SurviveJS - Webpack and React

From apprentice to master

Juho Vepsäläinen

SurviveJS - Webpack and React

From apprentice to master

Juho Vepsäläinen

Contents

Introduction . . . . **i**

What is Webpack? i

What is React? i

How is This Book Organized? ii

What is Kanban? iii

Who is This Book for? iii

Extra Material iii

Getting Support iv

Announcements iv

Acknowledgments v

I Setting Up Webpack 1

1. Webpack Compared **2**

1.1 Make 2

1.2 Grunt 4

1.3 Gulp 5

1.4 Browserify 7

1.5 Webpack 8

1.6 JSPM 10

1.7 Why Use Webpack? 10

1.8 Module Formats Supported by Webpack 11

1.9 Conclusion 13

2. Developing with Webpack **14**

2.1 Setting Up the Project 14

2.2 Installing Webpack 15

2.3 Directory Structure 15

2.4 Setting Up Assets 16

2.5 Setting Up Webpack Configuration 16

2.6 Setting Up *webpack-dev-server* 18

2.7 Refreshing CSS 20

2.8 Making the Configuration Extensible 23

2.9 Linting the Project 26
2.10 Conclusion 26

3. Webpack and React 27
3.1 What is React? 27
3.2 Babel 30
3.3 Developing the First React View 32
3.4 Activating Hot Loading for Development 35
3.5 React Component Styles 37
3.6 Conclusion 37

II Developing Kanban Application 38

4. Implementing a Basic Note Application 39
4.1 Initial Data Model 39
4.2 On Ids 40
4.3 Connecting Data with `App` 40
4.4 Fixing `Note` 42
4.5 Extracting `Notes` 43
4.6 Pushing `notes` to the `App` `state` 45
4.7 Adding New Items to `Notes` list 47
4.8 Editing Notes 49
4.9 Removing `Notes` 56
4.10 Styling Notes 59
4.11 Understanding React Components 62
4.12 React Component Conventions 63
4.13 Conclusion 64

Introduction

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

What is Webpack?

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

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

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

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

What is React?

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

Given React focuses only on the view you'll likely have to complement it with other libraries to give you the missing bits. This provides an interesting contrast to framework based approaches as they give you a lot more out of the box.

[1]https://webpack.github.io/
[2]https://facebook.github.io/react/

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

How is This Book Organized?

This book will guide you through a small example project. After completing it, we discuss more theoretical aspects of web development. The project in question will be a small Kanban[3] application.

We will start by building a Webpack based configuration. After that we will develop a small clone of a famous Todo application[4]. We will generalize from there and put in place Flux architecture[5] within our application. We will apply some Drag and Drop (DnD) magic[6] and start dragging things around. Finally we will get a production grade build done.

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

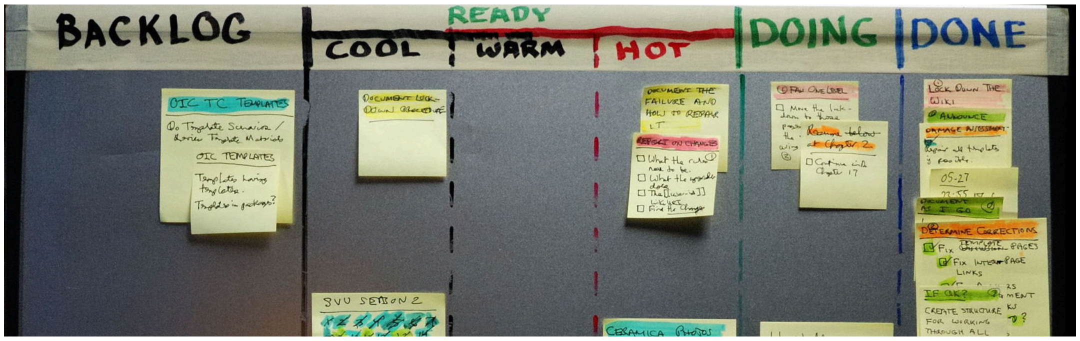
- Deal with typing in React.
- Test your components and logic.

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

- Lint your code effectively using ESLint[7] and some other tools.
- Author libraries at npm[8].
- Style React in various emerging ways.

[3]https://en.wikipedia.org/wiki/Kanban
[4]http://todomvc.com/
[5]https://facebook.github.io/flux/docs/overview.html
[6]https://gaearon.github.io/react-dnd/
[7]http://eslint.org/
[8]https://www.npmjs.com/

What is Kanban?



Kanban by Dennis Hamilton (CC BY)

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

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

As the system gets more sophisticated you can start applying concepts such as a limit on Work In Progress (WIP). The effect of this is that you are forced to focus on getting tasks done. That is one of the good consequences of using Kanban. Moving those notes around is satisfying. As a bonus you get visibility and know what is yet to be done.

A good example of Kanban in action on the web is Trello[9]. Sprintly has open sourced their React implementation of Kanban[10]. Ours won't be as sophisticated, but it will be enough to get started.

Who is This Book for?

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

Extra Material

The book content and source are available at GitHub[11]. This allows you to start from any chapter you want.

[9]https://trello.com/
[10]https://github.com/sprintly/sprintly-kanban
[11]https://github.com/survivejs/webpack_react

You can also find alternative implementations of the application using mobservable[12], Redux[13], and Cerebral/Baobab[14]. Studying those can give you a good idea of how different architectures work out using the same example.

Getting Support

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

- GitHub Issue Tracker[15]
- Gitter Chat[16]
- Twitter - @survivejs[17] or poke me through @bebraw[18]
- Email - info@survivejs.com[19]

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

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

Announcements

I announce SurviveJS related news through a couple of channels:

- Mailing list[20]
- Twitter[21]
- Blog RSS[22]

Feel free to subscribe.

[12] https://github.com/survivejs/mobservable-demo
[13] https://github.com/survivejs/redux-demo
[14] https://github.com/survivejs/cerebral-demo
[15] https://github.com/survivejs/webpack_react/issues
[16] https://gitter.im/survivejs/webpack_react
[17] https://twitter.com/survivejs
[18] https://twitter.com/bebraw
[19] mailto:info@survivejs.com
[20] http://eepurl.com/bth1v5
[21] https://twitter.com/survivejs
[22] http://survivejs.com/atom.xml

Acknowledgments

Big thanks to Christian Alfoni[23] for starting the react-webpack-cookbook[24] with me. That work eventually lead to this book.

The book wouldn't be half as good as it is without patient editing and feedback by my editor Jesús Rodríguez Rodríguez[25]. Thank you.

Special thanks to Steve Piercy for numerous contributions. Thanks to Prospect One[26] for helping with the logo and graphical outlook. Thanks for proofreading to Ava Mallory and EditorNancy from fiverr.com.

Numerous individuals have provided support and feedback along the way. Thank you in no particular order Vitaliy Kotov, @af7, Dan Abramov, @dnmd, James Cavanaugh, Josh Perez, Nicholas C. Zakas, Ilya Volodin, Jan Nicklas, Daniel de la Cruz, Robert Smith, Andreas Eldh, Brandon Tilley, Braden Evans, Daniele Zannotti, Partick Forringer, Rafael Xavier de Souza, Dennis Bunskoek, Ross Mackay, Jimmy Jia, Michael Bodnarchuk, Ronald Borman, Guy Ellis, Mark Penner, Cory House, Sander Wapstra, Nick Ostrovsky, Oleg Chiruhin, Matt Brookes, Devin Pastoor, Yoni Weisbrod, Guyon Moree, Wilson Mock, Herryanto Siatono, Héctor Cascos, Erick Bazán, Fabio Bedini, Gunnari Auvinen, Aaron McLeod, John Nguyen, Hasitha Liyanage, Mark Holmes, Brandon Dail, Ahmed Kamal, Jordan Harband, Michel Weststrate, Ives van Hoorne, Luca DeCaprio, @dev4Fun, Fernando Montoya, Hu Ming, @mpr0xy, David Gómez, Aleksey Guryanov, Elio D'antoni, Yosi Taguri, Ed McPadden, Wayne Maurer, Adam Beck, Omid Hezaveh, Connor Lay, Nathan Grey, Avishay Orpaz, Jax Cavalera, Juan Diego Hernández, Peter Poulsen, and Harro van der Klauw. If I'm missing your name, I might have forgotten to add it.

[23]http://www.christianalfoni.com/
[24]https://github.com/christianalfoni/react-webpack-cookbook
[25]https://github.com/Foxandxss
[26]http://prospectone.pl/

I Setting Up Webpack

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

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

1. Webpack Compared

You can understand Webpack better by putting it into an historical context. This will show you why its approach is powerful. Back in the day it was enough just to concatenate some scripts together. Times have changed, though. Now distributing your JavaScript code can be a complex endeavor.

This problem has been escalated by the rise of single page applications (SPAs). They tend to rely on various heavy 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[1], the Node.js package manager, provides more context. Before npm it was difficult to consume dependencies. Now that npm has become popular for front-end development, the situation has changed. Now we have nice ways to manage the dependencies of our projects.

Historically speaking there have been many build systems. Make[2] is perhaps the most known one and still a viable option. In the front-end world Grunt[3] and Gulp[4] have particularly gained popularity. Plugins made available through npm make both approaches powerful.

Browserify[5] took one step further. It provides powerful npm based bundling. You can complement it with many smaller utilities. This is a nice contrast to the approach Webpack uses.

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

1.1 Make

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

[1]https://www.npmjs.com/
[2]https://en.wikipedia.org/wiki/Make_%28software%29
[3]http://gruntjs.com/
[4]http://gulpjs.com/
[5]http://browserify.org/
[6]http://jspm.io/
[7]https://github.com/systemjs/systemjs
[8]https://www.youtube.com/watch?t=33&v=iukBMY4apvI

Even though Make is mostly used with C projects, it's not tied to it in any way. James Coglan discusses in detail how to use Make with JavaScript[9]. Consider the abbreviated code based on James' post below:

Makefile

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

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

libraries    := vendor/jquery.js

.PHONY: all clean test

all: $(app_bundle)

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

$(app_bundle): $(libraries) $(build_files)
    uglifyjs -cmo $@ $^

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

clean:
    rm -rf build
```

With Make you will model your tasks using some Make specific syntax and terminal commands. This allows to use it with Webpack easily.

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

1.2 Grunt

Grunt

Grunt went mainstream before Gulp. Its plugin architecture especially contributed towards its popularity. At the same time this 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[10]:

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

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

[10]http://gruntjs.com/sample-gruntfile

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

In this sample, we define two basic tasks related to *jshint*, a linting tool that helps you spot 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 grunt-webpack[11] plugin allows you to use Webpack in a Grunt environment. You can leave the heavy lifting to Webpack.

1.3 Gulp

Gulp

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

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

[11]https://www.npmjs.com/package/grunt-webpack

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

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

// Not all tasks need to use streams
// A gulpfile is just another node program and you can use all packages 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 modules as Gulp plugins, and so on. Compared to Grunt, you have a clearer idea of what's going on. You still end up writing a lot of boilerplate for casual tasks, though. That is where some newer approaches come in.

gulp-webpack[12] allows you to use Webpack in a Gulp environment.

1.4 Browserify

Browserify

Dealing with JavaScript modules has always been a bit of a problem. The language actually didn't have the concept of modules till ES6. Ergo we are stuck in the '90s when it comes to browser environments. Various solutions, including AMD[13], have been proposed.

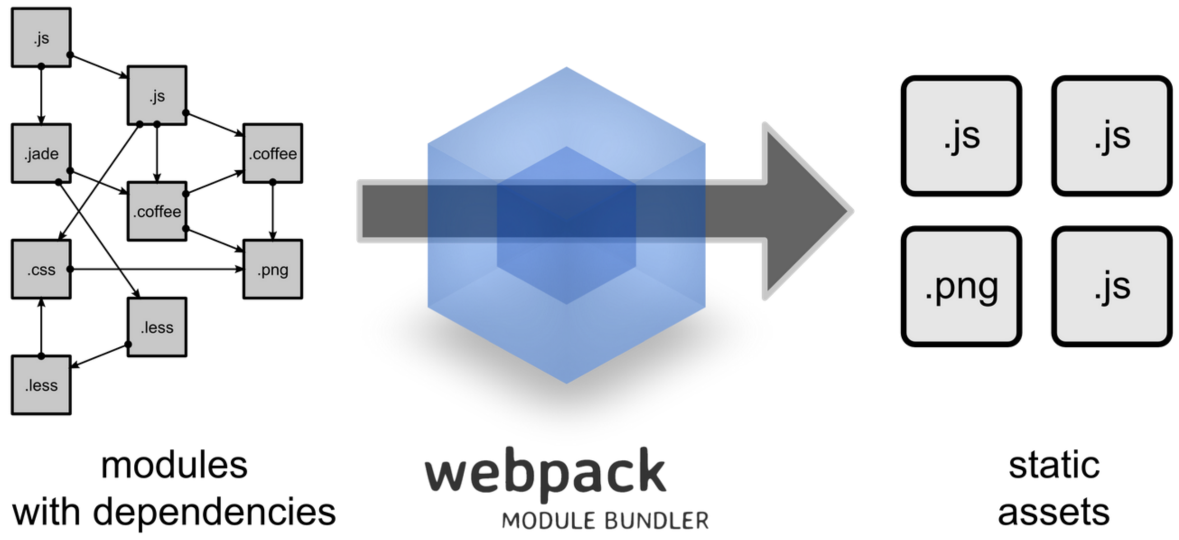
In practice it can be useful just to use CommonJS, the Node.js format, and let the tooling deal with the rest. The advantage is that you can often hook into npm and avoid reinventing the wheel.

Browserify[14] solves this 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[15] provides a file watcher that creates bundles for you during development. This will save some effort and no doubt is a good solution up to a point.

[12]https://www.npmjs.com/package/gulp-webpack
[13]http://requirejs.org/docs/whyamd.html
[14]http://browserify.org/
[15]https://www.npmjs.com/package/watchify

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

1.5 Webpack



webpack

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

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

All this is relies on configuration. It can be difficult to understand what's going on if you haven't seen it before. Fortunately there's certain logic involved. Here is a sample configuration adapted from the official webpack tutorial[16]:

webpack.config.js

[16]http://webpack.github.io/docs/tutorials/getting-started/

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

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

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

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

[17]https://christianalfoni.github.io/react-webpack-cookbook/

1.6 JSPM

JSPM

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

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

1.7 Why Use Webpack?

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

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

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

[18]https://github.com/systemjs/systemjs#plugins
[19]https://github.com/gaearon/react-hot-loader
[20]http://livereload.com/
[21]http://www.browsersync.io/

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

It is possible to achieve some of these tasks with other tools. The problem is that it would definitely take a lot more work to pull off. In Webpack it's a matter of configuration. Note that you can get HMR to Browserify through livereactload[22], so it's not a feature that's exclusive to Webpack.

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

To get a better idea how it compares to some other tools, check out the official comparison[23].

1.8 Module Formats Supported by Webpack

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

CommonJS

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

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

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

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

ES6

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

[22] https://github.com/milankinen/livereactload
[23] https://webpack.github.io/docs/comparison.html

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

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

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

AMD

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

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

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

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

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

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

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

UMD

UMD, universal module definition, takes it all to the next level. It is a monster of a format that aims to make the aforementioned formats compatible with each other. I will spare your eyes from it. Never write it yourself, leave it to the tools. If that didn’t scare you off, check out the official definitions[24].

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

1.9 Conclusion

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

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

[24]https://github.com/umdjs/umd

2. Developing with Webpack

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

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

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

2.1 Setting Up the Project

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

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

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

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

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

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

.gitignore

[1]https://github.com/webpack/css-loader/issues/144

[2]http://nodejs.org/

[3]https://docs.npmjs.com/files/package.json

```
node_modules
```

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

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

2.2 Installing Webpack

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

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

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

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

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

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

2.3 Directory Structure

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

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

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

2.4 Setting Up Assets

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

app/component.js

```
module.exports = function () {
  var element = document.createElement('h1');

  element.innerHTML = 'Hello world';

  return element;
};
```

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

app/index.js

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

document.body.appendChild(app);

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

2.5 Setting Up Webpack Configuration

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

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

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

to install it to the project.

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

webpack.config.js

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

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

module.exports = {
  entry: PATHS.app,
  output: {
    path: PATHS.build,
    filename: 'bundle.js'
  },
  plugins: [
    new HtmlwebpackPlugin({
      title: 'Kanban app'
    })
  ]
};
```

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

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

[4]https://www.npmjs.com/package/html-webpack-plugin

[5]https://nodejs.org/api/path.html

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

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

2.6 Setting Up *webpack-dev-server*

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

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

Hit

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

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

package.json

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

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

webpack.config.js

[6]http://livereload.com/
[7]http://www.browsersync.io/

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

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

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

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

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

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

Hello world

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

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

Alternative Ways to Use *webpack-dev-server*

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

Note that there are slight differences[10] between the CLI and Node.js API and they may behave slightly differently at times. This is the reason why some prefer to use solely Node.js API.

Customizing Server *host* and *port*

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

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

2.7 Refreshing CSS

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

[8]https://webpack.github.io/docs/webpack-dev-middleware.html
[9]https://webpack.github.io/docs/webpack-dev-server.html#api
[10]https://github.com/webpack/webpack-dev-server/issues/106
[11]https://www.npmjs.com/package/node-ip

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

If you are using Node.js 0.10, this is a good time to get a ES6 Promise polyfill[12] set up.

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

webpack.config.js

```
...

module.exports = {
  entry: PATHS.app,
  output: {
    path: PATHS.build,
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.css$/,
        loaders: ['style', 'css'],
        include: PATHS.app
      }
    ]
  },
  devServer: {
    historyApiFallback: true,
    hot: true,
    inline: true,
    progress: true,

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

    // parse host and port from env so this is easy
    // to customize
    host: process.env.HOST,
    port: process.env.PORT
```

[12]https://github.com/jakearchibald/es6-promise#auto-polyfill

```
  },
  plugins: [
    new webpack.HotModuleReplacementPlugin(),
    new HtmlwebpackPlugin({
      title: 'Kanban app'
    })
  ]
};
```

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

Loaders are transformations that are applied to source files, and return the new source. Loaders can be chained together, like using a pipe in Unix. See Webpack's What are loaders?[13] and list of loaders[14].

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

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

app/main.css

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

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

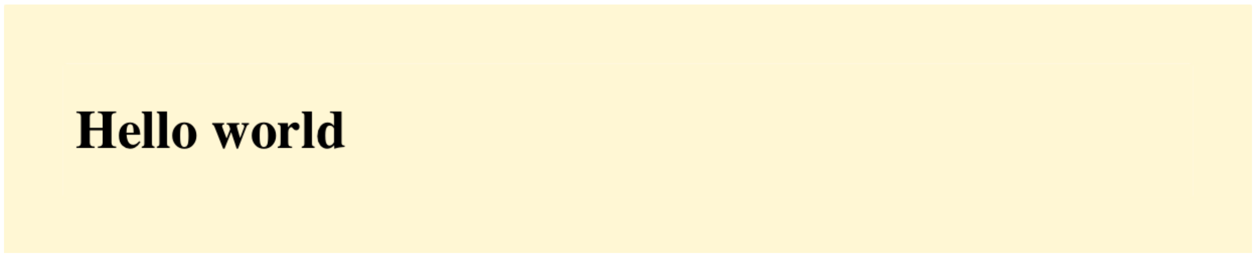
app/index.js

[13] http://webpack.github.io/docs/using-loaders.html
[14] http://webpack.github.io/docs/list-of-loaders.html

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

...
```

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

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



Hello cornsilk world

2.8 Making the Configuration Extensible

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

- Maintain configuration in multiple files and point Webpack to each through `--config` parameter. Share configuration through module imports. You can see this approach in action at webpack/react-starter[15].
- Push configuration to a library which you then consume. Example: HenrikJoreteg/hjs-webpack[16].
- Maintain configuration within a single file and branch there. If we trigger a script through *npm* (i.e., `npm run test`), npm sets this information to an environment variable. We can match against it and return the configuration we want. I prefer this approach as it allows me to understand what's going on easily. We'll be using this approach.

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

[15]https://github.com/webpack/react-starter
[16]https://github.com/HenrikJoreteg/hjs-webpack
[17]https://antwarjs.github.io/

Setting Up Configuration Target for `npm start`

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

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

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

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

webpack.config.js

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

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

var common = {
  entry: PATHS.app,
  module: {
    loaders: [
      {
        test: /\.css$/,
        loaders: ['style', 'css'],
        include: PATHS.app
      }
    ]
  },
  plugins: [
    // important! move HotModuleReplacementPlugin below
    //new webpack.HotModuleReplacementPlugin(),
    new HtmlwebpackPlugin({
      title: 'Kanban app'
    })
  ]
};

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

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

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

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

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

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

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

[18] https://webpack.github.io/docs/configuration.html#devtool

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

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

2.9 Linting the Project

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

2.10 Conclusion

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

[19]https://developer.chrome.com/devtools/docs/javascript-debugging
[20]https://developer.mozilla.org/en-US/docs/Tools/Debugger/How_to/Use_a_source_map

3. Webpack and React

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

Common editors (Sublime Text, Visual Studio Code, vim, emacs, Atom and such) have good support for React. Even IDEs such as WebStorm[1] support it up to an extent. Nuclide[2], an Atom based IDE, has been developed with React in mind.

3.1 What is React?

React

Facebook's React[3] has changed the way we think about front-end development. Also, thanks to React Native[4] the approach isn't limited just to web. Although simple to learn, React provides plenty of power.

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

[1]https://www.jetbrains.com/webstorm/
[2]http://nuclide.io/
[3]https://facebook.github.io/react/
[4]https://facebook.github.io/react-native/

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

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

JSX and Virtual DOM

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

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

```
function render() {
  const names = ['John', 'Jill', 'Jack'];

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

      <ul className="names">{
        names.map((name) =>
          <li className="name">{name}</li>
        )
      }</ul>
    </div>
  );
}
```

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

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

[5]https://github.com/Matt-Esch/virtual-dom
[6]https://facebook.github.io/react/docs/top-level-api.html
[7]https://facebook.github.io/jsx/
[8]https://medium.com/@housecor/react-s-jsx-the-other-side-of-the-coin-2ace7ab62b98

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

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

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

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

Better with Friends

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

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

[9]https://facebook.github.io/react/tips/maximum-number-of-jsx-root-nodes.html
[10]https://facebook.github.io/react-native/
[11]https://github.com/Flipboard/react-canvas
[12]https://github.com/Yomguithereal/react-blessed

3.2 Babel

Babel is a JavaScript compiler.
Use next generation JavaScript, today.

Babel

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

Babel provides support for certain experimental features[14] from ES7 beyond standard ES6. Some of these might make it to the core language while some might be dropped altogether. The language proposals have been categorized within stages:

- **Stage 0** - Strawman
- **Stage 1** - Proposal
- **Stage 2** - Draft - Features starting from *stage 2* have been enabled by default
- **Stage 3** - Candidate
- **Stage 4** - Finished

I would be careful with **stage 0** features. The problem is that if the feature changes or gets removed you will end up with broken code and will need to rewrite it. In smaller experimental projects it may be worth the risk. In our project we'll enable **stage 1**. This allows us to use decorators and property spreading. These features will make our code a little tidier.

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

Configuring `babel-loader`

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

[13]https://babeljs.io/
[14]https://babeljs.io/docs/usage/experimental/
[15]https://babeljs.io/repl/
[16]https://www.npmjs.com/package/babel-loader

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

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

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

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

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

webpack.config.js

```
...

var common = {
  entry: PATHS.app,
  /* add resolve.extensions */
  resolve: {
    extensions: ['', '.js', '.jsx']
  },
  module: {
    loaders: [
      ...
      /* set up jsx */
      {
        test: /\.jsx?$/,
        loaders: ['babel'],
        include: PATHS.app
      }
    ]
  },
  plugins: [
    new HtmlwebpackPlugin({
      title: 'Kanban app'
    })
  ]
};

...
```

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

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

Also, we are going to need a .babelrc[17]. You could pass Babel settings through Webpack (i.e., `babel?stage=1`), but then it would be just for Webpack only. That's why we are going to push our Babel settings to this specific dotfile. The same idea applies for other tools such as ESLint. Set it up as follows to enable support for class properties[18], decorators[19], and object rest spread[20]. We'll be using these features in our project:

.babelrc

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

Alternatively we could have a declaration such as `"stage": 1`. The problem is that this doesn't document well which experimental features we are using at our code base. It might work for small projects. Documenting your Babel usage this way will help in maintenance.

There are other possible .babelrc options[21]. Now we are just keeping it simple. You could, for instance, enable the features you want to use explicitly.

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

3.3 Developing the First React View

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

[17] https://babeljs.io/docs/usage/babelrc/
[18] https://github.com/jeffmo/es-class-static-properties-and-fields
[19] https://github.com/wycats/javascript-decorators
[20] https://github.com/sebmarkbage/ecmascript-rest-spread
[21] https://babeljs.io/docs/usage/babelrc/

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

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

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

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

app/components/App.jsx

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

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

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

It may be worth your while to install React Developer Tools[22] extension to your browser. Currently Chrome and Firefox are supported. This will make it easier to understand what's going on while developing.

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

Setting Up Note

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

app/components/Note.jsx

[22]https://github.com/facebook/react-devtools

```
import React from 'react';

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

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

Rendering Through `index.jsx`

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

app/index.jsx

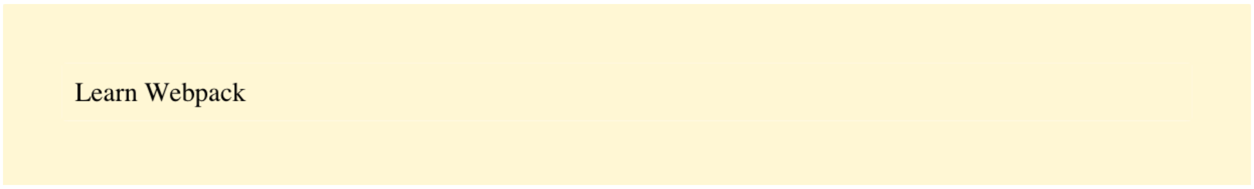
```
import './main.css';

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

main();

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

  document.body.appendChild(app);

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

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

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

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

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


Learn Webpack

Hello React

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

3.4 Activating Hot Loading for Development

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

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

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

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

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

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

webpack.config.js

[23]https://github.com/gaearon/babel-plugin-react-transform
[24]https://github.com/gaearon/react-transform-hmr

```
...

process.env.BABEL_ENV = TARGET;

var common = {
  ...
};

...
```

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

.babelrc

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

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

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

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

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

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

3.5 React Component Styles

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

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

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

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

3.6 Conclusion

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

[25]https://github.com/gaearon/react-transform-catch-errors
[26]https://code.google.com/p/chromium/issues/detail?id=492902
[27]https://github.com/brigand/react-mixin

II Developing Kanban Application

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

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

4. Implementing a Basic Note Application

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

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

4.1 Initial Data Model

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

```
[
  {
    id: '4a068c42-75b2-4ae2-bd0d-284b4abbb8f0',
    task: 'Learn Webpack'
  },
  {
    id: '4e81fc6e-bfb6-419b-93e5-0242fb6f3f6a',
    task: 'Learn React'
  },
  {
    id: '11bbffc8-5891-4b45-b9ea-5c99aadf870f',
    task: 'Do laundry'
  }
];
```

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

4.2 On Ids

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

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

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

at the project root to get it installed.

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

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

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

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

4.3 Connecting Data with `App`

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

app/components/App.jsx

[1]https://www.ietf.org/rfc/rfc4122.txt

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

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

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

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

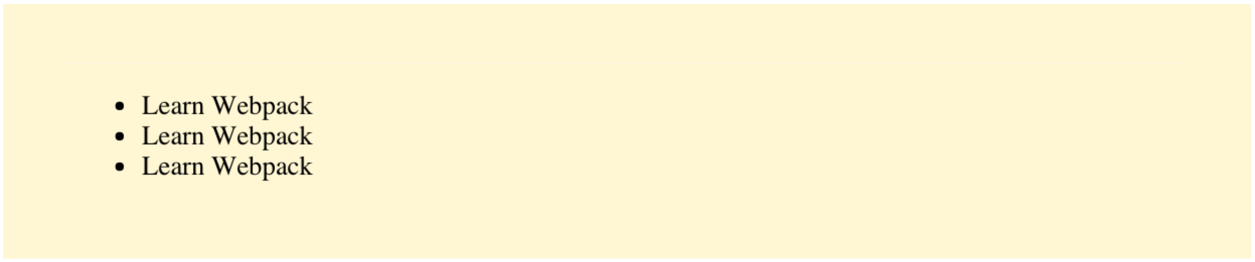
We are using various important features of React in the snippet above. Understanding them is invaluable. I have annotated important parts below:

- `<ul>{notes.map(this.renderNote)}</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}>` - In order to tell React in which order to render the elements, we use the `key` property. It is important that this is unique or else React won't be able to figure

out the correct order in which to render. If not set, React will give a warning. See Multiple Components[3] for more information.

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

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



- Learn Webpack
- Learn Webpack
- Learn Webpack

Almost done

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

4.4 Fixing `Note`

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

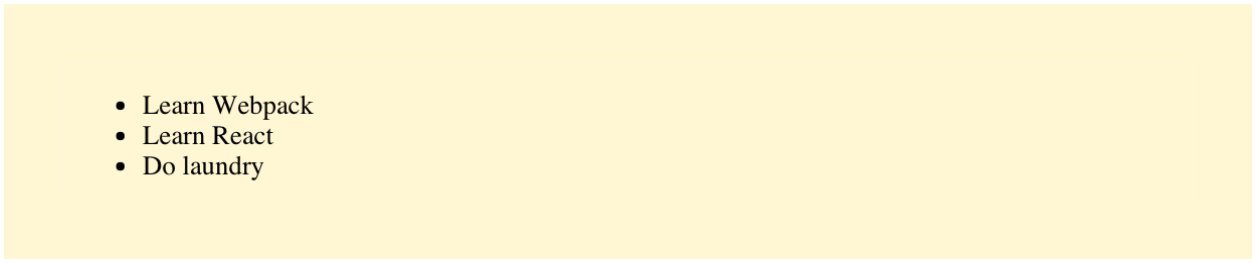
app/components/Note.jsx

[3]https://facebook.github.io/react/docs/multiple-components.html

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

```
import React from 'react';

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

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


- Learn Webpack
- Learn React
- Do laundry

Notes render now

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

4.5 Extracting Notes

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

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

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

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

app/components/Notes.jsx

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

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

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

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

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

app/components/App.jsx

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

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

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

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

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

4.6 Pushing `notes` to the `App` state

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

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

app/components/App.jsx

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

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

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

    ...
  }
}
```

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

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

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

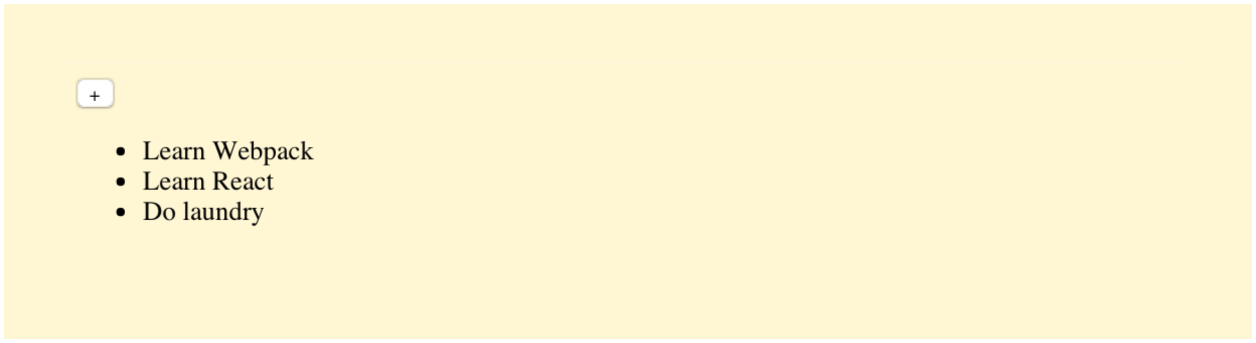
4.7 Adding New Items to `Notes` list

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

app/components/App.jsx

```
...

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

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

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



- Learn Webpack
- Learn React
- Do laundry

Notes with plus

Connecting `addNote` with Data Model

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

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

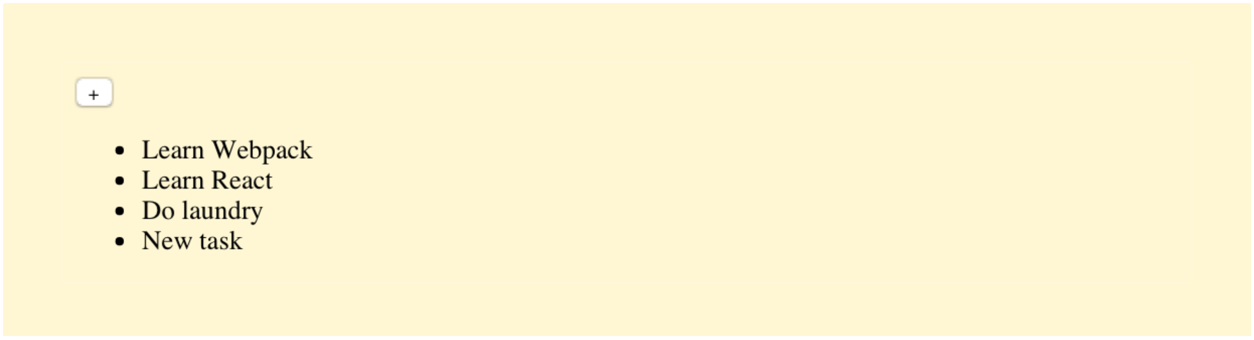
app/components/App.jsx

```
...

export default class App extends React.Component {
  constructor(props) {
    ...
  }
  render() {
    ...
  }
  // We are using an experimental feature known as property
  // initializer here. It allows us to bind the method `this`
  // to point at our *App* instance.
  //
  // Alternatively we could `bind` at `constructor` using
  // a line such as this.addNote = this.addNote.bind(this);
  addNote = () => {
    this.setState({
      notes: this.state.notes.concat([{
        id: uuid.v4(),
        task: 'New task'
      }])
    });
  }
}
```

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

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

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



- Learn Webpack
- Learn React
- Do laundry
- New task

Notes with a new item

4.8 Editing Notes

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

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

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

app/components/Note.jsx

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

```
import React from 'react';

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

    this.state = {
      editing: false
    };
  }
  render() {
    const editing = this.state.editing;

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

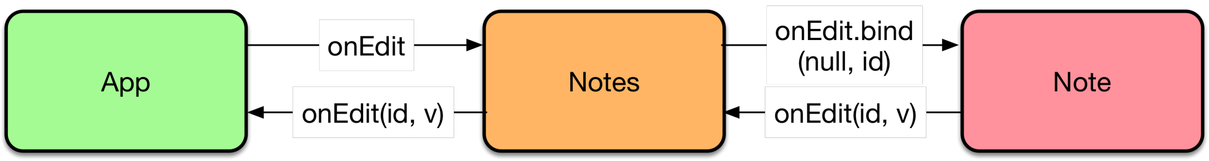
If you try to edit a `Note` now, you will see an error (`this.props.onEdit is not a function`) at the console. We'll fix this shortly.

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

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

Adding `onEdit` Stub

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



`onEdit` flow

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

app/components/App.jsx

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

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

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

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

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

app/components/Notes.jsx

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

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

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

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

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

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

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

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

Understanding `findIndex`

We’ll be using an ES6 function known as findIndex[9]. It accepts an array and a callback. The function will return either `-1` (no match) or the zero-based index number (match) depending on the result.

Babel provides an easy way to polyfill this feature using `import 'babel-core/polyfill';`. The problem is that it bloats our final bundle somewhat as it enables all core-js[10] features. As we need just one shim, we’ll be using a specific shim for this instead. Hit

[6]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind
[7]https://facebook.github.io/react/docs/context.html
[8]https://github.com/casesandberg/react-context/
[9]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex
[10]https://github.com/zloirock/core-js

```
npm i array.prototype.findindex --save
```

You can see how it behaves through Node.js cli. Here's a sample session:

```
> require('array.prototype.findindex')
{}
> a = [12, 412, 30]
[ 12, 412, 30 ]
> a.findIndex(function(v) {return v === 12;})
0
> a.findIndex(function(v) {return v === 121;})
-1
```

es5-shim[11], es6-shim[12] and es7-shim[13] are good alternatives to `core-js`. At the moment they don't allow you to import the exact shims you need in a granular way. That said, using a whole library at once can be worth the convenience.

We also need to attach the polyfill to our application.

app/index.jsx

```
import 'array.prototype.findindex';
...
```

After this you can use `findIndex` against arrays at your code. Note that this will bloat our final bundle a tiny bit (around 4 kB), but the convenience is worth it.

Implementing `onEdit` Logic

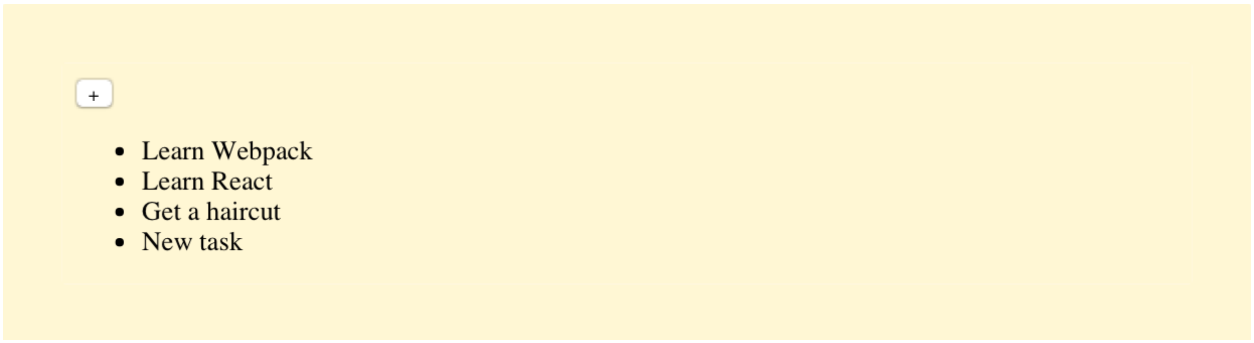
The only thing that remains is gluing this all together. We'll need to take the data and find the specific `Note` by its indexed value. Finally, we need to modify and commit the `Note`'s data to the component state through `setState`.

app/components/App.jsx

[11]https://www.npmjs.com/package/es5-shim
[12]https://www.npmjs.com/package/es6-shim
[13]https://www.npmjs.com/package/es7-shim

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

export default class App extends React.Component {
  ...
  editNote = (id, task) => {
    const notes = this.state.notes;
    const noteIndex = this.findNote(id);

    if(noteIndex < 0) {
      return;
    }

    notes[noteIndex].task = task;

    // shorthand - {notes} is the same as {notes: notes}
    this.setState({notes});
  }
  findNote = (id) => {
    const notes = this.state.notes;
    const noteIndex = notes.findIndex((note) => note.id === id);

    if(noteIndex < 0) {
      console.warn('Failed to find note', notes, id);
    }

    return noteIndex;
  }
}
```

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



- Learn Webpack
- Learn React
- Get a haircut
- New task

Edited a note

4.9 Removing Notes

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

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

app/components/App.jsx

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

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

    return (
      <div>
        <button className="add-note" onClick={this.addNote}>+</button>
        <Notes items={notes}
          onEdit={this.editNote} onDelete={this.deleteNote} />
      </div>
    );
  }
  deleteNote = (id) => {
    const notes = this.state.notes;
    const noteIndex = this.findNote(id);

    if(noteIndex < 0) {
      return;
    }

    this.setState({
      notes: notes.slice(0, noteIndex).concat(notes.slice(noteIndex + 1))
    });
  }
  ...
}
```

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

app/components/Notes.jsx

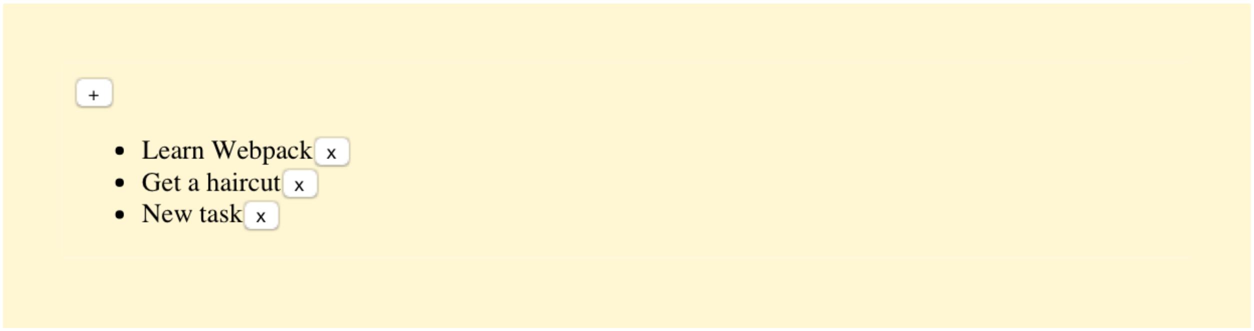
```
export default class Notes extends React.Component {
  render() {
    ...
  }
  renderNote = (note) => {
    return (
      <li className="note" key={note.id}>
        <Note
          task={note.task}
          onEdit={this.props.onEdit.bind(null, note.id)}
          onDelete={this.props.onDelete.bind(null,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

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

app/components/Note.jsx

```
...

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

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

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



Deleted a note

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

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

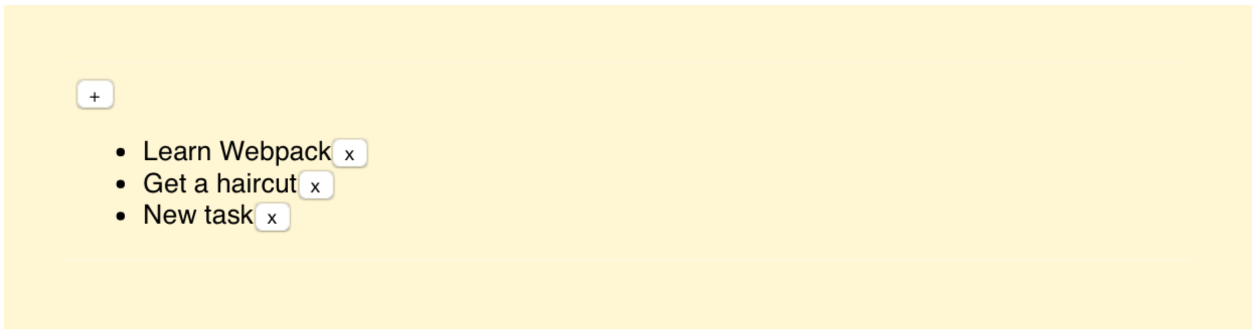
4.10 Styling Notes

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

app/main.css

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

Looking a little nicer now:



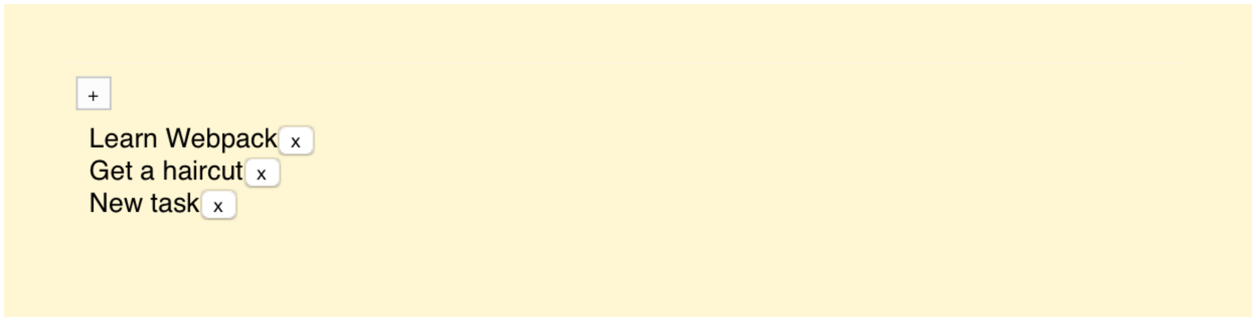
Sans serif

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:



```
+
Learn Webpack x
Get a haircut x
New task x
```

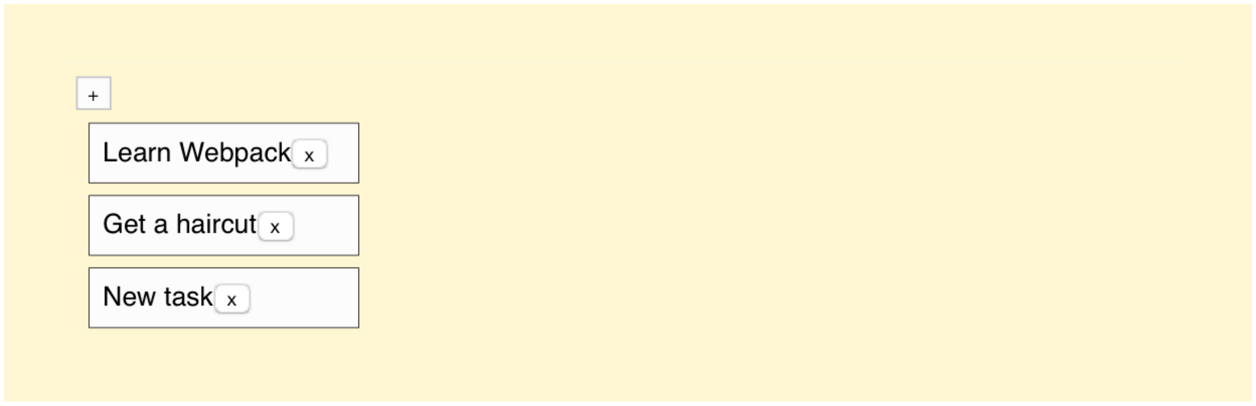
No bullets

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

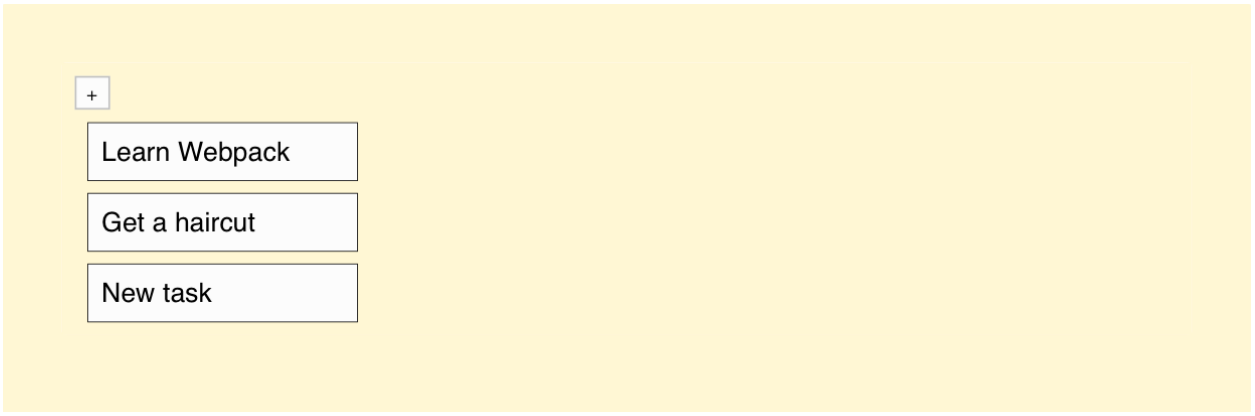
I animated `Note` shadow in the process. This way the user gets a better indication of what `Note` is being hovered upon. This won't work on touch based interfaces, but it's a nice touch for the desktop.

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 {
  float: right;

  padding: 0;

  background-color: #fdfdfd;
  border: none;

  cursor: pointer;

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

No more of those pesky delete buttons:



Delete on hover

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

4.11 Understanding React Components

Understanding how props and state work is important. Component lifecycle is another big concept you'll want to understand well. 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.

To quote the official documentation[14], React provides the following `React.createClass` component specifications:

- `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`.
- `propTypes` - `propTypes` allow you to document your props and catch potential issues during development. To dig deeper, read the *Typing with React* chapter.
- `mixins` - `mixins` contains an array of mixins to apply to components.
- `statics` - `statics` contains static properties and method for a component. In ES6 you would assign them to the class like below:

[14]https://facebook.github.io/react/docs/component-specs.html

```
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 component types support `render()`. As seen above, this is the workhorse of React. It describes what the component should look like. In case you don't want to render anything, return either `null` or `false`.

In addition 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 `should-ComponentUpdate` and before `render()`. It is not possible to use `setState` here, but you can set class properties, for instance.
- `componentDidUpdate` is triggered after rendering. You can modify the DOM here. This can be useful for adapting other code to work with React.
- `componentWillUnmount` is triggered just before a component is unmounted from the DOM. This is the ideal place to perform cleanup (e.g., remove running timers, custom DOM elements, and so on).

4.12 React Component Conventions

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

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

This can be useful in a team environment. It decreases the amount of friction when working on code written by others. Even on personal projects, using tools to verify syntax and standards for you can be useful. It lessens the amount and severity of mistakes.

4.13 Conclusion

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