

SurviveJS - Webpack and React

From apprentice to master



Juho Vepsäläinen

SurviveJS - Webpack and React

From apprentice to master

Juho Vepsäläinen

This book is for sale at http://leanpub.com/survivejs_webpack_react

This version was published on 2015-12-13



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-NonCommercial-NoDerivs 3.0 Unported License](#)

Contents

Introduction	i
What is Webpack?	i
What is React?	i
How is This Book Organized?	ii
What is Kanban?	iii
Who is This Book for?	iii
Extra Material	iii
Getting Support	iv
Announcements	iv
Acknowledgments	v

I Setting Up Webpack **1**

1. Webpack Compared	2
1.1 Make	2
1.2 Grunt	4
1.3 Gulp	5
1.4 Browserify	7
1.5 Webpack	8
1.6 JSPM	10
1.7 Why Use Webpack?	10
1.8 Module Formats Supported by Webpack	11
1.9 Conclusion	13
2. Developing with Webpack	14
2.1 Setting Up the Project	14
2.2 Installing Webpack	15
2.3 Directory Structure	15
2.4 Setting Up Assets	16
2.5 Setting Up Webpack Configuration	16
2.6 Setting Up <i>webpack-dev-server</i>	18
2.7 Refreshing CSS	21
2.8 Making the Configuration Extensible	23

CONTENTS

2.9	Linting the Project	26
2.10	Conclusion	26
3.	Webpack and React	27
3.1	What is React?	27
3.2	Babel	30
3.3	Developing the First React View	33
3.4	Activating Hot Loading for Development	36
3.5	React Component Styles	38
3.6	Conclusion	38
II	Developing Kanban Application	39
4.	Implementing a Basic Note Application	40
4.1	Initial Data Model	40
4.2	On Ids	41
4.3	Connecting Data with App	41
4.4	Fixing Note	43
4.5	Extracting Notes	44
4.6	Pushing notes to the App state	46
4.7	Adding New Items to Notes list	48
4.8	Editing Notes	50
4.9	Removing Notes	55
4.10	Styling Notes	58
4.11	Understanding React Components	61
4.12	React Component Conventions	63
4.13	Conclusion	63
5.	React and Flux	64
5.1	Introduction to Flux	64
5.2	Porting to Alt	66
5.3	Defining a Store for Notes	67
5.4	Gluing It All Together	71
5.5	Implementing Persistency over localStorage	73
5.6	Using the AltContainer	77
5.7	Dispatching in Alt	78
5.8	Relay?	78
5.9	Conclusion	78
6.	From Notes to Kanban	79
6.1	Extracting Lanes	79
6.2	Modeling Lane	82
6.3	Making Lanes Responsible of Notes	84

CONTENTS

6.4	Implementing Edit/Remove for Lane	90
6.5	Styling Kanban Board	96
6.6	On Namespacing Components	98
6.7	Conclusion	99
7.	Implementing Drag and Drop	100
7.1	Setting Up React DnD	100
7.2	Preparing Notes to Be Sorted	101
7.3	Allowing Notes to Be Dragged	102
7.4	Developing onMove API for Notes	105
7.5	Adding Action and Store Method for Moving	106
7.6	Implementing Note Drag and Drop Logic	108
7.7	Dragging Notes to an Empty Lanes	110
7.8	Conclusion	114
8.	Building Kanban	115
8.1	Setting Up a Build Target	115
8.2	Optimizing Build Size	116
8.3	Splitting app and vendor Bundles	119
8.4	Cleaning the Build	123
8.5	Separating CSS	124
8.6	Analyzing Build Statistics	127
8.7	Deployment	128
8.8	Conclusion	130
9.	Testing React	131
9.1	Levels of Testing	131
9.2	Setting Up Webpack	133
9.3	Testing Kanban Components	140
9.4	Testing Kanban Stores	148
9.5	Conclusion	151
10.	Typing with React	152
10.1	propTypes and defaultProps	152
10.2	Typing Kanban	154
10.3	Type Checking with Flow	158
10.4	Converting propTypes to Flow Checks	161
10.5	Babel Typecheck	161
10.6	TypeScript	162
10.7	Conclusion	162

III	Advanced Techniques	163
11.	Linting in Webpack	164
11.1	Brief History of Linting in JavaScript	164
11.2	Webpack and JSHint	164
11.3	Setting Up ESLint	166
11.4	Customizing ESLint	170
11.5	Linting CSS	173
11.6	Checking JavaScript Style with JSCS	175
11.7	EditorConfig	177
11.8	Conclusion	177
12.	Authoring Libraries	178
12.1	Anatomy of a npm Package	178
12.2	Understanding <i>package.json</i>	179
12.3	npm Workflow	181
12.4	Library Formats	185
12.5	npm Lifecycle Hooks	187
12.6	Keeping Dependencies Up to Date	188
12.7	Sharing Authorship	189
12.8	Conclusion	189
13.	Styling React	190
13.1	Old School Styling	190
13.2	CSS Methodologies	191
13.3	Less, Sass, Stylus, PostCSS, cssnext	193
13.4	React Based Approaches	198
13.5	CSS Modules	203
13.6	Conclusion	205
	Understanding Decorators	206
	Implementing Logging Decorator	206
	Implementing @connect	207
	Decorator Ideas	209
	Conclusion	210
	Troubleshooting	211
	EPEERINVALID	211
	Module parse failed	212
	Project Fails to Compile	212

Introduction

Front-end development moves forward fast. In this book we'll discuss [Webpack](https://webpack.github.io/)¹ and [React](https://facebook.github.io/react/)². Combined, these tools allow you to build all sorts of web applications swiftly. Knowledge of Webpack is useful beyond React. Understanding React will allow you to see the alternatives in a different light.

What is Webpack?

Web browsers have been designed to consume HTML, JavaScript, and CSS. The simplest way to develop is simply to write files that the browser understands directly. The problem is that this becomes unwieldy eventually. This is particularly true when you are developing web applications.

There are multiple ways to approach this problem. You can start splitting up your JavaScript and CSS to separate files for example. You could load dependencies through `script` tags. Even though better, it is still a little problematic. If you want to use technologies that compile to these target formats, you will need to introduce preprocessing steps. Task runners, such as Grunt and Gulp, allow you to achieve this but even then you need to write a lot of configuration by hand.

Webpack takes another route. It allows you to treat your project as a dependency graph. You could have a `index.js` in your project that pulls in the dependencies the project needs through standard `import` statements. You can refer to your style files and other assets the same way.

Webpack does all the preprocessing for you and gives you the bundles you specify through configuration. This declarative approach is powerful but a little difficult to learn. Once you begin to understand how Webpack works, it becomes an indispensable tool. This book has been designed to get through that initial learning curve.

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.

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.

¹<https://webpack.github.io/>

²<https://facebook.github.io/react/>

Both framework and library based approaches have their merits. You may even use React with a framework so it's not an either-or proposition. Ideas introduced by React have influenced the development of the frameworks so you can find familiar concepts there. Most importantly it has helped us to understand how well component based thinking fits web applications.

How is This Book Organized?

This book will guide you through a small example project. After completing it, we discuss more theoretical aspects of web development. The project in question will be a small [Kanban](https://en.wikipedia.org/wiki/Kanban)³ application.

We will start by building a Webpack based configuration. After that we will develop a small clone of a famous [Todo application](http://todomvc.com/)⁴. We will generalize from there and put in place [Flux architecture](https://facebook.github.io/flux/docs/overview.html)⁵ within our application. We will apply some [Drag and Drop \(DnD\) magic](https://gaearon.github.io/react-dnd/)⁶ and start dragging things around. Finally we will get a production grade build done.

After that we have a couple of Leanpub exclusive chapters in which we discuss how to:

- Deal with typing in React.
- Test your components and logic.

The final part of the book focuses on the tooling. Through the chapters included you will learn to:

- Lint your code effectively using [ESLint](http://eslint.org/)⁷ and some other tools.
- Author libraries at [npm](https://www.npmjs.com/)⁸.
- Style React in various emerging ways.

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

⁴<http://todomvc.com/>

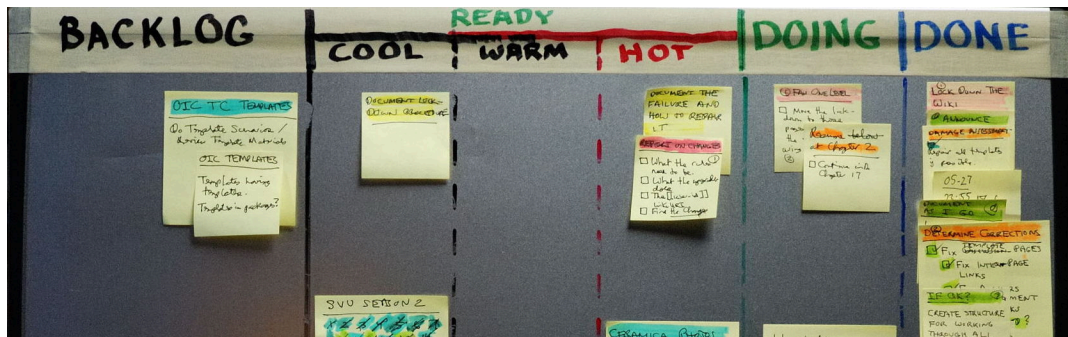
⁵<https://facebook.github.io/flux/docs/overview.html>

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

⁷<http://eslint.org/>

⁸<https://www.npmjs.com/>

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 as 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.

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.

As the system gets more sophisticated you can start applying concepts such as a limit on Work In Progress (WIP). 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.

A good example of Kanban in action on the web is [Trello](https://trello.com/)⁹. Sprintsly has open sourced their [React implementation of Kanban](https://github.com/sprintsly/sprintsly-kanban)¹⁰. Ours won't be as sophisticated, 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. If you know something about Webpack or React, that's great. By reading this book you will deepen your understanding of these tools.

Extra Material

The book content and source are available at [GitHub](https://github.com/survivejs/webpack_react)¹¹. This allows you to start from any chapter you want.

⁹<https://trello.com/>

¹⁰<https://github.com/sprintsly/sprintsly-kanban>

¹¹https://github.com/survivejs/webpack_react

You can also find alternative implementations of the application using [mobxobservable](#)¹², [Redux](#)¹³, and [Cerebral/Baobab](#)¹⁴. 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:

- [GitHub Issue Tracker](#)¹⁵
- [Gitter Chat](#)¹⁶
- Twitter - [@survivejs](#)¹⁷ or poke me through [@bebraw](#)¹⁸
- Email - info@survivejs.com¹⁹

If you post questions to Stack Overflow, tag them using `survivejs` so I will get notified of them.

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](#)²⁰
- [Twitter](#)²¹
- [Blog RSS](#)²²

Feel free to subscribe.

¹²<https://github.com/survivejs/mobobservable-demo>

¹³<https://github.com/survivejs/redux-demo>

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

¹⁵https://github.com/survivejs/webpack_react/issues

¹⁶https://gitter.im/survivejs/webpack_react

¹⁷<https://twitter.com/survivejs>

¹⁸<https://twitter.com/bebraw>

¹⁹<mailto:info@survivejs.com>

²⁰<http://eepurl.com/bth1v5>

²¹<https://twitter.com/survivejs>

²²<http://survivejs.com/atom.xml>

Acknowledgments

Big thanks to [Christian Alfoni](http://www.christianalfoni.com/)²³ for starting the [react-webpack-cookbook](https://github.com/christianalfoni/react-webpack-cookbook)²⁴ with me. That work eventually lead to this book.

The book wouldn't be half as good as it is without patient editing and feedback by my editor [Jesús Rodríguez Rodríguez](https://github.com/Foxandxss)²⁵. Thank you.

Special thanks to Steve Piercy for numerous contributions. Thanks to [Prospect One](http://prospectone.pl/)²⁶ 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 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, and Diego Toro. If I'm missing your name, I might have forgotten to add it.

²³<http://www.christianalfoni.com/>

²⁴<https://github.com/christianalfoni/react-webpack-cookbook>

²⁵<https://github.com/Foxandxss>

²⁶<http://prospectone.pl/>

I Setting Up Webpack

Webpack is a powerful module bundler. It hides a lot of power behind configuration. Once you understand its fundamentals, it becomes much easier to use this power. Initially it can be a confusing tool to adopt, but once you break the ice it gets better.

In this part we will develop a Webpack based project configuration that provides a solid foundation for the Kanban project and React development overall.

1. Webpack Compared

You can understand better why Webpack's approach is powerful by putting it into historical context. Back in the day, it was enough just to concatenate some scripts together. Times have changed, though, and now distributing your JavaScript code can be a complex endeavor.

This problem has escalated with the rise of single page applications (SPAs). They tend to rely on numerous hefty libraries. The last thing you want to do is to load them all at once. There are better solutions, and Webpack works with many of those.

The popularity of Node.js and [npm](https://www.npmjs.com/)¹, the Node.js package manager, provides more context. Before npm it was difficult to consume dependencies. Now that npm has become popular for front-end development, the situation has changed. Now we have nice ways to manage the dependencies of our projects.

Historically speaking there have been many build systems. [Make](#)² is perhaps the best known, and is still a viable option. To make things easier, specialized *task runners*, such as [Grunt](#)³ and [Gulp](#)⁴ appeared. Plugins available through npm made both task runners powerful.

Task runners are great tools on a high level. They allow you to perform operations in a cross-platform manner. The problems begin when you need to splice various assets together and produce bundles. This is the reason we have *bundlers*, such as [Browserify](#)⁵ or Webpack.

Continuing further on this path, [JSPM](#)⁶ pushes package management directly to the browser. It relies on [System.js](#)⁷, a dynamic module loader. Unlike Browserify and Webpack, it skips the bundling step altogether during development. You can generate a production bundle using it, however. Glen Maddern goes into good detail at his [video about JSPM](#)⁸.

1.1 Make

You could say Make goes way back. It was initially released in 1977. Even though it's an old tool, it has remained relevant. Make allows you to write separate tasks for various purposes. For instance, you might have separate tasks for creating a production build, minifying your JavaScript or running tests. You can find the same idea in many other tools.

¹<https://www.npmjs.com/>

²https://en.wikipedia.org/wiki/Make_%28software%29

³<http://gruntjs.com/>

⁴<http://gulpjs.com/>

⁵<http://browserify.org/>

⁶<http://jspm.io/>

⁷<https://github.com/systemjs/systemjs>

⁸<https://www.youtube.com/watch?t=33&v=iukBMY4apvI>

Even though Make is mostly used with C projects, it's not tied to it in any way. James Coglan discusses in detail [how to use Make with JavaScript](https://blog.jcoglan.com/2014/02/05/building-javascript-projects-with-make/)⁹. Consider the abbreviated code based on James' post below:

Makefile

```
PATH := node_modules/.bin:$(PATH)
SHELL := /bin/bash

source_files := $(wildcard lib/*.coffee)
build_files  := $(source_files:%.coffee=build/%.js)
app_bundle   := build/app.js
spec_coffee  := $(wildcard spec/*.coffee)
spec_js      := $(spec_coffee:%.coffee=build/%.js)

libraries    := vendor/jquery.js

.PHONY: all clean test

all: $(app_bundle)

build/%.js: %.coffee
    coffee -co $(dir $@) $<

$(app_bundle): $(libraries) $(build_files)
    uglifyjs -c -m $@ $^

test: $(app_bundle) $(spec_js)
    phantomjs phantom.js

clean:
    rm -rf build
```

With Make, you model your tasks using Make-specific syntax and terminal commands. This allows it to integrate easily with Webpack.

⁹<https://blog.jcoglan.com/2014/02/05/building-javascript-projects-with-make/>

1.2 Grunt



GRUNT

The JavaScript Task Runner

Grunt

Grunt went mainstream before Gulp. Its plugin architecture, especially, contributed towards its popularity. At the same time, this architecture is the Achilles' heel of Grunt. I know from experience that you **don't** want to end up having to maintain a 300-line `Gruntfile`. Here's an example from [Grunt documentation](http://gruntjs.com/sample-gruntfile)¹⁰:

```
module.exports = function(grunt) {
  grunt.initConfig({
    jshint: {
      files: ['Gruntfile.js', 'src/**/*.js', 'test/**/*.js'],
      options: {
        globals: {
          jQuery: true
        }
      }
    },
    watch: {
      files: ['<%= jshint.files %>'],
      tasks: ['jshint']
    }
  });

  grunt.loadNpmTasks('grunt-contrib-jshint');
  grunt.loadNpmTasks('grunt-contrib-watch');
```

¹⁰<http://gruntjs.com/sample-gruntfile>

```
grunt.registerTask('default', ['jshint']);  
};
```

In this sample, we define two basic tasks related to *jshint*, a linting tool that locates possible problem spots in your JavaScript source code. We have a standalone task for running *jshint*. Also, we have a watcher based task. When we run Grunt, we'll get warnings in real-time in our terminal as we edit and save our source code.

In practice, you would have many small tasks for various purposes, such as building the project. The example shows how these tasks are constructed. An important part of the power of Grunt is that it hides a lot of the wiring from you. Taken too far, this can get problematic, though. It can become hard to thoroughly understand what's going on under the hood.



Note that the [grunt-webpack](https://www.npmjs.com/package/grunt-webpack)¹¹ plugin allows you to use Webpack in a Grunt environment. You can leave the heavy lifting to Webpack.

1.3 Gulp



Gulp

Gulp takes a different approach. Instead of relying on configuration per plugin, you deal with actual code. Gulp builds on top of the tried and true concept of piping. If you are familiar with Unix, it's the same idea here. You simply have sources, filters, and sinks.

Sources match to files. Filters perform operations on sources (e.g., convert to JavaScript). Finally, the results get passed to sinks (e.g., your build directory). Here's a sample `Gulpfile` to give you a better idea of the approach, taken from the project's README. It has been abbreviated a bit:

¹¹<https://www.npmjs.com/package/grunt-webpack>


```
var gulp = require('gulp');
var coffee = require('gulp-coffee');
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');
var sourcemaps = require('gulp-sourcemaps');
var del = require('del');

var paths = {
  scripts: ['client/js/**/*.coffee', '!client/external/**/*.coffee']
};

// Not all tasks need to use streams
// A gulpfile is just another node program and you can use all packages available on npm
gulp.task('clean', function(cb) {
  // You can use multiple globbing patterns as you would with `gulp.src`
  del(['build'], cb);
});

gulp.task('scripts', ['clean'], function() {
  // Minify and copy all JavaScript (except vendor scripts)
  // with sourcemaps all the way down
  return gulp.src(paths.scripts)
    .pipe(sourcemaps.init())
    .pipe(coffee())
    .pipe(uglify())
    .pipe(concat('all.min.js'))
    .pipe(sourcemaps.write())
    .pipe(gulp.dest('build/js'));
});

// Rerun the task when a file changes
gulp.task('watch', function() {
  gulp.watch(paths.scripts, ['scripts']);
});

// The default task (called when you run `gulp` from CLI)
gulp.task('default', ['watch', 'scripts']);
```

Given the configuration is code, you can always just hack it if you run into troubles. You can wrap existing Node.js modules as Gulp plugins, and so on. Compared to Grunt, you have a clearer idea of what's going on. You still end up writing a lot of boilerplate for casual tasks, though. That is where some newer approaches come in.



[gulp-webpack¹²](#) allows you to use Webpack in a Gulp environment.

1.4 Browserify



Browserify

Dealing with JavaScript modules has always been a bit of a problem. The language itself actually didn't have the concept of modules till ES6. Ergo we have been stuck in the '90s when it comes to browser environments. Various solutions, including [AMD¹³](#), have been proposed.

In practice, it can be useful just to use CommonJS, the Node.js format, and let the tooling deal with the rest. The advantage is that you can often hook into npm and avoid reinventing the wheel.

[Browserify¹⁴](#) is one solution to the module problem. It provides a way to bundle CommonJS modules together. You can hook it up with Gulp. There are smaller transformation tools that allow you to move beyond the basic usage. For example, [watchify¹⁵](#) provides a file watcher that creates bundles for you during development. This will save some effort and no doubt is a good solution up to a point.

¹²<https://www.npmjs.com/package/gulp-webpack>

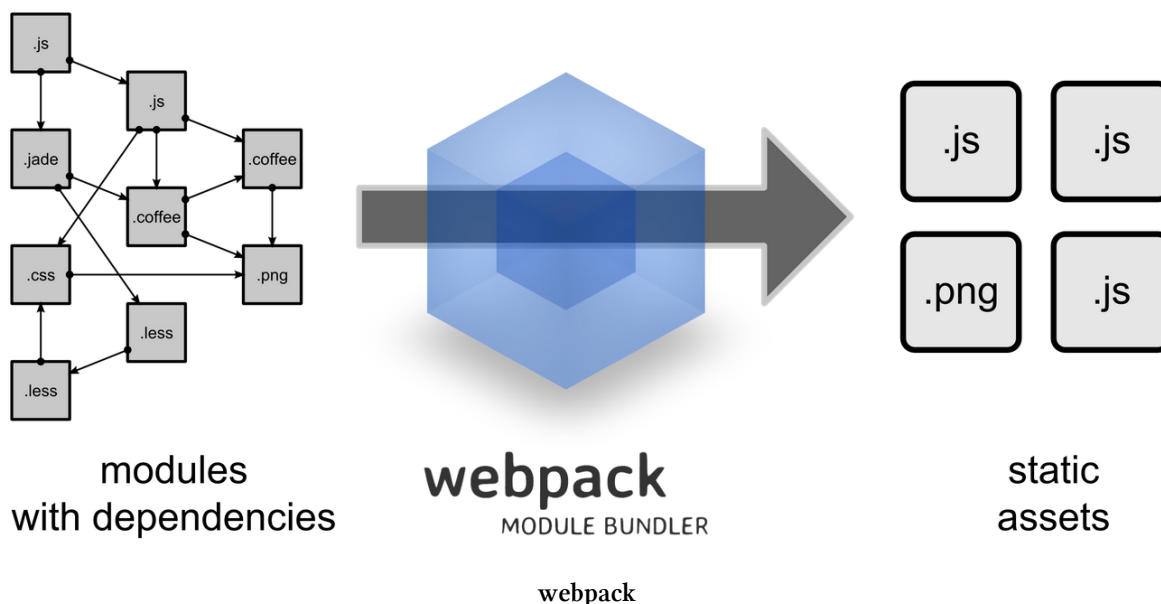
¹³<http://requirejs.org/docs/whyamd.html>

¹⁴<http://browserify.org/>

¹⁵<https://www.npmjs.com/package/watchify>

The Browserify ecosystem is composed of a lot of small modules. In this way, Browserify adheres to the Unix philosophy. Browserify is a little easier to adopt than Webpack, and is, in fact, a good alternative to it.

1.5 Webpack



You could say Webpack (or just *webpack*) takes a more monolithic approach than Browserify. You simply get more out of the box. Webpack extends `require` and allows you to customize its behavior using loaders. For example, `require('html!./file.html')` loads the contents of *file.html* and processes it through an HTML loader. It is a good idea to keep loader declarations such as this out of your source, though. Instead, use Webpack configuration to deal with it.

Webpack will traverse through the `require` statements of your project and will generate the bundles you want. You can even load your dependencies in a dynamic manner using a custom `require.ensure` statement. The loader mechanism works for CSS as well and `@import` is supported. There are also plugins for specific tasks, such as minification, localization, hot loading, and so on.

All this relies on configuration. It can be difficult to understand what's going on if you haven't seen it before. Fortunately there's certain logic involved. Here is a sample configuration adapted from [the official webpack tutorial](http://webpack.github.io/docs/tutorials/getting-started/)¹⁶:

webpack.config.js

¹⁶<http://webpack.github.io/docs/tutorials/getting-started/>

```
var webpack = require('webpack');

module.exports = {
  entry: './entry.js',
  output: {
    path: __dirname,
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.css$/,
        loaders: ['style', 'css']
      }
    ]
  },
  plugins: [
    new webpack.optimize.UglifyJsPlugin()
  ]
};
```

Given the configuration is written in JavaScript, it's quite malleable. As long as it's JavaScript, Webpack is fine with it.

The configuration model may make Webpack feel a bit opaque at times. It can be difficult to understand what it's doing. This is particularly true for more complicated cases. I have compiled [a webpack cookbook](https://christianalfoni.github.io/react-webpack-cookbook/)¹⁷ with Christian Alfoni that goes into more detail when it comes to specific problems.

¹⁷<https://christianalfoni.github.io/react-webpack-cookbook/>

1.6 JSPM



JSPM

Using JSPM is quite different than earlier tools. It comes with a little CLI tool of its own that is used to install new packages to the project, create a production bundle, and so on. It supports [SystemJS plugins](#)¹⁸ that allow you to load various formats to your project.

Given JSPM is still a young project, there might be rough spots. That said, it may be worth a look if you are adventurous. As you know by now, tooling tends to change quite often in front-end development, and JSPM is definitely a worthy contender.

1.7 Why Use Webpack?

Why would you use Webpack over tools like Gulp or Grunt? It's not an either-or proposition. Webpack deals with the difficult problem of bundling, but there's so much more. I picked up Webpack because of its support for hot module replacement (HMR). This is a feature used by [react-hot-loader](#)¹⁹. I will show you later how to set it up.

You might be familiar with tools such as [LiveReload](#)²⁰ or [Browsersync](#)²¹ already. These tools refresh the browser automatically as you make changes. HMR takes things one step further. In the case of React, it allows the application to maintain its state. This sounds simple, but it makes a big difference in practice.

Aside from the HMR feature, Webpack's bundling capabilities are extensive. It allows you to split bundles in various ways. You can even load them dynamically as your application gets executed. This sort of lazy loading comes in handy, especially for larger applications. You can load dependencies as you need them.

¹⁸<https://github.com/systemjs/systemjs#plugins>

¹⁹<https://github.com/gaearon/react-hot-loader>

²⁰<http://livereload.com/>

²¹<http://www.browsersync.io/>

With Webpack, you can easily inject a hash to each bundle name. This allows you to invalidate bundles on the client side as changes are made. Bundle splitting allows the client to reload only a small part of the data in the ideal case.

It is possible to achieve some of these tasks with other tools. The problem is that it would definitely take a lot more work to pull off. In Webpack, it's a matter of configuration. Note that HMR is available in Browserify via [livereactload](https://github.com/milankinen/livereactload)²², so it's not a feature that's exclusive to Webpack.

All these smaller features add up. Surprisingly, you can get many things done out of the box. And if you are missing something, there are loaders and plugins available that allow you to go further. Webpack comes with a significant learning curve. Even still, it's a tool worth learning, given it saves so much time and effort over the long term.

To get a better idea how it compares to some other tools, check out [the official comparison](https://webpack.github.io/docs/comparison.html)²³.

1.8 Module Formats Supported by Webpack

Webpack allows you to use different module formats, but under the hood they all work the same way.

CommonJS

If you have used Node.js, it is likely that you are familiar with CommonJS already. Here's a brief example:

```
var MyModule = require('./MyModule');

// export at module root
module.exports = function() { ... };

// alternatively, export individual functions
exports.hello = function() { ... };
```

ES6

ES6 is the format we all have been waiting for since 1995. As you can see, it resembles CommonJS a little bit and is quite clear!

²²<https://github.com/milankinen/livereactload>

²³<https://webpack.github.io/docs/comparison.html>

```
import MyModule from './MyModule.js';

// export at module root
export default function () { ... };

// or export as module function,
// you can have multiple of these per module
export function hello() {...};
```

AMD

AMD, or asynchronous module definition, was invented as a workaround. It introduces a define wrapper:

```
define(['./MyModule.js'], function (MyModule) {
  // export at module root
  return function() {};
});

// or
define(['./MyModule.js'], function (MyModule) {
  // export as module function
  return {
    hello: function() {...}
  };
});
```

Incidentally, it is possible to use `require` within the wrapper like this:

```
define(['require'], function (require) {
  var MyModule = require('./MyModule.js');

  return function() {...};
});
```

This approach definitely eliminates some of the clutter. You will still end up with some code that might feel redundant. Given there's ES6 now, it probably doesn't make much sense to use AMD anymore unless you really have to.

UMD

UMD, universal module definition, takes it all to the next level. It is a monster of a format that aims to make the aforementioned formats compatible with each other. I will spare your eyes from

it. Never write it yourself, leave it to the tools. If that didn't scare you off, check out [the official definitions](#)²⁴.

Webpack can generate UMD wrappers for you (`output.libraryTarget: 'umd'`). This is particularly useful for library authors. We'll get back to this later when discussing npm and library authorship in detail.

1.9 Conclusion

I hope this chapter helped you understand why Webpack is a valuable tool worth learning. It solves a fair share of common web development problems. If you know it well, it will save a great deal of time. In the following chapters we'll examine Webpack in more detail. You will learn to develop a simple development configuration. We'll also get started with our Kanban application.

You can, and probably should, use Webpack with some other tools. It won't solve everything. It does solve the difficult problem of bundling. That's one less worry during development. Just using *package.json*, *scripts*, and Webpack takes you far, as we will see soon.

²⁴<https://github.com/umdjs/umd>

2. Developing with Webpack

If you are not one of those people who likes to skip the introductions, you might have some clue what Webpack is. In its simplicity, it is a module bundler. It takes a bunch of assets in and outputs assets you can give to your client.

This sounds simple, but in practice, it can be a complicated and messy process. You definitely don't want to deal with all the details yourself. This is where Webpack fits in. Next, we'll get Webpack set up and your first project running in development mode.



Before getting started, make sure you are using a recent version of Node.js. Especially Node.js 0.10 has [issues with css-loader](#)¹. This will save you some trouble.

2.1 Setting Up the Project

Webpack is one of those tools that depends on [Node.js](#)². Make sure you have it installed and that you have npm available at your terminal. Set up a directory for your project, navigate there, hit `npm init` and fill in some details. You can just hit *return* for each and it will work. Here are the commands:

```
mkdir kanban_app
cd kanban_app
npm init
# hit return a few times till you have gone through the questions
```

As a result, you should have *package.json* at your project root. You can still tweak it manually to make further changes. We'll be doing some changes through *npm* tool, but it's fine to tweak the file to your liking. The official documentation explains various [package.json options](#)³ in more detail. I also cover some useful library authoring related tricks later in this book.



You can set those `npm init` defaults at `~/.npmrc`. See the “Authoring Libraries” for more information about npm and its usage.

If you are into version control, as you should, this would be a good time to set up your repository. You can create commits as you progress with the project.

If you are using git, I recommend setting up a *.gitignore* to the project root:

.gitignore

¹<https://github.com/webpack/css-loader/issues/144>

²<http://nodejs.org/>

³<https://docs.npmjs.com/files/package.json>

```
node_modules
```

At the very least you should have *node_modules* here as you probably don't want that to end up in the source control. The problem with that is that as some modules need to be compiled per platform, it gets rather messy to collaborate. Ideally your `git status` should look clean. You can extend *.gitignore* as you go.



You can push operating system level ignore rules such as *.DS_Store* and **.log* to *~/.gitignore*. This will keep your project level rules simpler.

2.2 Installing Webpack

Next, you should get Webpack installed. We'll do a local install and save it as a project dependency. This will allow us to maintain Webpack's version per project. Hit

```
npm i webpack --save-dev
```

This is a good opportunity to try to run Webpack for the first time. Hit `node_modules/.bin/webpack`. You should see a version log, a link to the command line interface guide and a long list of options. We won't be using most of those, but it's good to know that this tool is packed with functionality if nothing else.

Webpack works using a global install as well (`-g` or `--global` flag during installation). It is preferred to keep it as a project dependency like this. The arrangement helps to keep your life simpler. This way you have direct control over the version you are running.

We will be using `--save` and `--save-dev` to separate application and development dependencies. The separation keeps project dependencies more understandable. This will come in handy when we generate a vendor bundle later on.



There are handy shortcuts for `--save` and `--save-dev`. `-S` maps to `--save` and `-D` to `--save-dev`. So if you want to optimize for characters written, consider using these instead.

2.3 Directory Structure

As projects with just *package.json* are boring, we should set up something more concrete. To get started, we can implement a little web site that loads some JavaScript which we then build using Webpack. Set up a structure like this:

- /app
 - index.js
 - component.js
- package.json
- webpack.config.js

In this case, we'll generate *bundle.js* using Webpack based on our */app*. To make this possible, we should set up some assets and *webpack.config.js*.

2.4 Setting Up Assets

As you never get tired of Hello world, we might as well model a variant of that. Set up a component like this:

app/component.js

```
module.exports = function () {  
  var element = document.createElement('h1');  
  
  element.innerHTML = 'Hello world';  
  
  return element;  
};
```

Next, we are going to need an entry point for our application. It will simply require our component and render it through the DOM:

app/index.js

```
var component = require('./component');  
var app = document.createElement('div');  
  
document.body.appendChild(app);  
  
app.appendChild(component());
```

2.5 Setting Up Webpack Configuration

We'll need to tell Webpack how to deal with the assets we just set up. For this purpose we'll build *webpack.config.js*. Webpack and its development server will be able to discover this file through convention.

To keep things simple, we'll generate an entry point to our application using [html-webpack-plugin](#)⁴. We could create *index.html* by hand. Maintaining that could become troublesome as the project grows, though. *html-webpack-plugin* is able to create links to our assets keeping our life simple. Hit

```
npm i html-webpack-plugin --save-dev
```

to install it to the project.

To map our application to *build/bundle.js* and generate *build/index.html* we need configuration like this:

webpack.config.js

```
var path = require('path');
var HtmlWebpackPlugin = require('html-webpack-plugin');

const PATHS = {
  app: path.join(__dirname, 'app'),
  build: path.join(__dirname, 'build')
};

module.exports = {
  // Entry accepts a path or an object of entries.
  // The build chapter contains an example of the latter.
  entry: PATHS.app,
  output: {
    path: PATHS.build,
    filename: 'bundle.js'
  },
  plugins: [
    new HtmlWebpackPlugin({
      title: 'Kanban app'
    })
  ]
};
```

Given Webpack expects absolute paths we have some good options here. I like to use `path.join`, but `path.resolve` would be a good alternative. `path.resolve` is equivalent to navigating the file system through `cd`. `path.join` gives you just that, a join. See [Node.js path API](#)⁵ for the exact details.

⁴<https://www.npmjs.com/package/html-webpack-plugin>

⁵<https://nodejs.org/api/path.html>

If you hit `node_modules/.bin/webpack`, you should see a Webpack build at your output directory. You can open the `index.html` found there directly through a browser. On OS X you can use `open build/index.html` to see the result.

Another way to achieve this would be to serve the contents of the directory through a server such as `serve` (`npm i serve -g`). In this case you would execute `serve` at the output directory and head to `localhost:3000` at your browser. You can configure the port through the `--port` parameter if you want to use some other port.



Note that you can pass a custom template to *html-webpack-plugin*. In our case, the default template it uses is fine for our purposes for now.

2.6 Setting Up *webpack-dev-server*

Now that we have the basic building blocks together, we can set up a development server. *webpack-dev-server* is a development server running in-memory that automatically refreshes content in the browser while you develop the application. You should use *webpack-dev-server* strictly for development. If you want to host your application, consider other, standard solutions such as Apache or Nginx.

This makes it roughly equivalent to tools such as [LiveReload](http://livereload.com/)⁶ or [Browsersync](http://www.browsersync.io/)⁷. The greatest advantage Webpack has over these tools is Hot Module Replacement (HMR). We'll discuss it when we go through React.

Hit

```
npm i webpack-dev-server --save-dev
```

at the project root to get the server installed. We will be invoking our development server through npm. It allows us to set up scripts at *package.json*. The following configuration is enough:

package.json

```
...  
"scripts": {  
  "start": "webpack-dev-server"  
},  
...
```

⁶<http://livereload.com/>

⁷<http://www.browsersync.io/>

We also need to do some configuration work. We are going to use a simplified setup here. Beyond defaults we will enable Hot Module Replacement (HMR) and HTML5 History API fallback. The former will come in handy when we discuss React in detail. The latter allows HTML5 History API routes to work. *inline* setting embeds the *webpack-dev-server* runtime into the bundle allowing HMR to work easily. Otherwise we would have to set up more entry paths. Here's the setup:

webpack.config.js

```
...
var webpack = require('webpack');

const PATHS = {
  app: path.join(__dirname, 'app'),
  build: path.join(__dirname, 'build')
};

module.exports = {
  ...
  devServer: {
    historyApiFallback: true,
    hot: true,
    inline: true,
    progress: true,

    // Display only errors to reduce the amount of output.
    stats: 'errors-only',

    // Parse host and port from env so this is easy to customize.
    host: process.env.HOST,
    port: process.env.PORT
  },
  plugins: [
    new webpack.HotModuleReplacementPlugin(),
    ...
  ]
};
```

Hit `npm start` and surf to **localhost:8080**. You should see something familiar there. Try modifying *app/component.js* while the server is running and see what happens. Quite neat, huh?

Hello world

Hello world

You should be able to access the application alternatively through **localhost:8080/webpack-dev-server/** instead of root. It provides an iframe showing a status bar that indicates the status of the rebundling process.



Note that the current setup won't output a bundle at all given the development setup runs in-memory. We will set up a proper bundle at the build chapter.



In case the amount of console output annoys you, you can set `quiet: true` at `devServer` configuration to keep it minimal.



If you are using an environment, such as Cloud9, you should set `HOST` to `0.0.0.0`. The default `localhost` isn't the best option always.

Alternative Ways to Use *webpack-dev-server*

We could have passed *webpack-dev-server* options through the command line interface (CLI). I find it clearer to manage it within Webpack configuration as that helps to keep *package.json* nice and tidy. Alternatively we could have set up an Express server of our own and used *webpack-dev-server* as a [middleware](https://webpack.github.io/docs/webpack-dev-middleware.html)⁸. There's also a [Node.js API](https://webpack.github.io/docs/webpack-dev-server.html#api)⁹.



Note that there are [slight differences](https://github.com/webpack/webpack-dev-server/issues/106)¹⁰ between the CLI and Node.js API and they may behave slightly differently at times. This is the reason why some prefer to use solely Node.js API.

⁸<https://webpack.github.io/docs/webpack-dev-middleware.html>

⁹<https://webpack.github.io/docs/webpack-dev-server.html#api>

¹⁰<https://github.com/webpack/webpack-dev-server/issues/106>

Customizing Server *host* and *port*

It is possible to customize host and port settings through the environment in our setup (i.e., export `PORT=3000` on Unix or `SET PORT=3000` on Windows). This can be useful if you want to access your server within the same network. The default settings are enough on most platforms.

To access your server, you'll need to figure out the ip of your machine. On Unix this can be achieved using `ifconfig`. On Windows `ipconfig` can be used. A npm package, such as [node-ip](https://www.npmjs.com/package/node-ip)¹¹ may come in handy as well. Especially on Windows you may need to set your `HOST` to match your ip to make it accessible.

2.7 Refreshing CSS

We can extend the approach to work with CSS. Webpack allows us to change CSS without forcing a full refresh. To load CSS into a project, we'll need to use a couple of loaders. To get started, invoke

```
npm i css-loader style-loader --save-dev
```



If you are using Node.js 0.10, this is a good time to get a [ES6 Promise polyfill](https://www.npmjs.com/package/es6-promise)¹² set up.

Now that we have the loaders we need, we'll need to make sure Webpack is aware of them. Configure as follows.

webpack.config.js

...

```
module.exports = {
  entry: PATHS.app,
  output: {
    path: PATHS.build,
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        // Test expects a RegExp! Note the slashes!
        test: /\.css$/,
```

¹¹<https://www.npmjs.com/package/node-ip>

¹²<https://github.com/jakearchibald/es6-promise#auto-polyfill>


```

    loaders: ['style', 'css'],
    // Include accepts either a path or an array of paths.
    include: PATHS.app
  }
],
},
devServer: {
  historyApiFallback: true,
  hot: true,
  inline: true,
  progress: true,

  // Display only errors to reduce the amount of output.
  stats: 'errors-only',

  // Parse host and port from env so this is easy to customize.
  host: process.env.HOST,
  port: process.env.PORT
},
plugins: [
  new webpack.HotModuleReplacementPlugin(),
  new HtmlWebpackPlugin({
    title: 'Kanban app'
  })
]
};

```

The configuration we added means that files ending with `.css` should invoke given loaders. `test` matches against a JavaScript style regular expression. The loaders are evaluated from right to left. In this case, `css-loader` gets evaluated first, then `style-loader`. `css-loader` will resolve `@import` and `url` statements in our CSS files. `style-loader` deals with `require` statements in our JavaScript. A similar approach works with CSS preprocessors, like Sass and Less, and their loaders.



Loaders are transformations that are applied to source files, and return the new source. Loaders can be chained together, like using a pipe in Unix. See Webpack's [What are loaders?](http://webpack.github.io/docs/using-loaders.html)¹³ and [list of loaders](http://webpack.github.io/docs/list-of-loaders.html)¹⁴.



If `include` isn't set, Webpack will traverse all files within the base directory. This can hurt performance! It is a good idea to set up `include` always. There's also `exclude` option that may come in handy.

¹³<http://webpack.github.io/docs/using-loaders.html>

¹⁴<http://webpack.github.io/docs/list-of-loaders.html>

We are missing just one bit, the actual CSS itself:

app/main.css

```
body {  
  background: cornsilk;  
}
```

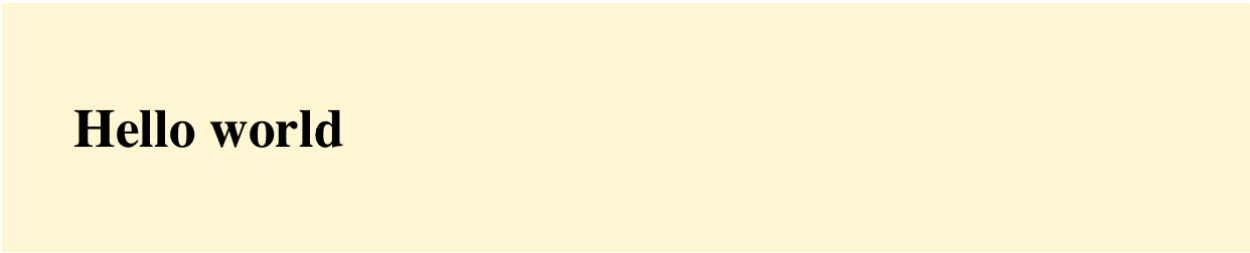
Also, we'll need to make Webpack aware of this file:

app/index.js

```
require('./main.css');  
  
...
```

Hit `npm start` now. Point your browser to **localhost:8080** if you are using the default port.

Open up *main.css* and change the background color to something like `lime` (`background: lime`). Develop styles as needed to make it look a little nicer.



Hello world

Hello cornsilk world

2.8 Making the Configuration Extensible

Our current configuration is enough as long as we're interested in just developing our application. But what if we want to deploy it to the production or test our application? We need to define separate **build targets** for these purposes. Given Webpack uses a module based format for its configuration, there are multiple possible approaches. At least the following are feasible:

- Maintain configuration in multiple files and point Webpack to each through `--config` parameter. Share configuration through module imports. You can see this approach in action at [webpack/react-starter](https://github.com/webpack/react-starter)¹⁵.

¹⁵<https://github.com/webpack/react-starter>

- Push configuration to a library which you then consume. Example: [HenrikJoreteg/hjs-webpack](https://github.com/HenrikJoreteg/hjs-webpack)¹⁶.
- Maintain configuration within a single file and branch there. If we trigger a script through *npm* (i.e., `npm run test`), *npm* sets this information to an environment variable. We can match against it and return the configuration we want. I prefer this approach as it allows me to understand what's going on easily. We'll be using this approach.



Webpack works well as a basis for more advanced tools. I've helped to develop a static site generator known as [Antwar](https://github.com/antwarjs)¹⁷. It builds upon Webpack and React and hides a lot of the complexity from the user.

Setting Up Configuration Target for `npm start`

To keep things simple, I've defined a custom merge function that concatenates arrays and merges objects. This is convenient with Webpack. Hit

```
npm i webpack-merge --save-dev
```

to add it to the project. We will detect the *npm* lifecycle event (`start`, `build`, ...) and then return configuration for each case. We will define more of these later on as we expand the project.

To improve the debuggability of the application, we can set up sourcemaps while we are at it. They allow you to see exactly where an error was raised. In Webpack this is controlled through the `devtool` setting. We can use decent defaults as follows:

webpack.config.js

```
var path = require('path');
var HtmlWebpackPlugin = require('html-webpack-plugin');
var webpack = require('webpack');
var merge = require('webpack-merge');

const TARGET = process.env.npm_lifecycle_event;
const PATHS = {
  app: path.join(__dirname, 'app'),
  build: path.join(__dirname, 'build')
};
```

¹⁶<https://github.com/HenrikJoreteg/hjs-webpack>

¹⁷<https://antwarjs.github.io/>

```
var common = {
  entry: PATHS.app,
  // Given webpack-dev-server runs in-memory, we can drop
  // `output`. We'll look into it again once we get to the
  // build chapter.
  /*output: {
    path: PATHS.build,
    filename: 'bundle.js'
  },*/
  module: {
    loaders: [
      {
        test: /\.css$/,
        loaders: ['style', 'css'],
        include: PATHS.app
      }
    ]
  },
  plugins: [
    // Important! move HotModuleReplacementPlugin below
    //new webpack.HotModuleReplacementPlugin(),
    new HtmlWebpackPlugin({
      title: 'Kanban app'
    })
  ]
};

if(TARGET === 'start' || !TARGET) {
  module.exports = merge(common, {
    devtool: 'eval-source-map',
    devServer: {
      historyApiFallback: true,
      hot: true,
      inline: true,
      progress: true,

      // Display only errors to reduce the amount of output.
      stats: 'errors-only',

      // Parse host and port from env so this is easy to customize.
      host: process.env.HOST,
      port: process.env.PORT
    }
  });
}
```

```
    },  
    plugins: [  
      new webpack.HotModuleReplacementPlugin()  
    ]  
  });  
}
```

if(TARGET === 'start' || !TARGET) { provides a default in case we're running Webpack outside of npm.

If you run the development build now using `npm start`, Webpack will generate sourcemaps. Webpack provides many different ways to generate them as discussed in the [official documentation](#)¹⁸. In this case, we're using `eval-source-map`. It builds slowly initially, but it provides fast rebuild speed and yields real files.



If `new webpack.HotModuleReplacementPlugin()` is added twice to the plugins declaration, Webpack will return `Uncaught RangeError: Maximum call stack size exceeded while hot loading!`

Faster development specific options such as `cheap-module-eval-source-map` and `eval` produce lower quality sourcemaps. Especially `eval` is fast and is the most suitable for large projects.

It is possible you may need to enable sourcemaps at your browser for this to work. See [Chrome](#)¹⁹ and [Firefox](#)²⁰ instructions for further details.

Configuration could contain more sections such as these based on your needs. Later on we'll develop another section to generate a production build.

2.9 Linting the Project

I discuss linting in detail in the *Linting in Webpack* chapter. Consider integrating the setup to your project to save some time. It will allow you to pick certain categories of errors earlier.

2.10 Conclusion

In this chapter you learned to build an effective development configuration using Webpack. Webpack deals with the heavy lifting for you now. The current setup can be expanded to support more scenarios. Next, we will see how to expand it to work with React.

¹⁸<https://webpack.github.io/docs/configuration.html#devtool>

¹⁹<https://developer.chrome.com/devtools/docs/javascript-debugging>

²⁰https://developer.mozilla.org/en-US/docs/Tools/Debugger/How_to/Use_a_source_map

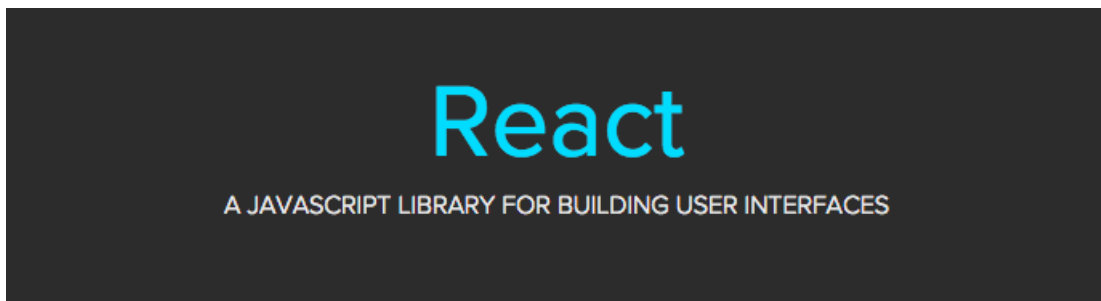
3. Webpack and React

Combined with Webpack, React becomes a joy to work with. Even though you can use React with other build tools, Webpack is a good fit and quite straightforward to set up. In this chapter we'll expand our configuration. After that we have a good starting point for developing our application further.



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/)¹ support it up to an extent. [Nuclide](http://nuclide.io/)², an Atom based IDE, has been developed with React in mind.

3.1 What is React?



React

Facebook's [React](https://facebook.github.io/react/)³ has changed the way we think about front-end development. Also, thanks to [React Native](https://facebook.github.io/react-native/)⁴ the approach isn't limited just to web. Although simple to learn, React provides plenty of power.

React isn't a framework like Angular.js or Ember. Frameworks tend to provide a lot of solutions out of the box. With React you will have to assemble your application from separate libraries. Both approaches have their merits. Frameworks may be faster to pick up, but they can become harder to work with as you hit their boundaries. In a library based approach you have more flexibility, but also responsibility.

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

²<http://nuclide.io/>

³<https://facebook.github.io/react/>

⁴<https://facebook.github.io/react-native/>

React introduced a concept known as virtual DOM to web developers. React maintains a DOM of its own unlike all the libraries and frameworks before it. As changes are made to virtual DOM, React will batch the changes to the actual DOM as it sees best.



Libraries such as [Matt-Esch/virtual-dom](https://github.com/Matt-Esch/virtual-dom)⁵ focus entirely on Virtual DOM. If you are interested in the theory, check it out.

JSX and Virtual DOM

React provides a [high level API](#)⁶ for generating virtual DOM. Generating complex structures using the API becomes cumbersome fast. Thus people usually don't write it by hand. Instead, they use some intermediate format that is converted into it. Facebook's [JSX](#)⁷ is one popular format.

JSX is a superset of JavaScript that allows you to mix XMLish syntax with JavaScript. Consider the example below:

```
function render() {  
  const names = ['John', 'Jill', 'Jack'];  
  
  return (  
    <div>  
      <h2>Names</h2>  
  
      <ul className="names">{  
        names.map((name) =>  
          <li className="name">{name}</li>  
        )  
      }</ul>  
    </div>  
  );  
}
```

If you haven't seen JSX before it will likely look strange. It isn't uncommon to experience "JSX shock" until you start to understand it. After that it all makes sense.

Cory House goes into more detail [about the shock](#)⁸. Briefly summarized, JSX gives us a level of validation we haven't encountered earlier. It takes a while to grasp, but once you get it, it's hard to go back.

⁵<https://github.com/Matt-Esch/virtual-dom>

⁶<https://facebook.github.io/react/docs/top-level-api.html>

⁷<https://facebook.github.io/jsx/>

⁸<https://medium.com/@housecor/react-s-jsx-the-other-side-of-the-coin-2ace7ab62b98>



Note that `render()` **must return a single node**⁹. Returning multiple won't work!

In JSX we are mixing something that looks a bit like HTML with JavaScript. Note how we treat attributes. Instead of using `class` as we would in vanilla HTML, we use `className`, which is the DOM equivalent. Even though JSX will feel a little weird to use at first, it will become second nature over time.

The developers of React have decoupled themselves from the limitations of the DOM. As a result, React is highly performant. This comes with a cost, though. The library isn't as small as you might expect. You can expect bundle sizes for small applications to be around 150-200k, React included. That is considerably less when gzipped over the wire, but it's still something.



The interesting side benefit of this approach is that React doesn't depend on the DOM. In fact, React can use other targets, such as **mobile**¹⁰, **canvas**¹¹, or **terminal**¹². The DOM just happens to be the most relevant one for web developers.

Better with Friends

React isn't the smallest library out there. It does manage to solve fundamental problems, though. It is a pleasure to develop thanks to its relative simplicity and a powerful API. You will need to complement it with a set of tools, but you can pick these based on actual need. It's far from a "one size fits all" type of solution which frameworks tend to be.

The approach used by React allowed Facebook to develop React Native on top of the same ideas. This time instead of the DOM, we are operating on mobile platform rendering. React Native provides abstraction over components and a layout system. It provides you the setup you already know from the web. This makes it a good gateway for web developers wanting to go mobile.

⁹<https://facebook.github.io/react/tips/maximum-number-of-jsx-root-nodes.html>

¹⁰<https://facebook.github.io/react-native/>

¹¹<https://github.com/Flipboard/react-canvas>

¹²<https://github.com/Yomguithereal/react-blessed>

3.2 Babel



Babel

[Babel](#)¹³ has made a big impact on the community. It allows us to use features from the future of JavaScript. It will transform your futuristic code to a format browsers understand. You can even use it to develop your own language features. Babel's built-in JSX support will come in handy here.

Babel provides support for certain [experimental features](#)¹⁴ 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
- **Stage 2** - Draft - Features starting from *stage 2* have been enabled by default
- **Stage 3** - Candidate
- **Stage 4** - Finished

I would be 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.



You can [try out Babel online](#)¹⁵ to see what kind of code it generates.

Configuring babel - loader

You can use Babel with Webpack easily through [babel-loader](#)¹⁶. It takes our ES6 module definition based code and turn it into ES5 bundles. Install *babel-loader* with

¹³<https://babeljs.io/>

¹⁴<https://babeljs.io/docs/usage/experimental/>

¹⁵<https://babeljs.io/repl/>

¹⁶<https://www.npmjs.com/package/babel-loader>

```
npm i babel-loader@5.x --save-dev
```



We're using Babel 5 here for now as *babel-plugin-react-transform* still needs to receive its Babel 6 fixes. The configuration will change considerably with Babel 6!

Besides, we need to add a loader declaration to the *loaders* section of configuration. It matches against *.js* and *.jsx* using a regular expression (*/\.jsx?\$/*).

To keep everything performant we restrict the loader to operate within *./app* directory. This way it won't traverse *node_modules*. An alternative would be to set up an *exclude* rule against *node_modules* explicitly. I find it more useful to *include* instead as that's more explicit. You never know what files might be in the structure after all.

Here's the relevant configuration we need to make Babel work:

webpack.config.js

```
...

var common = {
  entry: PATHS.app,
  // Add resolve.extensions. '' is needed to allow imports an extension
  // Note the .'s before extensions!!! Without those matching will fail
  resolve: {
    extensions: ['', '.js', '.jsx']
  },
  module: {
    loaders: [
      ...
      // Set up jsx. This accepts js too thanks to regex.
      {
        test: /\.jsx?$/,
        loaders: ['babel'],
        include: PATHS.app
      }
    ]
  },
  plugins: [
    new HtmlwebpackPlugin({
      title: 'Kanban app'
    })
  ]
};
```

...

Note that `resolve.extensions` setting will allow you to refer to JSX files without an extension now. I'll be using the extension for clarity, but for now you can omit it.



As `resolve.extensions` gets evaluated from left to right, we can use it to control which code gets loaded for given configuration. For instance, you could have `.web.js` to define web specific parts and then have something like `['', '.web.js', '.js', '.jsx']`. If a “web” version of the file is found, Webpack would use that instead of the default.

Setting Up *.babelrc*

Also, we are going to need a *.babelrc*¹⁷. You could pass Babel settings through Webpack (i.e., `babel?stage=1`), but then it would be just for Webpack only. That's why we are going to push our Babel settings to this specific dotfile. The same idea applies for other tools, such as ESLint.

We are going to enable three specific features as these will allow us to keep our project neat:

- **class properties**¹⁸ - Example: `renderNote = (note) => {`. This binds `renderNote` method to instances automatically. The feature makes more sense as we get to use it.
- **decorators**¹⁹ - Example: `@DragDropContext(HTML5Backend)`. These annotation allow us to attach functionality to classes and their methods.
- **object rest spread**²⁰ - Example: `const {a, b, ...props} = this.props`. This syntax allows us to extract specific properties from an object easily.

Set up a *.babelrc* to your project root as follows in order to enable the features:

.babelrc

¹⁷<https://babeljs.io/docs/usage/babelrc/>

¹⁸<https://github.com/jeffmo/es-class-static-properties-and-fields>

¹⁹<https://github.com/wycats/javascript-decorators>

²⁰<https://github.com/sebmarkbage/ecmascript-rest-spread>

```
{  
  "optional": [  
    "es7.classProperties",  
    "es7.decorators",  
    "es7.objectRestSpread"  
  ]  
}
```

Alternatively we could have used a declaration such as "stage": 1. The problem is that this doesn't document well which additional features we are using at our code base. It might work for small projects but I do not suggest this for production grade code. Documenting your Babel usage this way will help in the maintenance effort.

There are other possible [.babelrc options](#)²¹. For now we are keeping it simple.



It is possible to use Babel features at your Webpack configuration. Simply rename *webpack.config.js* as *webpack.config.babel.js* and Webpack will pick it up provided Babel has been set up with your project. It will respect the contents of *.babelrc*.

3.3 Developing the First React View

It is time to add a first application level dependency to our project. Hit

```
npm i react react-dom --save
```

to get React installed. This will save React to the dependencies section of *package.json*. Later on we'll use this information to generate a vendor build for the production version. It's a good practice to separate application and development level dependencies this way.

react-dom is needed as React can be used to target multiple systems (DOM, mobile, terminal, i.e.). Given we're dealing with the browser, *react-dom* is the correct choice here.

Now that we got that out of the way, we can start to develop our Kanban application. First we should define the App. This will be the core of our application. It represents the high level view of our app and works as an entry point. Later on it will orchestrate the entire app. We can get started by using React's function based component definition syntax:

app/components/App.jsx

²¹<https://babeljs.io/docs/usage/babelrc/>

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

export default () => {
  return <Note />;
};
```



You can import portions from react using syntax `import React, {Component} from 'react'`; Then you can do `class App extends Component`. It is important that you import React as well because that JSX will get converted to `React.createElement` calls. You may find this alternative a little neater regardless.



It may be worth your while to install [React Developer Tools](https://github.com/facebook/react-devtools)²² extension to your browser. Currently Chrome and Firefox are supported. This will make it easier to understand what's going on while developing.

Setting Up Note

We also need to define the Note component. In this case, we will just want to show some text like `Learn Webpack.Hello world` would work if you are into clichés.

app/components/Note.jsx

```
import React from 'react';

export default class Note extends React.Component {
  render() {
    return <div>Learn Webpack</div>;
  }
}
```



Note that we're using `jsx` extension here. It helps us to tell modules using JSX syntax apart from regular ones. It is not absolutely necessary, but it is a good convention to have.



It is important to note that ES6 based class approach **doesn't** support autobinding behavior. Apart from that you may find ES6 classes neater than `React.createClass`. See the end of this chapter for a comparison.

²²<https://github.com/facebook/react-devtools>

Rendering Through `index.jsx`

We'll need to adjust our `index.js` to render the component correctly. Note that I've renamed it as `index.jsx` given we have JSX content there. First the rendering logic creates a DOM element where it will render. Then it renders our application through React.

`app/index.jsx`

```
import './main.css';

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

main();

function main() {
  const app = document.createElement('div');

  document.body.appendChild(app);

  ReactDOM.render(<App />, app);
}
```

I'll be using `const` whenever possible. It will give me a guarantee that the reference to the object won't get changed inadvertently. It does allow you to change the object contents, though, in that you can still push new items to an array and so on.

If I want something mutable, I'll use `let` instead. `let` is scoped to the code block and is another new feature introduced with ES6. These both are good safety measures.



Avoid rendering directly to `document.body`. This can cause strange problems when relying on it. Instead give React a little sandbox of its own. That way everyone, including React, will stay happy.

If you hit `npm start` now, you should see something familiar at **localhost:8080**.

Learn Webpack

Hello React

Before moving on, this is a good chance to get rid of the old `component.js` file. It might be hanging around at app root.

3.4 Activating Hot Loading for Development

Note that every time you perform a modification, the browser updates with a flash. That's unfortunate because this means our application loses state. It doesn't matter yet, but as we keep on expanding the application this will become painful. It is annoying to manipulate the user interface back to the state in which it was to test something.

We can work around this problem using hot loading. [babel-plugin-react-transform](https://github.com/gaearon/babel-plugin-react-transform)²³ allow us to instrument React components in various ways. Hot loading is one of these. It is enabled through [react-transform-hmr](https://github.com/gaearon/react-transform-hmr)²⁴.

react-transform-hmr will swap React components one by one as they change without forcing a full refresh. Given it just replaces methods, it won't catch every possible change. This includes changes made to class constructors. There will be times when you will need to force a refresh, but it will work most of the time.

To enable hot loading for React, you should first install the packages using

```
npm i babel-plugin-react-transform react-transform-hmr --save-dev
```

We also need to make Babel aware of HMR. First we should pass target environment to Babel through our Webpack configuration:

webpack.config.js

```
...

process.env.BABEL_ENV = TARGET;

var common = {
  ...
};

...
```

In addition we need to expand Babel configuration to include the plugin we need during development:

.babelrc

²³<https://github.com/gaearon/babel-plugin-react-transform>

²⁴<https://github.com/gaearon/react-transform-hmr>

```

{
  "optional": [
    "es7.classProperties",
    "es7.decorators",
    "es7.objectRestSpread"
  ],
  "env": {
    "start": {
      "plugins": [
        "react-transform"
      ],
      "extra": {
        "react-transform": {
          "transforms": [
            {
              "transform": "react-transform-hmr",
              "imports": ["react"],
              "locals": ["module"]
            }
          ]
        }
      }
    }
  }
}

```

Try hitting `npm start` again and modifying the component. Note what doesn't happen this time. There's no flash! It might take a while to sink in, but in practice, this is a powerful feature. Small things such as this add up and make you more effective.

Note that Babel determines the value of `env` like this:

1. Use the value of `BABEL_ENV` if set.
2. Use the value of `NODE_ENV` if set.
3. Default to development.



If you want to show errors directly in the browser, you can configure [react-transform-catch-errors](https://github.com/gaearon/react-transform-catch-errors)²⁵. At the time of writing it works reliably only with `devtool: 'eval'`, but regardless it may be worth a look.

²⁵<https://github.com/gaearon/react-transform-catch-errors>



Note that sourcemaps won't get updated in [Chrome](#)²⁶ and Firefox due to browser level bugs! This may change in the future as the browsers get patched, though.

3.5 React Component Styles

Beyond ES6 classes, React allows you to construct components using `React.createClass()` and functions. `React.createClass()` was the original way to create components and it is still in use. The approaches aren't equivalent by default.

When you are using `React.createClass` it is possible to inject functionality using mixins. This isn't possible in ES6 by default. Yet, you can use a helper such as [react-mixin](#)²⁷. In later chapters we will go through various alternative approaches. They allow you to reach roughly equivalent results as you can achieve with mixins. Often a decorator is all you need.

Also, ES6 class based components won't bind their methods to `this` context by default. This is the reason why it's good practice to bind the context at the component constructor. We will use this convention in this book. It leads to some extra code, but later on it is likely possible to refactor it out.

The class based approach decreases the amount of concepts you have to worry about. `constructor` helps to keep things simpler than in `React.createClass` based approach. There you need to define separate methods to achieve the same result.

3.6 Conclusion

You should understand how to set up React with Webpack now. Hot loading is one of those features that sets Webpack apart. Now that we have a good development environment, we can focus on React development. In the next chapter you will see how to implement a little note taking application. That will be improved in the subsequent chapters into a full blown Kanban table.

²⁶<https://code.google.com/p/chromium/issues/detail?id=492902>

²⁷<https://github.com/brigand/react-mixin>

II Developing Kanban Application

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

In this part we will implement a small Kanban application. During the process you will learn basics of React. As React is just a view library we will discuss supporting technology. Alt, a Flux framework, provides a good companion to React and allows you to keep your components clean. You will also see how to use React DnD to add drag and drop functionality to the board.

4. Implementing a Basic Note Application

Given we have a nice development setup now, 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.



Hot loading isn't fool proof always. Given it operates by swapping methods dynamically, it won't catch every change. This is problematic with property initializers and `bind`. This means you may need to force a manual refresh at the browser for some changes to show up!

4.1 Initial Data Model

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

```
[
  {
    id: '4a068c42-75b2-4ae2-bd0d-284b4abbb8f0',
    task: 'Learn Webpack'
  },
  {
    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.

4.2 On Ids

We could have skipped ids in our definition. This would become problematic as we grow our application, though. If you are referring to data based on array indices and the data changes, each reference has to change too. We can avoid that.

Normally the problem is solved by a back-end. As we don't have one yet, we'll need to improvise something. A standard known as [RFC4122](https://www.ietf.org/rfc/rfc4122.txt)¹ allows us to generate unique ids. We'll be using a Node.js implementation known as *node-uuid*. Invoke

```
npm i node-uuid --save
```

at the project root to get it installed.

If you open up the Node.js CLI (node) and try the following, you can see what kind of ids it outputs.

```
> uuid = require('node-uuid')
{ [Function: v4]
  v1: [Function: v1],
  v4: [Circular],
  parse: [Function: parse],
  unparse: [Function: unparse],
  BufferClass: [Function: Array] }
> uuid.v4()
'1c8e7a12-0b4c-4f23-938c-00d7161f94fc'
```

`uuid.v4()` will help us to generate the ids we need for the purposes of this project. It is guaranteed to return a unique id with a high probability. If you are interested in the math behind this, check out [the calculations at Wikipedia](https://en.wikipedia.org/wiki/Universally_unique_identifier#Random_UUID_probability_of_duplicates)² for details. You'll see that the possibility for collisions is somewhat miniscule.



You can exit Node.js CLI by hitting CTRL-D once.

4.3 Connecting Data with App

Next, we need to connect our data model with App. The simplest way to achieve that is to push the data directly to `render()` for now. This won't be efficient, but it will allow us to get started. The implementation below shows how this works out in React terms:

`app/components/App.jsx`

¹<https://www.ietf.org/rfc/rfc4122.txt>

²https://en.wikipedia.org/wiki/Universally_unique_identifier#Random_UUID_probability_of_duplicates

```

import uuid from 'node-uuid';
import React from 'react';
import Note from './Note.jsx';

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

export default class App extends React.Component {
  render() {
    return (
      <div>
        <ul>{notes.map(this.renderNote)}</ul>
      </div>
    );
  }
  renderNote(note) {
    return (
      <li key={note.id}>
        <Note task={note.task} />
      </li>
    );
  }
}

```

We are using various important features of React in the snippet above. Understanding them is invaluable. I have annotated important parts below:

- `{notes.map(this.renderNote)}` - `{}`'s allow us to mix JavaScript syntax within JSX. `map` returns a list of `li` elements for React to render.
- `<li key={note.id}>` - 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](#)³ for more information.



You can import portions from react using syntax `import React, {Component} from 'react';`. Then you can do `class App extends Component`. You may find this alternative a little neater.

If you run the application now, you can see it almost works. There's a small glitch, but we'll fix that next.

- Learn Webpack
- Learn Webpack
- Learn Webpack

Almost done



If you want to examine your application state, it can be useful to attach a [debugger](#)⁴ statement to the place you want to study. It has to be placed on a line that will get executed for the browser to pick it up! The statement will cause the browser debugging tools to trigger and allow you to study the current call stack and scope. You can attach breakpoints like this through browser, but this is a good alternative.

4.4 Fixing Note

The problem is that we haven't taken `task` prop into account at `Note`. In React terms *props* is a data structure that's passed to a component from outside. It is up to the component how it uses this data. In the code below I extract the value of a prop and render it.

`app/components/Note.jsx`

³<https://facebook.github.io/react/docs/multiple-components.html>

⁴<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/debugger>

```
import React from 'react';

export default class Note extends React.Component {
  render() {
    return <div>{this.props.task}</div>;
  }
}
```

If you check out the application now, you should see we're seeing results that are more like it. This is only the start, though. Our App is getting cramped. It feels like there's a component waiting to be extracted.

- Learn Webpack
- Learn React
- Do laundry

Notes render now



If you want to attach comments to your JSX, just use `{/* no comments */}`.

4.5 Extracting Notes

If we keep on growing App like this we'll end up in trouble soon. Currently App deals with too many concerns. It shouldn't have to know what Notes look like. That's a perfect candidate for a component. As earlier, we'll want something that will accept a prop, say `items`, and is able to render them in a list. We already have logic for that in App. It needs to be moved out.



Recognizing components is an important skill when working with React. There's small overhead to creating them and it allows you to model your problems in exact terms. At higher levels, you will just worry about layout and connecting data. As you go lower in the architecture, you start to see more concrete structures.

A good first step towards a neater App is to define Notes. It will rely on the rendering logic we already set up. We are just moving it to a component of its own. Specifically we'll want to perform `<Notes items={notes} />` at `render()` method of App. That's just nice.

You probably have the skills to implement Notes by now. Extract the logic from App and push it to a component of its own. Remember to attach `this.props.items` to the rendering logic. This way our interface works as expected. I've included complete implementation below for reference:

app/components/Notes.jsx

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

export default class Notes extends React.Component {
  render() {
    const notes = this.props.items;

    return <ul className="notes">{notes.map(this.renderNote)}</ul>;
  }
  renderNote(note) {
    return (
      <li className="note" key={note.id}>
        <Note task={note.task} />
      </li>
    );
  }
}
```

It is a good idea to attach some CSS classes to components to make it easier to style them. React provides other styling approaches beyond this. I will discuss them later in this book. There's no single right way to style and you'll have to adapt based on your preferences. In this case, we'll just focus on keeping it simple.

We also need to replace the old App logic to use our new component. You should remove the old rendering logic, import Notes, and update `render()` to use it. Remember to pass `notes` through `items` prop and you might see something familiar. I have included the full solution below for completeness:

app/components/App.jsx


```
import uuid from 'node-uuid';
import React from 'react';
import Notes from './Notes.jsx';

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

export default class App extends React.Component {
  render() {
    return (
      <div>
        <Notes items={notes} />
      </div>
    );
  }
}
```

Logically, we have exactly the same App as earlier. There's one great difference. Our application is more flexible. You could render multiple Notes with data of their own easily.

Even though we improved `render()` and reduced the amount of markup, it's still not neat. We can push the data to the App's state. Besides making the code neater, this will allow us to implement logic related to it.

4.6 Pushing notes to the App state

As seen earlier React components can accept props. In addition, they may have a state of their own. This is something that exists within the component itself and can be modified. You can think of these two in terms of immutability. As you should not modify props you can treat them as immutable. The state, however, is mutable and you are free to alter it. In our case, pushing notes to the state makes sense. We'll want to tweak them through the user interface.

In ES6's class syntax the initial state can be defined at the constructor. We'll assign the state we want to `this.state`. After that we can refer to it. The example below illustrates how to convert our notes into state.

app/components/App.jsx

```
import uuid from 'node-uuid';
import React from 'react';
import Notes from './Notes.jsx';

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

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

    ...
  }
}
```

After this change our application works the same way as before. We have gained something in return, though. We can begin to alter the state.



Note that *babel-plugin-react-transform* doesn't pick the change made to the constructor. Technically it just replaces methods. As a result the constructor won't get invoked. You will have to force a refresh in this case!



In earlier versions of React you achieved the same result with `getInitialState`. 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.

4.7 Adding New Items to Notes list

Adding new items to the notes list is a good starting point. To get started, we could render a button element and attach a dummy `onClick` handler to it. We will expand the actual logic into that.

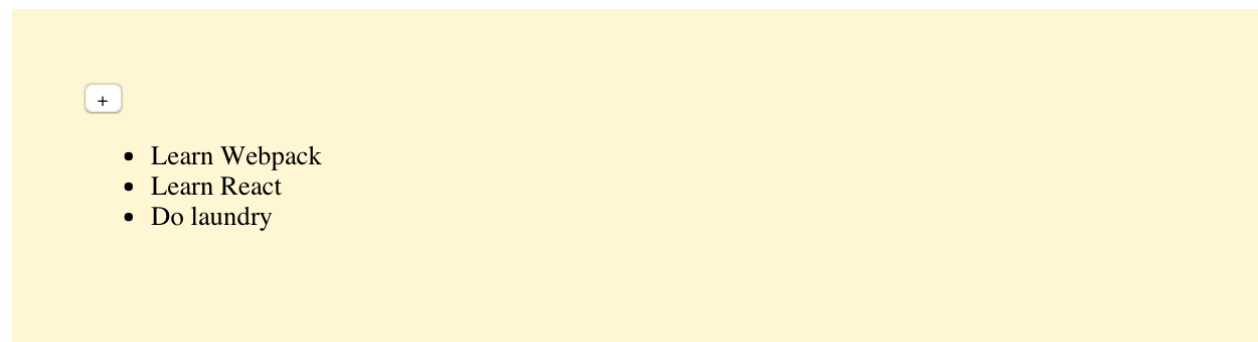
`app/components/App.jsx`

```
...

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

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

If you click the plus button now, you should see something in your browser console. The next step is to connect this stub with our data model.



Notes with plus

Connecting addNote with Data Model

React provides one simple way to change the state, namely `this.setState(data, cb)`. It is an asynchronous method that updates `this.state` and triggers `render()` eventually. It accepts data and an optional callback. The callback is triggered after the process has completed.

It is best to think of state as immutable and alter it always through `setState`. In our case, adding a new note can be done through a `concat` operation as below:

app/components/App.jsx

...

```
export default class App extends React.Component {
  constructor(props) {
    ...
  }
  render() {
    ...
  }
  // We are using an experimental feature known as property
  // initializer here. It allows us to bind the method `this`
  // to point at our *App* instance.
  //
  // In Babel 5 this is enabled using `es7.classProperties` key
  // at `optional` array of *.babelrc*.
  //
  // Alternatively we could `bind` at `constructor` using
  // a line such as this.addNote = this.addNote.bind(this);
  addNote = () => {
    // It would be possible to write this in an imperative style.
    // I.e., through this.state.notes.push`.
```

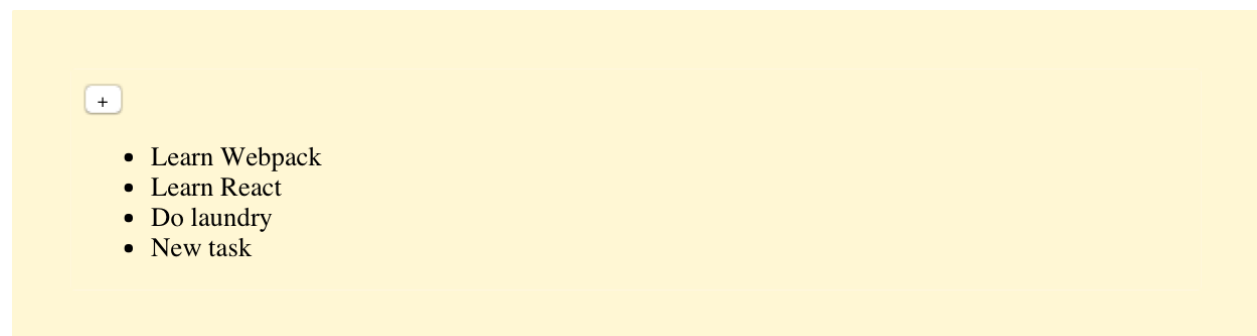
```
//
// I tend to favor functional style whenever that makes sense.
// Even though it might take more code sometimes, I feel
// the benefits (easy to reason about, no side effects)
// more than make up for it.
//
// Libraries, such as Immutable.js, go one notch further.
this.setState({
  notes: this.state.notes.concat([
    {
      id: uuid.v4(),
      task: 'New task'
    }
  ])
});
}
```

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.state.notes, {id: uuid.v4(), task: 'New task'}] to achieve the same result. This [spread operator](#)⁵ can be used with function parameters as well.

If you hit the button a few times now, you should see new items. It might not be pretty yet, but it works.



Notes with a new item

4.8 Editing Notes

Our Notes list is almost useful now. We just need to implement editing and we're almost there. One simple way to achieve this is to detect a click event on a Note, and then show an input containing its state. Then when the editing has been confirmed, we can return it back to normal.

⁵https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator

This means we'll need to extend `Note` somehow and communicate possible changes to `App`. That way it knows to update the data model. Additionally, `Note` needs to keep track of its edit state. It has to show the correct element (`div` or `input`) based on its state.

We can achieve these goals using a callback and a ternary expression. Here's a sample implementation of the idea:

`app/components/Note.jsx`

```
import React from 'react';

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

    this.state = {
      editing: false
    };
  }
  render() {
    if(this.state.editing) {
      return this.renderEdit();
    }

    return this.renderNote();
  }
  renderEdit = () => {
    return <input type="text"
      autoFocus={true}
      defaultValue={this.props.task}
      onBlur={this.finishEdit}
      onKeyPress={this.checkEnter} />;
  }
  renderNote = () => {
    return <div onClick={this.edit}>{this.props.task}</div>;
  }
  edit = () => {
    this.setState({
      editing: true
    });
  }
  checkEnter = (e) => {
    if(e.key === 'Enter') {
      this.finishEdit(e);
    }
  }
}
```

```

    }
  }
  finishEdit = (e) => {
    this.props.onEdit(e.target.value);

    this.setState({
      editing: false
    });
  }
}

```

If you try to edit a Note now, you will see an error (`this.props.onEdit` is not a function) at the console. We'll fix this shortly.

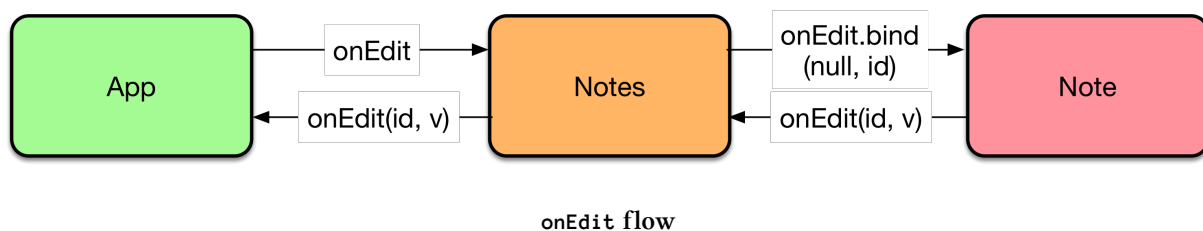
The rest of the code deals with events. If we click the component while it is in its initial state, we will enter the edit mode. If we confirm the editing, we hit the `onEdit` callback. As a result, we go back to the default state.



It is 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.

Adding onEdit Stub

Given we are currently dealing with the logic at App, we can deal with `onEdit` there as well. We will need to trigger this callback at Note and delegate the result to App level. The diagram below illustrates the idea:



A good first step towards this behavior is to create a stub. As `onEdit` is defined on App level, we'll need to pass `onEdit` handler through Notes. So for the stub to work, changes in two files are needed. Here's what it should look like for App.

app/components/App.jsx

```
import uuid from 'node-uuid';
import React from 'react';
import Notes from './Notes.jsx';

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

    return (
      <div>
        <button className="add-note" onClick={this.addNote}></button>
        <Notes items={notes} onEdit={this.editNote} />
      </div>
    );
  }
  ...
  editNote(noteId, task) {
    console.log('note edited', noteId, task);
  }
}
```

The idea is that Notes will return via our callback the id of the note being modified and the new state of the task. We'll need to use this data soon in order to patch the state.

We also need to make Notes work according to this idea. It will bind the id of the note in question. When the callback is triggered, the remaining parameter receives a value and the callback gets called.

app/components/Notes.jsx

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

export default class Notes extends React.Component {
  render() {
    const notes = this.props.items;

    // We are setting the context (`this`) of `this.renderNote` to `this`
    // explicitly! Alternatively we could use a property initializer here,
    // but this will work as well while being compatible with hot loading.
    return <ul className="notes">{notes.map(this.renderNote, this)}</ul>;
  }
}
```



```

    }
    renderNote(note) {
      return (
        <li className="note" key={note.id}>
          <Note
            task={note.task}
            onEdit={this.props.onEdit.bind(null, note.id)} />
        </li>
      );
    }
  }
}

```

If you edit a Note now, you should see a log at the console.



Besides allowing you to set context, [bind](#)⁶ makes it possible to fix parameters to certain values. Here we fix the first parameter to the value of `note.id`.

It would be nice to push the state to Notes. The problem is that doing this would break the encapsulation. We would still need to wire up the “add note” button with the same state. This would mean we would have to communicate the changes to Notes somehow. We’ll discuss a better way to solve this very issue in the next chapter.

We are missing one final bit, the actual logic. Our state consists of Notes each of which has an id (string) and a task (string) attached to it. Our callback receives both of these. In order to edit a Note it should find the Note to edit and patch its task using the new data.



Some of the prop related logic could be potentially extracted to a [context](#)⁷. That would help us to avoid some of the prop passing. It is especially useful for implementing features such as internationalization (i18n) or [feature detection](#)⁸. A component interested in it may simply query for a translator instance.

Implementing onEdit Logic

The only thing that remains is gluing this all together. We’ll need to take the data and find the specific Note by its indexed value. Finally, we need to modify and commit the Note’s data to the component state through `setState`.

`app/components/App.jsx`

⁶https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind

⁷<https://facebook.github.io/react/docs/context.html>

⁸<https://github.com/casesandberg/react-context/>

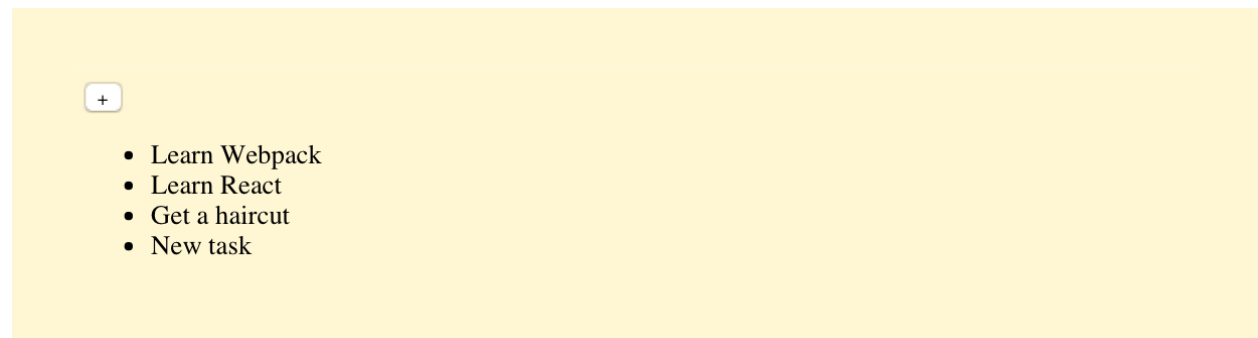
```
import uuid from 'node-uuid';
import React from 'react';
import Notes from './Notes.jsx';

export default class App extends React.Component {
  ...
  editNote = (id, task) => {
    const notes = this.state.notes.map((note) => {
      if(note.id === id) {
        note.task = task;
      }

      return note;
    });

    this.setState({notes});
  }
}
```

If you try to edit a Note now, the modification should stick. The same idea can be used to implement a lot of functionality and this is a pattern you will see a lot.



Edited a note

4.9 Removing Notes

We are still missing one vital function. It would be nice to be able to delete notes. We could implement a button per Note and trigger the logic using that. It will look a little rough initially, but we will style it later.

As before we'll need to define some logic on App level. Deleting a note can be achieved by first looking for a Note to remove based on id. After we know which Note to remove, we can construct a new state without it.

app/components/App.jsx

```

import uuid from 'node-uuid';
import React from 'react';
import Notes from './Notes.jsx';

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

    return (
      <div>
        <button className="add-note" onClick={this.addNote}></button>
        <Notes items={notes}
          onEdit={this.editNote} onDelete={this.deleteNote} />
      </div>
    );
  }
  deleteNote = (id) => {
    this.setState({
      notes: this.state.notes.filter((note) => note.id !== id)
    });
  }
  ...
}

```

In addition to App level logic, we'll need to trigger onDelete logic at Note level. The idea is the same as before. We'll bind the id of the Note at Notes. A Note will simply trigger the callback when the user triggers the behavior.

app/components/Notes.jsx

```

export default class Notes extends React.Component {
  render() {
    ...
  }
  renderNote(note) {
    return (
      <li className="note" key={note.id}>
        <Note
          task={note.task}
          onEdit={this.props.onEdit.bind(null, note.id)}
          onDelete={this.props.onDelete.bind(null, note.id)} />
      </li>
    );
  }
}

```

```

    );
  }
}

```

In order to invoke the previous `onDelete` callback we need to connect it with `onClick` for `Note`. If the callback doesn't exist, it makes sense to avoid rendering the delete button. An alternative way to solve this would be to push it to a component of its own.

app/components/Note.jsx

```

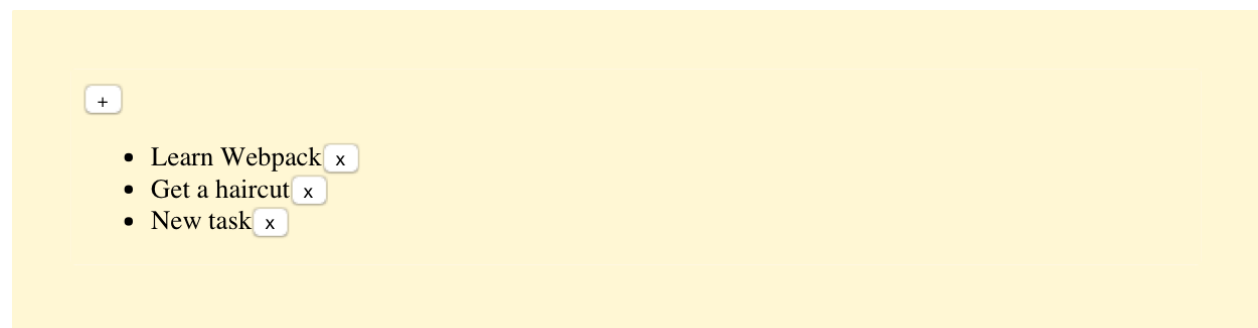
...

export default class Note extends React.Component {
  ...
  renderNote = () => {
    const onDelete = this.props.onDelete;

    return (
      <div onClick={this.edit}>
        <span className="task">{this.props.task}</span>
        {onDelete ? this.renderDelete() : null }
      </div>
    );
  }
  renderDelete = () => {
    return <button className="delete" onClick={this.props.onDelete}>x</button>;
  }
  ...
}

```

After these changes you should be able to delete notes as you like.



Deleted a note

We have a fairly well working little application now. We can create, update and delete Notes now. During this process we learned something about props and state. There's more than that to React, though.



Now delete is sort of blunt. One interesting way to develop this further would be to add confirmation. One simple way to achieve this would be to show yes/no buttons before performing the action. The logic would be more or less the same as for editing. This behavior could be extracted into a component of its own.

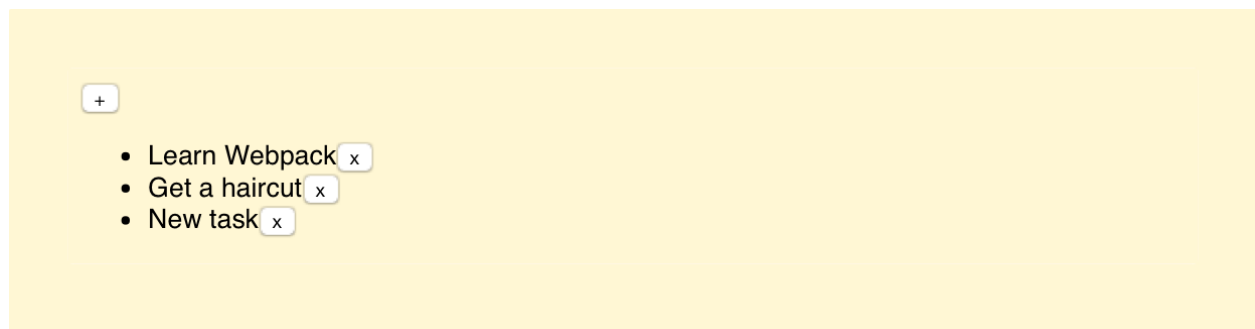
4.10 Styling Notes

Aesthetically our current application is very barebones. As pretty applications are more fun to use, we can do a little something about that. The first step is to get rid of that horrible *serif* font.

app/main.css

```
body {  
  background: cornsilk;  
  font-family: sans-serif;  
}
```

Looking a little nicer now:



Sans serif

A good next step would be to constrain the Notes container a little and get rid of those list bullets.

app/main.css

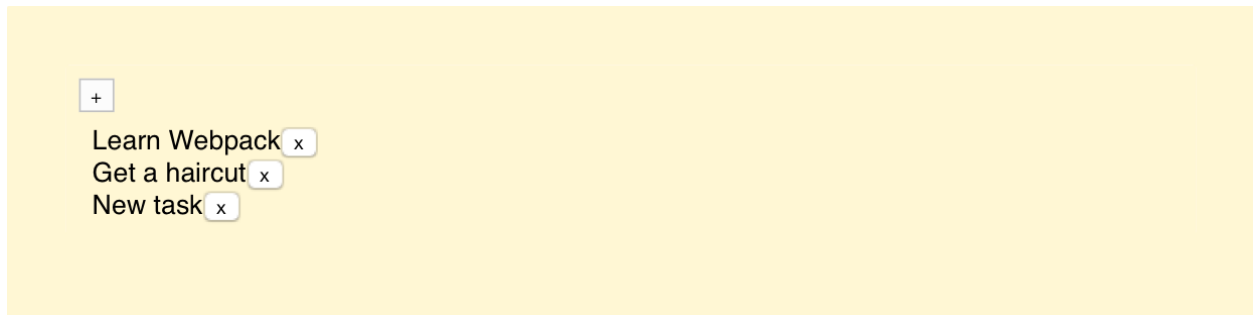
```
...

.add-note {
  background-color: #fdfdfd;
  border: 1px solid #ccc;
}

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

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

Removing bullets helps:



No bullets

To make individual Notes stand out we can apply a couple of rules.

app/main.css

```
...

.note {
  margin-bottom: 0.5em;
  padding: 0.5em;

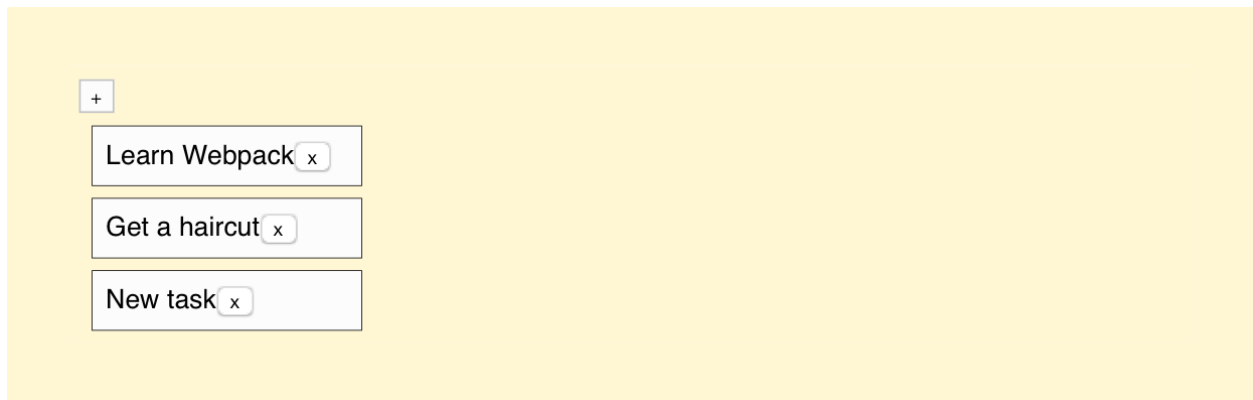
  background-color: #fdfdfd;
  box-shadow: 0 0 0.3em 0.03em rgba(0, 0, 0, 0.3);
}

.note:hover {
  box-shadow: 0 0 0.3em 0.03em rgba(0, 0, 0, 0.7);
}
```

```
    transition: 0.6s;
  }

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

Now the notes stand out a bit:



Styling notes

I animated Note shadow in the process. This way the user gets a better indication of what Note is being hovered upon. This won't work on touch based interfaces, but it's a nice touch for the desktop.

Finally, we should make those delete buttons stand out less. One way to achieve this is to hide them by default and show them on hover. The gotcha is that delete won't work on touch, but we can live with that.

app/main.css

```
...

.note .delete {
  float: right;

  padding: 0;

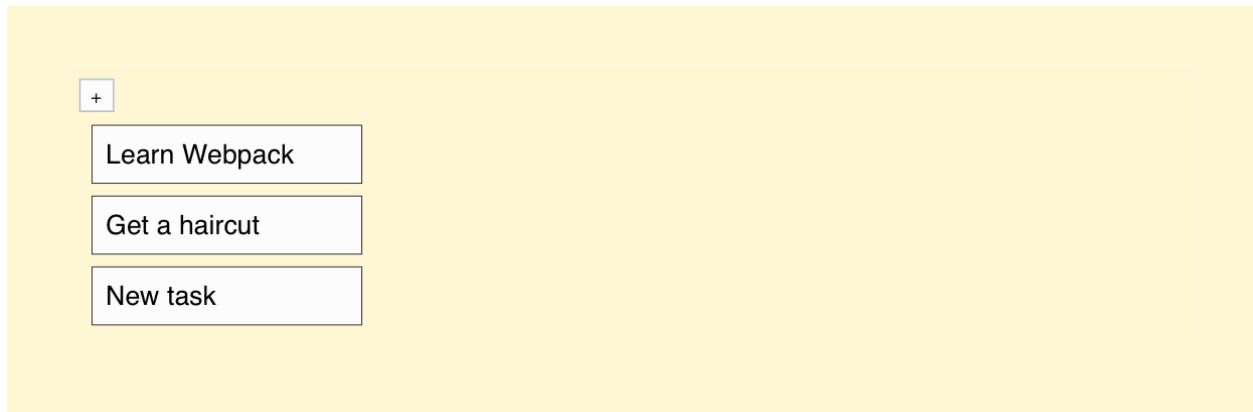
  background-color: #fdfdfd;
  border: none;

  cursor: pointer;

  visibility: hidden;
```

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

No more of those pesky delete buttons:



Delete on hover

After these few steps we have an application that looks passable. We'll be improving its appearance as we add functionality, but at least it's somewhat visually appealing.

4.11 Understanding React Components

Understanding how props and state work is important. Component lifecycle is another key concept. We already touched on it earlier, but it's a good idea to understand it in more detail. You can achieve most tasks in React by applying these concepts throughout your application. React provides the following lifecycle hooks:

- `componentWillMount()` gets triggered once before any rendering. One way to use it would be to load data asynchronously there and force rendering through `setState`.
- `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.
- `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.

- `componentDidUpdate()` is triggered after rendering. You can modify the DOM here. This can be useful for adapting other code to work with React.
- `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).

Beyond the lifecycle methods, there are a variety of [properties and methods](#)⁹ you should be aware of 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.
- `getInitialState()` - In class based approach the same can be achieved through constructor.
- `getDefaultProps()` - In classes you can set these in constructor.
- `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;
```

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.

Both class and `React.createClass` based components allow you to document the interface of your component using `propTypes`. To dig deeper, read the *Typing with React* chapter.

Both support `render()`, the workhorse of React. In function based definition `render()` is the function itself. `render()` simply describes what the component should look like. In case you don't want to render anything, return either `null` or `false`.

⁹<https://facebook.github.io/react/docs/component-specs.html>

4.12 React Component Conventions

I prefer to have the constructor first, followed by lifecycle hooks, `render()`, and finally methods used by `render()`. I like this top-down approach as it makes it straightforward to follow code. Some prefer to put the methods used by `render()` before it. There are also various naming conventions. It is possible to use `_` prefix for event handlers, too.

In the end you will have to find conventions that you like and which work the best for you. I go into more detail about this topic in the linting chapter where I introduce various code quality related tools. Through the use of these tools, it is possible to enforce coding style to some extent.

This can be useful in a team environment. It 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.

4.13 Conclusion

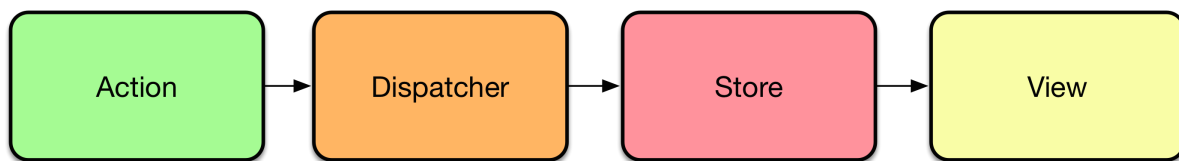
You can get quite far just with vanilla React. The problem is that we are starting to mix data related concerns and logic with our view components. We'll improve the architecture of our application by introducing Flux to it.

5. React and Flux

You can get pretty far by keeping everything in components. Eventually that will become painful. [Flux application architecture](https://facebook.github.io/flux/docs/overview.html)¹ helps to bring clarity to our React applications.

Flux will allow us to separate data and application state from our views. This helps us to keep them clean and the application maintainable. Flux was designed with large teams in mind. As a result, you might find it quite verbose. This comes with great advantages, though, as it can be straightforward to work with.

5.1 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.

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

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

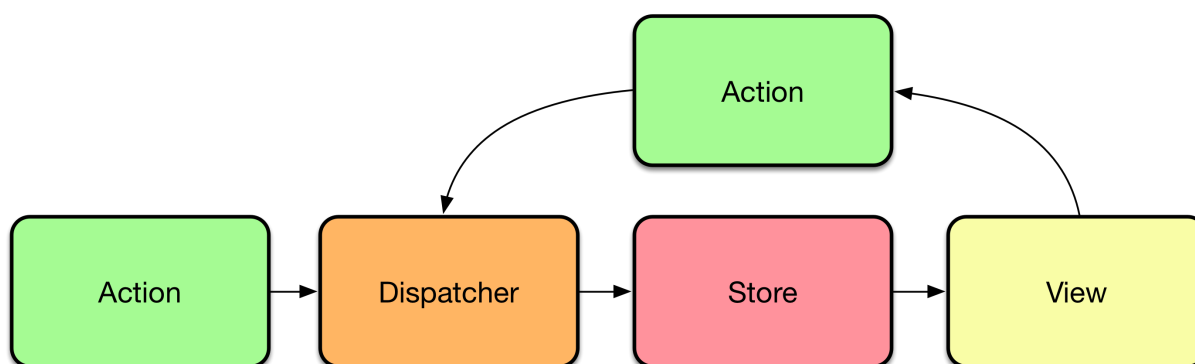
At the simplest level, actions can just pass the message to dispatcher as is. They can also trigger asynchronous queries and hit dispatcher based on the result eventually. This allows us to deal with received data and possible errors.

Once the dispatcher has dealt with the action, stores that are 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.

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

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.

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.



Flux dataflow with cycle

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 i18n 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: 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, "explicit is better than implicit".

Which Flux Implementation to Use?

The library situation is constantly changing. There is no single right way to interpret the architecture. You will find implementations that fit different tastes. [voronianski/flux-comparison](https://github.com/voronianski/flux-comparison)² 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.

²<https://github.com/voronianski/flux-comparison>

5.2 Porting to Alt



Alt

In this chapter we'll be using a library known as [Alt](http://alt.js.org/)³. It is a flexible, full-featured implementation that has been designed with isomorphic rendering in mind.

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 isomorphic rendering.

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 get started, we should add Alt to our project. We'll also install *alt-utils* as it contains some special functionality we'll need later on:

```
npm i alt alt-utils --save
```

To keep things simple, we'll be treating all Alt components as a [singleton](https://en.wikipedia.org/wiki/Singleton_pattern)⁴. 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. Set it up as follows:

app/libs/alt.js

³<http://alt.js.org/>

⁴https://en.wikipedia.org/wiki/Singleton_pattern

```
import Alt from 'alt';
//import chromeDebug from 'alt-utils/lib/chromeDebug';

const alt = new Alt();
//chromeDebug(alt);

export default alt;
```

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



There is a Chrome plugin known as [alt-devtool](https://github.com/goatslacker/alt-devtool)⁵. After it is installed, you can connect to Alt by uncommenting the related lines above. You can use it to debug the state of your stores, search, and travel in time.

Defining CRUD API for Notes

Next we'll need to define a basic API for operating over the Note data. To keep this simple, we can CRUD (Create, Read, Update, Delete) it. Given Read is implicit, we won't be needing that. We can model the rest as actions, though. Alt provides a shorthand known as `generateActions`. We can use it like this:

`app/actions/NoteActions.js`

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

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

5.3 Defining a Store for Notes

A store is a single source of truth for a part of your application state. In this case, we need one to maintain the state of the notes. We will connect all the actions we defined above using the `bindActions` function.

We have the logic we need for our store already at App. We will move that logic to `NoteStore`.

Setting Up a Skeleton

As a first step, we can set up a skeleton for our store. We can fill in the methods we need after that. Alt uses standard ES6 classes, so it's the same syntax as we saw earlier with React components. Here's a starting point:

`app/stores/NoteStore.js`

⁵<https://github.com/goatslacker/alt-devtool>

```
import uuid from 'node-uuid';
import alt from '../libs/alt';
import NoteActions from '../actions/NoteActions';

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

    this.notes = [];
  }
  create(note) {

  }
  update({id, task}) {

  }
  delete(id) {

  }
}

export default alt.createStore(NoteStore, 'NoteStore');
```

We call `bindActions` to map each action to a method by name. We trigger the appropriate logic at each method based on that. Finally we connect the store with Alt using `alt.createStore`.

Note that assigning a label to a store (`NoteStore` in this case) isn't required. It is a good practice as it protects the code against minification and possible collisions. These labels become important when we persist the data.

Implementing create

Compared to the earlier logic, `create` will generate an `id` for a `Note` automatically. This is a detail that can be hidden within the store:

`app/stores/NoteStore.js`

```
import uuid from 'node-uuid';
import alt from '../libs/alt';
import NoteActions from '../actions/NoteActions';

class NoteStore {
  constructor() {
    ...
  }
  create(note) {
    const notes = this.notes;

    note.id = uuid.v4();

    this.setState({
      notes: notes.concat(note)
    });
  }
  ...
}

export default alt.createStore(NoteStore, 'NoteStore');
```

To keep the implementation clean, we are using `this.setState`. It is a feature of Alt that allows us to signify that we are going to alter the store state. Alt will signal the change to possible listeners.

Implementing update

`update` follows the earlier logic apart from some renaming. Most importantly we commit the new state through `this.setState`:

app/stores/NoteStore.js

```
...

class NoteStore {
  ...
  update({id, task}) {
    const notes = this.notes.map((note) => {
      if(note.id === id) {
        note.task = task;
      }
    })
  }
}
```



```

        return note;
    });

    this.setState({notes});
  }
  delete(id) {

  }
}

export default alt.createStore(NoteStore, 'NoteStore');
```

We have one final operation left, delete.



{notes} is known as a an ES6 feature known as [property shorthand](#)⁶. This is equivalent to {notes: notes}.

Implementing delete

delete is straightforward. Seek and destroy, as earlier, and remember to commit the change:
app/stores/NoteStore.js

```

...

class NoteStore {
  ...
  delete(id) {
    this.setState({
      notes: this.notes.filter((note) => note.id !== id)
    });
  }
}

export default alt.createStore(NoteStore, 'NoteStore');
```

Instead of slicing and concatenating data, it would be possible to operate directly on it. For example a mutable variant such as `this.notes.splice(targetId, 1)` would work. We could also use a shorthand, such as `[...notes.slice(0, noteIndex), ...notes.slice(noteIndex + 1)]`. The exact solution depends on your preferences. I prefer to avoid mutable solutions (i.e., `splice`) myself.

⁶https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer

It is recommended that you use `setState` with `Alt` to keep things clean and easy to understand. Manipulating `this.notes` directly would work, but that would miss the intent and could become problematic in larger scale as mutation is difficult to debug. `setState` provides a nice analogue to the way React works so it's worth using.

We have almost integrated Flux with our application now. We have a set of actions that provide an API for manipulating Notes data. We also have a store for actual data manipulation. We are missing one final bit, integration with our view. It will have to listen to the store and be able to trigger actions to complete the cycle.



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. [Flow⁷](http://flowtype.org/) based gradual typing provides one way to do this. Alternatively you could write nice tests. That's a good idea regardless.

5.4 Gluing It All Together

Gluing this all together is a little complicated as there are multiple concerns to take care of. Dealing with actions is going to be easy. For instance, to create a Note, we would need to trigger `NoteActions.create({task: 'New task'})`. That would cause the associated store to change and, as a result, all the components listening to it.

Our `NoteStore` provides two methods in particular that are going to be useful. These are `NoteStore.listen` and `NoteStore.unlisten`. They will allow views to subscribe to the state changes.

As you might remember from the earlier chapters, React provides a set of lifecycle hooks. We can subscribe to `NoteStore` within our view at `componentDidMount` and `componentWillUnmount`. By unsubscribing, we avoid possible memory leaks.

Based on these ideas we can connect App with `NoteStore` and `NoteActions`:

app/components/App.jsx

```
import React from 'react';
import Notes from './Notes.jsx';
import NoteActions from '../actions/NoteActions';
import NoteStore from '../stores/NoteStore';

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

⁷<http://flowtype.org/>

```

    this.state = NoteStore.getState();
  }
  componentDidMount() {
    NoteStore.listen(this.storeChanged);
  }
  componentWillUnmount() {
    NoteStore.unlisten(this.storeChanged);
  }
  storeChanged = (state) => {
    // Without a property initializer `this` wouldn't
    // point at the right context (defaults to `undefined` in strict mode).
    this.setState(state);
  }
  render() {
    const notes = this.state.notes;

    return (
      <div>
        <button className="add-note" onClick={this.addNote}></button>
        <Notes items={notes}
          onEdit={this.editNote} onDelete={this.deleteNote} />
      </div>
    );
  }
  addNote() {
    NoteActions.create({task: 'New task'});
  }
  editNote(id, task) {
    NoteActions.update({id, task});
  }
  deleteNote(id) {
    NoteActions.delete(id);
  }
}

```

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.

What's the Point?

Even though integrating Alt 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? It would be natural to plug that into our `NoteStore`. Alternatively we could do something more generic as we'll be doing next.
2. What if we had many components relying on the data? We would just consume `NoteStore` 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.

This is what makes Flux a strong architecture when used with React. It isn't hard to find answers to questions like these. Even though there is more code, it is easier to reason about. Given we are dealing with a unidirectional flow we have something that is simple to debug and test.

5.5 Implementing Persistency over `localStorage`

We will modify our implementation of `NoteStore` to persist the data on change. This way we don't lose our data after a refresh. One way to achieve this is to use `localStorage`⁸. It is a well supported feature that allows you to persist data to the browser.

Understanding `localStorage`

`localStorage` has a sibling known as `sessionStorage`. Whereas `sessionStorage` loses its data when the browser is closed, `localStorage` retains its data. They both share [the same API](#)⁹ 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.

Note that it is convenient to operate on the API using your browser developer tools. For instance, in Chrome you can see the state of the storages through the *Resources* tab. *Console* tab allows you to

⁸<https://developer.mozilla.org/en/docs/Web/API/Window/localStorage>

⁹https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API/Using_the_Web_Storage_API

perform direct operations on the data. You can even use `storage.key` and `storage.key = 'value'` shorthands for quick modifications.

`localStorage` and `sessionStorage` can use up to 10 MB of data combined. Even though they are well supported, there are certain corner cases with interesting failures. 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 Overflow¹⁰](https://stackoverflow.com/questions/14555347/html5-localstorage-error-with-safari-quota-exceeded-err-dom-exception-22-an) for details.

Implementing a Wrapper for `localStorage`

To keep things simple and manageable, we can implement a little wrapper for storage. It will wrap all of these complexities. The API expects strings.

As objects are convenient, we'll use `JSON.parse` and `JSON.stringify` for serialization. We need just `storage.get(k)` and `storage.set(k, v)` as seen in the implementation below:

`app/libs/storage.js`

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

The implementation could be generalized further. You could convert it into a factory `((storage) => { ... })` and make it possible to swap the storage. Now we are stuck with `localStorage` unless we change the code.

¹⁰<https://stackoverflow.com/questions/14555347/html5-localstorage-error-with-safari-quota-exceeded-err-dom-exception-22-an>



We're operating with `localStorage` directly to keep the implementation simple. An alternative would be to use [localForage](https://github.com/mozilla/localForage)¹¹ to hide all the complexity. You could even integrate it behind our interface.

Persisting Application Using `FinalStore`

Besides this little utility, we'll need to adapt our application to use it. `Alt` provides a built-in store called `FinalStore` which is perfect for this purpose. We can persist the entire state of our application using `FinalStore`, bootstrapping, and snapshotting. `FinalStore` is a store that listens to all existing stores. Every time some store changes, `FinalStore` will know about it. This makes it ideal for persistency.

We can take a snapshot of the entire app state and push it to `localStorage` every time `FinalStore` changes. That solves one part of the problem. Bootstrapping solves the remaining part as `alt.bootstrap` allows us to set state of the all stores. In our case, we'll fetch the data from `localStorage` and invoke it to populate our stores. This is handy for other cases as well. The data can come from elsewhere, through a `WebSocket` for instance.



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

In order to integrate this idea to our application we will need to implement a little module to manage it. We take the possible initial data into account there and trigger the new logic.

`app/libs/persist.js` does the hard part. It will set up a `FinalStore`, deal with bootstrapping (restore data) and snapshotting (save data). I have included an escape hatch in the form of the debug flag. If it is set, the data won't get saved to `localStorage`. The reasoning is that by doing this, you can set the flag (`localStorage.setItem('debug', 'true')`), hit `localStorage.clear()` and refresh the browser to get a clean slate. The implementation below illustrates these ideas:

`app/libs/persist.js`

¹¹<https://github.com/mozilla/localForage>

```

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

export default function(alt, storage, storeName) {
  const finalStore = makeFinalStore(alt);

  try {
    alt.bootstrap(storage.get(storeName));
  }
  catch(e) {
    console.error('Failed to bootstrap data', e);
  }

  finalStore.listen(() => {
    if(!storage.get('debug')) {
      storage.set(storeName, alt.takeSnapshot());
    }
  });
}

```

Finally, we need to trigger the persistency logic at initialization. We will need to pass the relevant data to it (Alt instance, storage, storage name) and off we go.

app/index.jsx

```

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

main();

function main() {
  persist(alt, storage, 'app');

  ...
}

```

If you try refreshing the browser now, the application should retain its state. The solution should scale with minimal effort if we add more stores to the system. Integrating a real back-end wouldn't be a problem. There are hooks in place for that now.

You could, for instance, pass the initial payload as a part of your HTML (isomorphic rendering), load it up, and then persist the data to the back-end. You have a great deal of control over how to do this, and you can use `localStorage` as a backup if you want.

Isomorphic rendering is a powerful technique that allows you to use React to improve the performance of your application while gaining SEO benefits. Rather than leaving all rendering to the front-end, we perform a part of it at the back-end side. We render the initial application markup at back-end and provide it to the user. React will pick that up. This can also include data that can be loaded to your application without having to perform extra queries.



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. There are no easy solutions. Regardless, this is something to keep in mind when developing something more sophisticated. The lesson here is that the more you inject state to your application, the more complicated it gets.

5.6 Using the AltContainer

The [AltContainer](http://alt.js.org/docs/components/altContainer/)¹² wrapper allows us to simplify connection logic greatly and cut down the amount of logic needed. To get started, install it using:

```
npm i alt-container --save
```

The implementation below illustrates how to bind it all together. Note how much code we can remove!

app/components/App.jsx

```
import AltContainer from 'alt-container';
import React from 'react';
import Notes from './Notes.jsx';
import NoteActions from '../actions/NoteActions';
import NoteStore from '../stores/NoteStore';

export default class App extends React.Component {
  render() {
    return (
      <div>
        <button className="add-note" onClick={this.addNote}></button>
        <AltContainer
          stores={[NoteStore]}
          inject={{
```

¹²<http://alt.js.org/docs/components/altContainer/>


```

        items: () => NoteStore.getState().notes
      }}
    >
    <Notes onEdit={this.editNote} onDelete={this.deleteNote} />
  </AltContainer>
</div>
);
}
...
}

```

The `AltContainer` allows us to bind data to its immediate children. In this case it injects the `items` property in to `Notes`. The pattern allows us to set up arbitrary connections to multiple stores and manage them. You can find another possible approach at the appendix about decorators.

Integrating the `AltContainer` tied this component to `Alt`. If you wanted something forward-looking, you could push it into a component of your own. That facade would hide `Alt` and allow you to replace it with something else later on.

5.7 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.

5.8 Relay?

Facebook's [Relay](https://facebook.github.io/react/blog/2015/02/20/introducing-relay-and-graphql.html)¹³ 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.

5.9 Conclusion

In this chapter you saw how to port our simple application to use Flux architecture. In the process we learned about basic concepts of Flux. Now we are ready to start adding more functionality to our application.

¹³<https://facebook.github.io/react/blog/2015/02/20/introducing-relay-and-graphql.html>

6. From Notes to Kanban



Kanban board

So far we have managed to set up a nice little development environment. We have developed an application for keeping track of notes in `localStorage`. We still have work to do to turn this into a real Kanban as pictured above. Most importantly our system is missing 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.

6.1 Extracting 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 uuid from 'node-uuid';
import alt from '../libs/alt';
import LaneActions from '../actions/LaneActions';

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

    this.lanes = [];
  }
  create(lane) {
    const lanes = this.lanes;

    lane.id = uuid.v4();
    lane.notes = lane.notes || [];

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

export default alt.createStore(LaneStore, 'LaneStore');
```

We are also going to need a stub for `Lanes`. We will expand this later. For now we just want something simple to show up.

app/components/Lanes.jsx

```
import React from 'react';

export default class Lanes extends React.Component {
  render() {
    return (
      <div className="lanes">
        lanes should go here
      </div>
    );
  }
}
```

Next we need to make room for Lanes at App. We will simply replace Notes references with Lanes, set up actions, and store as needed:

app/components/App.jsx

```
import AltContainer from 'alt-container';
import React from 'react';
import Lanes from './Lanes.jsx';
import LaneActions from '../actions/LaneActions';
import LaneStore from '../stores/LaneStore';

export default class App extends React.Component {
  render() {
    return (
      <div>
        <button className="add-lane" onClick={this.addItem}></button>
        <AltContainer
          stores={[LaneStore]}
          inject={{
            items: () => LaneStore.getState().lanes || []
          }}
        >
          <Lanes />
        </AltContainer>
      </div>
    );
  }
  addItem() {
    LaneActions.create({name: 'New lane'});
  }
}
```

The current implementation doesn't do much. It just shows a plus button and *lanes should go here* text. Even the add button doesn't work yet. We still need to model Lane and attach Notes to that to make this all work.

6.2 Modeling Lane

The Lanes container will render each Lane separately. Each Lane in turn will then render associated Notes just like our App did earlier. Lanes is analogous to Notes in this manner. The example below illustrates how to set this up:

app/components/Lanes.jsx

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

export default class Lanes extends React.Component {
  render() {
    const lanes = this.props.items;

    return <div className="lanes">{lanes.map(this.renderLane)}</div>;
  }
  renderLane(lane) {
    return <Lane className="lane" key={lane.id} lane={lane} />;
  }
}
```

We are also going to need a Lane component to make this work. It will render the Lane name and associated Notes. The example below has been modeled largely after our earlier implementation of App. It will render an entire lane including its name and associated notes:

app/components/Lane.jsx

```
import AltContainer from 'alt-container';
import React from 'react';
import Notes from './Notes.jsx';
import NoteActions from '../actions/NoteActions';
import NoteStore from '../stores/NoteStore';

export default class Lane extends React.Component {
  render() {
    const {lane, ...props} = this.props;
```

```

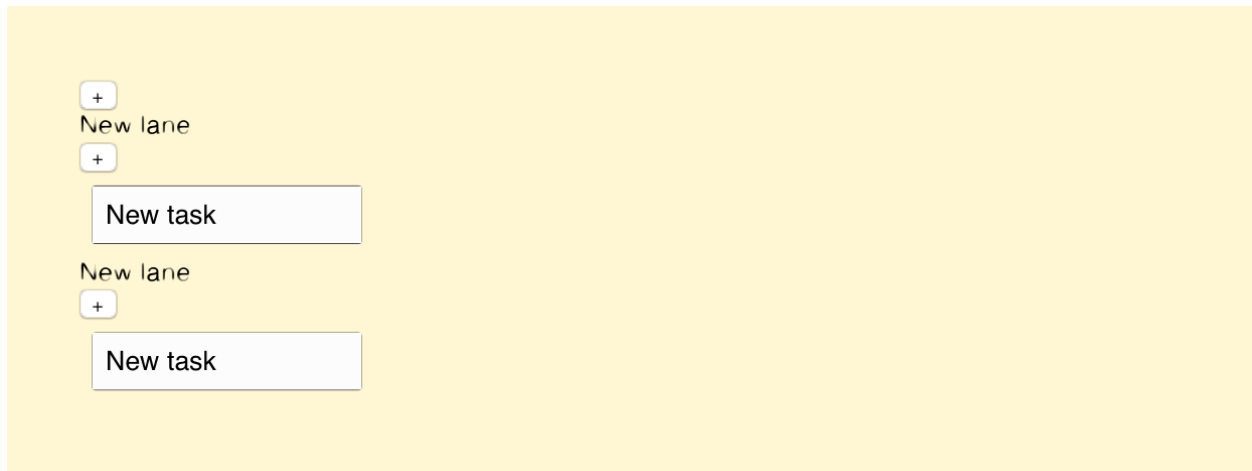
return (
  <div {...props}>
    <div className="lane-header">
      <div className="lane-name">{lane.name}</div>
      <div className="lane-add-note">
        <button onClick={this.addNote}>+</button>
      </div>
    </div>
    <AltContainer
      stores={[NoteStore]}
      inject={{
        items: () => NoteStore.getState().notes || []
      }}
    >
      <Notes onEdit={this.editNote} onDelete={this.deleteNote} />
    </AltContainer>
  </div>
);
}
addNote() {
  NoteActions.create({task: 'New task'});
}
editNote(id, task) {
  NoteActions.update({id, task});
}
deleteNote(id) {
  NoteActions.delete(id);
}
}

```

I am using [Object rest spread syntax \(stage 2\)](https://github.com/sebmarkbage/ecmascript-rest-spread)¹ (`const {a, b, ...props} = this.props`) in the example. This allows us to attach a `className` to `Lane` and we avoid polluting it with HTML attributes we don't need. The syntax expands Object key value pairs as props so we don't have to write each prop we want separately.

If you run the application, you can see there's something wrong. If you add new Notes to a Lane, the Note appears to each Lane. Also if you modify a Note, the other Lanes update, too.

¹<https://github.com/sebmarkbage/ecmascript-rest-spread>



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.

6.3 Making Lanes Responsible of Notes

Currently our `Lane` model is simple. We are just storing an array of objects. Each of the objects knows its *id* and *name*. We'll need something more. 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.

Setting Up `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>})`. As a `Note` needs to exist before this association can be made, this method needs to trigger `waitFor`.

`waitFor` tells the dispatcher that it should wait before going on. A line such as `this.waitFor(NoteStore);` at `LaneStore` would make the operation wait. It will resume only after `NoteStore` has finished its work. Here's an example of how it would work:

```
NoteActions.create({task: 'New task'});
```

```
// triggers waitFor
```

```
LaneActions.attachToLane({laneId});
```

`waitFor` is a feature that should be used carefully. Always consider other alternatives before using it. In this particular case, there's no easy way around it.

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');
```

The next step takes more code. We need to take care of attaching:

app/stores/LaneStore.js

```
import uuid from 'node-uuid';
import alt from '../libs/alt';
import LaneActions from '../actions/LaneActions';
import NoteStore from './NoteStore';

class LaneStore {
  ...
  attachToLane({laneId, noteId}) {
    if(!noteId) {
      this.waitFor(NoteStore);

      noteId = NoteStore.getState().notes.slice(-1)[0].id;
    }

    const lanes = this.lanes.map((lane) => {
      if(lane.id === laneId) {
        if(lane.notes.indexOf(noteId) === -1) {
          lane.notes.push(noteId);
        }
        else {
          console.warn('Already attached note to lane', lanes);
        }
      }
    });

    return lane;
  });

  this.setState({lanes});
}

export default alt.createStore(LaneStore, 'LaneStore');
```


`attachToLane` has been coded defensively to guard against possible problems. Unless we pass `noteId`, we wait for one. Passing one explicitly becomes useful when we implement drag and drop, so it's good to have it in place.

The rest of the code deals with the logic. First we try to find a matching lane. If found, we attach the note id to it unless it has been attached already.

Setting Up `detachFromLane`

`deleteNote` is the opposite operation of `addNote`. When removing a Note, it's important to remove its association with a Lane as well. For this purpose we can implement `LaneActions.detachFromLane({laneId: <id>})`. We would use it like this:

```
LaneActions.detachFromLane({laneId, noteId});
NoteActions.delete(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 uuid from 'node-uuid';
import alt from '../libs/alt';
import LaneActions from '../actions/LaneActions';
import NoteStore from './NoteStore';

class LaneStore {
  attachToLane({laneId, noteId}) {
    ...
  }
  detachFromLane({laneId, noteId}) {
    const lanes = this.lanes.map((lane) => {
      if(lane.id === laneId) {
        lane.notes = lane.notes.filter((note) => note !== noteId);
      }
    });
  }
}
```

```

        return lane;
    });

    this.setState({lanes});
  }
}

export default alt.createStore(LaneStore, 'LaneStore');
```

Again, the implementation has been coded with drag and drop in mind. Later on we'll want to pass `noteId` explicitly, so it doesn't hurt to have it there. You've seen the rest of the code earlier in different contexts.

Implementing a Getter for NoteStore

Given our lanes contain references to notes through ids, we are going to need some way to resolve those ids to actual notes. One neat way to do this is to implement a public method `NoteStore.get(notes)` for the purpose. It accepts an array of Note ids, and returns corresponding objects.

This can be achieved using the `map` operation. First we need to get the ids of all notes to match against. After that, we can perform a lookup for each note id passed in using `indexOf`.

Just implementing the method isn't enough. We also need to make it public. In Alt this can be achieved using `this.exportPublicMethods`. It takes an object that describes the public interface of the store in question. Consider the implementation below:

app/stores/NoteStore.jsx

```

import uuid from 'node-uuid';
import alt from '../libs/alt';
import NoteActions from '../actions/NoteActions';

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

    this.notes = [];

    this.exportPublicMethods({
      get: this.get.bind(this)
    });
  }
}
```

```

...
get(ids) {
  return (ids || []).map(
    (id) => this.notes.filter((note) => note.id === id)
  ).filter((a) => a).map((a) => a[0]);
}
}

export default alt.createStore(NoteStore, 'NoteStore');

```

Note that the implementation filters possible non-matching ids from the result.

Connecting Lane with the Logic

Now that we have the logical bits together, we can integrate it with Lane. We'll need to take the newly added props (id, notes) into account, and glue this all together:

app/components/Lane.jsx

```

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

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

    const id = props.lane.id;

    this.addNote = this.addNote.bind(this, id);
    this.deleteNote = this.deleteNote.bind(this, id);
  }
  render() {
    const {lane, ...props} = this.props;

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

```

```

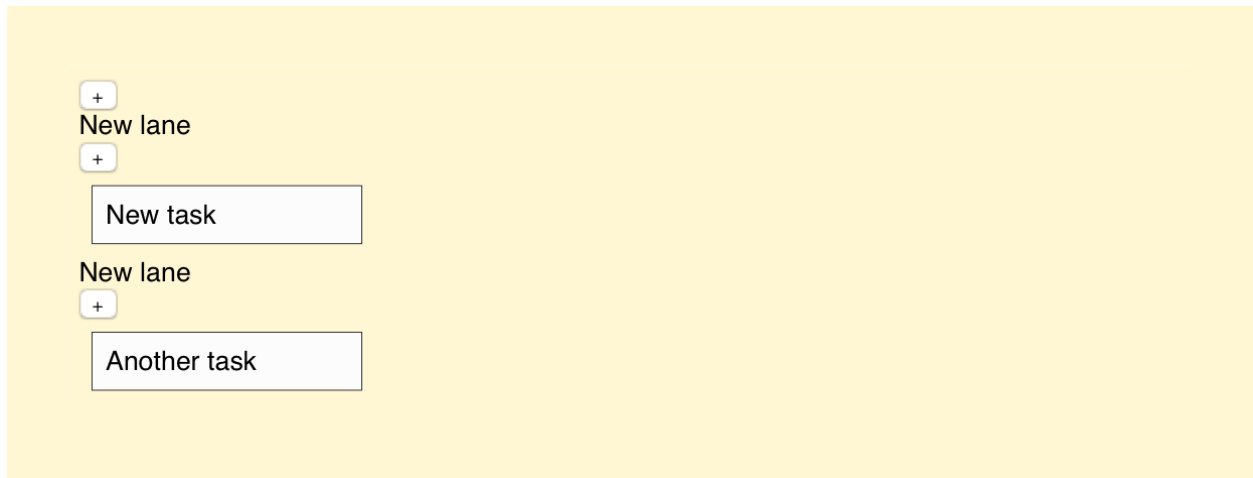
    <AltContainer
      stores={[NoteStore]}
      inject={{
        items: () => NoteStore.get(lane.notes)
      }}
    >
      <Notes
        onEdit={this.editNote}
        onDelete={this.deleteNote} />
      </AltContainer>
    </div>
  );
}
addNote(laneId) {
  NoteActions.create({task: 'New task'});
  LaneActions.attachToLane({laneId});
}
editNote(id, task) {
  NoteActions.update({id, task});
}
deleteNote(laneId, noteId) {
  LaneActions.detachFromLane({laneId, noteId});
  NoteActions.delete(noteId);
}
}

```

There are two important changes:

- `items: () => NoteStore.get(notes)` - Our new getter is used to filter notes.
- `addNote`, `deleteNote` - These operate now based on the new logic we specified. Note that we trigger `detachFromLane` before `delete` at `deleteNote`. Otherwise we may try to render non-existent notes. You can try swapping the order to see warnings.

After these changes, we now have a system that can maintain relations between Lanes and 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.



Separate notes

6.4 Implementing Edit/Remove for Lane

We are still missing some basic functionality such as editing and removing lanes. Copy *Note.jsx* as *Editable.jsx*. We'll get back to that original *Note.jsx* later in this project. For now, we just want to get *Editable* into a good condition. Tweak the code as follows to generalize the implementation:

app/components/Editable.jsx

```
import React from 'react';

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

    this.state = {
      editing: false
    };
  }
  render() {
    const {value, onEdit, ...props} = this.props;
    const editing = this.state.editing;

    return (
      <div {...props}>
        {editing ? this.renderEdit() : this.renderValue()}
      </div>
    );
  }
}
```

```

    }
    renderEdit = () => {
      return <input type="text"
        autoFocus={true}
        defaultValue={this.props.value}
        onBlur={this.finishEdit}
        onKeyDown={this.checkEnter} />;
    }
    renderValue = () => { // drop renderNote
      const onDelete = this.props.onDelete;

      return (
        <div onClick={this.edit}>
          <span className="value">{this.props.value}</span>
          {onDelete ? this.renderDelete() : null }
        </div>
      );
    }
    renderDelete = () => {
      return <button className="delete" onClick={this.props.onDelete}>x</button>;
    }
    ...
  }

```

There are a couple of important changes:

- `{editing ? this.renderEdit() : this.renderValue()}` - This ternary selects what to render based on the editing state. Previously we had `Task`. Now we are using the term `Value` as that's more generic.
- `const {value, onEdit, ...props} = this.props;` - We changed `task` to `value` here as well.
- `renderValue()` - Formerly this was known as `renderNote()`. Again, an abstraction step. Note that we refer to `this.props.value` and not `this.props.task`.

Because the class name changes, *main.css* needs a small tweak:

app/main.css

```

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

```



Our current implementation of `Editable` encapsulates its editing state. This makes it impossible to control outside of the component. One way to resolve this issue would be to eliminate the internal state and provide enough API to control it. This would mean controlling editing state through a prop and allowing the consumer to react to the editing states through hooks such as `onBeginEdit` and `onFinishEdit`.

Pointing Notes to Editable

Next we need to make `Notes.jsx` point at the new component. We'll need to alter the import and the component name at `render()`:

`app/components/Notes.jsx`

```

import React from 'react';
import Editable from './Editable.jsx';

export default class Notes extends React.Component {
  ...
  renderNote = (note) => {
    // note that we pass task to Editable through `value` prop now!
    return (
      <li className="note" key={note.id}>
        <Editable
          value={note.task}
          onEdit={this.props.onEdit.bind(null, note.id)}
          onDelete={this.props.onDelete.bind(null, note.id)} />
      </li>
    );
  }
}

```

Connecting Lane with Editable

Next we can use this generic component to allow a Lane's name to be modified. This will give a hook for our logic. We'll need to alter `<div className='lane-name'>{name}</div>` as follows:

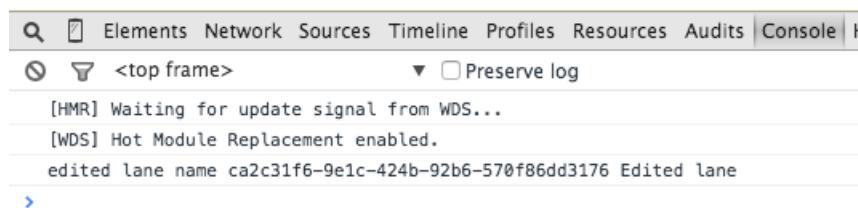
`app/components/Lane.jsx`

```
...
import Editable from './Editable.jsx';

export default class Lane extends React.Component {
  constructor(props) {
    ...
    this.editName = this.editName.bind(this, id);
  }
  render() {
    const {lane, ...props} = this.props;

    return (
      <div {...props}>
        <div className="lane-header">
          <Editable className="lane-name" value={lane.name}
            onEdit={this.editName} />
          <div className="lane-add-note">
            <button onClick={this.addNote}>+</button>
          </div>
        </div>
        ...
      </div>
    )
  }
  ...
  editName(id, name) {
    console.log('edited lane name', id, name);
  }
}
```

If you try to edit a lane name now, you should see a log at the console.



Logging lane name editing

Defining Editable Logic

We will need to define some logic to make this work. To follow the same idea as with Note, we can model the remaining CRUD actions here. We'll need to set up update and delete actions in particular.

app/actions/LaneActions.js

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

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

We are also going to need LaneStore level implementations for these. They can be modeled based on what we have seen in NoteStore earlier:

app/stores/LaneStore.js

```
...

class LaneStore {
  ...
  update({id, name}) {
    const lanes = this.lanes.map((lane) => {
      if(lane.id === id) {
        lane.name = name;
      }

      return lane;
    });

    this.setState({lanes});
  }
  delete(id) {
    this.setState({
      lanes: this.lanes.filter((lane) => lane.id !== id)
    });
  }
  attachToLane({laneId, noteId}) {
    ...
  }
}
```

```

    ...
  }

  export default alt.createStore(LaneStore, 'LaneStore');

```



If a lane is deleted, it would be a good idea to get rid of the associated notes as well. In the current implementation they are left hanging in the NoteStore. It doesn't hurt the functionality but it's one of those details that you may want to be aware of.

Now that we have resolved actions and store, we need to adjust our component to take these changes into account:

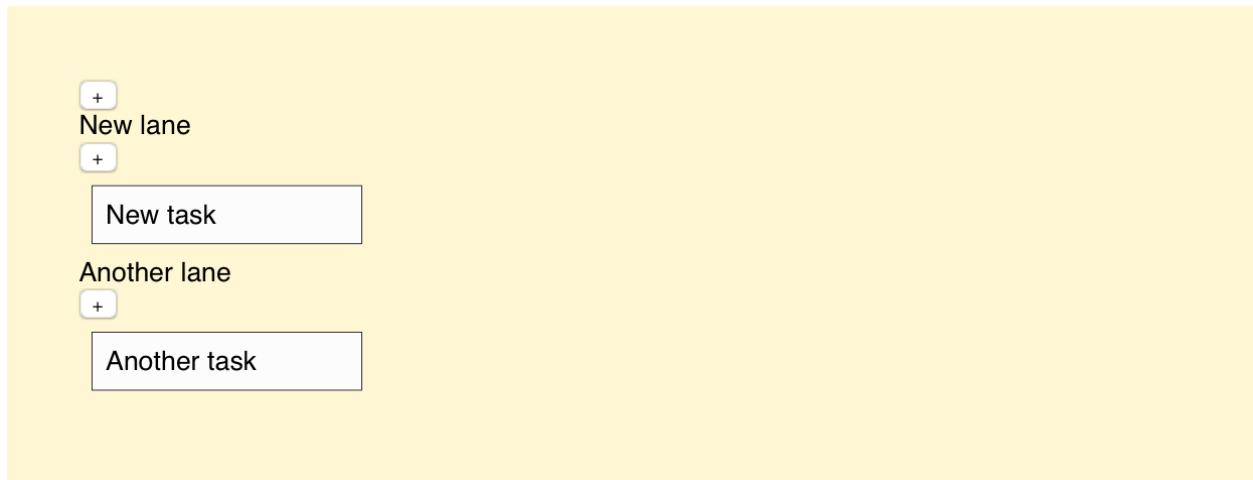
app/components/Lane.jsx

```

...
export default class Lane extends React.Component {
  ...
  editName(id, name) {
    if(name) {
      LaneActions.update({id, name});
    }
    else {
      LaneActions.delete(id);
    }
  }
}

```

Try modifying a lane name now. Modifications now should get saved the same way as they do for notes. Deleting lanes should be possible as well.



Editing a lane name

6.5 Styling Kanban Board

As we added Lanes to the application, the styling went a bit off. Add the following styling to make it a little nicer:

app/main.css

```
body {  
  background: cornsilk;  
  
  font-family: sans-serif;  
}  
  
.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;
}

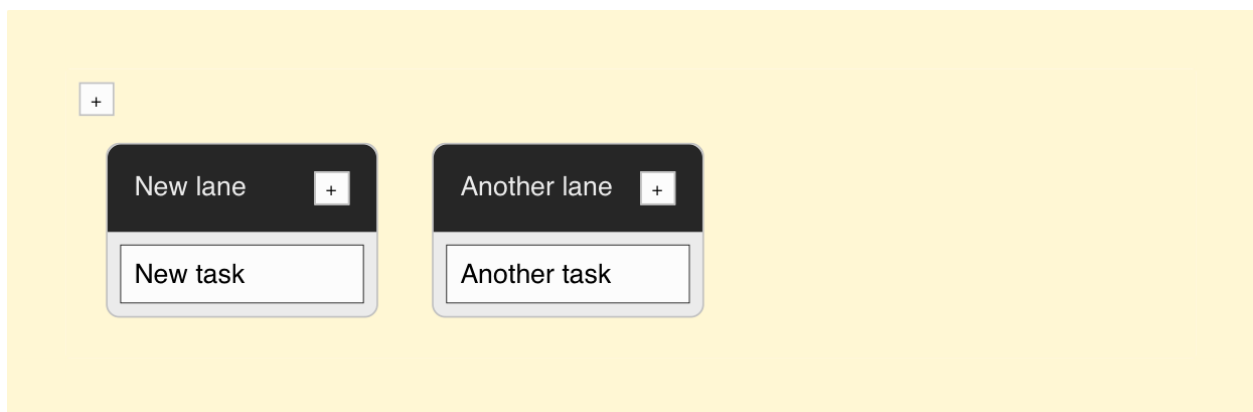
.lane-add-note {
    float: right;

    margin-left: 0.5em;
}

.add-lane, .lane-add-note button {
    background-color: #fdfdfd;
    border: 1px solid #ccc;
}

...
```

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`).

Besides keeping things nice and tidy, Webpack's lazy loading machinery can pick this up. As a result, the initial CSS your user has to load will be smaller. I go into further detail later as I discuss styling.

6.6 On Namespacing Components

So far we've been defining a component per file. That's not the only way. It may be handy to treat a file as a namespace and expose multiple components from it. React provides [namespaces components](https://facebook.github.io/react/docs/jsx-in-depth.html#namespaced-components)² just for this purpose. In this case we could apply namespacing to the concept of Lane or Note. This would add some flexibility to our system while keeping it simple to manage. By using namespacing we could do something like this:

`app/components/Lanes.jsx`

...

```
export default class Lanes extends React.Component {
  render() {
    const lanes = this.props.items;

    return <div className="lanes">{lanes.map(this.renderLane)}</div>;
  }
  renderLane(lane) {
    // new
    return (
      <Lane className="lane" key={lane.id}>
        <Lane.Header id={lane.id} name={lane.name} />
        <Lane.Notes id={lane.id} notes={lane.notes} />
      </Lane>
    );

    // old
    // return <Lane className="lane" key={lane.id} lane={lane} />;
  }
}
```

`app/components/Lane.jsx`

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

```
...

class Lane extends React.Component {
  ...
}

Lane.Header = class LaneHeader extends React.Component {
  ...
}
Lane.Notes = class LaneNotes extends React.Component {
  ...
}

export default Lane;
```

Now we have pushed the control over Lane formatting to a higher level. In this case the change isn't worth it, but it can make sense in a more complex case.

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.

6.7 Conclusion

The current design has been optimized with drag and drop operations in mind. Moving notes within a lane is a matter of swapping ids. Moving notes from one lane to another is again an operation over ids. This structure leads to some complexity as we need to track ids, but it will pay off in the next chapter.

There isn't always a clear cut way to model data and relations. In other scenarios, we could push the references elsewhere. For instance, the note to lane relation could be inversed and pushed to `Note` level. We would still need to track their order within a lane somehow. We would be pushing the complexity elsewhere by doing this.

Currently `NoteStore` is treated as a singleton. Another way to deal with it would be to create `NoteStore` per `Notes` dynamically. Even though this simplifies dealing with the relations somewhat, this is a Flux anti-pattern better avoided.

We still cannot move notes between lanes or within a lane. We will solve that in the next chapter as we implement drag and drop.

7. Implementing Drag and Drop

Our Kanban application is almost usable now. It looks alright and there's some basic functionality in place. In this chapter I'll show you how to take it to the next level. We will integrate some drag and drop functionality as we set up [React DnD](https://github.com/react-dnd/react-dnd)¹. After this chapter you should be able to sort notes within a lane and drag them from one lane to another.

7.1 Setting Up React DnD

Before going further hit

```
npm i react-dnd react-dnd-html5-backend --save
```

to add React DnD to the project.

As a first step we'll need to connect it with our project. Currently it provides an HTML5 Drag and Drop API specific back-end. There's no official support for touch yet, but it's possible to add later on. In order to set it up, we need to use the `DragDropContext` decorator and provide the back-end to it:

app/components/App.jsx

```
...
import {DragDropContext} from 'react-dnd';
import HTML5Backend from 'react-dnd-html5-backend';

@DragDropContext(HTML5Backend)
export default class App extends React.Component {
  ...
}
```

After this change the application should look exactly the same as before. We are now ready to add some sweet functionality to it.



Decorators provide us simple means to annotate our components. Alternatively we could use syntax such as `DragDropContext(HTML5Backend)(App)` but this would get rather unwieldy when we want to apply multiple decorators. It's a valid alternative, though. See the decorator appendix to understand in detail how they work and how to implement them yourself.

¹<https://github.com/gaearon.github.io/react-dnd/>



Back-ends allow us to customize React DnD behavior. For instance, we can add [support for touch²](#) to our application using one. There's also a testing specific one available.

7.2 Preparing Notes to Be Sorted

Next we will need to tell React DnD what can be dragged and where. Since we want to move notes, we'll need to annotate them accordingly. In addition, we'll need some logic to tell what happens during this process.

Earlier we extracted editing functionality from `Note` and ended up dropping `Note`. It seems like we'll want to add that concept back to allow drag and drop.

We can use a handy little technique here that allows us to avoid code duplication. We can implement `Note` as a wrapper component. It will accept `Editable` and render it. This will allow us to keep DnD related logic in `Note`. This avoids having to duplicate any logic related to `Editable`.

The magic lies in a single property known as `children`. React will render possible child components in the slot `{this.props.children}`. Set up `Note.jsx` as shown below:

app/components/Note.jsx

```
import React from 'react';

export default class Note extends React.Component {
  render() {
    return (
      <li {...this.props}>{this.props.children}</li>
    );
  }
}
```

We also need to tweak `Notes` to use our wrapper component. We will simply wrap `Editable` using `Note`, and we are good to go. We will pass `note` data to the wrapper as we'll need that later when dealing with logic:

app/components/Notes.jsx

²<https://github.com/gaearon/react-dnd/pull/240>


```

...
import Note from './Note.jsx';

export default class Notes extends React.Component {
  ...
  renderNote = (note) => {
    return (
      <Note className="note" id={note.id} key={note.id}>
        <Editable
          value={note.task}
          onEdit={this.props.onEdit.bind(null, note.id)}
          onDelete={this.props.onDelete.bind(null, note.id)} />
      </Note>
    );
  }
}

```

After this change the application should look exactly the same as before. We have achieved nothing yet. Fortunately we can start adding functionality, now that we have the foundation in place.

7.3 Allowing Notes to Be Dragged

React DnD uses constants to tell different 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 to the system.

Next we need to tell our Note that it's possible to drag and drop it. This is done through `@DragSource` and `@DropTarget` annotations.

Setting Up Note @DragSource

Marking a component as a `@DragSource` simply means that it can be dragged. Set up the annotation as follows:

app/components/Note.jsx

```

...
import {DragSource} from 'react-dnd';
import ItemTypes from '../constants/itemTypes';

const noteSource = {
  beginDrag(props) {
    console.log('begin dragging note', props);

    return {};
  }
};

@DragSource(ItemTypes.NOTE, noteSource, (connect) => ({
  connectDragSource: connect.dragSource()
}))
export default class Note extends React.Component {
  render() {
    const {connectDragSource, id, onMove, ...props} = this.props;

    return connectDragSource(
      <li {...props}>{props.children}</li>
    );
  }
}

```

There are a couple of important changes:

- We set up imports for the new logic.
- We defined a `noteSource`. It contains a `beginDrag` handler. We can set the initial state for dragging here. For now we just have a debug log there.
- `@DragSource` connects `NOTE` item type with `noteSource`.
- `id` and `onMove` props are extracted from `this.props`. We'll use these later on to set up a callback so that the parent of a `Note` can deal with the moving related logic.
- Finally `connectDragSource` prop wraps the element at `render()`. It could be applied to a specific part of it. This would be handy for implementing handles for example.

If you drag a `Note` now, you should see a debug log at the console.

We still need to make sure `Note` works as a `@DropTarget`. Later on this will allow swapping them as we add logic in place.



Note that React DnD doesn't support hot loading perfectly. You may need to refresh the browser to see the logs you expect!

Setting Up Note @DropTarget

@DropTarget allows a component to receive components annotated with @DragSource. As @DropTarget triggers, we can perform actual logic based on the components. Expand as follows:

app/components/Note.jsx

```
...
import {DragSource, DropTarget} from 'react-dnd';
import ItemTypes from '../constants/itemTypes';

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);
  }
};

@DragSource(ItemTypes.NOTE, noteSource, (connect) => ({
  connectDragSource: connect.dragSource()
}))
@DropTarget(ItemTypes.NOTE, noteTarget, (connect) => ({
  connectDropTarget: connect.dropTarget()
}))
export default class Note extends React.Component {
  render() {
    const {connectDragSource, connectDropTarget,
      id, onMove, ...props} = this.props;

    return connectDragSource(connectDropTarget(
      <li {...props}>{props.children}</li>
    ));
  }
}
```

Besides the initial debug log, we should see way more logs as we drag a Note around. Note that both decorators give us access to the Note props. In this case we are using `monitor.getItem()` to access them at `noteTarget`.

7.4 Developing onMove API for Notes

Now that we can move notes around, we still need to define logic. The following steps are needed:

- Capture Note id on `beginDrag`.
- Capture target Note id on `hover`.
- Trigger `onMove` callback on `hover` so that we can deal with the logic at a higher level.

You can see how this translates to code below:

app/components/Note.jsx

```
...

const noteSource = {
  beginDrag(props) {
    return {
      id: props.id
    };
  }
};

const noteTarget = {
  hover(targetProps, monitor) {
    const targetId = targetProps.id;
    const sourceProps = monitor.getItem();
    const sourceId = sourceProps.id;

    if(sourceId !== targetId) {
      targetProps.onMove({sourceId, targetId});
    }
  }
};

...
```

If you run the application now, you'll likely get a bunch of `onMove` related errors. We should make Notes aware of that:

app/components/Notes.jsx

```

...

export default class Notes extends React.Component {
  ...
  renderNote = (note) => {
    return (
      <Note className="note" onMove={this.onMoveNote}
        id={note.id} key={note.id}>
        <Editable
          value={note.task}
          onEdit={this.props.onEdit.bind(null, note.id)}
          onDelete={this.props.onDelete.bind(null, note.id)} />
      </Note>
    );
  }
  onMoveNote({sourceId, targetId}) {
    console.log('source', sourceId, 'target', targetId);
  }
}

```

If you drag a Note around now, you should see logs like `source <id> target <id>` in the console. We are getting close. We still need to figure out what to do with these ids, though.

7.5 Adding Action and Store Method for Moving

The logic of drag and drop is quite simple. 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 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 onMove hook we just defined:

app/components/Notes.jsx

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

export default class Notes extends React.Component {
  ...
  renderNote = (note) => {
    return (
      <Note className="note" onMove={LaneActions.move}
        id={note.id} key={note.id}>
        <Editable
          value={note.task}
          onEdit={this.props.onEdit.bind(null, note.id)}
          onDelete={this.props.onDelete.bind(null, note.id)} />
      </Note>
    );
  }
}
```

We should also define a stub at LaneStore to see that we wired it up correctly:

app/stores/LaneStore.js

```
...

class LaneStore {
  ...
  move({sourceId, targetId}) {
    console.log('source', sourceId, 'target', targetId);
  }
}

export default alt.createStore(LaneStore, 'LaneStore');
```

You should see the same logs 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.

7.6 Implementing Note Drag and Drop Logic

Moving within a lane itself is more 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)³ from React as that solves the problem in one pass. It is included in a separate package. To get started, install it using:

```
npm i react-addons-update --save
```

It is possible to solve the lane to lane case using [splice](#)⁴. 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';

export default class LaneStore {
  ...
  move({sourceId, targetId}) {
    const lanes = this.lanes;
    const sourceLane = lanes.filter((lane) => {
      return lane.notes.indexOf(sourceId) >= 0;
    })[0];
    const targetLane = lanes.filter((lane) => {
      return lane.notes.indexOf(targetId) >= 0;
    })[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: [
```

³<https://facebook.github.io/react/docs/update.html>

⁴https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/splice

```

        [sourceNoteIndex, 1],
        [targetNoteIndex, 0, sourceId]
      ]
    });
  }
  else {
    // get rid of the source
    sourceLane.notes.splice(sourceNoteIndex, 1);

    // and move it to target
    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. The presentation could be better, though.

It would be better if indicated the note target better. We can do this by hiding the dragged note from the list. React DnD provides us the hooks we need.

Indicating Where to Move

React DnD provides a feature known as state monitors. We can use `monitor.isDragging()` to detect which note we are currently dragging. It can be set up as follows:

app/components/Note.jsx

```

...

@DragSource(ItemTypes.NOTE, noteSource, (connect, monitor) => ({
  connectDragSource: connect.dragSource(),
  isDragging: monitor.isDragging() // map isDragging() state to isDragging prop
}))
@DropTarget(ItemTypes.NOTE, noteTarget, (connect) => ({
  connectDropTarget: connect.dropTarget()
}))
export default class Note extends React.Component {
  render() {
    const {connectDragSource, connectDropTarget, isDragging,
      onMove, id, ...props} = this.props;
  }
}

```



```

    return connectDragSource(connectDropTarget(
      <li style={{
        opacity: isDragging ? 0 : 1
      }} {...props}>{props.children}</li>
    ));
  }
}

```

If you drag a note within a lane, you should see the behavior we expect. The target should be shown as blank. If you try moving the note to another lane and move it there, you will see this doesn't quite work.

The problem is that our note component gets unmounted during this process. This makes it lose `isDragging` state. Fortunately we can override the default behavior by implementing a `isDragging` check of our own to fix the issue. Perform the following addition:

app/components/Note.jsx

```

...

const noteSource = {
  beginDrag(props) {
    return {
      id: props.id
    };
  },
  isDragging(props, monitor) {
    return props.id === monitor.getItem().id;
  }
};

...

```

This tells React DnD to perform our custom check instead of relying on the default logic. After this change, unmounting won't be an issue and the feature works as we expect.

There is one little problem in our system. We cannot drag notes to an empty lane yet.

7.7 Dragging Notes to an Empty Lanes

To drag notes to an empty lane, we should allow lanes 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 {DropTarget} from 'react-dnd';
import ItemTypes from '../constants/itemTypes';

const noteTarget = {
  hover(targetProps, monitor) {
    const targetId = targetProps.lane.id;
    const sourceProps = monitor.getItem();
    const sourceId = sourceProps.id;

    console.log('source', sourceId, 'target', targetId);
  }
};

@DropTarget(ItemTypes.NOTE, noteTarget, (connect) => ({
  connectDropTarget: connect.dropTarget()
})))
export default class Lane extends React.Component {
  ...
  render() {
    const {connectDropTarget, lane, ...props} = this.props;

    return connectDropTarget(
      ...
    );
  }
}

```

If you drag a note to a lane now, you should see logs at your console. The question is what to do with this data? Before actually moving the note to a lane, we should check whether it's empty or not. If it has content already, the operation doesn't make sense. Our existing logic can deal with that.

This is a simple check to make. Given we know the target lane at our noteTarget hover handler, we can check its notes array as follows:

app/components/Lane.jsx

```
const noteTarget = {
  hover(targetProps, monitor) {
    const sourceProps = monitor.getItem();
    const sourceId = sourceProps.id;

    if(!targetProps.lane.notes.length) {
      console.log('source', sourceId, 'target', targetProps);
    }
  }
};
```

If you refresh your browser and drag around now, the log should appear only when you drag a note to a lane that doesn't have any notes attached to it yet.

Triggering move Logic

Now we know what Note to move into which Lane. `LaneStore.attachToLane` is ideal for this purpose. Adjust Lane as follows:

app/components/Lane.jsx

```
const noteTarget = {
  hover(targetProps, monitor) {
    const sourceProps = monitor.getItem();
    const sourceId = sourceProps.id;

    if(!targetProps.lane.notes.length) {
      LaneActions.attachToLane({
        laneId: targetProps.lane.id,
        noteId: sourceId
      });
    }
  }
};
```

There is one problem, though. What happens to the old instance of the Note? In the current solution, the old lane will have an id pointing to it. As a result we have duplicate data in the system.

Earlier we resolved this using `detachFromLane`. The problem is that we don't know to which lane the note belonged. We could pass this data through the component hierarchy, but that doesn't feel particularly nice.

We could resolve this on store level instead by implementing an invariant at `attachToLane` that makes sure any possible earlier references get removed. This can be achieved by implementing `this.removeNote(noteId)` check:

app/stores/LaneStore.js

```
...

class LaneStore {
  ...
  attachToLane({laneId, noteId}) {
    if(!noteId) {
      this.waitFor(NoteStore);

      noteId = NoteStore.getState().notes.slice(-1)[0].id;
    }

    this.removeNote(noteId);

    ...
  }
  removeNote(noteId) {
    const lanes = this.lanes;
    const removeLane = lanes.filter((lane) => {
      return lane.notes.indexOf(noteId) >= 0;
    })[0];

    if(!removeLane) {
      return;
    }

    const removeNoteIndex = removeLane.notes.indexOf(noteId);

    removeLane.notes = removeLane.notes.slice(0, removeNoteIndex).
      concat(removeLane.notes.slice(removeNoteIndex + 1));
  }
  ...
}
```

`removeNote(noteId)` goes through `LaneStore` data. If it finds a note by id, it will get rid of it. After that we have a clean slate, and we can add a note to a lane. This change allows us to drop `detachFromLane` from the system entirely, but I'll leave that up to you.

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!

7.8 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.

In the next chapter we'll set up a production level build for our application. You can use the same techniques in your own projects.

8. Building Kanban

Now that we have a nice Kanban application up and running, we can worry about showing it to the public. The goal of this chapter is to set up a nice production grade build. There are various techniques we can apply to bring the bundle size down. We can also leverage browser caching.

8.1 Setting Up a Build Target

In our current setup, we always serve the application through `webpack-dev-server`. To create a build, we'll need to extend the `scripts` section in `package.json`.

package.json

```
{  
  ...  
  "scripts": {  
    "build": "webpack",  
    ...  
  },  
  ...  
}
```

We'll also need some build specific configuration to make Webpack pick up our JSX. We can set up sourcemaps while we're at it. I'll be using the `source-map` option here as that's a good choice for production.

webpack.config.js

```
...  
  
if(TARGET === 'build') {  
  module.exports = merge(common, {  
    output: {  
      path: PATHS.build,  
      filename: 'bundle.js'  
    },  
    devtool: 'source-map'  
  });  
}
```

After these changes, `npm run build` should yield something like the following:

Hash: 109cade2bfe58cda116f

Version: webpack 1.12.9

Time: 5424ms

Asset	Size	Chunks		Chunk Names
bundle.js	1.12 MB	0	[emitted]	main
bundle.js.map	1.31 MB	0	[emitted]	main
index.html	184 bytes		[emitted]	
+ 334 hidden modules				

1.12 MB is quite a lot. We should do something about that.

8.2 Optimizing Build Size

There are a couple of basic things we can do to slim down our build. We can apply some minification to it. We can also tell React to optimize itself. Doing both will result in significant size savings. Provided we apply gzip compression on the content when serving it, further gains may be made.

Minification

Minification will convert our code into a smaller format without losing any meaning. Usually this means some amount of rewriting code through predefined transformations. Sometimes this can break code as it can rewrite pieces of code you inadvertently depend upon. This is the reason why we gave explicit ids to our stores for instance.

The easiest way to enable minification is to call `webpack -p` (-p as in production). As Uglify will output a lot of warnings and they don't provide value in this case, we'll be disabling them. Add the following section to your Webpack configuration:

webpack.config.js

```
if(TARGET === 'build') {
  module.exports = merge(common, {
    output: {
      path: PATHS.build,
      filename: 'bundle.js'
    },
    devtool: 'source-map',
    plugins: [
      new webpack.optimize.UglifyJsPlugin({
        compress: {
          warnings: false
        }
      })
    ]
  })
}
```

```

    })
  ]
});
}

```



Uglify warnings can help you to understand how it processes the code. Therefore it may be beneficial to have a peek at the output every once in a while.

If you hit `npm run build` now, you should see better results:

Hash: 30b18807737f5bc0e462

Version: webpack 1.12.9

Time: 12451ms

Asset	Size	Chunks	Chunk Names
bundle.js	370 kB	0 [emitted]	main
bundle.js.map	2.77 MB	0 [emitted]	main
index.html	184 bytes	[emitted]	
+ 334 hidden modules			

Given it needs to do more work, it took longer. But on the plus side the build is much smaller now.



It is possible to push minification further by enabling variable name mangling. It comes with some extra complexity to worry about, but it may be worth it when you are pushing for minimal size. See [the official documentation](https://webpack.github.io/docs/list-of-plugins.html#uglifyjsplugin)¹ for details.

`process.env.NODE_ENV`

We can perform one more step to decrease build size further. React relies on `process.env.NODE_ENV` based optimizations. If we force it to production, React will get built in an optimized manner. This will disable some checks (e.g., property type checks). Most importantly it will give you a smaller build and improved performance.

In Webpack terms, you can add the following snippet to the `plugins` section of your configuration:

webpack.config.js

¹<https://webpack.github.io/docs/list-of-plugins.html#uglifyjsplugin>


```

if(TARGET === 'build') {
  module.exports = merge(common, {
    output: {
      path: PATHS.build,
      filename: 'bundle.js'
    },
    devtool: 'source-map',
    plugins: [
      // Setting DefinePlugin affects React library size!
      new webpack.DefinePlugin({
        'process.env.NODE_ENV': JSON.stringify('production')
      }),
      ...
    ]
  });
}

```

This is a useful technique for your own code. If you have a section of code that evaluates as false after this process, the minifier will remove it from the build completely.



It can be useful to set `'process.env.NODE_ENV': JSON.stringify('development')` for your development target to force it to build in *development* environment no matter what.

You can attach debugging specific utilities and such to your code easily this way. For instance, you could build a powerful logging system just for development. Here's a small example of what that could look like:

```

if(process.env.NODE_ENV === 'development') {
  console.log('developing like an ace');
}

```

If you prefer something more terse, you could use `__DEV__ === 'dev'` kind of syntax instead.



That `JSON.stringify` is needed, as Webpack will perform string replace “as is”. In this case, we'll want to end up with strings, as that's what various comparisons expect, not just production. The latter would just cause an error. An alternative would be to use a string such as `'"production"'`. Note the double quotation marks (“”).

Hit `npm run build` again, and you should see improved results:

Version: webpack 1.12.9

Time: 11983ms

Asset	Size	Chunks	Chunk Names
bundle.js	309 kB	0 [emitted]	main
bundle.js.map	2.68 MB	0 [emitted]	main
index.html	184 bytes	[emitted]	
+ 330 hidden modules			

So we went from 1.12 MB to 370 kB, and finally to 309 kB. The final build is a little faster than the previous one. As that 309 kB can be served gzipped, it is quite reasonable. gzipping will drop around another 40%. It is well supported by browsers.

We can do a little better, though. We can split app and vendor bundles and add hashes to their filenames.

8.3 Splitting app and vendor Bundles

The main advantage of splitting the application into two separate bundles is that it allows us to benefit from client caching. We might, for instance, make most of our changes to the small app bundle. In this case, the client would have to fetch only the app bundle, assuming the vendor bundle has already been loaded.

This scheme won't load as fast as a single bundle initially due to the extra request. Thanks to client-side caching, we might not need to reload all the data for every request. This is particularly true if a bundle remains unchanged. If only app updates, only that may need to be downloaded.

Defining a vendor Entry Point

To get started, we need to define a vendor entry point:

webpack.config.js

```
var path = require('path');
var HtmlWebpackPlugin = require('html-webpack-plugin');
var webpack = require('webpack');
var merge = require('webpack-merge');

// Load *package.json* so we can use `dependencies` from there
var pkg = require('./package.json');

...

if(TARGET === 'build') {
```

```

module.exports = merge(common, {
  // Define entry points needed for splitting
  entry: {
    app: PATHS.app,
    vendor: Object.keys(pkg.dependencies).filter(function(v) {
      // Exclude alt-utils as it won't work with this setup
      // due to the way the package has been designed
      // (no package.json main).
      return v !== 'alt-utils';
    })
  },
  output: {
    path: PATHS.build,
    // Output using entry name
    filename: '[name].js'
  },
  ...
});
}

```

This tells Webpack that we want a separate *entry chunk* for our project vendor level dependencies. Beyond this, it's possible to define chunks that are loaded dynamically. This can be achieved through [require.ensure²](#).

If you trigger the build now using `npm run build`, you should see something along this:

Hash: a9866869773710dead0f

Version: webpack 1.12.9

Time: 18362ms

Asset	Size	Chunks		Chunk Names
app.js	309 kB	0	[emitted]	app
vendor.js	285 kB	1	[emitted]	vendor
app.js.map	2.68 MB	0	[emitted]	app
vendor.js.map	2.55 MB	1	[emitted]	vendor
index.html	224 bytes		[emitted]	
[0] multi vendor	112 bytes	{1}	[built]	
+ 330 hidden modules				

Now we have separate *app* and *vendor* bundles. There's something wrong, however. If you examine the files, you'll see that *app.js* contains *vendor* dependencies. We need to do something to tell Webpack to avoid this situation. This is where `CommonsChunkPlugin` comes in.

²<https://webpack.github.io/docs/code-splitting.html>

Setting Up CommonsChunkPlugin

CommonsChunkPlugin allows us to extract the code we need for the vendor bundle. In addition we will use it to extract a *manifest*. It is a file that tells Webpack how to map each module to each file. We will need this in the next step for setting up long term caching. Here's the setup:

webpack.config.js

```
if(TARGET === 'build') {
  module.exports = merge(common, {
    ...
    module: {
      ...
    },
    plugins: [
      // Extract vendor and manifest files
      new webpack.optimize.CommonsChunkPlugin({
        names: ['vendor', 'manifest']
      }),
      ...
    ]
  });
}
```

If you run `npm run build` now, you should see output as below:

Hash: 0a6856bc6ac50a758aba

Version: webpack 1.12.9

Time: 12045ms

Asset	Size	Chunks		Chunk Names
app.js	24.6 kB	0, 2	[emitted]	app
vendor.js	285 kB	1, 2	[emitted]	vendor
manifest.js	780 bytes	2	[emitted]	manifest
app.js.map	127 kB	0, 2	[emitted]	app
vendor.js.map	2.55 MB	1, 2	[emitted]	vendor
manifest.js.map	8.72 kB	2	[emitted]	manifest
index.html	269 bytes		[emitted]	
[0] multi vendor	112 bytes	{1}	[built]	
+ 330 hidden modules				

The situation is far better now. Note how small app bundle compared to the vendor bundle. In order to really benefit from this split, we should set up caching. This can be achieved by adding cache busting hashes to filenames.

Adding Hashes to Filenames

Webpack provides placeholders that can be used to access different types of hashes and entry name as we saw before. The most useful ones are:

- `[name]` - Returns entry name.
- `[hash]` - Returns build hash.
- `[chunkhash]` - Returns a chunk specific hash.

Using these placeholders you could end up with filenames such as:

```
app.d587bbd6e38337f5accd.js  
vendor.dc746a5db4ed650296e1.js
```

If the file contents are different, the hash will change as well, thus invalidating the cache, or more accurately the browser will send a new request for the new file. This means if only app bundle gets updated, only that file needs to be requested again.



An alternative way to achieve the same would be to generate static filenames and invalidate the cache through a querystring (i.e., `app.js?d587bbd6e38337f5accd`). The part behind the question mark will invalidate the cache. This method is not recommended. According to [Steve Souders](http://www.stevesouders.com/blog/2008/08/23/revving-filenames-dont-use-querystring/)³, attaching the hash to the filename is a more performant way to go.

We can use the placeholder idea within our configuration like this:

webpack.config.js

```
if(TARGET === 'build') {  
  module.exports = merge(common, {  
    entry: {  
      app: PATHS.app,  
      vendor: Object.keys(pkg.dependencies)  
    },  
    /* important! */  
    output: {  
      path: PATHS.build,  
      filename: '[name].[chunkhash].js',  
      chunkFilename: '[chunkhash].js'  
    },  
    ...  
  });  
}
```

If you hit `npm run build` now, you should see output like this.

³<http://www.stevesouders.com/blog/2008/08/23/revving-filenames-dont-use-querystring/>

Hash: 60db5ab202527c9b3f25

Version: webpack 1.12.9

Time: 12067ms

Asset	Size	Chunks		Chunk Names
app.61448a510c24c28317d0.js	24.6 kB	0, 2	[emitted]	app
vendor.edadb6384837b591ae83.js	285 kB	1, 2	[emitted]	vendor
manifest.39215342321849b1f4e1.js	821 bytes	2	[emitted]	manifest
app.61448a510c24c28317d0.js.map	127 kB	0, 2	[emitted]	app
vendor.edadb6384837b591ae83.js.map	2.55 MB	1, 2	[emitted]	vendor
manifest.39215342321849b1f4e1.js.map	8.78 kB	2	[emitted]	manifest
index.html	332 bytes		[emitted]	
[0] multi vendor 112 bytes {1} [built]				
+ 330 hidden modules				

Our files have neat hashes now. To prove that it works, you could try altering *app/index.jsx* and include a `console.log` there. After you build, only app and manifest related bundles should change.

One more way to improve the build further would be to load popular dependencies, such as React, through a CDN. That would decrease the size of the vendor bundle even further while adding an external dependency on the project. The idea is that if the user has hit the CDN earlier, caching can kick in just like here.

8.4 Cleaning the Build

Our current setup doesn't clean the build directory between builds. As this can get annoying if we change our setup, we can use a plugin to clean the directory for us. Execute

```
npm i clean-webpack-plugin --save-dev
```

to install the plugin. Change the build configuration as follows to integrate it:

webpack.config.js

```
...
var Clean = require('clean-webpack-plugin');
...

if(TARGET === 'build') {
  module.exports = merge(common, {
    ...
    plugins: [
```

```
    new Clean([PATHS.build]),  
    ...  
  ]  
});  
}
```

After this change our `build` directory should remain nice and tidy when building. See [clean-webpack-plugin](https://www.npmjs.com/package/clean-webpack-plugin)⁴ for further options.



An alternative would be to use your terminal (`rm -rf ./build/`) and set that up in the `scripts` section of `package.json`.

8.5 Separating CSS

Even though we have a nice build set up now, where did all the CSS go? As per our configuration, it has been inlined to JavaScript! Even though this can be convenient during development, it doesn't sound ideal. The current solution doesn't allow us to cache CSS. In some cases we might suffer from a flash of unstyled content (FOUC).

It just so happens that Webpack provides a means to generate a separate CSS bundle. We can achieve this using the `ExtractTextPlugin`. It comes with overhead during the compilation phase, and it won't work with Hot Module Replacement (HMR) by design. Given we are using it only for production, that won't be a problem.

It will take some configuration to make it work. Hit

```
npm i extract-text-webpack-plugin --save-dev
```

to get started. Next we need to get rid of our current CSS related declaration at `common` configuration. After that, we need to split it up between `build` and `dev` configuration sections as follows:

webpack.config.js

⁴<https://www.npmjs.com/package/clean-webpack-plugin>

```
...
var ExtractTextPlugin = require('extract-text-webpack-plugin');

...

var common = {
  entry: PATHS.app,
  resolve: {
    extensions: ['', '.js', '.jsx']
  },
  module: {
    loaders: [
      // Remove CSS specific section here
      {
        test: /\.jsx?$/,
        loaders: ['babel'],
        include: PATHS.app
      }
    ]
  },
  plugins: [
    new HtmlwebpackPlugin({
      title: 'Kanban app'
    })
  ]
};

if(TARGET === 'start' || !TARGET) {
  module.exports = merge(common, {
    ...
    module: {
      loaders: [
        // Define development specific CSS setup
        {
          test: /\.css$/,
          loaders: ['style', 'css'],
          include: PATHS.app
        }
      ]
    },
    ...
  });
}
```



```

}

if(TARGET === 'build') {
  module.exports = merge(common, {
    ...
    devtool: 'source-map',
    module: {
      loaders: [
        // Extract CSS during build
        {
          test: /\.css$/,
          loader: ExtractTextPlugin.extract('style', 'css'),
          include: PATHS.app
        }
      ]
    },
    plugins: [
      new Clean(['build']),
      // Output extracted CSS to a file
      new ExtractTextPlugin('styles.[chunkhash].css'),
      ...
    ]
  });
}

```

Using this setup we can still benefit from the HMR during development. For a production build, we generate a separate CSS. `html-webpack-plugin` will pick it up automatically and inject it into our `index.html`.



Definitions such as `loaders: [ExtractTextPlugin.extract('style', 'css')]` won't work and will cause the build to error instead! So when using `ExtractTextPlugin`, use the loader form instead.



If you want to pass more loaders to the `ExtractTextPlugin`, you should use `!` syntax. Example: `ExtractTextPlugin.extract('style', 'css!autoprefixer-loader')`.

After running `npm run build` you should see output similar to the following:

Hash: ae2c206e1c5c06e7b884

Version: webpack 1.12.9

Time: 12976ms

Asset	Size	Chunks		Chunk Names
app.750beafb82d440b285.js	19 kB	0, 2	[emitted]	app
vendor.c61cac7ff701f2733703.js	289 kB	1, 2	[emitted]	vendor
manifest.bc3f15afc68e28cfee64.js	821 bytes	2	[emitted]	manifest
styles.750beafb82d440b285.css	937 bytes	0, 2	[emitted]	app
app.750beafb82d440b285.js.map	85 kB	0, 2	[emitted]	app
styles.750beafb82d440b285.css.map	108 bytes	0, 2	[emitted]	app
vendor.c61cac7ff701f2733703.js.map	2.62 MB	1, 2	[emitted]	vendor
manifest.bc3f15afc68e28cfee64.js.map	8.78 kB	2	[emitted]	manifest
index.html	404 bytes		[emitted]	

[0] multi vendor 112 bytes {1} [built]
 + 355 hidden modules

Child extract-text-webpack-plugin:
 + 2 hidden modules



If you are getting `Module build failed: CssSyntaxError: error`, make sure your common configuration doesn't have CSS related section set up!

Now our styling has been pushed to a separate CSS file. As a result, our JavaScript bundles have become slightly smaller. If we modify only our CSS now, those bundles shouldn't become invalidated anymore.



If you have a complex project with a lot of dependencies, it is likely a good idea to use the `DedupePlugin`. It will find possible duplicate files and deduplicate them. Use `new webpack.optimize.DedupePlugin()` in your plugins definition to enable it.

8.6 Analyzing Build Statistics

Analyzing build statistics is a good step towards understanding Webpack better. We can get statistics from it easily and we can visualize them using a tool. This shows us the composition of our bundles.

In order to get suitable output we'll need to do a couple of tweaks to our configuration:

`package.json`

```
{
  ...
  "scripts": {
    "stats": "webpack --profile --json > stats.json",
    ...
  },
  ...
}
```

webpack.config.js

```
...

//if(TARGET === 'build') {
if(TARGET === 'build' || TARGET === 'stats') {
  ...
}

...
```

If you hit `npm run stats` now, you should find `stats.json` at your project root after it has finished processing. We can take this file and pass it to [the online tool](http://webpack.github.io/analyse/)⁵. Note that the tool works only over HTTP! If your data is sensitive, consider using [the standalone version](https://github.com/webpack/analyse)⁶ instead.

Besides helping you to understand your bundle composition, the tool can help you to optimize your output further.

8.7 Deployment

There's no one right way to deploy our application. `npm run build` provides us something static to host. If you drop that on a suitable server, it will just work. One neat way to deal with it for small demos is to piggyback on GitHub Pages.

Hosting on GitHub Pages

A package known as [gh-pages](https://github.com/webpack/gh-pages)⁷ allows us to achieve this easily. You point it to your build directory first. It will then pick up the contents and push them to the `gh-pages` branch. To get started, hit

⁵<http://webpack.github.io/analyse/>

⁶<https://github.com/webpack/analyse>

⁷<https://www.npmjs.com/package/gh-pages>

```
npm i gh-pages --save-dev
```

We are also going to need an entry point at *package.json*:

package.json

```
{
  ...
  "scripts": {
    "deploy": "node ./lib/deploy.js",
    ...
  },
  ...
}
```

In order to get access to our build path, we need to expose something useful for deploy case. We can match for the case like this so we get useful configuration:

webpack.config.js

```
...

//if(TARGET === 'build' || TARGET === 'stats') {
if(TARGET === 'build' || TARGET === 'stats' || TARGET === 'deploy') {
  ...
}

...
```

To glue it all together, we need a deployment script like this:

lib/deploy.js

```
var ghpages = require('gh-pages');
var config = require('../webpack.config');

main();

function main() {
  ghpages.publish(config.output.path, console.error.bind(console));
}
```

If you hit `npm run deploy` now and everything goes fine, you should have your application hosted through GitHub Pages. You should find it at `https://<name>.github.io/<project>` (`github.com/<name>/<project>` at GitHub) assuming it worked.

8.8 Conclusion

Beyond the features discussed Webpack allows you to [lazy load](https://webpack.github.io/docs/code-splitting.html)⁸ content through `require.ensure`. This is handy if you happen to have a specific dependency on some view and want to load it when you need it.

Our Kanban application is now ready to be served. We went from a chunky build to a slim one. Even better the production version can benefit from caching and it is able to invalidate it. You can also understand how isomorphic rendering works on a basic level.



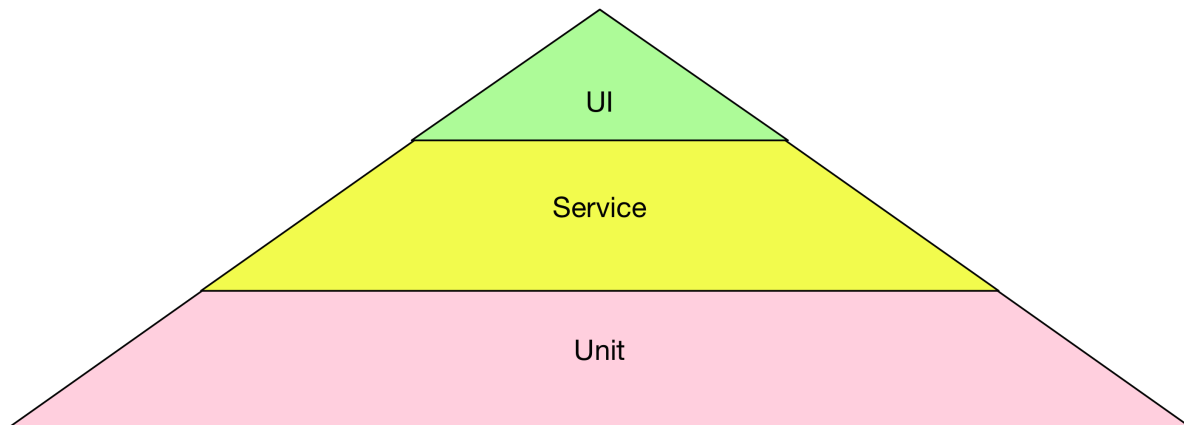
It can be a good idea to read the *Authoring Libraries* chapter for more ideas on how to improve your project. Although the chapter has been designed libraries in mind, understanding the fundamentals of npm doesn't hurt. A lot of the same concepts apply to both applications and libraries after all.

⁸<https://webpack.github.io/docs/code-splitting.html>

9. 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 is deployed into production. It is still possible to break things, but tests allow us to catch mistakes early in the development cycle.

9.1 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.

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). 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.

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.

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 from the user's perspective. 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/)¹ 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.

¹<http://www.seleniumhq.org/>

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 example we could end up with 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 [QuickCheck²](#) made the approach well-known and there are implementations for other languages as well.

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.

9.2 Setting Up Webpack

There are multiple approaches for testing with Webpack. I'll be discussing one based on [Karma³](#) and [Mocha⁴](#). This approach has been adapted based on Cesar Andreu's [web-app⁵](#) as it works well.

²<https://hackage.haskell.org/package/QuickCheck>

³<https://karma-runner.github.io>

⁴<https://mochajs.org/>

⁵<https://github.com/cesarandreu/web-app>

Test Runner

Karma is a test runner. It allows you to execute tests against a browser. In this case we'll be using [PhantomJS](http://phantomjs.org/)⁶, a popular Webkit based headless browser. Karma performs most of the hard work for us. It is possible to configure it to work with various testing tools depending on your preferences.

[Testem](https://github.com/airportyh/testem)⁷ is a valid alternative to Karma. You'll likely find a few others as there's no lack of testing tools for JavaScript.

Unit Testing

Mocha will be used for structuring tests. It follows a simple describe, it format. It doesn't specify assertions in any way. We'll be using Node.js [assert](https://nodejs.org/api/assert.html)⁸ as that's enough for our purposes. Alternatives such as [Chai](http://chaijs.com/)⁹ provide more powerful and expressive syntax.

Facebook's [Jest](https://facebook.github.io/jest/)¹⁰ is a popular alternative to Mocha. It is based on [Jasmine](https://jasmine.github.io/)¹¹, another popular tool, and takes less setup than Mocha. Unfortunately there are some Node.js version related issues and there are a few features (mainly auto-mocking) that can be a little counter-intuitive. It's a valid alternative, though.



[rewire-webpack](https://github.com/jhnns/rewire-webpack)¹² allows you to manipulate your module behavior to make unit testing easier. It uses [rewire](https://github.com/jhnns/rewire)¹³ 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](https://www.npmjs.com/package/babel-plugin-rewire)¹⁴.

Code Coverage

In order to get a better idea of the *code coverage* of our tests, we'll be using [Istanbul](https://gotwarlost.github.io/istanbul/)¹⁵. It will provide us with an HTML report to study. This will help us to understand what parts of code could use tests. It doesn't tell us anything about the quality of the tests, however. It just tells that we have hit some particular branches of the code with them.

[Blanket](http://blanketjs.org/)¹⁶ is a valid alternative to Istanbul.

⁶<http://phantomjs.org/>

⁷<https://github.com/airportyh/testem>

⁸<https://nodejs.org/api/assert.html>

⁹<http://chaijs.com/>

¹⁰<https://facebook.github.io/jest/>

¹¹<https://jasmine.github.io/>

¹²<https://github.com/jhnns/rewire-webpack>

¹³<https://github.com/jhnns/rewire>

¹⁴<https://www.npmjs.com/package/babel-plugin-rewire>

¹⁵<https://gotwarlost.github.io/istanbul/>

¹⁶<http://blanketjs.org/>

Installing Dependencies

Our test setup will require a lot of new dependencies. Execute the following command to get them installed:

```
npm i react-addons-test-utils isparta-instrumenter-loader@0.2.1 karma karma-cove\
rage karma-mocha karma-phantomjs-launcher karma-sourcemap-loader karma-spec-repo\
rter karma-webpack mocha phantomjs phantomjs-polyfill --save-dev
```

Besides these dependencies, we need to do a bit of configuration work.

Setting Up *package.json* scripts

As usual, we'll be wrapping the tools behind *package.json* scripts. We'll do one for running tests once and another for running them constantly. Set up as follows:

package.json

```
{
  ...
  "scripts": {
    "build": "webpack",
    "start": "webpack-dev-server",
    "test": "karma start",
    "tdd": "karma start --auto-watch --no-single-run"
  },
  ...
}
```

`npm run test` or `npm test` will simply execute our tests. `npm run tdd` will keep on running the tests as we work on the project. That's what you'll be relying upon during development a lot.

Testing File Layout

To keep things simple, we'll use a separate directory for tests. Here's an example of the structure we'll end up with:

- /tests
 - demo_test.js
 - editable_test.js
 - note_store_test.js

– note_test.js

There are multiple available conventions for this. One alternative is to push your tests to the component level. For instance you could have a directory per component. That directory would contain the component, associated styling, and tests. The tests directory and the tests could be named *specs* instead. In this case you would have */specs* and *demo_spec.js* for example.

Configuring Karma

We are still missing a couple of important bits to make this setup work. We'll need to configure both Karma and Webpack. We can set up Karma first:

karma.conf.js

```
// Reference: http://karma-runner.github.io/0.13/config/configuration-file.html
module.exports = function karmaConfig (config) {
  config.set({
    frameworks: [
      // Reference: https://github.com/karma-runner/karma-mocha
      // Set framework to mocha
      'mocha'
    ],

    reporters: [
      // Reference: https://github.com/mlex/karma-spec-reporter
      // Set reporter to print detailed results to console
      'spec',

      // Reference: https://github.com/karma-runner/karma-coverage
      // Output code coverage files
      'coverage'
    ],

    files: [
      // Reference: https://www.npmjs.com/package/phantomjs-polyfill
      // Needed because React.js requires bind and phantomjs does not support it
      'node_modules/phantomjs-polyfill/bind-polyfill.js',

      // Grab all files in the tests directory that contain _test.
      'tests/**/*.test.*'
    ]
  })
}
```

```
preprocessors: {
  // Reference: http://webpack.github.io/docs/testing.html
  // Reference: https://github.com/webpack/karma-webpack
  // Convert files with webpack and load sourcemaps
  'tests/**/*.test.*': ['webpack', 'sourcemap']
},

browsers: [
  // Run tests using PhantomJS
  'PhantomJS'
],

singleRun: true,

// Configure code coverage reporter
coverageReporter: {
  dir: 'build/coverage/',
  type: 'html'
},

// Test webpack config
webpack: require('./webpack.config'),

// Hide webpack build information from output
webpackMiddleware: {
  noInfo: true
}
});
```

As you can see from the comments, you can configure Karma in a variety of ways. For example, you could point it to Chrome or even multiple browsers at once.

`require('babel/register');` is needed as our configuration relies on Babel features. Normally you can skip that. This also depends on the version of Node.js you are using. Node.js 4.0 includes many ES6 features. It wouldn't work for our case, but might work for some.

We still need to write some Webpack specific configuration to make this all work correctly.

Configuring Webpack

Webpack will require some special configuration of its own. In order to make Karma find the code we want, we need to point Webpack to it. In addition we need to configure `isparta-instrumenter-`

loader so that our code coverage report generation will work through Istanbul. *isparta* is needed given we are using Babel features. Consider the test configuration below:

webpack.config.js

```
...
const TARGET = process.env.npm_lifecycle_event;
const PATHS = {
  app: path.join(__dirname, 'app'),
  build: path.join(__dirname, 'build'),
  test: path.join(__dirname, 'test')
};

process.env.BABEL_ENV = TARGET;

var common = {
  ...
};

...

if(TARGET === 'test' || TARGET === 'tdd') {
  module.exports = merge(common, {
    entry: {}, // karma will set this
    output: {}, // karma will set this
    devtool: 'inline-source-map',
    resolve: {
      alias: {
        'app': PATHS.app
      }
    },
    module: {
      preLoaders: [
        {
          test: /\.jsx?$/,
          loaders: ['isparta-instrumenter'],
          include: PATHS.app
        }
      ],
      loaders: [
        {
          test: /\.jsx?$/,
          loaders: ['babel'],
```

```

        include: PATHS.test
      }
    ]
  }
});
}

...

```

We have a basic testing setup together now.

If you hit `npm test` now, you should see something like this:

```
> karma start
```

```

11 09 2015 15:09:13.055:WARN [watcher]: Pattern ".../kanban_app/tests/**/*.test.*" does not match any file.
11 09 2015 15:09:13.075:INFO [karma]: Karma v0.13.9 server started at http://localhost:9876/
11 09 2015 15:09:13.085:INFO [launcher]: Starting browser PhantomJS
11 09 2015 15:09:14.212:INFO [PhantomJS 1.9.8 (Mac OS X 0.0.0)]: Connected on socket vk5nMd7Qpb10E-WqAAAA with id 18811718

```

```
PhantomJS 1.9.8 (Mac OS X 0.0.0): Executed 0 of 0 ERROR (0.001 secs / 0 secs)
```

```
npm ERR! Test failed.  See above for more details.
```

Given there are no tests yet, the setup is supposed to fail, so it's all good. Even `npm run tdd` works. Note that you can kill that process using `ctrl-c`.

Writing the First Test

To prove that everything works with a test, we can write one. Just asserting a simple math statement is enough. Try this:

```
tests/demo_test.js
```

```
import assert from 'assert';

describe('add', () => {
  it('adds', () => {
    assert.equal(1 + 1, 2);
  });
});
```

If you hit `npm test` now, you should see something like this:

```
> karma start
```

```
11 09 2015 15:14:22.705:INFO [karma]: Karma v0.13.9 server started at http://localhost:9876/
11 09 2015 15:14:22.718:INFO [launcher]: Starting browser PhantomJS
11 09 2015 15:14:23.939:INFO [PhantomJS 1.9.8 (Mac OS X 0.0.0)]: Connected on socket J8pa59_mf9m_EfzWAAAA with id 35083296
```

```
add
  ✓ adds
```

```
PhantomJS 1.9.8 (Mac OS X 0.0.0): Executed 1 of 1 SUCCESS (0.012 secs / 0 secs)
TOTAL: 1 SUCCESS
```

Even better, we can try `npm run tdd` now. Hit it and try tweaking the test. Make it fail for example. As you can see, this provides us a nice testing workflow. Now that we've tested that testing works, we can focus on real work.

9.3 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.

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.

Test Plan for `Editable`

There are a couple of things to test in `Editable`:

- Does it render the given value correctly?
- Does it enter the edit mode on click?
- Does it trigger `onEdit` callback on edit?
- Does it trigger `onDelete` callback on delete?

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 from the DOM.
3. Check that the value is what we expect.

In terms of code it would look like this:

`tests/editable_test.js`


```
import React from 'react';
import {
  renderIntoDocument,
  findRenderedDOMComponentWithClass
} from 'react-addons-test-utils';
import assert from 'assert';
import Editable from 'app/components/Editable.jsx';

describe('Editable', () => {
  it('renders value', () => {
    const value = 'value';
    const component = renderIntoDocument(
      <Editable value={value} />
    );

    const valueComponent = findRenderedDOMComponentWithClass(component, 'value');

    assert.equal(valueComponent.textContent, value);
  });
});
```

There are a couple of important parts here that are good to understand as they'll repeat later:

- `renderIntoDocument` is a testing utility React provides. It renders the given markup to a document we can then inspect. It expects the DOM in order to work.
- `findRenderedDOMComponentWithClass` allows us to traverse the DOM and seek for components matching the given class. It expects to find exactly one match, else it will throw an exception. `scry` variant is more generic and returns a list of matches.
- `valueComponent.textContent` provides us access to the text content of the DOM node.

These functions are a part of React [Test Utilities](https://facebook.github.io/react/docs/test-utils.html)¹⁷ API.

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.



Given React test API can be somewhat verbose, people have developed lighter alternatives to it. See [jqunse/teaspoon](https://github.com/jquense/teaspoon)¹⁸, [Legitcode/tests](https://github.com/Legitcode/tests)¹⁹, and [react-test-tree](https://github.com/QubitProducts/react-test-tree)²⁰ for example.

¹⁷<https://facebook.github.io/react/docs/test-utils.html>

¹⁸<https://github.com/jquense/teaspoon>

¹⁹<https://github.com/Legitcode/tests>

²⁰<https://github.com/QubitProducts/react-test-tree>



React provides a lighter way to assert component behavior without DOM. [Shallow rendering](#)²¹ is still considered experimental but may provide a lighter way to test React components in the future.

Editable Enters the Edit Mode

We can follow a similar idea here as before. This case is more complex, though. First we need to enter the edit mode somehow. After that we need to check that the input displays the correct value. Consider the implementation below:

`tests/editable_test.js`

```
import React from 'react';
import {
  renderIntoDocument,
  findRenderedDOMComponentWithClass,
  findRenderedDOMComponentWithTag,
  Simulate
} from 'react-addons-test-utils';
import assert from 'assert';
import Editable from 'app/components/Editable.jsx';

describe('Editable', () => {
  ...

  it('enters edit mode', () => {
    const value = 'value';
    const component = renderIntoDocument(
      <Editable value={value} />
    );

    const valueComponent = findRenderedDOMComponentWithClass(component, 'value');
    Simulate.click(valueComponent);

    const input = findRenderedDOMComponentWithTag(component, 'input');

    assert.equal(input.value, value);
  });
});
```

We are using a couple of new features:

²¹<https://facebook.github.io/react/docs/test-utils.html#shallow-rendering>

- `Simulate.click` triggers the `onClick` behavior we've defined at `Editable`. There are methods like this for simulating other user input as well.
- `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.

There's still some work left to do. We'll want to check out `onEdit` behavior next.

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:

`tests/editable_test.js`

```
...

describe('Editable', () => {
  ...

  it('triggers onEdit', () => {
    let triggered = false;
    const newValue = 'value';
    const onEdit = (val) => {
      triggered = true;
      assert.equal(val, newValue);
    };
    const component = renderIntoDocument(
      <Editable value={'value'} onEdit={onEdit} />
    );

    let valueComponent = findRenderedDOMComponentWithClass(component, 'value');
    Simulate.click(valueComponent);

    const input = findRenderedDOMComponentWithTag(component, 'input');
    input.value = newValue;

    Simulate.blur(input);

    assert.equal(triggered, true);
  });
});
```

Compared to the earlier tests, there isn't much new here. We perform an assertion at `onEdit` and trigger the behavior through `Simulate.blur`, but apart from that we're in a familiar territory. You could probably start refactoring some common parts of the tests into separate functions now, but we can live with the current solution. At least we're being verbose about what we are doing.

One more test to go.

Editable Allows Deletion

Checking that `Editable` allows deletion is a similar case as triggering `onEdit`. We just check that the callback triggered:

`tests/editable_test.js`

```
describe('Editable', () => {
  ...

  it('allows deletion', () => {
    let deleted = false;
    const onDelete = () => {
      deleted = true;
    };
    const component = renderIntoDocument(
      <Editable value={'value'} onDelete={onDelete} />
    );

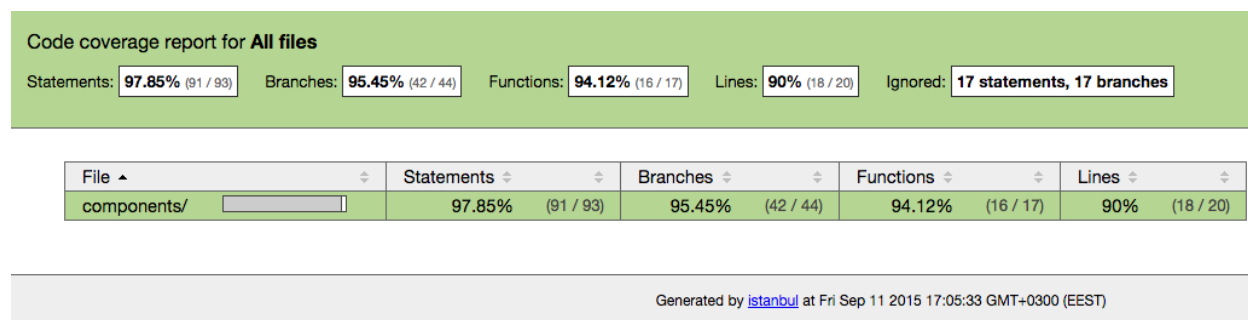
    let deleteComponent = findRenderedDOMComponentWithClass(component, 'delete');
    Simulate.click(deleteComponent);

    assert.equal(deleted, true);
  });
});
```

We have some basic tests in place now, but what about test coverage?

Checking Test Coverage

Now is a good time to check our coverage report. Serve `build/coverage/PhantomJS .../` and inspect it in your browser. You should see something like this:



Istanbul coverage

Based on the statistics, we're quite good. We are still missing some branches, but we cover most of `Editable`, so that's nice. You can try removing tests to see how the statistics change. You can also try to figure out which branches aren't covered.

You can see the component specific report by checking out `components/Editable.jsx.html` in your browser. That will show you that `checkEnter` isn't covered by any test yet. It would be a good idea to implement the missing test for that at some point.

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](#)²² 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. Hit

```
npm i react-dnd-test-backend --save-dev
```

to get the backend installed. The test below illustrates the basic idea:

`tests/note_test.js`

²²<https://gaearon.github.io/react-dnd/docs-testing.html>

```

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.jsx';

const {renderIntoDocument} = React.addons.TestUtils;

describe('Note', () => {
  it('renders children', () => {
    const test = 'test';
    const NoteContent = wrapInTestContext(Note);
    const component = renderIntoDocument(
      <NoteContent>{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.

9.4 Testing Kanban Stores

Alt provides a nice means for testing both [actions](#)²³ and [stores](#)²⁴. 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?
- 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';

describe('NoteStore', () => {
  it('creates notes', () => {
    const task = 'test';

    NoteActions.create({task});

    const state = NoteStore.getState();

    assert.equal(state.notes.length, 1);
```

²³<http://alt.js.org/docs/testing/actions/>

²⁴<http://alt.js.org/docs/testing/stores/>

```

    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.

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', () => {
  ...

  it('updates notes', () => {
    const task = 'test';
    const updatedTask = 'test 2';

    NoteActions.create({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', () => {
  beforeEach(() => {
    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', () => {
  ...

  it('deletes notes', () => {
    NoteActions.create({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.

NoteStore Allows get

There's only one test left for `get`. Given it's a public method of a `NoteStore`, we can go directly through `NoteStore.get`:

`tests/note_store_test.js`

```
describe('NoteStore', () => {  
  ...  
  
  it('gets notes', () => {  
    const task = 'test';  
    NoteActions.create({task: task});  
  
    const note = NoteStore.getState().notes[0];  
    const notes = NoteStore.get([note.id]);  
  
    assert.equal(notes.length, 1);  
    assert.equal(notes[0].task, task);  
  });  
});
```

This test proves that the logic works on a basic level. It would be a good idea to specify what happens with invalid data, though. We might want to react to that somehow.



[Legitcode/tests²⁵](https://github.com/Legitcode/tests#testing-alt-stores) provides handy shortcuts for testing Alt stores.

9.5 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. [Nightwatch²⁶](http://nightwatchjs.org/) 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.

²⁵<https://github.com/Legitcode/tests#testing-alt-stores>

²⁶<http://nightwatchjs.org/>

10. Typing with React

Just like linting, 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](#)¹ for React, but we won't go into that.

10.1 `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 in various syntaxes below:

ES5

¹<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, stage 0)

```
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 as objects.
- `React.PropTypes.shape({<name>: <propType>})` - Checks that given object is in a particular object shape with certain `propTypes`.

In addition it's 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](#)² goes into further detail.

10.2 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 `items`. We can make it optional and default to an empty list. This means we can simplify `App` a little:

app/components/App.jsx

```
// instead of
items: () => LaneStore.getState().lanes || []

// we can do
items: () => LaneStore.getState().lanes
```

In terms of `propTypes`, our annotation looks like this:

app/components/Lanes.jsx

²<https://facebook.github.io/react/docs/reusable-components.html>

```

class Lanes extends React.Component {
  ...
}
Lanes.propTypes = {
  items: React.PropTypes.array
};
Lanes.defaultProps = {
  items: []
};

export default Lanes;

```

We've also documented what Lanes expects. App is a little neater as well. This doesn't help much, though, if the lane items have an invalid format. We should annotate Lane and its children to achieve this.

Annotating Lane

As per our implicit definition, Lane expects an id, a name, and connectDropSource. id should be required, as a lane without one doesn't make any sense. The rest can remain optional. Translated to propTypes we would end up with this:

app/components/Lane.jsx

```

class Lane extends React.Component {
  ...
}
Lane.propTypes = {
  lane: React.PropTypes.shape({
    id: React.PropTypes.string.isRequired,
    name: React.PropTypes.string,
    notes: React.PropTypes.array
  }).isRequired,
  connectDropTarget: React.PropTypes.func
};
Lane.defaultProps = {
  name: '',
  notes: []
};

export default Lane;

```

If our basic data model is wrong somehow now, we'll know about it. To harden our system further, we should annotate notes contained by lanes.

Annotating Notes

As you might remember from the implementation, `Notes` accepts `items`, `onEdit`, and `onDelete` handlers. We can apply the same logic to `items` 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

```
class Notes extends React.Component {
  ...
}
Notes.propTypes = {
  items: React.PropTypes.array,
  onEdit: React.PropTypes.func,
  onDelete: React.PropTypes.func
};
Notes.defaultProps = {
  items: [],
  onEdit: () => {},
  onDelete: () => {}
};

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

```
class Note extends React.Component {  
  ...  
}  
Note.propTypes = {  
  id: React.PropTypes.string.isRequired,  
  connectDragSource: React.PropTypes.func,  
  connectDropSource: React.PropTypes.func,  
  onMove: React.PropTypes.func  
};  
Note.defaultProps = {  
  onMove: () => {}  
};  
  
export default Note;
```

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. `onDelete` is optional. Using the annotation syntax we get the following:

app/components/Editable.jsx

```
class Editable extends React.Component {  
  ...  
}  
Editable.propTypes = {  
  value: React.PropTypes.string,  
  onEdit: React.PropTypes.func.isRequired,  
  onDelete: React.PropTypes.func  
};  
Editable.defaultProps = {  
  value: '',  
  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.

10.3 Type Checking with Flow



Flow

Facebook's [Flow](http://flowtype.org/)³ 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/)⁴ 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.



At the time of this writing, major editors and IDEs have poor support for Flow annotations. This may change in the future.

³<http://flowtype.org/>

⁴<https://tryflow.org/>

Setting Up Flow

There are [pre-built binaries](#)⁵ 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. Hit `flow init`. That will generate a `.flowconfig` file that can be used for [advanced configuration](#)⁶.

Since we want avoid parsing `node_modules`, we should tweak the configuration as follows:

.flowconfig

```
[ignore]
.*/*node_modules
```

```
[include]
```

```
[libs]
```

```
[options]
```

Now it won't go through `node_modules` when executing.

Running Flow

Running Flow is simple, just hit `flow check`. This will likely yield a message such as `Found 0 errors` since we haven't really done anything yet.

An alternative way to run Flow is to simply hit `flow`. This will start `flow` in a server mode and it will start running in the background when hit initially. If you hit `flow` again, it will give instant results. This process may be closed using `flow stop`.

To keep things neat, we can push Flow behind npm. Add the following bit to your `package.json`:

package.json

⁵<http://flowtype.org/docs/getting-started.html>

⁶<http://flowtype.org/docs/advanced-configuration.html>

```
{  
  ...  
  "scripts": {  
    ...  
    "flow": "flow check"  
  },  
  ...  
}
```

After this we can hit `npm run 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 flow --silent` instead to chomp it.

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`!

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');
```

If this worked correctly, you should see something like this:

```
flow_app $ npm run flow --silent
```

```
.../flow_app/demo.js:9:1,17: function call
```

```
Error:
```

```
.../flow_app/demo.js:9:5,9: string
```

```
This type is incompatible with
```

```
.../flow_app/demo.js:2:17,22: number
```

```
.../flow_app/demo.js:9:1,17: function call
```

```
Error:
```

```
.../flow_app/demo.js:9:12,16: string
```

```
This type is incompatible with
```

```
.../flow_app/demo.js:2:28,33: number
```

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 useful to know.

10.4 Converting propTypes to Flow Checks

Unfortunately Flow [doesn't support](#)⁷ all ES6 and ES7 features we're using as of now. We'll have to settle with propTypes in our code until the functionality is there.

10.5 Babel Typecheck

While flow itself is a static checker that you have to run separately, [babel-plugin-typecheck](#)⁸ provides runtime checks during development. Hit

```
npm i babel-plugin-typecheck --save-dev
```

To make Babel aware of it, we'll need to tweak `.babelrc`:

```
.babelrc
```

⁷<https://github.com/facebook/flow/issues/560>

⁸<https://github.com/codemix/babel-plugin-typecheck>

```
{
  "stage": 1,
  "env": {
    "build": {
      "plugins": ["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.

10.6 TypeScript

Microsoft's [TypeScript](http://www.typescriptlang.org/)⁹ is yet another alternative. Starting from the version 1.6 it will gain JSX support. It will be documented in greater detail as the support lands.

10.7 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 upcoming. The situation may change in the near future as these solutions stabilize.

⁹<http://www.typescriptlang.org/>

III Advanced Techniques

There are a variety of advanced Webpack and React techniques that are good to be aware of. Linting can improve the quality of your code as it allows you to spot potential issues earlier. We will also discuss various ways you can use Webpack to bundle your application.

Besides consuming libraries, it can be fun to create them. As a result, I will discuss common authoring related concerns and show how to get libraries out there with minimal effort. There are a bunch of smaller tricks that you should be aware of and that will make your life as a library author easier.

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.

11. Linting in Webpack

Nothing is easier than making mistakes when coding in JavaScript. Linting is one of those techniques that can help you to make less mistakes. You can spot issues before they become actual problems.

Better yet, modern editors and IDEs offer strong support for popular tools. This means you can spot possible issues as you are developing. Despite this, it is a good idea to set them up with Webpack. That allows you to cancel a production build that might not be up to your standards for example.

11.1 Brief History of Linting in JavaScript

The linter that started it all for JavaScript is Douglas Crockford's [JSLint](http://www.jshint.com/)¹. It is opinionated like the man himself. The next step in evolution was [JSHint](http://jshint.com/)². It took the opinionated edge out of JSLint and allowed for more customization. [ESLint](http://eslint.org/)³ is the newest tool in vogue.

ESLint goes to the next level as it allows you to implement custom rules, parsers, and reporters. ESLint works with Babel and JSX syntax making it ideal for React projects. The project rules have been documented well and you have full control over their severity. These features alone make it a powerful tool.

Besides linting for issues, it can be useful to manage the code style on some level. Nothing is more annoying than having to work with source code that has mixed tabs and spaces. Stylistically consistent code reads better and is easier to work with.

[JSCS](http://jscs.info/)⁴ makes it possible to define a style guide for JavaScript code. It is easy to integrate into your project through Webpack, although ESLint implements a large part of its functionality.

11.2 Webpack and JSHint

Interestingly no JSLint loader seems to exist for Webpack yet. Fortunately there's one for JSHint. You could set it up on a legacy project easily. Install [jshint-loader](https://www.npmjs.com/package/jshint-loader)⁵ to your project first:

```
npm i jshint jshint-loader --save-dev
```

In addition, you will need a little bit of configuration:

¹<http://www.jshint.com/>

²<http://jshint.com/>

³<http://eslint.org/>

⁴<http://jscs.info/>

⁵<https://www.npmjs.com/package/jshint-loader>

```
var common = {
  ...
  module: {
    preLoaders: [
      {
        test: /\.js?$/,
        loaders: ['jshint'],
        // define an include so we check just the files we need
        include: PATHS.app
      }
    ]
  },
};
```

preLoaders section of the configuration gets executed before loaders. If linting fails, you'll know about it first. There's a third section, postLoaders, that gets executed after loaders. You could include code coverage checking there during testing, for instance.

JSHint will look into specific rules to apply from `.jshintrc`. You can also define custom settings within a `jshint` object at your Webpack configuration. Exact configuration options have been covered at [the JSHint documentation](http://jshint.com/docs/)⁶ in detail. `.jshintrc` could look like this:

`.jshintrc`

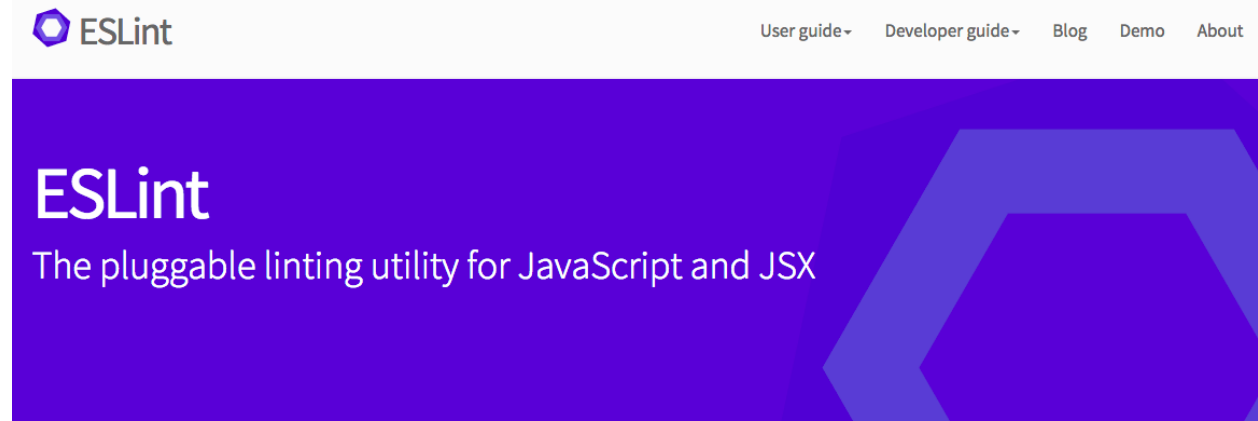
```
{
  "browser": true,
  "camelcase": false,
  "esnext": true,
  "indent": 2,
  "latedef": false,
  "newcap": true,
  "quotmark": "double"
}
```

This tells JSHint we're operating within browser environment, don't care about linting for camelcase naming, want to use double quotes everywhere and so on.

If you try running JSHint on our project, you will get a lot of output. It's not the ideal solution for React projects. ESLint will be more useful so we'll be setting it up next for better insights. Remember to remove JSHint configuration before proceeding further.

⁶<http://jshint.com/docs/>

11.3 Setting Up ESLint



ESLint

[ESLint](http://eslint.org/)⁷ is a recent linting solution for JavaScript. It builds on top of ideas presented by JSLint and JSHint. More importantly it allows you to develop custom rules. As a result, a nice set of rules have been developed for React in the form of [eslint-plugin-react](https://www.npmjs.com/package/eslint-plugin-react)⁸.



Since *v1.4.0* ESLint supports a feature known as [autofixing](http://eslint.org/blog/2015/09/eslint-v1.4.0-released/)⁹. It allows you to perform certain rule fixes automatically. To activate it, pass the flag `--fix` to the tool.

Connecting ESLint with *package.json*

In order to integrate ESLint with our project, we'll need to do a couple of little tweaks. First we'll need to hit

```
npm i babel-eslint eslint eslint-plugin-react --save-dev
```

This will add ESLint and the plugin we want to use as our project development dependencies. *babel-eslint* allows us to use Babel specific language features with ESLint. Next, we'll need to do some configuration:

package.json

⁷<http://eslint.org/>

⁸<https://www.npmjs.com/package/eslint-plugin-react>

⁹<http://eslint.org/blog/2015/09/eslint-v1.4.0-released/>

```
"scripts": {
  ...
  "lint": "eslint . --ext .js --ext .jsx"
}
...
```

This will trigger ESLint against all JS and JSX files of our project. That's definitely too much, so we'll need to restrict it to avoid going through a possible production build. Set up *.eslintignore* to the project root like this:

.eslintignore

```
build/
```

Next, we'll need to activate [babel-eslint](https://www.npmjs.com/package/babel-eslint)¹⁰ so that ESLint works with our Babel code. In addition, we need to activate React specific rules and set up a couple of our own. You can adjust these to your liking. For details see the official [ESLint rules documentation](http://eslint.org/docs/rules/)¹¹.

.eslintrc

```
{
  "parser": "babel-eslint",
  "env": {
    "browser": true,
    "node": true
  },
  "plugins": [
    "react"
  ],
  "rules": {
    "new-cap": 0,
    "strict": 0,
    "no-underscore-dangle": 0,
    "no-use-before-define": 0,
    "eol-last": 0,
    "quotes": [2, "single"],
    "jsx-quotes": 1,
    "react/jsx-no-undef": 1,
    "react/jsx-uses-react": 1,
    "react/jsx-uses-vars": 1
  }
}
```

¹⁰<https://www.npmjs.com/package/babel-eslint>

¹¹<http://eslint.org/docs/rules/>



ESLint supports ES6 features through configuration. You will have to specify the features to use through the `ecmaFeatures`¹² property.

The severity of an individual rule is defined by a number as follows:

- 0 - The rule has been disabled.
- 1 - The rule will emit a warning.
- 2 - The rule will emit an error.

Some rules, such as quotes, accept an array instead. This allows you to pass extra parameters to them. Refer to the rule's documentation for specifics.

The `react/` rules listed above are just a small subset of all available rules. Pick rules from `eslint-plugin-react`¹³ as needed.



Note that you can write ESLint configuration directly to `package.json`. Set up a `eslintConfig` field, and write your declarations below it.



It is possible to generate a sample `.eslintrc` using `eslint --init` (or `node_modules/.bin/eslint --init` for local install). This can be useful on new projects.

Dealing with ELIFECYCLE Error

In case the linting process fails, npm will give you a nasty looking ELIFECYCLE error. A good way to achieve a tidier output is to invoke `npm run lint --silent`. That will hide the ELIFECYCLE bit. You can define an alias for this purpose. In Unix you would do `alias run='npm run --silent'` and then run `<script>`.

Alternatively, you could pipe output to `true` like this:

package.json

¹²<http://eslint.org/docs/user-guide/configuring.html#specifying-language-options>

¹³<https://www.npmjs.com/package/eslint-plugin-react>

```
"scripts": {  
  ...  
  "lint": "eslint . --ext .js --ext .jsx || true"  
}
```

The problem with this approach is that if you invoke `lint` through some other command, it will pass even if there are failures. If you have another script that does something like `npm run lint && npm run build`, it will build regardless of the output of the first command!

Connecting ESLint with Webpack

We can make Webpack emit ESLint messages for us by using [eslint-loader](https://www.npmjs.com/package/eslint-loader)¹⁴. As the first step hit

```
npm i eslint-loader --save-dev
```



Note that `eslint-loader` will use a globally installed version of ESLint unless you have one included with the project itself! Make sure you have ESLint as a development dependency to avoid strange behavior.

Next, we need to tweak our development configuration to include it. Add the following section to it:

webpack.config.js

```
var common = {  
  ...  
  module: {  
    preLoaders: [  
      {  
        test: /\.jsx?$/,  
        loaders: ['eslint'],  
        include: PATHS.app  
      }  
    ]  
  },  
};
```

¹⁴<https://www.npmjs.com/package/eslint-loader>

We are including the configuration to `common` so that linting always gets performed. This way you can make sure your production build passes your rules while making sure you benefit from linting during development.

If you execute `npm start` now and break some linting rule while developing, you should see that in the terminal output. The same should happen when you build the project.

11.4 Customizing ESLint

Even though you can get very far with vanilla ESLint, there are several techniques you should be aware of. For instance, sometimes you might want to skip some particular rules per file. You might even want to implement rules of your own. We'll cover these cases briefly next.

Skipping ESLint Rules

Sometimes you'll want to skip certain rules per file or per line. This can be useful when you happen to have some exceptional case in your code where some particular rule doesn't make sense. As usual exception confirms the rule. Consider the following examples:

```
// everything
/* eslint-disable */
...
/* eslint-enable */

// specific rule
/* eslint-disable no-unused-vars */
...
/* eslint-enable no-unused-vars */

// tweaking a rule
/* eslint no-comma-dangle:1 */

// disable rule per line
alert('foo'); // eslint-disable-line no-alert
```

Note that the rule specific examples assume you have the rules in your configuration in the first place! You cannot specify new rules here. Instead you can modify the behavior of existing rules.

Setting Environment

Sometimes you may want to run ESLint in a specific environment, such as Node.js or Mocha. These environments have certain conventions of their own. For instance, Mocha relies on custom keywords (e.g., `describe`, `it`) and it's good if the linter doesn't choke on those.

ESLint provides two ways to deal with this: local and global. If you want to set it per file, you can use a declaration at the beginning of a file:

```
/*eslint-env node, mocha */
```

Global configuration is possible as well. In this case you can use `env` key like this:

.eslintrc

```
{
  "env": {
    "browser": true,
    "node": true,
    "mocha": true
  },
  ...
}
```

Writing Your Own Rules

ESLint rules rely on Abstract Syntax Tree (AST) definition of JavaScript. It is a data structure that describes JavaScript code after it has been lexically analyzed. There are tools such as [recast](https://github.com/benjamn/recast)¹⁵ that allow you to perform transformations on JavaScript code by using AST transformations. The idea is that you match some structure, then transform it somehow and convert AST back to JavaScript.

To get a better idea of how AST works and what it looks like, you can check [Esprima online JavaScript AST visualization](http://esprima.org/demo/parse.html)¹⁶ or [JS AST Explorer by Felix Kling](http://felix-king.de/esprima_ast_explorer/)¹⁷. Alternatively you can install `recast` and examine the output it gives. That is the structure we'll be working with for ESLint rules.

In ESLint's case we just want to check the structure and report in case something is wrong. Getting a simple rule done is surprisingly simple:

1. Set up a new project named `eslint-plugin-custom`. You can replace `custom` with whatever you want. ESLint follows this naming convention.
2. Hit `npm init` to create a dummy `package.json`
3. Set up `index.js` in the project root with content like this:

eslint-plugin-custom/index.js

¹⁵<https://github.com/benjamn/recast>

¹⁶<http://esprima.org/demo/parse.html>

¹⁷http://felix-king.de/esprima_ast_explorer/

```

module.exports = {
  rules: {
    demo: function(context) {
      return {
        Identifier: function(node) {
          context.report(node, 'This is unexpected!');
        }
      };
    }
  }
};

```

In this case, we just report for every identifier found. In practice, you'll likely want to do something more complex than this, but this is a good starting point.

Next, you need to hit `npm link` within `eslint-plugin-custom`. This will make your plugin visible within your system. `npm link` allows you to easily consume a development version of a library you are developing. To reverse the link you can hit `npm unlink` when you feel like it.



If you want to do something serious, you should point to your plugin through *package.json*.

We need to alter our project configuration to make it find the plugin and the rule within.

.eslintrc

```

"plugins": {
  "custom"
},
"rules": {
  "custom/demo": 1,
  ...
}

```

If you invoke ESLint now, you should see a bunch of warnings. Mission accomplished!

Of course the rule doesn't do anything useful yet. To move forward, I recommend checking out the official documentation about [plugins](http://eslint.org/docs/developer-guide/working-with-plugins.html)¹⁸ and [rules](http://eslint.org/docs/developer-guide/working-with-rules.html)¹⁹.

You can also check out some of the existing rules and plugins for inspiration to see how they achieve certain things. ESLint allows you to [extend these rulesets](http://eslint.org/docs/user-guide/configuring.html#extending-configuration-files)²⁰ through `extends` property. It accepts either a path to it (`"extends": "../node_modules/coding-standard/.eslintrc"`) or an array of paths. The entries are applied in the given order and later ones override former.

¹⁸<http://eslint.org/docs/developer-guide/working-with-plugins.html>

¹⁹<http://eslint.org/docs/developer-guide/working-with-rules.html>

²⁰<http://eslint.org/docs/user-guide/configuring.html#extending-configuration-files>

ESLint Resources

Besides the official documentation available at eslint.org²¹, you should check out the following blog posts:

- [Lint Like It's 2015](#)²² - This post by Dan Abramov shows how to get ESLint to work well with Sublime Text.
- [Detect Problems in JavaScript Automatically with ESLint](#)²³ - A good tutorial on the topic.
- [Understanding the Real Advantages of Using ESLint](#)²⁴ - Evan Schultz's post digs into details.
- [eslint-plugin-smells](#)²⁵ - This plugin by Elijah Manor allows you to lint against various JavaScript smells. Recommended.

If you just want some starting point, you can pick one of [eslint-config- packages](#)²⁶ or go with the [standard](#)²⁷ style. By the looks of it, standard has [some issues with JSX](#)²⁸ so be careful with that.

11.5 Linting CSS

[stylelint](#)²⁹ allows us to lint CSS. It can be used with Webpack through [postcss-loader](#)³⁰.

```
npm i stylelint postcss-loader --save-dev
```

Next, we'll need to integrate it with our configuration:

webpack.config.js

²¹<http://eslint.org/>

²²https://medium.com/@dan_abramov/lint-like-it-s-2015-6987d44c5b48

²³<http://davidwalsh.name/eslint>

²⁴<http://rangle.io/blog/understanding-the-real-advantages-of-using-eslint/>

²⁵<https://github.com/elijahmanor/eslint-plugin-smells>

²⁶<https://www.npmjs.com/search?q=eslint-config>

²⁷<https://www.npmjs.com/package/standard>

²⁸<https://github.com/feross/standard/issues/138>

²⁹<http://stylelint.io/>

³⁰<https://www.npmjs.com/package/postcss-loader>


```

...
var stylelint = require('stylelint');

...

var common = {
  ...
  module: {
    preLoaders: [
      {
        test: /\.css$/,
        loaders: ['postcss'],
        include: PATHS.app
      },
      ...
    ],
    ...
  },
  postcss: function () {
    return [stylelint({
      rules: {
        'color-hex-case': 'lower'
      }
    })];
  },
  ...
}

```

If you define a CSS rule, such as `background-color: #EFEFEF;`, you should see a warning at your terminal. See stylelint documentation for a full list of rules. npm lists [possible stylelint rulesets](https://www.npmjs.com/search?q=stylelint-config)³¹. You consume them as your project dependency like this:

```

var configSuitcss = require('stylelint-config-suitcss');

...

stylelint(configSuitcss)

```

Given stylelint is still under development, there's no CLI tool available yet. `.stylelintrc` type functionality is planned.

³¹<https://www.npmjs.com/search?q=stylelint-config>

11.6 Checking JavaScript Style with JSCS



JSCS — JavaScript Code Style.

[Overview](#)[Rules](#)[Contributing](#)[Changelog](#)

JSCS is a code style linter for programmatically enforcing your style guide. You can configure JSCS for your project in detail using over 90 validation rules, including presets from popular style guides like jQuery, Airbnb, Google, and more.

JSCS

Especially in a team environment, it can be annoying if one guy uses tabs and another uses spaces. There can also be discrepancies between space usage. Some like to use two spaces, and some like four for indentation. In short, it can get pretty messy without any discipline. To solve this issue, JSCS allows you to define a style guide for your project.



Just like ESLint, also JSCS has autofixing capabilities. To fix certain issues, you can invoke `jscs --fix` and it will modify your code.

JSCS can be installed through

```
npm i jscs jscs-loader --save-dev
```

[jscs-loader](#)³² provides Webpack hooks to the tool. Integration is similar as in the case of ESLint. You would define a `.jscsrc` with your style guide rules and use configuration like this:

³²<https://github.com/unindented/jscs-loader>

```

module: {
  preLoaders: [
    {
      test: /\.jsx?$/,
      loaders: ['eslint', 'jscss'],
      include: PATHS.app
    }
  ]
}

```

Here's a sample configuration:

.jscsrc

```

{
  "esnext": true,
  "preset": "google",

  "fileExtensions": [".js", ".jsx"],

  "requireCurlyBraces": true,
  "requireParenthesesAroundIIFE": true,

  "maximumLineLength": 120,
  "validateLineBreaks": "LF",
  "validateIndentation": 2,

  "disallowKeywords": ["with"],
  "disallowSpacesInsideObjectBrackets": null,
  "disallowImplicitTypeConversion": ["string"],

  "safeContextKeyword": "that",

  "excludeFiles": [
    "dist/**",
    "node_modules/**"
  ]
}

```

JSCS supports *package.json* based configuration through `jscssConfig` field.



ESLint implements a large part of the functionality provided by JSCS. It is possible you can skip JSCS altogether provided you configure ESLint correctly. There's a large amount of presets available for both.

11.7 EditorConfig

[EditorConfig](http://editorconfig.org/)³³ allows you to maintain a consistent coding style across different IDEs and editors. Some even come with built-in support. For others you have to install a separate plugin. In addition to this you'll need to set up a `.editorconfig` file like this:

`.editorconfig`

```
root = true

# General settings for whole project
[*]
indent_style = space
indent_size = 4

end_of_line = lf
charset = utf-8
trim_trailing_whitespace = true
insert_final_newline = true

# Format specific overrides
[*.md]
trim_trailing_whitespace = false

[app/**/*.js]
indent_style = space
indent_size = 2
```

11.8 Conclusion

In this chapter you learned how to lint your code using Webpack in various ways. It is one of those techniques that yields benefits over the long term. You can fix possible problems before they become actual issues.

³³<http://editorconfig.org/>

12. Authoring Libraries

[npm](#)¹ is one of the reasons behind the popularity of Node.js. Even though it was used initially for managing back-end packages, it has become increasingly popular for front-end usage as well. As you saw in the earlier chapters, it is easy to consume npm packages through Webpack.

Eventually you may want to author packages of your own. Publishing one is relatively easy. There are a lot of smaller details to know, though. This chapter goes through those so that you can avoid some of the common problems.

12.1 Anatomy of a npm Package

Most of the available npm packages are small and include just a select few files such as:

- *index.js* - On small projects it's enough to have the code at the root. On larger ones you may want to start splitting it up further.
- *package.json* - npm metadata in JSON format
- *README.md* - README is the most important document of your project. It is written in Markdown format and provides an overview. For simple projects the whole documentation can fit there. It will be shown at the package page at *npmjs.com*.
- *LICENSE* - You should include licensing information within your project. You can refer to it from *package.json*.

In larger projects you may find the following:

- *CONTRIBUTING.md* - A guide for potential contributors. How should the code be developed and so on.
- *CHANGELOG.md* - This document describes major changes per version. If you do major API changes, it can be a good idea to cover them here. It is possible to generate the file based on Git commit history, provided you write nice enough commits.
- *.travis.yml* - [Travis CI](#)² is a popular continuous integration platform that is free for open source projects. You can run the tests of your package over multiple systems using it. There are other alternatives of course, but Travis is very popular.
- *.gitignore* - Ignore patterns for Git, i.e., which files shouldn't go under version control. It can be useful to ignore npm distribution files here so they don't clutter your repository.

¹<https://www.npmjs.com/>

²<https://travis-ci.org/>

- *.npmignore* - Ignore patterns for npm. This describes which files shouldn't go to your distribution version.
- *.eslintignore* - Ignore patterns for ESLint. Again, tool specific.
- *.eslintrc* - Linting rules. You can use *.jshintrc* and such based on your preferences.
- *webpack.config.js* - If you are using a simple setup, you might as well have the configuration at project root.

In addition, you'll likely have various directories for source, tests, demos, documentation, and so on.

12.2 Understanding *package.json*

All packages come with a *package.json* that describes metadata related to them. This includes information about the author, various links, dependencies, and so on. The [official documentation](#)³ covers them in detail.

I've annotated a part of *package.json* of my [React component boilerplate](#)⁴ below.

```
{
  /* Name of the project */
  "name": "react-component-boilerplate",
  /* Brief description */
  "description": "Boilerplate for React.js components",
  /* Who is the author + optional email + optional site */
  "author": "Juho Vepsalainen <email goes here> (site goes here)",
  /* Version of the package */
  "version": "0.0.0",
  /* `npm run <name>` */
  /* You can namespace these if you want. Example: build:dist, build:gh-pages */
  "scripts": {
    "start": "webpack-dev-server",
    "test": "karma start",
    "tdd": "karma start --auto-watch --no-single-run",
    "gh-pages": "webpack",
    "deploy-gh-pages": "node ./lib/deploy_gh_pages.js",
    "dist": "webpack",
    "dist-min": "webpack",
    "dist-modules": "babel ./src --out-dir ./dist-modules",
    "lint": "eslint . --ext .js --ext .jsx",
    "preversion": "npm run test && npm run lint && npm run dist && npm run dist-\\"
  }
}
```

³<https://docs.npmjs.com/files/package.json>

⁴<https://github.com/survivejs/react-component-boilerplate>

```

min && git commit --allow-empty -am \"Update dist\\",
  "prepublish": "npm run dist-modules",
  "postpublish": "npm run gh-pages && npm run deploy-gh-pages",
  /* If your library is installed through Git, you may want to transpile it */
  "postinstall": "node lib/post_install.js"
},
/* Entry point for terminal (i.e., <package name>) */
/* Don't set this unless you intend to allow CLI usage */
"bin": "./index.js",
/* Entry point (defaults to index.js) */
"main": "dist-modules",
/* Package dependencies */
"dependencies": {
  "react": "^0.14.0",
  "react-dom": "^0.14.0"
},
/* Package development dependencies */
"devDependencies": {
  "babel": "^5.8.23",
  "babel-eslint": "^4.1.3",
  ...
  "webpack": "^1.12.2",
  "webpack-dev-server": "^1.12.0",
  "webpack-merge": "^0.2.0"
},
/* Links to repository, homepage, and issue tracker */
"repository": {
  "type": "git",
  "url": "https://github.com/bebraw/react-component-boilerplate.git"
},
"homepage": "https://bebraw.github.io/react-component-boilerplate/",
"bugs": {
  "url": "https://github.com/bebraw/react-component-boilerplate/issues"
},
/* Keywords related to package. */
/* Fill this well to make the package findable. */
"keywords": [
  "react",
  "reactjs",
  "boilerplate"
],
/* Which license to use */

```

```
"license": "MIT"
}
```

As you can see *package.json* can contain a lot of information. You can attach non-npm specific metadata there that can be used by tooling. Given this can bloat *package.json*, it may be preferable to keep metadata at files of their own.

12.3 npm Workflow

Working with npm is surprisingly simple. To get started, you need to use [npm adduser](https://docs.npmjs.com/cli/adduser)⁵ (aliased to `npm login`). It allows you to set up an account. After this process has completed, it will create `~/.npmrc` and use that data for authentication. There's also [npm logout](https://docs.npmjs.com/cli/logout)⁶ that will clear the credentials.



`npm init` respects the values set at `~/.npmrc`. Hence it may be worth your while to set reasonable defaults there to save some time.

Publishing a Package

Provided you have logged in, creating new packages is just a matter of hitting `npm publish`. Given that the package name is still available and everything goes fine, you should have something out there! After this you can install your package through `npm install` or `npm i` as we've done so many times before in this book.

An alternative way to consume a library is to point at it directly in *package.json*. In that case you can do `"depName": "<github user>/<project>#<reference>"` where `<reference>` can be either commit hash, tag, or branch. This can be useful, especially if you need to hack around something and cannot wait for a fix.

Bumping a Version

In order to bump your package version, you'll just need to invoke `npm version <x.y.z>`. That will update *package.json* and create a version commit to git automatically. If you hit `npm publish`, you should have something new out there.

Note that in the example above I've set up version related hooks to make sure a version will contain a fresh version of a distribution build. I also run tests just in case.

⁵<https://docs.npmjs.com/cli/adduser>

⁶<https://docs.npmjs.com/cli/logout>

Publishing a Prerelease Version

Sometimes you might want to publish something preliminary for other people to test. There are certain conventions for this. You rarely see *alpha* releases at npm. *beta* and **rc* (release candidate) are common, though. For example, a package might have versions like this:

- v0.5.0-alpha1
- v0.5.0-beta1
- v0.5.0-beta2
- v0.5.0-rc1
- v0.5.0-rc2
- v0.5.0

The initial alpha release will allow the users to try out the upcoming functionality and provide feedback. The beta releases can be considered more stable. The release candidates (rc) are close to an actual release and won't introduce any new functionality. They are all about refining the release till it's suitable for general consumption.

The workflow in this case is straight-forward:

1. `npm version 0.5.0-alpha1` - Update *package.json* as discussed earlier.
2. `npm publish --tag alpha1` - Publish the package under *alpha1* tag.

In order to consume the test version, your users will have to use `npm i <your package name>@alpha1`.



It can be useful to utilize `npm link` during development. That will allow you to use a development version of your library from some other context. Node.js will resolve to the linked version unless local `node_modules` happens to contain a version. If you want to remove the link, use `npm unlink`.

On Naming Packages

Before starting to develop, it can be a good idea to spend a little bit of time on figuring out a good name for your package. It's not very fun to write a great package just to notice the name has been taken. A good name is easy to find through a search engine, and most importantly, is available at npm.

As of npm 2.7.0 it is possible to create [scoped packages](https://docs.npmjs.com/getting-started/scoped-packages)⁷. They follow format `@username/project-name`. Simply follow that when naming your project.

⁷<https://docs.npmjs.com/getting-started/scoped-packages>

Dealing with Preprocessing

It is possible to consume a library directly through Git. This can be problematic for library authors. One way to solve this is to set up a `postinstall` script that will generate a local npm version of your library in case it doesn't exist. This can be achieved through a `postinstall` script like this:

package.json

```
{
  ...
  "scripts": {
    ...
    "postinstall": "node lib/post_install.js"
  },
  "devDependencies": {
    ...
    /* You should install sync-exec through `npm i` to get a recent version */
    "sync-exec": "^0.6.2"
  }
}
```

In addition we need to define a little script to do the work for us. It will check whether our package contains the directory we expect and will then act based on that. If it doesn't exist, we'll generate it. You may need to tweak the script to fit your exact purposes. The idea is the same, though:

lib/post_install.js

```
// adapted based on rackt/history (MIT)
// Node 0.10+
var execSync = require('child_process').execSync;
var stat = require('fs').stat;

// Node 0.10 check
if (!execSync) {
  execSync = require('sync-exec');
}

function exec(command) {
  execSync(command, {
    stdio: [0, 1, 2]
  });
}
```

```
stat('dist-modules', function(error, stat) {
  if (error || !stat.isDirectory()) {
    exec('npm i babel@5.x');
    exec('npm run dist-modules');
  }
});
```

Even though setting up a script like this takes some time, it can be beneficial as it will make it a little easier to consume your library. After this you can point at your library directly through Git. If you are into shorthands, you could use a dependency declaration such as "my-project": "<name>/<project>#reference". *reference* can be a commit id or a tag name for instance.

Respect the SemVer

Even though it is simple to publish new versions out there, it is important to respect the SemVer. Roughly it states that you should not break backwards compatibility, given certain rules are met. For example, if your current version is 0.1.4 and you do a breaking change, you should bump to 0.2.0 and document the changes. You can understand SemVer much better by studying [the online tool⁸](#) and how it behaves.

Version Ranges

npm supports multiple version ranges. I've listed the common ones below:

- ~ - Tilde matches only patch versions. For example, ~1.2 would be equal to 1.2.x.
- ^ - Caret is the default you get using --save or --save-dev. It matches to It matches minor versions. This means ^0.2.0 would be equal to 0.2.x.
- * - Asterisk matches major releases. This is the most dangerous of the ranges. Using this recklessly can easily break your project in the future and I would advise against using it.

You can set the default range using `npm config set save-prefix='^'` in case you prefer something else than caret. Alternatively you can modify `~/.npmrc` directly.



Sometimes using version ranges can feel a little dangerous. What if some future version is broken? `npm shrinkwrap9` allows you to fix your project versions and have stricter control over the versions you are using in a production environment.

⁸<http://semver.npmjs.com/>

⁹<https://docs.npmjs.com/cli/shrinkwrap>

12.4 Library Formats

I output my React component in various formats at my boilerplate. I generate a version that's convenient to consume from Node.js by processing my component code through Babel. That will convert ES6 and other goodies to a format which is possible to consume from vanilla Node.js. This allows the user to refer to some specific module within the whole if needed.

In addition, I generate so called *distribution bundles*: `.js` and `.min.js`. There's a sourcemap (`.map`) useful for debugging for both. It is possible to consume these bundles standalone as they come with an UMD¹⁰ wrapper.

UMD makes it possible to consume them from various environments including global, AMD, and CommonJS (Node.js format). You can refresh your memory with these by checking the Getting Started chapter for examples.

It is surprisingly easy to generate the aforementioned bundles using Webpack. The following example should give you the basic idea:

webpack.config.js

```
...

var config = {
  paths: {
    dist: '...',
    src: '...',
  },
  filename: 'demo',
  library: 'Demo'
};

var commonDist = {
  devtool: 'source-map',
  output: {
    path: config.paths.dist,
    libraryTarget: 'umd',
    library: config.library
  },
  entry: config.paths.src,
  externals: {
    react: 'react'
    /* more complicated mapping for lodash */
    /* we need to access it differently depending */
  }
};
```

¹⁰<https://github.com/umdjs/umd>

```

    /* on the environment */
    lodash: {
      commonjs: 'lodash',
      commonjs2: 'lodash',
      amd: '_',
      root: '_'
    }
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        loaders: ['babel'],
        include: config.paths.src
      }
    ]
  }
};

if(TARGET === 'dist') {
  module.exports = merge(commonDist, {
    output: {
      filename: config.filename + '.js'
    },
  });
}

if(TARGET === 'dist-min') {
  module.exports = merge(commonDist, {
    output: {
      filename: config.filename + '.min.js'
    },
    plugins: [
      new webpack.optimize.UglifyJsPlugin({
        compress: {
          warnings: false
        }
      })
    ]
  });
}

```

Most of the magic happens thanks to devtool and output declarations. In addition, I have set up

externals as I want to avoid bundling React and lodash into my library. Instead, both will be loaded as external dependencies using the naming defined in the mapping.



The example uses the same merge utility we defined earlier on. You should check [the boilerplate](#)¹¹ itself for the exact configuration.



If your library is using ES6 exclusively, [rollup](#)¹² can be a valid, simple alternative to Webpack.

12.5 npm Lifecycle Hooks

npm provides various lifecycle hooks that can be useful. Suppose you are authoring a React component using Babel and some of its goodies. You could let the *package.json* main field point at the UMD version as generated above. This won't be ideal for those consuming the library through npm, though.

It is better to generate a ES5 compatible version of the package for npm consumers. This can be achieved using **babel** CLI tool:

```
babel ./lib --out-dir ./dist-modules
```

This will walk through the *./lib* directory and output a processed file for each library it encounters to *./dist-modules*.

Since we want to avoid having to run the command directly whenever we publish a new version, we can connect it to *prepublish* hook like this:

```
"scripts": {
  ...
  "prepublish": "babel ./lib --out-dir ./dist-modules"
}
```

Make sure you hit `npm i babel --save-dev` to include the tool into your project.

You probably don't want the directory content to end up in your Git repository. In order to avoid this and to keep your `git status` clean, consider this sort of *.gitignore*:

¹¹<https://github.com/bebraw/react-component-boilerplate>

¹²<https://www.npmjs.com/package/rollup>

```
dist-modules/  
...
```

Besides `prepublish`, npm provides a set of other hooks. The naming is always the same and follows the pattern `pre<hook>`, `<hook>`, `post<hook>` where `<hook>` can be `publish`, `install`, `test`, `stop`, `start`, `restart`, or `version`. Even though npm will trigger scripts bound to these automatically, you can trigger them explicitly through `npm run` for testing (i.e., `npm run prepublish`).

There are plenty of smaller tricks to learn for advanced usage. Those are better covered by [the official documentation](#)¹³. Often all you need is just a `prepublish` script for build automation.

12.6 Keeping Dependencies Up to Date

An important part of maintaining npm packages is keeping their dependencies up to date. How to do this depends a lot on the maturity of your package. Ideally you have a nice set of tests covering the functionality. If not, things can get a little hairier. There are a few ways to approach dependency updates:

- You can update all dependencies at once and hope for the best. Tools such as [npm-check-updates](#)¹⁴ can do this for you.
- Install the newest version of some specific dependency, e.g., `npm i lodash@* --save`. This is a more controlled way to approach the problem.
- Patch version information by hand by modifying `package.json` directly.

It is important to remember that your dependencies may introduce backwards incompatible changes. It can be useful to remember how SemVer works and study dependency release notes. They might not always exist, so you may have to go through the project commit history. There are a few services that can help you to keep track of your project dependencies:

- [David](#)¹⁵
- [versioneye](#)¹⁶
- [Gemnasium](#)¹⁷

These services provide badges you can integrate into your project *README.md*. These services may email you about important changes. They can also point out possible security issues that have been fixed.

¹³<https://docs.npmjs.com/misc/scripts>

¹⁴<https://www.npmjs.com/package/npm-check-updates>

¹⁵<https://david-dm.org/>

¹⁶<https://www.versioneye.com/>

¹⁷<https://gemnasium.com>

For testing your projects you can consider solutions such as [Travis CI](#)¹⁸ or [SauceLabs](#)¹⁹. [Coveralls](#)²⁰ gives you code coverage information and a badge.

These services are valuable as they allow you to test your updates against a variety of platforms quickly. Something that might work on your system might not work in some specific configuration. You'll want to know about that as fast as possible to avoid introducing problems.

12.7 Sharing Authorship

As packages evolve you may want to start developing with others. You could become the new maintainer of some project, or pass the torch to someone else. These things happen as packages evolve.

npm provides a few commands for these purposes. It's all behind `npm owner namespace`. More specifically you'll find `ls <package name>`, `add <user> <package name>` and `rm <user> <package name>` there (i.e., `npm owner ls`). That's about it.

See [npm documentation](#)²¹ for the most up to date information about the topic.

12.8 Conclusion

You should have a basic idea on how to author npm libraries with the help of Webpack now. It takes a lot of effort out of the process. Just keep the basic rules in mind when developing and remember to respect the SemVer.

¹⁸<https://travis-ci.org/>

¹⁹<https://saucelabs.com/>

²⁰<https://coveralls.io/>

²¹<https://docs.npmjs.com/cli/owner>

13. 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 different.

This change in the mindset has lead 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. It is difficult to provide any definite recommendations at the moment. Instead, I'm going through some common approaches so you can make up your mind based on your needs.

13.1 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.

Webpack Configuration for Vanilla CSS

It is easy to configure vanilla CSS in Webpack. Consider the example below:

`webpack.config.js`

```
var common = {
  ...
  module: {
    loaders: [
      {
        test: /\.css$/,
        loaders: ['style', 'css']
      }
    ]
  },
  ...
};
```

First [css-loader](#)¹ goes through possible `@import` and `url()` statements within the matched files and treats them as regular `require`. This allows us to rely on various other loaders such as [file-loader](#)² or [url-loader](#)³.

`file-loader` generates files, whereas `url-loader` can create inline data URLs for small resources. This can be useful for optimizing application loading. You avoid unnecessary requests while providing a slightly bigger payload. Small improvements can yield large benefits if you depend on a lot of small resources in your style definitions.

Finally `style-loader` picks up `css-loader` output and injects the CSS into the bundle. As we saw earlier in the build chapter, it is possible to use `ExtractTextPlugin` to generate a separate CSS file.

13.2 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](#)⁴ (Object-Oriented CSS), [SMACSS](#)⁵ (Scalable and Modular Approach for CSS), and [BEM](#)⁶ (Block Element Modifier) are well known. Each of them solves problems of vanilla CSS in their own way.

BEM

BEM originates from Yandex. The goal of BEM is to allow reusable components and code sharing. Sites such as [Get BEM](#)⁷ 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](#)⁸, [react-bem-render](#)⁹, and [bem-react](#)¹⁰.

Note that [postcss-bem-linter](#)¹¹ allows you to lint your CSS for BEM conformance.

¹<https://www.npmjs.com/package/css-loader>

²<https://www.npmjs.com/package/file-loader>

³<https://www.npmjs.com/package/url-loader>

⁴<http://oocss.org/>

⁵<https://smacss.com/>

⁶<https://en.bem.info/method/>

⁷<http://getbem.com/>

⁸<https://www.npmjs.com/package/react-bem-helper>

⁹<https://www.npmjs.com/package/react-bem-render>

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

¹¹<https://www.npmjs.com/package/postcss-bem-linter>

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

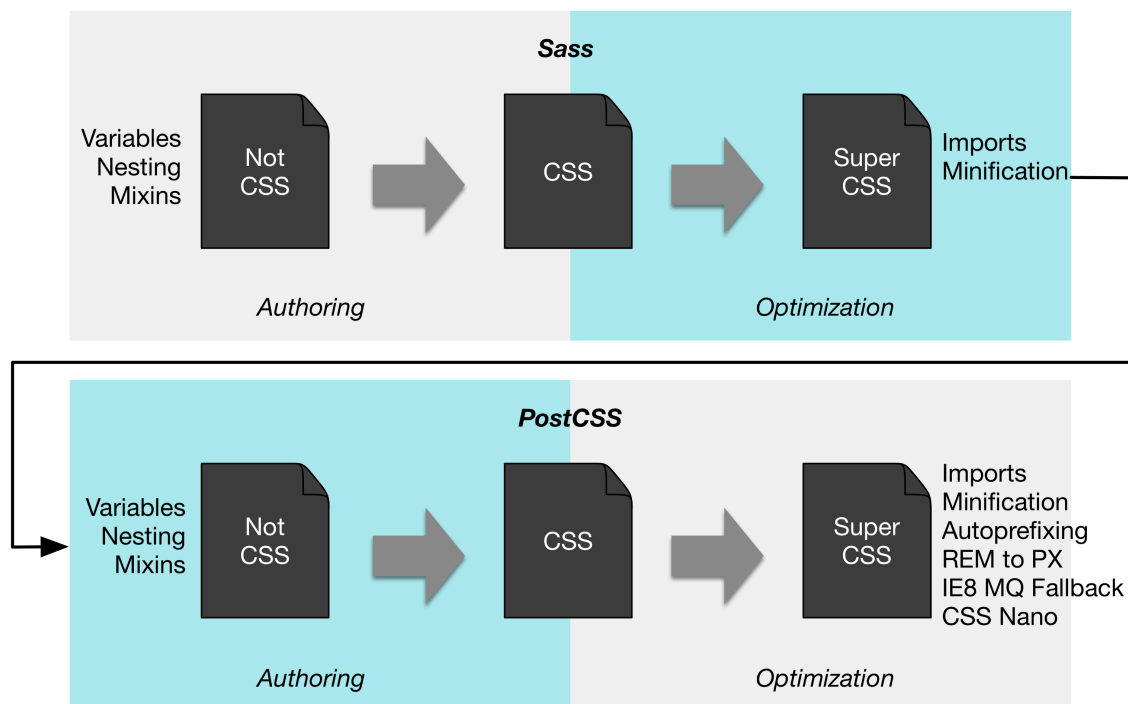
The primary benefit of adopting a methodology is that it brings structure to your project. Rather than writing ad hoc rules and hoping everything works, you will have something stronger to fall back onto. The methodologies overcome some of the basic issues and help you develop good software over the 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.

13.3 Less, Sass, Stylus, PostCSS, cssnext



CSS Processors

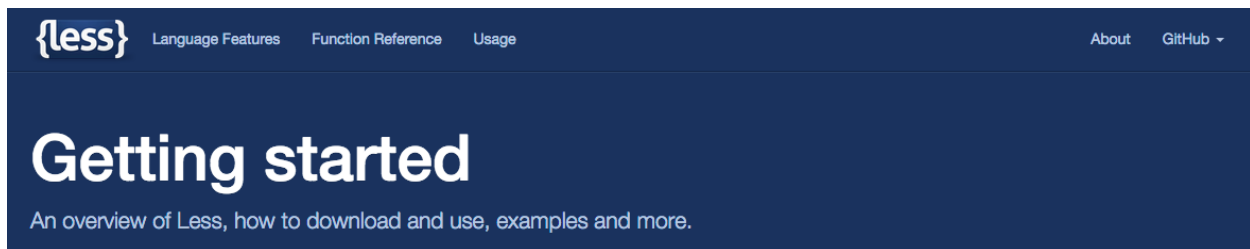
Vanilla CSS is missing some functionality that would make maintenance work easier. Consider something basic like variables, nesting, mixins, math or color functions, and so on. It would also be nice to be able to forget about browser specific prefixes. These are small things that add up quite fast and make it annoying to write vanilla CSS.

Sometimes you may see terms *preprocessor* or *postprocessor*. [Stefan Baumgartner¹²](https://medium.com/@ddprtt/deconfusing-pre-and-post-processing-d68e3bd078a3) 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.

¹²<https://medium.com/@ddprtt/deconfusing-pre-and-post-processing-d68e3bd078a3>

Less



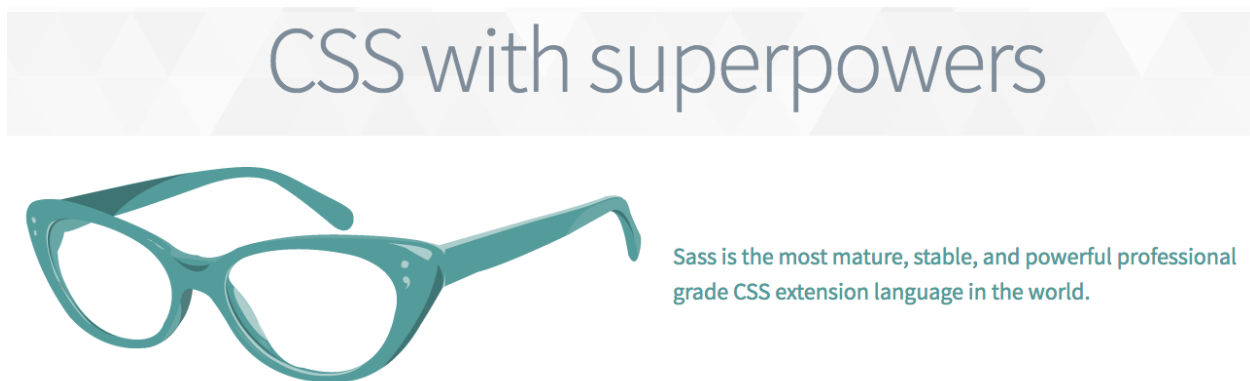
Less

[Less](#)¹³ is a popular CSS processor that is packed with functionality. In Webpack using Less doesn't take a lot of effort. [less-loader](#)¹⁴ deals with the heavy lifting:

```
{
  test: /\.less$/,
  loaders: ['style', 'css', 'less']
}
```

There is also support for Less plugins, sourcemaps, and so on. To understand how those work you should check out the project itself.

Sass



Sass

[Sass](#)¹⁵ is a popular alternative to Less. You should use [sass-loader](#)¹⁶ with it. Remember to install `node-sass` to your project as the loader has a peer dependency on that. Webpack doesn't take much configuration:

¹³<http://lesscss.org/>

¹⁴<https://www.npmjs.com/package/less-loader>

¹⁵<http://sass-lang.com/>

¹⁶<https://www.npmjs.com/package/sass-loader>

```
{  
  test: /\.scss$/,  
  loaders: ['style', 'css', 'sass']  
}
```

Check out the loader for more advanced usage.

Stylus



Stylus

[Stylus](https://learnboost.github.io/stylus/)¹⁷ is a Python inspired way to write CSS. Besides providing an indentation based syntax, it is a full-featured processor. When using Webpack, you can use [stylus-loader](https://www.npmjs.com/package/stylus-loader)¹⁸ to Stylus within your project. Configure as follows:

```
{  
  test: /\.styl$/,  
  loaders: ['style', 'css', 'stylus']  
}
```

You can also use Stylus plugins with it by setting `stylus.use: [plugin()]`. Check out the loader for more information.

¹⁷<https://learnboost.github.io/stylus/>

¹⁸<https://www.npmjs.com/package/stylus-loader>

PostCSS

[PostCSS](https://github.com/postcss/postcss)¹⁹ allows you to perform transformations over CSS through JavaScript plugins. You can even find plugins that provide you Sass-like features. PostCSS can be thought as the equivalent of Babel for styling. It can be used through [postcss-loader](https://www.npmjs.com/package/postcss-loader)²⁰ with Webpack as below:

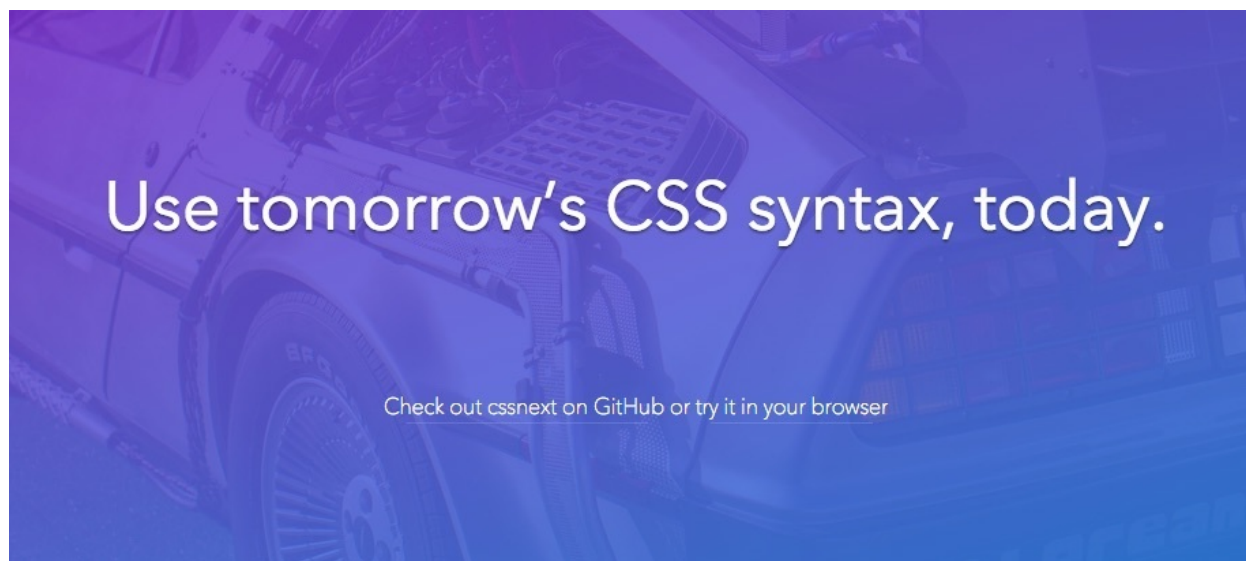
```
var autoprefixer = require('autoprefixer');
var precss = require('precss');

module.exports = {
  module: {
    loaders: [
      {
        test: /\.css$/,
        loaders: ['style', 'css', 'postcss']
      }
    ]
  },
  // PostCSS plugins go here
  postcss: function () {
    return [autoprefixer, precss];
  }
};
```

¹⁹<https://github.com/postcss/postcss>

²⁰<https://www.npmjs.com/package/postcss-loader>

cssnext



cssnext

[cssnext](https://cssnext.github.io/)²¹ is a PostCSS plugin that allows us to experience the future now. There are some restrictions, but it may be worth a go. In Webpack it is simply a matter of installing [cssnext-loader](https://www.npmjs.com/package/cssnext-loader)²² and attaching it to your CSS configuration. In our case, you would end up with the following:

```
{  
  test: /\.css$/,  
  loaders: ['style', 'css', 'cssnext']  
}
```

Alternatively you could consume it through *postcss-loader* as a plugin if you need more control.

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.

²¹<https://cssnext.github.io/>

²²<https://www.npmjs.com/package/cssnext-loader>

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.

13.4 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:

```
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](#)²³ for a more a comprehensive outlook of the situation.

Radium

[Radium](#)²⁴ 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%',
```

²³<https://github.com/MicheleBertoli/css-in-js>

²⁴<http://projects.formidablelabs.com/radium/>

```

      ':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.

React Style

[React Style](https://github.com/js-next/react-style)²⁵ uses the same syntax as React Native [StyleSheet](https://facebook.github.io/react-native/docs/stylesheets.html#content)²⁶. 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%'
    }
  }
});

```

²⁵<https://github.com/js-next/react-style>

²⁶<https://facebook.github.io/react-native/docs/stylesheets.html#content>

```

    }
  }
});

...

<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.

Interestingly there is a [React Style plugin for Webpack](#)²⁷. It 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.

JSS

[JSS](#)²⁸ 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](#)²⁹. There's also an experimental [jss-loader](#)³⁰ for Webpack. 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%'
    }
  },
};

```

²⁷<https://github.com/js-next/react-style-webpack-plugin>

²⁸<https://github.com/jsstyles/jss>

²⁹<https://www.npmjs.com/package/react-jss>

³⁰<https://www.npmjs.com/package/jss-loader>

```

    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>;
  }
}

```

The approach supports pseudoselectors, i.e., you could define a selector within such as `&:hover` within a definition and it would just work.

React Inline

[React Inline³¹](#) 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>;
  }
}

```

³¹<https://github.com/martinandert/react-inline>

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. As of the time of this writing, there's no Webpack loader available.

jsxstyle

Pete Hunt's [jsxstyle](https://github.com/petehunt/jsxstyle)³² 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:

```
// 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.

13.5 CSS Modules

As if there weren't enough styling options for React, there's one more that's worth mentioning. [CSS Modules](https://github.com/css-modules/css-modules)³³ 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](https://medium.com/seek-ui-engineering/the-end-of-global-css-90d2a4a06284)³⁴ 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

³²<https://github.com/petehunt/jsxstyle>

³³<https://github.com/css-modules/css-modules>

³⁴<https://medium.com/seek-ui-engineering/the-end-of-global-css-90d2a4a06284>

```
.primary {  
  background: 'green';  
}  
  
.warning {  
  background: 'yellow';  
}  
  
.button {  
  padding: 1em;  
}  
  
@media (max-width: 200px) {  
  .button {  
    width: 100%;  
  }  
}
```

button.jsx

```
import classNames from 'classnames';  
import styles from './style.css';  
  
...  
  
<button className={classNames(  
  styles.button, styles.primary  
)}>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](https://css-modules.github.io/webpack-demo/)³⁵ for more examples.



[gajus/react-css-modules](https://github.com/gajus/react-css-modules)³⁶ 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.

³⁵<https://css-modules.github.io/webpack-demo/>

³⁶<https://github.com/gajus/react-css-modules>

13.6 Conclusion

It is simple to try out various styling approaches with Webpack. 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.

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](#)³⁷ (stage 1) only class and method level wrapping is supported. Functions may become supported later on.

Implementing 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);
  };

  return descriptor;
}

const math = new Math();

// passed parameters should get logged now
math.add(2, 4);
```

³⁷<https://github.com/wycats/javascript-decorators>

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`³⁸ 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 for example 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.

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`

³⁸<https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/arguments>

```
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 [ES7 rest spread operator](https://github.com/sebmarkbage/ecmascript-rest-spread)³⁹. It expands the given object to separate key-value pairs, or props, as in this case.

You can connect the decorator with App like this for example:

app/components/App.jsx

³⁹<https://github.com/sebmarkbage/ecmascript-rest-spread>

```

...
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.

To get familiar with more approaches we'll be using the `AltContainer` in this project. Using the decorator is completely acceptable. It comes down to your personal preferences.

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 peer dependency 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 will be dealt with differently starting with npm 3. After that it's up to the package consumer (i.e., you) to deal with it. This particular error will go away.

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 hit `npm i` at an unfortunate time, you may end up with a project that doesn't compile.

A good first step is to hit `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)⁴⁰ goes into more detail on the topic.

⁴⁰<https://docs.npmjs.com/cli/shrinkwrap>