Function], name: 'demo object 1' }
console.log(obj2.context()); // {} in Node.js, Window in browser
```

As you can notice in the snippet above, the anonymous function has a `this` pointing to the context function in the `obj` object. In other words, it is binding the scope of the caller object `obj` to the context function.

This happens because `this` doesn't point to the object scopes that contains it, but the caller object scopes, as you can see it in the next snippet of code:

```
console.log(obj.context.call(obj2)); // { context: [Function], name: 'demo object 2' }
```

The arrow function in the object `obj2` doesn't bind any object to its context, following the normal lexical scoping rules resolving the reference to the nearest outer scope. In this case it happens to be `Node.js global` object.

Even though the behavior might seem a little weird, it is actually useful. In the past, if you wanted to access parent context, you either needed to bind it or attach the parent context to a variable `var that = this;`. The introduction of the arrow function syntax has mitigated this problem.

## Function Parameters

Historically, dealing with function parameters has been somewhat limited. There are various hacks, such as `values = values || [];`, but they aren't particularly nice and they are prone to errors. For example, using `||` can cause problems with zeros. ES6 solves this problem by introducing default parameters. We can simply write function `map(cb, values=[])` now.

There is more to that and the default values can even depend on each other. You can also pass an arbitrary amount of parameters through function `map(cb, values...)`. In this case, you would call the function through `map(a => a * 2, 1, 2, 3, 4)`. The API might not be perfect for `map`, but it might make more sense in some other scenario.

There are also convenient means to extract values out of passed objects. This is highly useful with React component defined using the function syntax:

```
export default ({name}) => {  
  // ES6 string interpolation. Note the back-ticks!  
  return <div>{`Hello ${name}!`}</div>;  
};
```

## String Interpolation

Earlier, dealing with strings was somewhat painful in JavaScript. Usually you just ended up using a syntax like 'Hello' + name + '!'. Overloading + for this purpose wasn't perhaps the smartest move as it can lead to strange behavior due to type coercion. For example, 0 + ' world' would yield 0 world string as a result.

Besides being clearer, ES6 style string interpolation provides us multi-line strings. This is something the old syntax didn't support. Consider the examples below:

```
const hello = `Hello ${name}!`;  
const multiline = `  
multiple  
lines of  
awesomeness  
`;
```

The back-tick syntax may take a while to get used to, but it's powerful and less prone to mistakes.

## Destructuring

That ... is related to the idea of destructuring. For example, `const {lane, ...props} = this.props;` would extract lane out of this.props while the rest of the object would go to props. This object based syntax is still experimental. ES6 specifies an official way to perform the same for arrays like this:

```
const [lane, ...rest] = ['foo', 'bar', 'baz'];  
  
console.log(lane, rest); // 'foo', ['bar', 'baz']
```

The spread operator (...) is useful for concatenating. You see syntax like this in Redux examples often. They rely on experimental [Object rest/spread syntax](https://github.com/sebmarkbage/ecmascript-rest-spread)<sup>7</sup>:

---

<sup>7</sup><https://github.com/sebmarkbage/ecmascript-rest-spread>



```
[...state, action.lane];

// This is equal to
state.concat([action.lane])
```

The same idea applies to React components:

```
...

render() {
  const {value, onEdit, ...props} = this.props;

  return <div {...props}>Spread demo</div>;
}

...
```



There are several gotchas related to the spread operator. Given it is *shallow* by default, it can lead to interesting behavior that might be unexpected. This is particularly true if you are trying to use it to clone an object using it. Josh Black discusses this problem in detail at his Medium post titled [Gotchas in ES2015+ Spread](https://medium.com/@joshblack/gotchas-in-es2015-spread-5db06dfb1e10)<sup>8</sup>.

## Object Initializers

In order to make it easier to work with objects, ES6 provides a variety of features just for this. To quote [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer)<sup>9</sup>, consider the examples below:

```
const a = 'demo';
const shorthand = {a}; // Same as {a: a}

// Shorthand methods
const o = {
  get property() {},
  set property(value) {},
  demo() {}
};
```

<sup>8</sup><https://medium.com/@joshblack/gotchas-in-es2015-spread-5db06dfb1e10>

<sup>9</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object\\_initializer](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer)

```
// Computed property names
const computed = {
  [a]: 'testing' // demo -> testing
};
```

## const, let, var

In JavaScript, variables are global by default. `var` binds them on *function level*. This is in contrast to many other languages that implement *block level* binding. ES6 introduces block level binding through `let`.

There's also support for `const`, which guarantees the reference to the variable itself cannot change. This doesn't mean, however, that you cannot modify the contents of the variable. So if you are pointing at an object, you are still allowed to tweak it!

I tend to favor to default to `const` whenever possible. If I need something mutable, `let` will do fine. It is hard to find any good use for `var` anymore as `const` and `let` cover the need in a more understandable manner. In fact, all of the book's code, apart from this appendix, relies on `const`. That just shows you how far you can get with it.

## Decorators

Given decorators are still an experimental feature and there's a lot to cover about them, there's an entire appendix dedicated to the topic. Read *Understanding Decorators* for more information.

## Conclusion

There's a lot more to ES6 and the upcoming specifications than this. If you want to understand the specification better, [ES6 Katas](http://es6katas.org/)<sup>10</sup> is a good starting point for learning more. Just having a good idea of the basics will take you far.

---

<sup>10</sup><http://es6katas.org/>

# Understanding Decorators

If you have used languages, such as Java or Python before, you might be familiar with the idea. Decorators are syntactic sugar that allow us to wrap and annotate classes and functions. In their [current proposal](#)<sup>11</sup> (stage 1) only class and method level wrapping is supported. Functions may become supported later on.

In Babel 6 you can enable this behavior through [babel-plugin-syntax-decorators](#)<sup>12</sup> and [babel-plugin-transform-decorators-legacy](#)<sup>13</sup> plugins. The former provides syntax level support whereas the latter gives the type of behavior we are going to discuss here.

The greatest benefit of decorators is that they allow us to wrap behavior into simple, reusable chunks while cutting down the amount of noise. It is definitely possible to code without them. They just make certain tasks neater, as we saw with drag and drop related annotations.

## Implementing a Logging Decorator

Sometimes, it is useful to know how methods are being called. You could of course attach `console.log` there but it's more fun to implement `@log`. That's a more controllable way to deal with it. Consider the example below:

```
class Math {
  @log
  add(a, b) {
    return a + b;
  }
}

function log(target, name, descriptor) {
  var oldValue = descriptor.value;

  descriptor.value = function() {
    console.log(`Calling "${name}" with`, arguments);

    return oldValue.apply(null, arguments);
  };
}
```

---

<sup>11</sup><https://github.com/wycats/javascript-decorators>

<sup>12</sup><https://www.npmjs.com/package/babel-plugin-syntax-decorators>

<sup>13</sup><https://www.npmjs.com/package/babel-plugin-transform-decorators-legacy>

```
    return descriptor;
}

const math = new Math();

// passed parameters should get logged now
math.add(2, 4);
```

The idea is that our log decorator wraps the original function, triggers a `console.log`, and finally, calls it again while passing the original [arguments](#)<sup>14</sup> to it. Especially if you haven't seen arguments or apply before, it might seem a little strange.

apply can be thought as another way to invoke a function while passing its context (`this`) and parameters as an array. `arguments` receives function parameters implicitly so it's ideal for this case.

This logger could be pushed to a separate module. After that, we could use it across our application whenever we want to log some methods. Once implemented decorators become powerful building blocks.

The decorator receives three parameters:

- `target` maps to the instance of the class.
- `name` contains the name of the method being decorated.
- `descriptor` is the most interesting piece as it allows us to annotate the method and manipulate its behavior. It could look like this:

```
const descriptor = {
  value: () => {...},
  enumerable: false,
  configurable: true,
  writable: true
};
```

As you saw above, `value` makes it possible to shape the behavior. The rest allows you to modify behavior on method level. For instance, a `@readonly` decorator could limit access. `@memoize` is another interesting example as that allows you to implement easy caching for methods.

---

<sup>14</sup><https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/arguments>

## Implementing @connect

@connect will wrap our component in another component. That, in turn, will deal with the connection logic (listen/unlisten/setState). It will maintain the store state internally and then pass it to the child component that we are wrapping. During this process, it will pass the state through props. The implementation below illustrates the idea:

app/decorators/connect.js

```
import React from 'react';

const connect = (Component, store) => {
  return class Connect extends React.Component {
    constructor(props) {
      super(props);

      this.storeChanged = this.storeChanged.bind(this);
      this.state = store.getState();

      store.listen(this.storeChanged);
    }
    componentWillUnmount() {
      store.unlisten(this.storeChanged);
    }
    storeChanged() {
      this.setState(store.getState());
    }
    render() {
      return <Component {...this.props} {...this.state} />;
    }
  };
};

export default (store) => {
  return (target) => connect(target, store);
};
```

Can you see the wrapping idea? Our decorator tracks store state. After that, it passes the state to the component contained through props.



... is known as a [spread operator](https://github.com/sebmarkbage/ecmascript-rest-spread)<sup>15</sup>. It expands the given object to separate key-value pairs, or props, as in this case.

<sup>15</sup><https://github.com/sebmarkbage/ecmascript-rest-spread>

You can connect the decorator with App like this:

**app/components/App.jsx**

```
...
import connect from '../decorators/connect';
...

@connect(NoteStore)
export default class App extends React.Component {
  render() {
    const notes = this.props.notes;

    ...
  }
  ...
}
```

Pushing the logic to a decorator allows us to keep our components simple. If we wanted to add more stores to the system and connect them to components, it would be trivial now. Even better, we could connect multiple stores to a single component easily.

## Decorator Ideas

We can build new decorators for various functionalities, such as undo, in this manner. They allow us to keep our components tidy and push common logic elsewhere out of sight. Well designed decorators can be used across projects.

### Alt's @connectToStores

Alt provides a similar decorator known as `@connectToStores`. It relies on static methods. Rather than normal methods that are bound to a specific instance, these are bound on class level. This means you can call them through the class itself (i.e., `App.getStores()`). The example below shows how we might integrate `@connectToStores` into our application.

```
...
import connectToStores from 'alt-utils/lib/connectToStores';

@connectToStores
export default class App extends React.Component {
  static getStores(props) {
    return [NoteStore];
  };
  static getPropsFromStores(props) {
    return NoteStore.getState();
  };
  ...
}
```

This more verbose approach is roughly equivalent to our implementation. It actually does more as it allows you to connect to multiple stores at once. It also provides more control over the way you can shape store state to props.

## Conclusion

Even though still a little experimental, decorators provide nice means to push logic where it belongs. Better yet, they provide us a degree of reusability while keeping our components neat and tidy.

# Troubleshooting

I've tried to cover some common issues here. This chapter will be expanded as common issues are found.

## EPEERINVALID

It is possible you may see a message like this:

```
npm WARN package.json kanban-app@0.0.0 No repository field.
npm WARN package.json kanban-app@0.0.0 No README data
npm WARN peerDependencies The peer dependency eslint@0.21 - 0.23 included from e\
slint-loader will no
npm WARN peerDependencies longer be automatically installed to fulfill the peerD\
ependency
npm WARN peerDependencies in npm 3+. Your application will need to depend on it \
explicitly.
```

...

```
npm ERR! Darwin 14.3.0
npm ERR! argv "node" "/usr/local/bin/npm" "i"
npm ERR! node v0.10.38
npm ERR! npm v2.11.0
npm ERR! code EPEERINVALID
```

```
npm ERR! peerinvalid The package eslint does not satisfy its siblings' peerDepen\
dencies requirements!
npm ERR! peerinvalid Peer eslint-plugin-react@2.5.2 wants eslint@>=0.8.0
npm ERR! peerinvalid Peer eslint-loader@0.14.0 wants eslint@0.21 - 0.23
```

```
npm ERR! Please include the following file with any support request:
```

...

In human terms, it means that some package, `eslint-loader` in this case, has a too strict `peerDependency` requirement. Our project has a newer version installed already. Given the required peer dependency is older than our version, we get this particular error.

There are a couple of ways to work around this:



1. Report the glitch to the package author and hope the version range will be expanded.
2. Resolve the conflict by settling to a version that satisfies the peer dependency. In this case, we could pin `eslint` to version `0.23` (`"eslint": "0.23"`), and everyone should be happy.
3. Fork the package, fix the version range, and point at your custom version. In this case, you would have a `"<package>": "<github user>/<project>#<reference>"` kind of declaration for your dependencies.



Note that peer dependencies are dealt with differently starting from npm 3. After that version, it's up to the package consumer (i.e., you) to deal with it. This particular error will go away.

## Warning: setState(...): Cannot update during an existing state transition

You might get this warning while using React. An easy way to end up getting it is to trigger `setState()` within a method, such as `render()`. Sometimes this can happen indirectly. One way to cause the warning is call a method instead of binding it. Example: `<input onKeyDown={this.checkEnter()} />`. Assuming `this.checkEnter` uses `setState()`, this code will fail. Instead, you should use `<input onKeyDown={this.checkEnter} />` as that will bind the method correctly without calling it.

## Warning: React attempted to reuse markup in a container but the checksum was invalid

You can get this warning through multiple means. Common causes below:

- You tried to mount React multiple times to the same container. Check your script loading and make sure your application is loaded only once.
- The existing markup on your template doesn't match the one rendered by React. This can happen especially if you are rendering the initial markup through a server.

## Module parse failed

When using Webpack, an error like this might come up:

```
ERROR in ./app/components/Demo.jsx
Module parse failed: .../app/components/Demo.jsx Line 16: Unexpected token <
```

This means there is something preventing Webpack to interpret the file correctly. You should check out your loader configuration carefully. Make sure the right loaders are applied to the right files. If you are using `include`, you should verify that the file is included within `include` paths.

## Project Fails to Compile

Even though everything should work in theory, sometimes version ranges can bite you, despite SemVer. If some core package breaks, let's say `babel`, and you happen to execute `npm i` in an unfortunate time, you may end up with a project that doesn't compile.

A good first step is to execute `npm update`. This will check out your dependencies and pull the newest matching versions into your SemVer declarations. If this doesn't fix the issue, you can try to nuke `node_modules` (`rm -rf node_modules`) from the project directory and reinstall the dependencies (`npm i`). Alternatively you can try to explicitly pin some of your dependencies to specific versions.

Often you are not alone with your problem. Therefore, it may be worth your while to check out the project issue trackers to see what's going on. You can likely find a good workaround or a proposed fix there. These issues tend to get fixed fast for popular projects.

In a production environment, it may be preferable to lock production dependencies using `npm shrinkwrap`. [The official documentation](https://docs.npmjs.com/cli/shrinkwrap)<sup>16</sup> goes into more detail on the topic.

---

<sup>16</sup><https://docs.npmjs.com/cli/shrinkwrap>