A book on U&I
UI for you & I

Farhad Ghayour

with React

U&I with React

On building U&I components with React

Farhad Ghayour

This book is for sale at http://leanpub.com/ui-react

This version was published on 2017-08-10

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

I'd like to thank Dr. Kevin Healy, my astronomy professor, who inspired within me a thirst for knowledge that will never be satisfied. I'd like to thank Dr. Bernard Kobes and Dr. John Devlin, my philosophy professors, who provided me tools, rigor and intuition that I've been able to use across all of my passions and interests. I'd like to thank the many wonderful folks at Hack Reactor and Famo.us who opened up the many doors of opportunities. I'd also like to thank my close friends and colleagues who have helped me not only become a better professional but also a better person.

Most importantly, I'd like to thank my loving and supporting family who have always been there for me.

Contents

Profile

Author: Farhad Ghayour

About the Author

Farhad Ghayour is a technology consultant based out of San Francisco, CA, where he helps transform Fortune 500 companies worldwide. Previously, he was a platform engineer at Famo.us building a 3D WebGL rendering engine and a lead software engineer at various innovative startups around the world. He is passionate about all things philosophy, math, code and design — and every so often you can find him in the high horsepower auto scene.

Contact Details

- Email: me@farhadg.com[1]
- Twitter: @farhadg_com[2]
- GitHub: github.com/farhadg[3]
- LinkedIn: linkedin.com/in/farhadg[4]
- Website: farhadg.com[5]

[1] mailto:me@farhadg.com
[2] http://www.twitter.com/farhadg_com
[3] http://www.github.com/farhadg
[4] https://www.linkedin.com/in/farhadg
[5] http://www.farhadg.com

Preface

Frontend development has undergone a major transformation with modern frontend technologies, such as React, Angular, Vue, and so forth. This is largely due to the wide adoption of component-based architecture provided by these wildly successful technologies.

This book aims to not only guide readers from the foundational building blocks of creating well structured interfaces but also provide an exhaustive list of different philosophies for creating modular, extendable and scalable **U&I** — i.e. UI intended for all developers, like **you** and **I**.

Topics

Even though the concepts in this book are taught via React, they are transferable to other frontend technologies. I chose React as our base because of simplicity, wide adoption, and undeniable power that it provides.

Chapter 1: Introduction

Explains a brief history of web development and the rise of component-based architecture.

Chapter 2: What is U&I?

Provides an in depth view of U&I and the criteria for building them.

Chapter 3: Getting Started

Outlines some of the base technologies we'll be using and then walks us through setting up a boilerplate codebase.

Chapter 4: Building our App

Guides us through building the foundation of our application.

Chapter 5: Using U&I Concepts

Guides us on refactoring our application using U&I best practices.

Chapter 6: Exploring CSS Preprocessors

Explains CSS preprocessors and guides us on refactoring our interface using Sass.

Chapter 7: Exploring CSS Modules

Introduces CSS modules and guides us on refactoring our interface using CSS modules.

Chapter 8: Exploring Inline Styles

Takes us on a journey through inline styles and guides us on refactoring our application using this dated philosophy in a modern setting.

Chapter 9: Adding Real Time Capabilities

Introduces an entirely new perspective of looking at interactive interfaces and helps bring our application to life with real time capabilities.

Chapter 10: Showcasing

Guides us on how we can build, document and showcase our components in an isolated environment.

Chapter 11: Looking Ahead

Compares the various methodologies covered throughout this book, with a glimpse into the future of UI development.

Requirements

The following is recommended for maximum productivity:

- Any modern computer with Linux, Mac OS, or Windows.
- Node
- NPM
- Git *(optional)*

All software mentioned in this book are free of charge and can be downloaded from the Internet. You can find the source code hosted on GitHub: https://github.com/FarhadG/ui-react[6].

Audience

This book is ideal for developers who are familiar with React and are looking for a comprehensive guide for building modular, extendable and scalable user interfaces. Even though this book is intended for intermediate to advanced React developers, anyone new to React can easily follow along given the progressive format of this book where each chapter builds on the chapters before.

The ideas covered in this book are not tied to React and are transferable to other frontend technologies.

[6] https://github.com/FarhadG/ui-react

Conventions

In this book, you will a find number of text formats that convey different types of information. Here are some examples for illustration:

A block of code:

```
function greet(name) {
  return 'Hello, ' + name;
}
```

When blocks of code are redundant from previous examples, you will see an ellipsis signifying that they have been omitted to be able to draw your attention to the appropriate blocks of code:

```
function greet(name) {
  ...
}
```

To signify added blocks of code, they will be highlighted in **bold**:

```
function greet(name) {
  ...
}

greet('rock star developer!');
```

To help locate the files we're working with, you will see titles at the top of code blocks describing the path to the file:

src/util/greet.js

```
function greet(name) {
  ...
}

...
```

It is often useful to provide inline comments within code examples, as you can see here:

src/util/greet.js

```
function greet(name) {
  // if name does not exist, it's set to a default value
  name = name || 'World';
  ...
}
```

Any command-line input or output is written as follows:

```
$ npm install --save react
$ npm install --save-dev webpack
$ npm install --save-dev style-loader
```

The `$` signifies a new command-line input. This does not need to be included with your commands as it only serves to distinguish new command-line inputs.

Lastly, any auxiliary information or useful tips can be found in the following format:

Webpack has many powerful and useful loaders listed on their website for reference.

Example Code

You can download the example code for this book by going to the GitHub repository, https://github.com/FarhadG/ui-react[7]. There are several ways to download the source code:

- **GIT**: You can clone the repo and go through the various chapters by navigating the different directories.
- **Zip download**: You can directly download the codebase from GitHub via their zip option.

The contents of this book can be found under the `manuscript` directory and the source code has been broken down by each chapter under the `code` directory. If you'd like to run the source code, you can do so by installing the chapter's dependencies and running the appropriate `npm` scripts.

Feedback

Your feedback is very important to me so please do not hesitate to reach out and let me know what you liked about this book, what are its deficiencies and how can I improve upon them. You can find my contact details in the **profile** section.

Errata

Although I have taken every care to ensure the accuracy of the contents in this book, mistakes do happen. If you spot any mistakes in the book's contents, please contact me or use the GitHub issues[8] page to report the issue. By reporting these mistakes, you will help improve the quality of the book for yourself and other readers in the future. Once your errata are verified, your submission will be accepted and the errata will be resolved. You can find my contact details in the **profile** section.

[7] https://github.com/FarhadG/ui-react
[8] https://github.com/FarhadG/ui-react

Piracy

Although I'm very supportive of open source, piracy of copyrighted material is a major problem. If you come across any illegal copies of this book, please contact me. You can find my contact details in the **profile** section.

Contributions

Given that all of the contents in this book are open source and intentionally provided on GitHub, contributions are welcome and encouraged. You can find all of the necessary information on GitHub[9].

Questions

If you have any questions, concerns or suggestions, please contact me directly. You can find my contact details in the **profile** section.

[9] https://github.com/FarhadG/ui-react

What are we waiting for? Let's get started...

Chapter 1: Introduction

Although we are about to embark on a new journey together, this journey extends many years back. Let's take a quick moment and go over some history covering the evolution of UI development.

A Brief History of Web Development

I remember the days when I started UI development by first designing interfaces in Photoshop and then "sliced" them using HTML, CSS and minimal JavaScript. The process of building a website would entail using HTML for markup, JavaScript for some very simple behavior and CSS for presentation. Even though we continue using the same technologies for delivering similar functionality, the methodologies have changed over time. Let's take a simple view of the various methodologies that have come to fruition through the lens of CSS.

Inline Styles

You can use the style attribute in the relevant tag. The style attribute can contain any CSS property.

index.html

```
<!DOCTYPE html>
<html>
  <body>
    <div style="background-color: black">
      <p style="color: red; font-size: 18px;">
        Hello, World!
      </p>
    </div>
  </body>
</html>
```

Internal CSS

Since inline styles mix content with presentation, many styles would be duplicated across elements. This is where CSS style sheets come to the rescue.

index.html

```
<!DOCTYPE html>
<html>
  <head>

    <!-- internal CSS stylesheet -->
    <style type="text/css">
      div {
        background-color: black;
      }

      p {
        color: red;
        font-size: 18px;
      }
    </style>

  </head>

  <body>
    <div>
      <p>Hello, World!</p>
    </div>
  </body>
</html>
```

External CSS

As websites grow in complexity and start to contain multiple pages, we can extract CSS styleheets into separate files. A common convention is to separate base or global

styles that span across pages into one stylesheet from the individual page style sheets. This has the advantage of being able to keep our code "DRY" so that we can leverage the same styles across multiple pages and easily change the styles of similar elements across our entire website.

styles.css

```
div {
  background-color: black;
}

p {
  color: red;
  font-size: 18px;
}
```

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <!-- references our styles file -->
    <link rel="stylesheet" type="text/css" href="./style.css">
  </head>

  <body>
    <div>
      <p>Hello, World!</p>
    </div>
  </body>
</html>
```

about.html

```
<!DOCTYPE html>
<html>
  <head>
    <!-- references our styles file -->
    <link rel="stylesheet" type="text/css" href="./style.css">
  </head>

  <body>
    <div>
      <p>About the world.</p>
      <p>Some useful information about our amazing website.</p>
    </div>
  </body>
</html>
```

CSS Classifiers

Extracting these styles works well, but we can't scale our application since all applied styles are global to all matching HTML elements. So, we use CSS classes and IDs to be able to target elements as needed.

styles.css

```
.container {
  background-color: black;
}

.message {
  color: red;
  font-size: 18px;
}

.info {
  color: yellow;
```

```
  font-size: 13px;
}
```

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="./style.css">
  </head>

  <body>
    <div class="container">
      <p class="message">Hello, World!</p>
    </div>
  </body>
</html>
```

about.html

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="./style.css">
  </head>

  <body>
    <div class="container">
      <p class="message">About the world.</p>
      <p class="info">Some useful information about our amazing webs\
ite.</p>
    </div>
  </body>
</html>
```

Surprisingly enough, applying and organizing our JavaScript went through a very similar history. As you may have guessed, we:

1. inlined them
2. extracted them to the top of our document using an internal `<script>` tag
3. organized them into their own separate `.js` files, and, ultimately
4. referenced individual elements by their classifiers.

A Necessary Change

The evolution of coding practices has been great and the abstractions made sense for their time, however, the major adoption of single page applications has started to shake our foundational ideas of UI development. Some of the pitfalls with our previous ideas include:

- **Everything is global**. Selectors are matched against everything in the DOM, so we require clever naming strategies that are hard to enforce and easy to break, to combat collisions.
- **The growing complexity is frightening**. Itâ€™s not unusual for developers to acknowledge that they are afraid to modify their own CSS and JavaScript. Given the frightening complexity of their code, they would rather add to the codebase, than risk breaking functionality by refactoring or deleting code.
- **Managing state and user interactions across UI elements is a mess**. Although we have simple JS libraries to help us manage and connect different interface components, interactions between components often make for a messy codebase.

Single page applications have demanded our attention for rethinking new paradigms for writing encapsulated and modular components. That is, discrete, functional and encapsulated UI elements. Those of us who treated CSS and JavaScript as first class citizens are starting to realize the power of UI components as the building blocks of our applications. We now aim at modular components that can be shared across platforms and applications, given the right sorts of constraints and environments.

The Rise of Components

React was one of the first UI libraries that got me thinking about component-based architecture. Component-based design is powerful for both designers and developers. It helps designers in building beautiful modular UI elements that can be designed, tested and reviewed in isolation and helps developers in thinking about the single most important function of each interface element.

React not only promotes many great patterns but it also pushes frontend developers to think about UI in a modular and scalable manner. As we'll come to see in later chapters, React lends itself well to modular components. That said, styling has been and continues to be a major issue when building these modular components. **We will cover many best practices for building components by covering various effective strategies**.

Many of the concepts in this book are covered through the lens of styles, but they are inherently tied to component-based architecture. That is, we plan, organize and build our markup, behavior and styles in alignment with the component spec.

Summary

In this chapter, we covered a quick history of UI development and set the stage for what's to come with a brief introduction to the concepts of modular components.

Prepare yourself for an exciting journey! We're about to enter U&I components and the blueprints for building them in the following chapter.

Chapter 2: What is U&I?

In the previous chapter we covered a brief history of UI development and were introduced to a few key terms that we'll be covering throughout this book. In this chapter, we'll cover what U&I components are and how to derive them in an axiomatic manner.

So, what is U&I?

In short, if we define our *UI components* in terms of modular components that are reusable in a given *application*, then *U&I components* can be understood as UI components that can be used across *applications*.

> U&I components are UI components that are reusable across applications.

One of my major motivations for thinking about U&I components was a talk[10] given by Christopher Chedeau[11], one of Facebook's very own, in November 2014 at NationJS. The talk covers many of the downfalls of CSS in UI development and how the Facebook team has been exploring inline styles to mitigate these issues. Even though the talk only talks about CSS, I've found many of the concepts to be great axioms for defining U&I components. Now that you've got a hint for what U&I components are, let's cover them in detail and how to go about building them.

UI Components

UI Components are modular UI blocks that can be used across an application. Their functionality and reusability differ from application to application, but their overall intent, from my experience, has been scoped to a single organization or application.

[10] https://vimeo.com/116209150

[11] https://twitter.com/Vjeux

For example, an application may have a `Header`, `Footer` and `Slider` component that is used throughout the application, however, the scope, modularity, extendibility and scalability of these components has always been understood in a single context — that is, a single application.

U&I Components

U&I components are UI components that are intended for several contexts. That is, they are components intended not just for a single application but several applications. This may sound simple, but the truth is that building reusable UI components in a single application is challenging enough, so building U&I components for several applications requires even more rigor. A valid concern is whether there is a difference between a U&I library and some of the traditional UI libraries we've seen over the years. There is a subtle difference that will become clearer as we define a sample U&I contract. In short, all U&I libraries are UI libraries but not all UI libraries are U&I libraries.

> All U&I libraries are UI libraries but not all UI libraries are U&I libraries.

Another valid concern is whether we really need to consider building such generalized components. I argue that we should, not because we necessarily need generalized components in all cases but rather that we should build upon maintainable and scalable foundations.

One of the key starting points for building U&I components is a well-defined contract. Contracts are carefully defined criteria for component publishers and consumers to reference. For example, if we define a U&I contract that includes unidirectional styles, then many of the traditional UI libraries would not pass our U&I spec.

Let's start with a sample U&I contract for an application that we'll be building throughout this book. I'd like to make clear that these requirements are neither an exhaustive list nor the criteria for building U&I components. It simply serves as a

list of requirements that we're defining to allow for us to build and consume a U&I library.

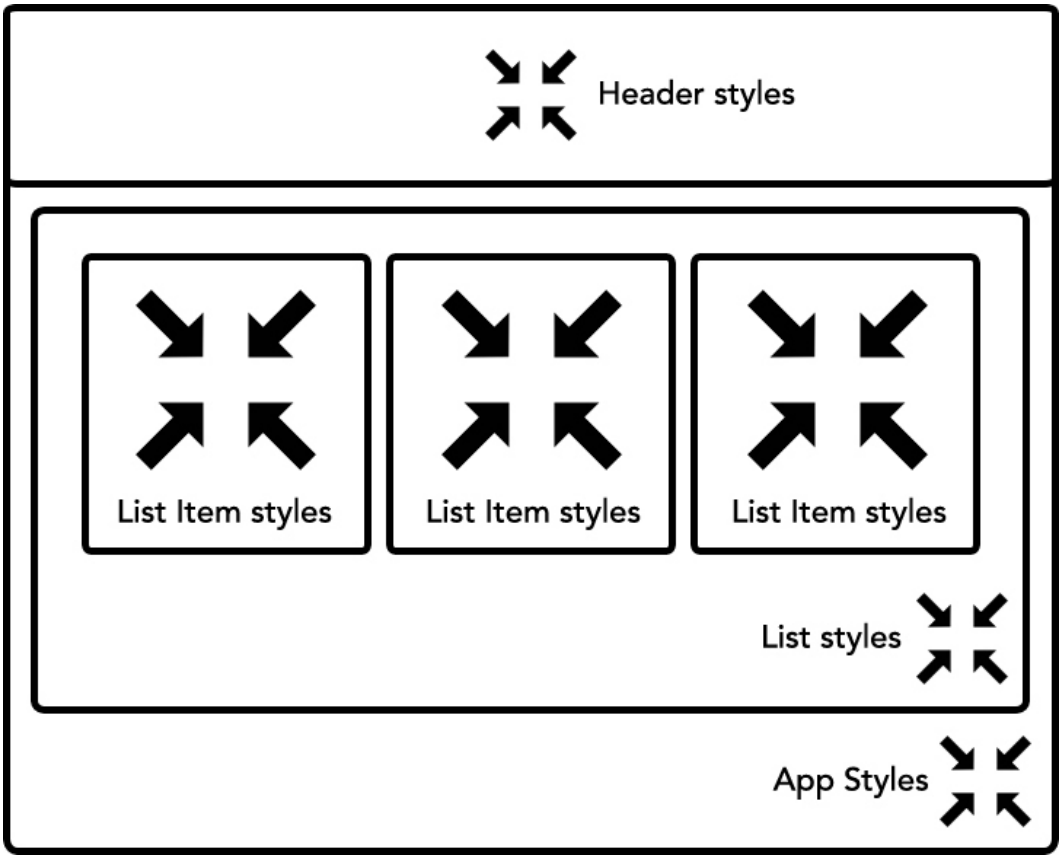
Sample U&I Contract

Here's an exhaustive list of the requirements we'll cover in this book. Some of these requirements will be temporarily ignored or left unchecked only to be fulfilled in later chapters when appropriate.

No global namespace

Since CSS selectors are global, it's necessary to avoid any possibility of naming collision.

Unidirectional styles



Unidirectional styles

Since components can be used across applications, it's necessary for component styles to be unidirectional; that is, child components cannot modify styles outside of their scope.

As a rule of thumb, layout focused CSS attributes like `margin`, `width`, `height`, etc. and CSS selectors that extend beyond the scope of their component should be avoided.

Dead code elimination

Since we want lean and maintainable code, it's necessary to be able programmatically remove unnecessary code.

Minification

Code size is important. Therefore, it's necessary to be able to minify our code, including our styles.

Shareable constants

Modern interfaces are highly interactive. Therefore, it's necessary to be able to share constants between HTML, CSS and JavaScript.

Deterministic resolution

Component behavior must be predictable. So, it's necessary to avoid situations where loading styles asynchronously can result in different results.

Isolation

Since components must be modular, it's necessary for their styles and behavior to be isolated and well encapsulated.

Extendable

Since components can be used in many contexts, it's necessary to be able to extend their styles and behavior.

Documentable

Since components can be used in many contexts, it's necessary for their interface to be documentable.

Presentable

Since components can be used in many contexts, it's necessary for them to be presentable in isolation.

Summary

In this chapter, we explored what U&I components are and how they conceptually differ from their traditional counterparts. We also defined a sample U&I contract that will guide us throughout this book.

It's perfectly fine if some of these concepts are foreign to you, because we're going to continue referencing them in a more practical setting as we start coding in the following chapters.

Chapter 3: Getting Started

In the previous chapter we covered a conceptual overview of U&I components. These U&I concepts aren't useful to us unless we ground our knowledge with practice. This will be the first chapter of a series in implementations covering several strategies for building our application. Due to drastic differences between these U&I strategies, it will be useful to experience first-hand their pros and cons.

In this chapter, we'll build a simple boilerplate to get us up-and-running. Boilerplates are tedious and cumbersome, especially in the JavaScript world; the number of tools and technologies to choose from can be daunting. In addition, React's modular and non-prescriptive eco system doesn't make this any easier. So, instead of dealing with complex build configurations and working up a headache, we'll rely on Facebook's convenient create-react-app[12] command line utility. If you're not familiar with this module, `create-react-app` is a very handy command-line module created by Facebook that encapsulates some of the latest and greatest tools in the React community for a pleasant development experience. For example, we'll have technologies like Webpack for bundling, Babel as our ES6 transpiler, hot module reloading for quicker development, ESLint to keep our code clean and consistent, and many others that are configured out-of-the-box.

If you don't say it, I will: **That's awesome!**

Their documentation[13] is thorough and useful. I highly recommend reading it, if you plan to go beyond the basics covered in this book.

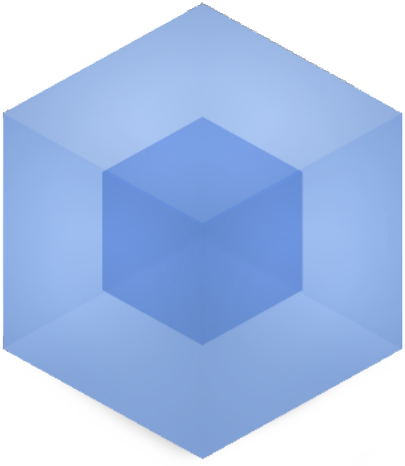
Be sure to have `node` and the other technologies mentioned in the preface. If you have any troubles, you can reference each chapter's source code on GitHub[14].

[12] https://github.com/facebookincubator/create-react-app
[13] https://github.com/facebookincubator/create-react-app
[14] https://github.com/FarhadG/ui-react

Tools

You may already be familiar with some of the tools in `create-react-app`, but we should cover the fundamentals to ensure we have a similar starting point.

Webpack



Webpack

Webpack[15] is a powerful module bundler. A *bundle* is a JavaScript file that incorporates *assets* that *belong* together and should be served to the client in a response to a single file request. A bundle can include JavaScript, CSS styles, HTML, and almost any other kind of file.

Webpack roams over your application source code, looking for `import` statements, building a dependency graph, and emitting one or more *bundles*. With plugins and rules, Webpack can preprocess and minify different non-JavaScript files such as TypeScript, Sass, LESS, and many others.

[15] https://webpack.github.io/

Modern JavaScript with Babel

Babel

Babel[16] is a transpiler for JavaScript, best known for its ability to turn ES6 and beyond (the next versions of JavaScript) into code that runs in your browser (or on your server) today.

JSX

JSX

JSX[17] is a preprocessor step that adds XML syntax to JavaScript. Just like XML, JSX

[16] http://babeljs.io/
[17] https://jsx.github.io/

tags have a tag name, attributes, and children. You can definitely use React without JSX but JSX makes React a lot more elegant.

ESLint

ESLint

ESLint[18] is an open source JavaScript linting utility. Code linting[19] is a type of static analysis that is frequently used to find problematic patterns or code that doesn't adhere to certain style guidelines. With ESLint, developers can write rules to configure a coding standard for a given project.

Others

As mentioned earlier, there are countless tools that `create-react-app` configures for us. If you're interested in learning more, you should explore its repository[20].

[18] http://eslint.org/

[19] http://en.wikipedia.org/wiki/Lint_(software)

[20] https://github.com/facebookincubator/create-react-app

Our Project

Are you ready? Let's get started! But, wait… what are we building? We're going to build a very simply application that displays a list of todos. We're going to keep the application simple, so that we focus on architecture and best practices as a basis for scalable UI.

That said, I recommended taking the building blocks of each chapter to build more complex functionality into the app. This will ensure that the concepts presented in this book are reinforced with creative and discoverable practice.

Boilerplate

Let's globally install `create-react-app` so that we can use it from the command line:

```
$ npm install -g create-react-app
```

Once installed, we can run the following command:

```
$ create-react-app ui-react
```

If everything was successful, our `ui-react` directory should be generated and all of the necessary dependencies was installed. You may also see a list of useful `scripts` after the installation process:

```
yarn start
# Starts the development server.

yarn build
# Bundles the app into static files for production.

yarn test
# Starts the test runner.

yarn eject
# Removes this tool and copies build dependencies,
# configuration files and scripts into the app
# directory. If you do this, you can't go back!
```

For all intents and purposes, we will treat `yarn` and `npm` as being the same. If you're interested in learning more about these package managers, there are a ton of resources that explain their differences. That said, you can execute commands with either `npm run [command]` or its `yarn` equivalent.

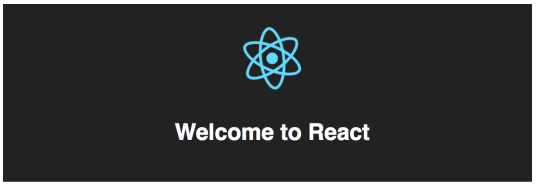
We'll use most of these scripts, except for `test` since there are countless resources online that do a great job explaining how to test your React application.

Let's run the following commands and start our application:

```
$ cd ui-react
$ npm start
```

You should be directed to http://localhost:3000/[21] to see the app running.

[21] http://localhost:3000/



It's quite remarkable that all of the tools ranging from a full test suite, linting, hot reloading, transpiling, etc. have been taken care of for us with a single command. This saves a ton of time, so that we can focus on the important bits of why you're reading this book.

Overall Structure

Go ahead and open up the application in the editor of your choice so that we can start building our application. We'll start off by building the generic version which we'll use as a means to go through the various U&I strategies covered this book.

In our application directory, we'll see the following directories and files:

```
.gitignore
package.json
public/
  favicon.ico
  index.html
README.md
src/
  App.css
  App.js
  App.test.js
  index.css
  index.js
  logo.svg
```

We'll spend most of our time inside of the `src` directory. A great feature that `create-react-app` provides is a *code-focused* seed by hiding the boilerplate code,

configurations and scripts. That said, we'll need to configure `Webpack` as we progress through the chapters, so we're going to `eject` now and have full control from the beginning of our development process.

Opting to `eject` from the beginning of the project is not mandatory, since it's useful to decrease the cognitive load of navigating a larger seed unless needed.

Let's `eject` from the `create-react-app` managed seed:

```
$ npm run eject
```

This command is permanent and irreversible.

After running the command, you should see that a few additional directories and files were injected into the seed project.

```
.gitignore
config/ # bundler configurations
package.json
public/
  favicon.ico
  index.html
README.md
scripts/ # node scripts
src/
  App.css
  App.js
  App.test.js
  index.css
  index.js
  logo.svg
```

A Clean Slate

We're going to start with a clean slate, so go ahead and remove everything from the `src` directory and create a new few file called `index.js`:

src/index.js

```
// dependencies
import React from 'react';
import { render } from 'react-dom';

const App = () => (
  <div>
    <h1>Hello, World!</h1>
  </div>
);

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

At the time of writing this book, React components must return a single React node, which is why we have to always ensure our components are wrapped in a single HTML element. This may change in the near future.

We'll be using some of the latest techniques from the JavaScript and React community, as needed, to build our application. As you can see above, we're using ES6 to deconstruct our dependencies and we're using React's pure component syntax for defining `App`.

If you're not familiar with ES6, I highly recommend these free resources[22] to become an ES6 ninja.

[22] https://github.com/getify/You-Dont-Know-JS

If you visit your browser, you should see our friendly `Hello, World!` message.

Summary

In this chapter, we covered some technologies that will be used in later chapters, along with Facebook's `create-react-app` command line utility to quickly bootstrap our application.

In the next chapter, we'll start building the first version of our application using a more traditional component-based strategy.

Chapter 4: Building our App

In the previous chapter we used `create-react-app` to quickly setup a seed project. In this chapter, we're going to start building the first version of our application.

We're going to start by creating everything inside of `index.js` and later refactor everything into separate components. That said, it's generally easier to build components in an isolated fashion, but given that we're just starting, it'll be easier to see everything in one file so that we can capture the entire journey together.

First pass

We'll first install Lodash[23], "a modern JavaScript utility library delivering modularity, performance & extras" with the following command:

```
$ npm install lodash --save
```

Once installed, we can start with code and cover it in detail afterwards:

src/index.js

```
1  // dependencies
2  import _ from 'lodash';
3  import React, { Component } from 'react';
4  import { render } from 'react-dom';
5
6  class App extends Component {
7
8    constructor(...args) {
9      super(...args);
```

[23] https://lodash.com/

```
10      this.state = {
11        todos: {}
12      };
13    }
14
15    componentDidMount() {
16      this.setState({
17        todos: {
18          1: { id: 1, completed: false, description: 'task 1' },
19          2: { id: 2, completed: false, description: 'task 2' },
20          3: { id: 3, completed: false, description: 'task 3' },
21          4: { id: 4, completed: false, description: 'task 4' }
22        }
23      });
24    }
25
26    render() {
27      const { todos } = this.state;
28      return (
29        <ul>
30          {_.map(todos, (todo, id) =>
31            <li key={id}>
32              {todo.description}
33            </li>
34          )}
35        </ul>
36      );
37    }
38  }
39
40  render(
41    <App />,
42    document.getElementById('root')
43  );
```

We've imported `lodash` to help us with common programming tasks in an elegant and concise fashion. We've converted our `App` component from a React pure component to its `class` interface, since we need internal component state for our todos. In a real world application, you'd use a state manager, such as Redux[24], Flux[25], MobX[26], etc but we won't need one here.

To minimize any external dependencies, we won't hit an API to fetch our data, but we'll treat setting our state as if we were. Inside of `componentDidMount`, you can see that we're updating our state's `todos` with a list of todos. The data structure for a todo item is as follows:

```
1: {
  id: 1,
  completed: false,
  description: 'task 1'
}
```

- `id`: a unique identifier attached to each todo
- `completed`: a boolean determining the todo's status
- `description`: the description for the todo

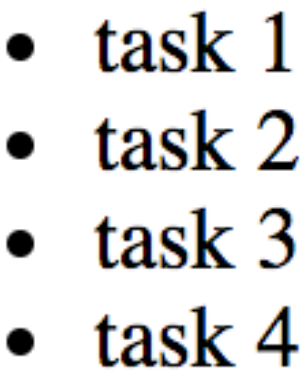
Why store the todos as an `object` rather than inside of an `array`? I opted to have them stored in an `object` with each todo `id` as the key, since it's easier to reference an individual todo and we can do so in constant time.

In our `render` method, we're grabbing and mapping over our collection of todos via Lodash's `_.map` and rendering them as `li` elements inside of an `ul`.

It is good practice to use the `key` prop on elements that we're looping over, so that React can intelligently track them and make internal optimizations.

For the time being, we'll keep these simple without much styling. If you visit your browser, you should see a list of `4` todos rendering in an `ul` tag.

[24] http://redux.js.org/
[25] https://facebook.github.io/flux/
[26] https://mobx.js.org/



Now that we have our todos, how do we toggle their status? Let's manually set one of our todos to complete and then define some styles for a completed todo:

src/index.js

```
...

  componentDidMount() {
    this.setState({
      todos: {
        1: { id: 1, completed: false, description: 'task 1' },
        // manually set to true
        2: { id: 2, completed: true, description: 'task 2' },
        3: { id: 3, completed: false, description: 'task 3' },
        4: { id: 4, completed: false, description: 'task 4' }
      }
    });
  }

...
```

Add a `completed` class to the `li` elements that are complete:

src/index.js

```
...

      <ul>
        {_.map(todos, (todo, id) =>
          <li key={id}
              className={todo.completed ? 'completed' : ''}>
            {todo.description}
          </li>
        )}
      </ul>

...
```

Let's create a `styles.css` file under the `src` directory with a basic set of styles:

src/styles.css

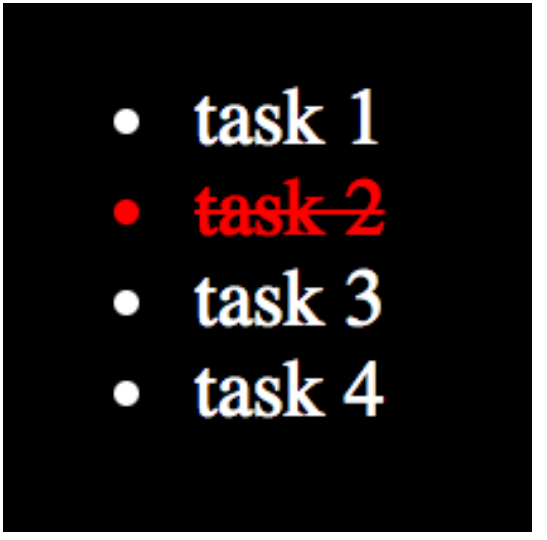
```
1  body {
2    background: black;
3    color: white;
4  }
5
6  .completed {
7    color: red;
8    text-decoration: line-through;
9  }
```

We're going to leverage Webpack and import our css within `index.js`:

src/index.js

```
// dependencies
...

// local dependencies
import './styles.css'

...
```

Visit your browser and you'll see the new styles applied.



Since we hard coded our `todo`'s completed state, how do we give the user the ability to be able to dynamically toggle that state?

src/index.js

```
...

  toggleTodo(id, e) {
    e.preventDefault();
    const todos = _.clone(this.state.todos);
    todos[id].completed = !todos[id].completed;
    this.setState({ todos });
  }

...
```

We add the following method, `toggleTodo`, to our `App` class, giving us the ability to toggle our todo. Instead of mutating the original state of our todos, we're cloning an entirely new collection so that we don't run into situations where mutations cause unpredictable behavior. If this is a foreign topic to you, I highly recommend reading more about immutability[27] within object-oriented and functional programming paradigms.

If you're interested in immutable JavaScript data structures, ImmutableJS[28] is a great open-source candidate.

We need to make our todos trigger our new `toggleTodo` method:

[27] https://en.wikipedia.org/wiki/Immutable_object
[28] https://facebook.github.io/immutable-js/

src/index.js

```
...

  render() {
    const { todos } = this.state;
    return (
      <ul>
        {_.map(todos, (todo, id) =>
          <li key={id}
              className={todo.completed ? 'completed' : ''}
              onClick={(e) => this.toggleTodo(id, e)}>
            {todo.description}
          </li>
        )}
      </ul>
    );
  }

...
```

If you've been following along, your `index.js` should look like this:

src/index.js

```
1  // dependencies
2  import _ from 'lodash';
3  import React, { Component } from 'react';
4  import { render } from 'react-dom';
5
6  // local dependencies
7  import './styles.css'
8
9  class App extends Component {
10
11   constructor(...args) {
```

```
12      super(...args);
13      this.state = {
14        todos: {}
15      };
16    }
17
18    componentDidMount() {
19      this.setState({
20        todos: {
21          1: { id: 1, completed: false, description: 'task 1' },
22          // set to true
23          2: { id: 2, completed: true, description: 'task 2' },
24          3: { id: 3, completed: false, description: 'task 3' },
25          4: { id: 4, completed: false, description: 'task 4' }
26        }
27      });
28    }
29
30    toggleTodo(id, e) {
31      e.preventDefault();
32      const todos = _.clone(this.state.todos);
33      todos[id].completed = !todos[id].completed;
34      this.setState({ todos });
35    }
36
37    render() {
38      const { todos } = this.state;
39      return (
40        <ul>
41          {_.map(todos, (todo, id) =>
42            <li key={id}
43                className={todo.completed ? 'completed' : ''}
44                onClick={(e) => this.toggleTodo(id, e)}>
45              {todo.description}
46            </li>
```

```
47          )}
48        </ul>
49      );
50    }
51  }
52
53  render(
54    <App />,
55    document.getElementById('root')
56  );
```

Viola! Check out your browser and todo away. Mark them todos like there ain't no tomorrow!

Styling our App

I know, I know! Our application looks hideous. I'll leave the creative work of making our application *totally fabulous* up to you.

If you want to share your beautified versions, pull requests to the codebase[29] are more than welcome.

However, we will make our application look somewhat decent with some basic styles so that our eyes don't hate us by the end of this book.

[29] https://github.com/FarhadG/ui-react

src/styles.css

```
html {
  box-sizing: border-box;
}

*, *:before, *:after {
  box-sizing: inherit;
}

body {
  background: #F1F1F1;
  font-size: 10px;
  font-family: "Helvetica Neue", "Arial", "sans-serif";
  margin: 0;
  padding: 0;
}

ul {
  list-style: none;
  margin: 50px auto;
  max-width: 800px;
  padding: 10px 15px;
}

li {
  background: #FAFAFA;
  border-radius: 5px;
  border: 1px solid #E1E1E1;
  box-shadow: 0 2px 1px 0 rgba(0, 0, 0, 0.2);
  color: #888888;
  cursor: pointer;
  font-size: 2rem;
  margin: 10px 0;
  padding: 15px 20px;
  position: relative;
```

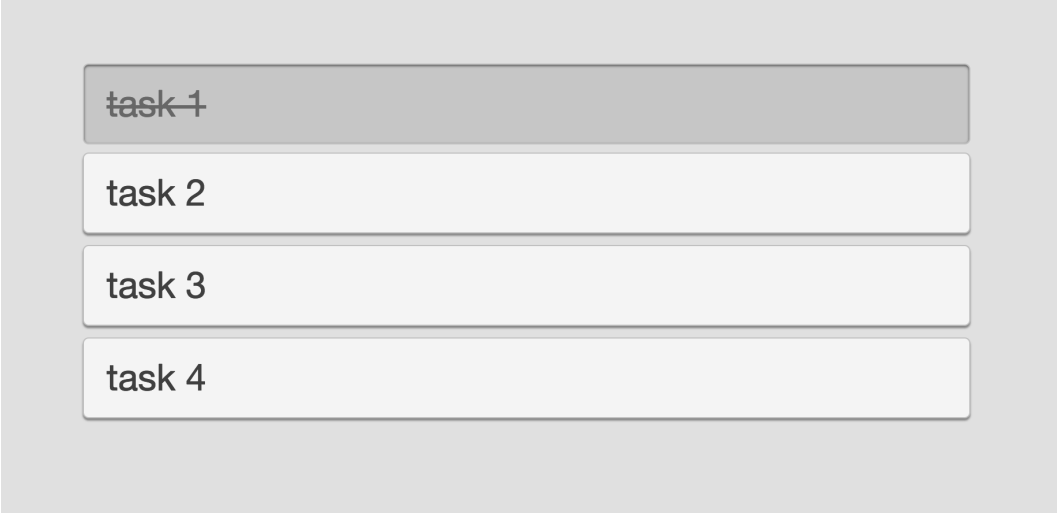
```
35    transition: all 0.2s ease;
36  }
37
38  li:hover {
39    opacity: 0.8;
40  }
41
42  .completed {
43    background: #E4E4E4;
44    box-shadow: inset 0 1px 2px 0 rgba(0, 0, 0, 0.3);
45    color: #AAAAAA;
46    text-decoration: line-through;
47    top: 3px;
48  }
```

These styles are lean enough to provide some elegance and are simple enough to be cross-browser compatible.

Build tools, like Webpack's Autoprefixer[30], helps keep your CSS cross-browser compatible.

If you visit your browser, you should see our newly refined application:

[30] https://github.com/postcss/autoprefixer



Summary

In this chapter, we laid out the foundation of our application. We introduced some basic concepts that are well established in the React ecosystem and left out others that we will cover in later chapters.

You may be wondering when modularity and encapsulation come into play? Don't worry! We'll be getting into that in the following chapter.

Chapter 5: Using U&I Concepts

In the previous chapter we built the first pass of our application using the foundations we laid out using `create-react-app`.

In this chapter, we'll refactor our application into components and cover some U&I concepts in that process.

Organization

We can't build large applications within a single file, so we won't do that either. Create a `components` directory to house the all of our future components:

```
src/
  components/
  index.js
```

Let's create our first component:

```
src/
  components/
    App/
      App.css
      App.js
      App.spec.js
  index.js
```

We'll encapsulate all `App` related code inside of the `App` directory. As you can see, we have a `css`, `js` and `spec.js` file describing our `App` component. It is conventional to co-locate all component related code together, so that we can scale and organize

our code in a component-centric manner. The focus of this book is not testing, so we won't be working with the `spec.js` files, but I want to provide an example of how you'd organize related files together. When organizing your components, keep modularity and encapsulation in mind; you could even go as far as putting the relevant assets such as images, icons, etc. within that directory as well.

Extract `App` from `index.js` and include it within `App.js`:

components/App/App.js

```
// dependencies
import _ from 'lodash';
import React, { Component } from 'react';

// local dependencies
import './App.css';

export default class App extends Component {

  constructor(...args) {
    super(...args);
    this.state = {
      todos: {}
    };
  }

  componentDidMount() {
    this.setState({
      todos: {
        1: { id: 1, completed: false, description: 'task 1' },
        // set to true
        2: { id: 2, completed: true, description: 'task 2' },
        3: { id: 3, completed: false, description: 'task 3' },
        4: { id: 4, completed: false, description: 'task 4' }
      }
    });
  }
```

```
28
29    toggleTodo(id, e) {
30      e.preventDefault();
31      const todos = _.clone(this.state.todos);
32      todos[id].completed = !todos[id].completed;
33      this.setState({ todos });
34    }
35
36    render() {
37      const { todos } = this.state;
38      return (
39        <ul>
40          {_.map(todos, (todo, id) =>
41            <li key={id}
42                className={todo.completed ? 'completed' : ''}
43                onClick={(e) => this.toggleTodo(id, e)}>
44              {todo.description}
45            </li>
46          )}
47        </ul>
48      );
49    }
50  }
```

Note the imported `App.css` which will contains all `App` related styles, as seen below:

components/App/App.css

```
1  ul {
2    list-style: none;
3    margin: 50px auto;
4    max-width: 800px;
5    padding: 10px 15px;
6  }
7
8  li {
```

```
  background: #FAFAFA;
  border-radius: 5px;
  border: 1px solid #E1E1E1;
  box-shadow: 0 2px 1px 0 rgba(0, 0, 0, 0.2);
  color: #888888;
  cursor: pointer;
  font-size: 2rem;
  margin: 10px 0;
  padding: 15px 20px;
  position: relative;
  transition: all 0.2s ease;
}

li:hover {
  opacity: 0.8;
}

.completed {
  background: #E4E4E4;
  box-shadow: inset 0 1px 2px 0 rgba(0, 0, 0, 0.3);
  color: #AAAAAA;
  text-decoration: line-through;
  top: 3px;
}
```

It contains nearly everything from our previous `styles.css` file, without the global styles that are not tied to `App`. Lastly, let's update `index.js` and `styles.css` to reflect these changes:

src/index.js

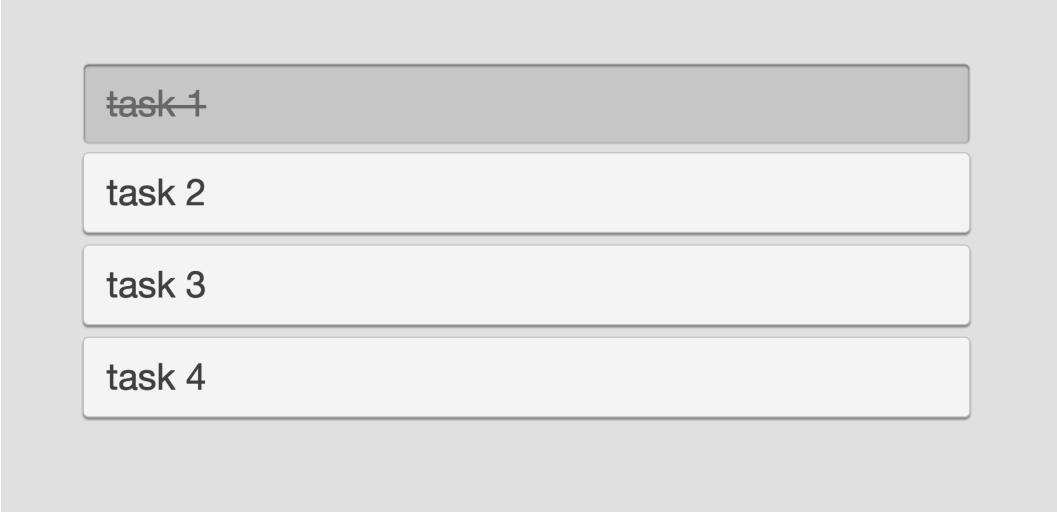
```
// dependencies
import React from 'react';
import { render } from 'react-dom';

// local dependencies
import App from './components/App/App';
import './styles.css';

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

We import our `App` component and render everything just like before. Let's remove the duplicated styles from `styles.css`:

src/styles.css

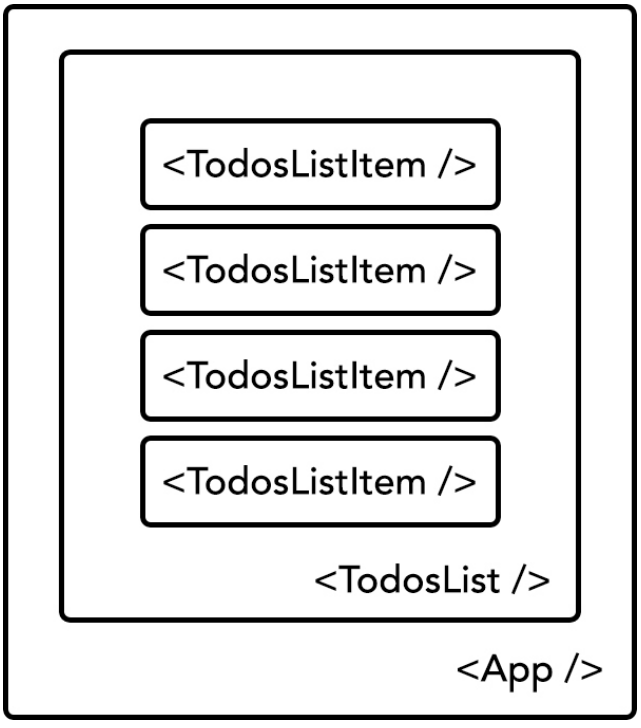
```
html {
  box-sizing: border-box;
}

*, *:before, *:after {
  box-sizing: inherit;
}

body {
  background: #F1F1F1;
  font-size: 10px;
  font-family: "Helvetica Neue", "Arial", "sans-serif";
  margin: 0;
  padding: 0;
}
```

If everything went according to plan, you should see our application render just as it did before.



Necessary Components

That's all nice and dandy, but we haven't done much with this refactor, other than co-locate our `App` related code. Before we move onto the more interesting bits of our refactoring, we need to outline the necessary components needed to build our application. This sort of planning is crucial to any U&I development.



Components Layout

On <App />

Our `App` component will be entry point for our app, instantiating our top-level components. Essentially, we'd like for `App` to do the following:

- `fetch` a set of todos
- `manage` these of todos
- `update` these todos
- `pass` the ability to `update` these todos to child components
- `render` todos

On <TodosList />

Since `App` doesn't deal with the actual todos directly, we'll create a `TodosList` component, which will serve the following purpose(s):

- `accept` a list of todos
- `render` a list of todo items

You can think of `TodosList` as an `ordered` or `unordered` list that simply renders any set of `children` passed down to it. You may be wondering why we need to modularize our `TodosList` in this fashion, but we'll later see the power of separating our actual todo items from the todos list where our `TodosList` can render any sort of items, not just our todo list items.

On <TodosListItem />

Our app instantiates and manages our todos. We have a list that renders a list of children, but we haven't defined the actual todo item. This is where `TodosListItem` comes into the picture. In our previous step, I mentioned why it's a good idea to separate our todo items from the todo list. Even though we won't be building different types of todo items, you can, for example, imagine cases where `TodosList` can render a variety of different todo items, such as `TodosListItem`, `TodosListItemWithIcon`, `TodosListItemThatNeverGetsFinishedCuzItsRequiresWayTooMuchWorkToComplete`, and so forth. These are, of course, examples but it's a good idea to have these separated, instead of having `TodosList` bound to rendering only one type of todo item component.

The point of our `TodosListItem` is quite simple:

- `accept` a todo
- `accept` a callback to be able to `toggle` the todo's status
- `render` the todo item

Others?

Any other necessary components? Nope! That's it! Well… at least for now.

As mentioned earlier, we're going to keep the app minimal so that we can focus on the strategies rather than the slue of other topics that can bleed into this domain.

Building <TodosList />

But first, let's finish up what we started and refactor our components to meet the layout spec we defined above. Let's create a `TodosList` directory along with its associated files.

```
src/
  components/
    App/
    TodosList/
      TodosList.css
      TodosList.js
      TodosList.spec.js
  index.js
```

Then, create `TodosList` as a `ul` that simply renders the `children` passed down via `props`:

src/components/TodosList/TodosList.js

```
1  // dependencies
2  import React from 'react';
3
4  // local dependencies
5  import './TodosList.css';
6
7  export default ({ children }) => (
8    <ul>
9      {children}
10   </ul>
11 );
```

We deconstruct our `props` and pick out `children`. We don't need any sort of lifecycle methods, so we define it as a React's pure component.

We've also imported `TodosList.css`, which extracts out the relevant styles from `App.css`:

src/components/TodosList/TodosList.css

```
ul {
  list-style: none;
  margin: 50px auto;
  max-width: 800px;
  padding: 10px 15px;
}
```

Update `App` to use `TodosList`:

src/components/App/App.js

```
// dependencies
...

// local dependencies
...
import TodosList from '../TodosList/TodosList';

...

  render() {
    const { todos } = this.state;
    return (
      <TodosList>
        {_.map(todos, (todo, id) =>
          <li key={id}
              className={todo.completed ? 'completed' : ''}
              onClick={(e) => this.toggleTodo(id, e)}>
            {todo.description}
          </li>
        )}
      </TodosList>
    );
  }
```

```
24
25  ...
```

We import our `TodosList` and replace our `ul` with `TodoList` which, for the time being, behaves very similar to the previous `ul` html element.

We also remove the styles associated with `TodosList` from `App.css`, so we don't have any sort of duplication or style clashes. Here's the updated `App.css` without the `ul` styles:

src/components/App/App.css

```
1  li {
2    background: #FAFAFA;
3    border-radius: 5px;
4    border: 1px solid #E1E1E1;
5    box-shadow: 0 2px 1px 0 rgba(0, 0, 0, 0.2);
6    color: #888888;
7    cursor: pointer;
8    font-size: 2rem;
9    margin: 10px 0;
10   padding: 15px 20px;
11   position: relative;
12   transition: all 0.2s ease;
13 }
14
15 li:hover {
16   opacity: 0.8;
17 }
18
19 .completed {
20   background: #E4E4E4;
21   box-shadow: inset 0 1px 2px 0 rgba(0, 0, 0, 0.3);
22   color: #AAAAAA;
23   text-decoration: line-through;
24   top: 3px;
25 }
```

Building <TodosListItem />

Let's create a new directory to contain the files associated with `TodosListItem`:

```
src/
  components/
    App/
    TodosList/
    TodosListItem/
      TodosListItem.css
      TodosListItem.js
      TodosListItem.spec.js
  index.js
```

Let's visit `TodosListItem.js`:

src/components/TodosListItem/TodosListItem.js

```
1  // dependencies
2  import React from 'react';
3
4  // local dependencies
5  import './TodosListItem.css';
6
7  export default ({ todo, handleClick }) => (
8    <li className={todo.completed ? 'completed' : ''}
9        onClick={(e) => handleClick(e, todo.id)}>
10     {todo.description}
11   </li>
12 );
```

We import the `css` associated with our component, opt for the React's pure component interface and deconstructed the props.

We removed the `key` attribute because the looping doesn't happen within the component but rather the component's consumer.

Lastly, we bring over the styles relevant to `TodosListItem`, which are, at the moment, all of the styles inside of `App.css`.

src/components/TodosListItem/TodosListItem.css

```
li {
  background: #FAFAFA;
  border-radius: 5px;
  border: 1px solid #E1E1E1;
  box-shadow: 0 2px 1px 0 rgba(0, 0, 0, 0.2);
  color: #888888;
  cursor: pointer;
  font-size: 2rem;
  margin: 10px 0;
  padding: 15px 20px;
  position: relative;
  transition: all 0.2s ease;
}

li:hover {
  opacity: 0.8;
}

.completed {
  background: #E4E4E4;
  box-shadow: inset 0 1px 2px 0 rgba(0, 0, 0, 0.3);
  color: #AAAAAA;
  text-decoration: line-through;
  top: 3px;
}
```

Let's update `App.js` to reflect these changes:

src/components/App/App.js

```
// dependencies
...

// local dependencies
...
import TodosListItem from '../TodosListItem/TodosListItem';

...

  render() {
    const { todos } = this.state;
    return (
      <TodosList>
        {_.map(todos, (todo, id) =>
          <TodosListItem key={id}
                         todo={todo}
                         handleClick={this.toggleTodo.bind(this)} />
        )}
      </TodosList>
    );
  }

...
```

We pass the necessary `props` to `TodosListItem` and `bind` our `handleClick` method to the correct context. It's best practice to bind these callbacks only once. You can do so in either the constructor or somewhere appropriate. We can also make use Babel's ES6+ presets to be able to `auto bind` our methods.

Let's see how that looks:

src/components/App/App.js

```
...

  toggleTodo(id, e) {
    ...
  }

  toggleTodo = (id, e) => {
    ...
  };

...
```

It's a subtle difference, but this syntax provided by Babel auto binds our methods to our class. Lastly, we need to update the prop reference:

src/components/App/App.js

```
...

          <TodosListItem key={todo.id}
                         todo={todo}
                         handleClick={this.toggleTodo} />

...
```

I've found this syntax elegant and preferable to the `::` syntax or binding within the constructor.

Another pattern I've started using are higher-order functions[31] when additional arguments are needed *(such as the index of an item in a loop)*. For example, we can update our method to the following:

[31] http://eloquentjavascript.net/05_higher_order.html

src/components/App/App.js

```
...

  toggleTodo = (id, e) => {
    ...
  };

  toggleTodo = (id) => (e) => {
    ...
  };

...
```

The first time our new `toggleTodo` gets called, we return a new function waiting to be called with `e` as the argument and the todo `id` stored in a closure.

We need to update how we call our new method inside of `TodosListItem`:

src/components/TodosListItem/TodosListItem.js

```
...

export default ({ todo, handleClick }) => (
  <li className={todo.completed ? 'completed' : ''}
      onClick={(e) => handleClick(e, todo.id)}>
    {todo.description}
  </li>
);

export default ({ todo, handleClick }) => (
  <li className={todo.completed ? 'completed' : ''}
      onClick={handleClick(todo.id)}>
    {todo.description}
  </li>
);
```

. . .

Open your browser and you should see everything working.

Considering U&I Concepts

So far, we've covered some best practices covering component encapsulation and composition, by co-locating component related code and having "focused" components.

That's great, but our styles are global and doomed to fail as we scale our application. **This is where the real pain resides**. We'll need to try a new strategy to guide us towards building U&I components.

Name Spacing Components

If you haven't already started thinking about how to name space components, I highly encourage you to do so. There are many useful techniques that help keep CSS styles from clashing but only a few work well with co-located styles.

Application Name Spacing

One powerful technique is name spacing your entire app. That is, every component rendered in your app is a child of the root app, so you can increase CSS specificity by name spacing it with a unique ID. This is great to help avoid situations where your application uses an external UI that may clash with your CSS selectors.

Component Name Spacing

You may already be familiar with naming conventions, such as BEM[32], SUIT[33], etc. which would play well with CEM (Component Element Modifier)[34]. We won't use

[32] http://getbem.com/

[33] https://github.com/suitcss/suit/blob/master/doc/naming-conventions.md

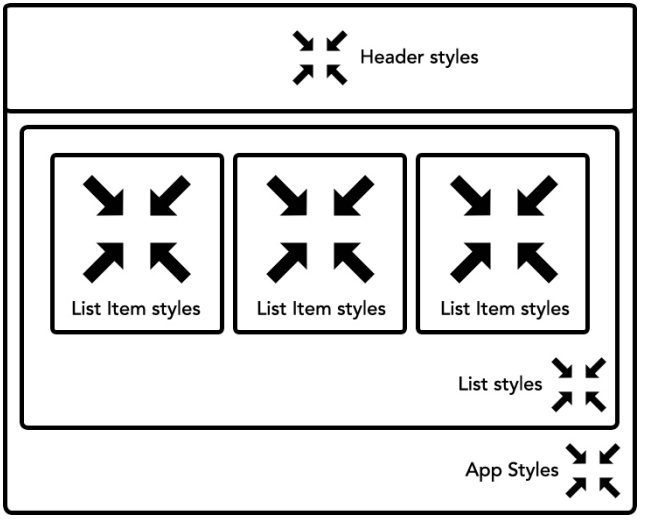
[34] https://atendesigngroup.com/blog/component-element-modifier-design-pattern

BEM here, but you may decide to follow BEM as a naming convention to mitigate CSS name collision. Since BEM is well covered, we'll use a simpler version of CEM, given that our application is quite lean.

It's more semantic to namespace components with CSS classes rather than IDs. Since components, by nature, may be rendered several times within a single application, CSS classes are semantically more appropriate.

Unidirectional styling

If you have any experience with React or any of its popular state managers, you'll know the principle of unidirectional data[35]. We've already seen this in action by not allowing child components, such as our `TodosListItem`, to mutate the state of its parent directly, but rather by having an interface to invoke a callback to perform those operations. That way, our `TodosListItem` is simply a representation of the data. As the data updates, it flow downwards, to the child components, in a unidirectional top-down fashion. Similarly, if we treat our styles as "data," we can employ a similar strategy to have our styles flow "downwards," as seen here:



[35] http://redux.js.org/docs/basics/DataFlow.html

As you can see from the diagram above, we should aim for styles to be "intra-focused" and cascade downwards. In principle, we should avoid any sort of styles that extend beyond their own scopes.

Extendibility

Following a rigorous component design strategy lends itself well for component consumers to be able to extend or modify a component's inner workings. One of the worst situations in UI development is using UI elements that are extremely difficult to modify or are inherently "locked." When building a U&I component, we need to design with empathy, by considering the component's consumers.

In Action

Now, onto the fun part where we get to apply these concepts.

Implementing App Name Spacing

You may have already noticed that we're mounting our entire application to an HTML element with an ID of `root`. Let's leverage what we already have and simply make a semantic change.

public/index.html

```
...

    <div id="root"></div>

    <div id="todos-app"></div>

...
```

We name space the application with `todos-app` where our entire React application is mounted into the DOM.

src/index.js

```
...

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

Update the app's entry point, ensuring we're targeting the correct mounting HTML element. Everything else remains the same. We've simply introduced a semantic change for name spacing our entire application under an identifiable CSS id.

You may be wondering that within our `styles.css`, we're targeting many global HTML elements. That is perfectly fine! Again, once we consider our unidirectional styling criteria, we quickly see that these styles are applied to all elements within our application. These sorts of resets and global styles are common in applications, so it's ideal to keep them sitting at the very top of our application.

Implementing Component Name Spacing

Let's see how other U&I concepts get applied to our components.

Our `index.js` and `styles.css` remain the same. So, let's move onto our first component: `App`:

src/components/App/App.js

```
1  // dependencies
2  ...
3
4  // local dependencies
5  ...
6
7  export default class App extends Component {
8
9    constructor(...args) {
```

```
    ...
  }

  componentDidMount() {
    ...
  }

  toggleTodo = (id) => (e) => {
    ...
  };

  generateTodosListItem = (todo, id) => (
    <TodosListItem key={id}
                   todo={todo}
                   handleClick={this.toggleTodo} />
  );

  render() {
    const { todos } = this.state;
    return (
      <div className="app">
        <TodosList>
          {_.map(todos, this.generateTodosListItem)}
        </TodosList>
      </div>
    );
  }
}
```

Our main `App` component remains mostly the same, except for two things:

- A function for generating our `TodosListItem` in order to keep our `render` method lean.
- Nesting our components within a top level `div` with the appropriate class name `app`.

We create another function for generating our `TodosListItem` to keep our `render` method lean. It can become difficult reading and debugging large `render` methods.

Let's visit our second component: `TodosList`.

src/components/TodosList/TodosList.js

```
// dependencies
import React, { PropTypes } from 'react';

// local dependencies
import './TodosList.css';

const TodosList = ({ children }) => (
  <ul className="todos-list">
    {children}
  </ul>
);

TodosList.propTypes = {
  children: PropTypes.oneOfType([
    PropTypes.array,
    PropTypes.element
  ])
};

export default TodosList;
```

Our `TodosList` remains mostly the same, except that we add a `todos-list` class to namespace our component. In addition, it's always a good idea to perform some `propTypes` validations.

The updated `TodosList` styles:

src/components/TodosList/TodosList.css

```
#todos-app .todos-list {
  list-style: none;
  margin: 50px auto;
  max-width: 800px;
  padding: 10px 15px;
}
```

Yup, quite simple! Instead of applying the styles to `ul`, we've targeted our component by first targeting our application `id` followed by the component `class`.

For our final component, `TodosListItem`, we'll need a new dependency, `classnames`[36], to help handle conditional styles in a more elegant fashion. Go ahead and install this new dependency:

```
$ npm install classnames --save
```

Now we can visit our `TodosListItem` component:

src/components/TodosListItem.js

```
// dependencies
import _ from 'lodash';
import classNames from 'classnames';
import PropTypes from 'prop-types';
import React from 'react';

// local dependencies
import './TodosListItem.css';

const TodosListItem = ({ todo, handleClick }) => (
  <li className={classNames('todos-list-item', {
        completed: todo.completed
```

[36] https://github.com/JedWatson/classnames

```
13        })}
14        onClick={handleClick(todo.id)}>
15      {todo.description}
16    </li>
17  );
18
19  TodosListItem.defaultProps = {
20    todo: {},
21    toggleTodo: _.noop
22  };
23
24  TodosListItem.propTypes = {
25    todo: PropTypes.shape({
26      completed: PropTypes.boolean,
27      description: PropTypes.string,
28      id: PropTypes.number
29    }),
30    handleClick: PropTypes.func
31  };
32
33  export default TodosListItem;
```

A few things were updated:

- We name space our component with a css class.
- Imported our new `classnames` module. This useful module allows for us to define complex classnames in an intuitive manner. For example, in `render`, we apply the `todos-list-item` class regardless, however, if the `todo.completed` state is `true`, we also add the `completed` class. That is, if the todo is completed, it gets `todos-list-item completed`; if not, only `todos-list-item` is returned. You can, obviously, add many more conditionals for more complex classnames. You can read up more on `classnames` in their documentation[37].

[37] https://github.com/JedWatson/classnames

- We set `defaultProps` to ensure way have sane prop values when `TodosListItem` is rendered. The `noop` function is a no-operation function which we'll use, so we don't get an error if no `handleClick` function is provided.
- As before, we validate our props with `propTypes` validations.

Of course, this component would not be complete until we updated its `.css` file.

src/components/TodosListItem/TodosListItem.css

```
#todos-app .todos-list-item {
  background: #FAFAFA;
  border-radius: 5px;
  border: 1px solid #E1E1E1;
  box-shadow: 0 2px 1px 0 rgba(0, 0, 0, 0.2);
  color: #888888;
  cursor: pointer;
  font-size: 2rem;
  margin: 10px 0;
  padding: 15px 20px;
  position: relative;
  transition: all 0.2s ease;
}

#todos-app .todos-list-item:hover {
  opacity: 0.8;
}

#todos-app .todos-list-item.completed {
  background: #E4E4E4;
  box-shadow: inset 0 1px 2px 0 rgba(0, 0, 0, 0.3);
  color: #AAAAAA;
  text-decoration: line-through;
  top: 3px;
}
```

Just like before, we target our elements through the application `id` followed by the component `class`.

Implementing Extendable & Unidirectional Styles

It may appear that we're done, however, we forgot two important concepts: extendable and unidirectional styles. Can you spot which component's styles are extending beyond their scope?

The `margins` set in `TodosList` and `TodosListItem` are extending beyond their own scope. That is, they affect the behavior outside of themselves. It may appear silly to point these out in a small application, but components may conflict if you don't follow the contract you've set out to fulfill in the first place. It may be perfectly reasonable to keep them as is, however, we'll go through the exercise to see how we'd resolve this issue. Since we're still in CSS land, we cascade our styles over our component name spaced interface.

Let's start from the children first:

src/components/TodosListItem/TodosListItem.css

```
#todos-app .todos-list-item {
  background: #FAFAFA;
  border-radius: 5px;
  border: 1px solid #E1E1E1;
  box-shadow: 0 2px 1px 0 rgba(0, 0, 0, 0.2);
  color: #888888;
  cursor: pointer;
  font-size: 2rem;
  margin: 10px 0;
  padding: 15px 20px;
  position: relative;
  transition: all 0.2s ease;
}

#todos-app .todos-list-item:hover {
  opacity: 0.8;
}

#todos-app .todos-list-item.completed {
  background: #E4E4E4;
```

```
21    box-shadow: inset 0 1px 2px 0 rgba(0, 0, 0, 0.3);
22    color: #AAAAAA;
23    text-decoration: line-through;
24    top: 3px;
25  }
```

We remove the `margin` from `TodosListItem.css` and include it in `TodosList`:

src/components/TodosList/TodosList.css

```
1   #todos-app .todos-list {
2     list-style: none;
3     padding: 10px 15px;
4   - margin: 50px auto;
5   - max-width: 800px;
6   }
7
8   #todos-app .todos-list .todos-list-item {
9     margin: 10px 0;
10  }
```

We also remove the `max-width` and `margin` from `TodosList` and include it in the parent component:

src/components/App/App.css

```
1   #todos-app .app {
2
3   }
4
5   #todos-app .app .todos-list {
6     margin: 50px auto;
7     max-width: 800px;
8   }
```

If you visit the browser, the entire application may appear the same, but the truth is that it has been heavily refactored to a U&I focused foundation.

Summary

In this chapter, we outlined some concepts that are a great starting point on the U&I journey. Even though these are great practices for building U&I components, they are not, technically, bulletproof. That is, we can still run into issues with style collisions.

Overall, they provide a solid foundation to tackle some of our U&I criteria. Let's take a quick look at where we stand in relation to the U&I contract we defined in Chapter 2.

	CSS
No global namespace	*
Unidirectional styles	✓
Dead code elimination	
Minification	
Shareable constants	
Deterministic resolution	*
Isolation	*
Extendable	*
Documentable	NA
Presentable	NA

[✓ Fulfilled] [* Pseudo fulfilled]

Some of them are `pseudo` completed, signifying that they don't completely fulfill the criteria but are *close enough* given clear guidelines and best practices. Additionally, we won't cover **Documentable** and **Presentable** until later chapters, so we'll ignore them for the time being.

Even though we weren't able to fulfill all of our U&I criteria and some were only

pseudo fulfilled, this may be more than enough for many U&I requirements. From my experience, following these basic principles is already way ahead of the many applications I've come across that contain high dependency between components with styles extending beyond their scope.

Phew! We're done! Well, sort of… Here's a quick recap of what we covered in this chapter:

- Modularize your components.
- Build composable components when appropriate.
- Name space, intelligently. That is, increase your CSS specificity and bring some structure to your app by name spacing your application and components.
- Keep your `render` methods lean and focused, so it's easy to read and debug.
- Keep your components focused, functional and sustainable.
- Be sure to define and protect your component API with default props and prop type validation.
- Most importantly, think about your component styles unidirectionally. That is, styles should flow downwards, from parent to child. We should try to avoid component styles extending beyond their scope, unless there is absolutely good reason for it.

Let's move onto our next chapter to see how we can further clean up our components.

Chapter 6: Exploring CSS Preprocessors

In the previous chapter we covered some U&I best practices to guide us on refactoring our application. In this chapter, we'll leverage a CSS preprocessor to help us adhere to those principles.

What is a CSS Preprocessor?

A preprocessor is a program transforms one type of data to another. In the case of CSS, we can leverage a preprocessors to extend CSS with variables, operators, interpolations, functions, mixins and many more other usable assets.

Why use a Preprocessor?

CSS is primitive and incomplete. CSS preprocessors offer us many useful and advanced features that help to achieve writing reusable, maintainable and extensible code in CSS. By using a preprocessor, you can easily increase your productivity and decrease the amount of code you are writing.

Meet Sass

Although there are many powerful CSS preprocessors, such as Stylus[38], Less[39], Myth[40], etc., we'll be using Sass[41]. Sass has grown in popularity for a variety of

[38] http://stylus-lang.com/
[39] http://lesscss.org/
[40] http://www.myth.io/
[41] http://sass-lang.com/

reasons — most notably where Bootstrap dropped Less in favor of Sass with their Bootstrap 4 release. That said, feel free to use any CSS preprocessor you feel most comfortable with, given that they offer similar features with very little difference in syntax and style.

Sass in Action

Since we bootstrapped our app using `create-react-app`, we'll need to modify our Webpack to be able to support `.scss` files. This is one of the reasons we executed the `eject` script, so that we could eventually modify `create-react-app`'s config.

Install a Webpack `sass` loader:

```
$ npm install sass-loader node-sass --save-dev
```

We only install these dependencies as `dev-dependencies`, since they're only needed in our build pipeline. Once installed, you can find the webpack configuration files inside `/config`. We're only concerned with `development`, but you can follow a similar set of steps for other environments as well.

config/webpack.config.dev.js

```
...

      {
        ~~test: /\.css$/,~~
        test: /\.(css|scss)$/,
        use: [
          require.resolve('style-loader'),
          {
            loader: require.resolve('css-loader'),
            options: {
              importLoaders: 1,
            },
          },
```

```
        {
          loader: require.resolve('postcss-loader'),
          options: {
            ident: 'postcss',
            plugins: () => [
              require('postcss-flexbugs-fixes'),
              autoprefixer({
                browsers: [
                  '>1%',
                  'last 4 versions',
                  'Firefox ESR',
                  'not ie < 9',
                ],
                flexbox: 'no-2009',
              }),
            ],
          },
        },
        require.resolve('sass-loader')
      ],
    },

...
```

Your Webpack configuration may vary slightly, but essentially, you'll need to find and update your css loader to be able to run the appropriate loaders over both `.css` and `.scss` files.

Go ahead and restart your project by running `npm start` and everything should work just as before. Now, it's time to refactor our styles, component by component, by leveraging Sass.

If this did not work with your version of `create-react-app`, you can try the following:

config/webpack.config.dev.js

```
      {
        exclude: [
          /\.html$/,
          /\.(js|jsx)(\?.*)?$/,
          /\.css$/,
          /\.(css|scss)$/,
          /\.json$/,
          /\.svg$/
        ],
        loader: 'url',
        query: {
          limit: 10000,
          name: 'static/media/[name].[hash:8].[ext]'
        }
      },
```

If you continue to encounter issues, please refer to the book's source code[42] to inspect the versions of the dependencies and the Webpack configuration used.

We won't be using all of the features that Sass provides, but we will cover a few fundamental features. For example, one powerful technique to increase CSS specificity is nesting. Of course, you don't necessarily need to use a preprocessor to do this as you can write out the classes with plain ol' CSS, but that's tedious and error prone.

Configuring our Styles

Create a `styles` directory to house global files (e.g. `theme.scss`, `typography.scss`, etc.) in a production-grade application.

[42] https://github.com/FarhadG/ui-react

src/styles/theme.scss

```
$APP: 'todos-app';

$main-background-color: #F1F1F1;
$main-font-family: "Helvetica Neue", "Arial", "sans-serif";

$light-gray: #FAFAFA;
$dark-gray: #888888;
```

We use a variable to define our app name, along with some color variables.

We move our `styles.css` file into our `styles` directory and rename it as `globals.scss`:

src/styles/globals.scss

```
html {
  box-sizing: border-box;
}

*, *:before, *:after {
  box-sizing: inherit;
}

body {
  background: #F1F1F1;
  font-size: 10px;
  font-family: "Helvetica Neue", "Arial", "sans-serif";
  margin: 0;
  padding: 0;
}
```

Since these will be the entry styles for our entire application, we'll create a new file, `main.scss` which imports all general `.scss` files to bootstrap our application.

src/styles/main.scss

```
@import "theme.scss";
@import "globals.scss";
```

Since we moved and renamed `styles.css`, we'll import `main.scss` to bootstrap our application:

src/index.js

```
// dependencies
...

// local dependencies
...
import './styles.css';

import './styles/main.scss';

...
```

That said, we really haven't used our `theme.scss` files, so let's use that within `globals.scss`.

src/styles/globals.scss

```
...

body {
  background: $main-background-color;
  font-size: 10px;
  font-family: $main-font-family;
  margin: 0;
  padding: 0;
}
```

Since we import `main.scss` file which contains `theme.scss` within its context, we can reference those variables.

Refactoring <App />

Update `App.css` to `App.scss` and update our `import` statement:

src/components/App/App.js

```
// dependencies
...

// local dependencies
...
~~import './App.css';~~

import './App.scss';

...
```

Then, make the following style changes:

src/components/App/App.scss

```
@import '../../styles/theme';

##{$APP} .app {

  .todos-list {
    margin: 50px auto;
    max-width: 800px;
  }

}
```

This is where our app ID becomes useful. We import `theme.scss` and we interpolate our `$APP` variable to name space our component. The syntax may look a bit strange but all that we're doing is setting an ID (i.e. `#`) and interpolating our `$APP` variable with the `#{VARIABLE}` syntax. Now, we can easily skin our entire application in different contexts by switching themes with one variable.

Refactoring <TodosList />

Update `TodosList.css` to `TodosList.scss` and update our import statement:

src/components/TodosList/TodosList.js

```
// dependencies
...

// local dependencies
...
import './TodosList.css';

import './TodosList.scss';

...
```

Then, make the following style changes:

src/components/TodosList/TodosList.scss

```
@import '../../styles/theme';

##{$APP} .todos-list {
  list-style: none;
  padding: 10px 15px;

  &-item {
    margin: 10px 0;
  }
}
```

We use the `&` operator to keep our code dry. The `&` operator inside of a selector concatenates the name of the parent in its place, so we are left with `.todos-list-item`.

Refactoring <TodosListItem />

Update `TodosListItem.css` to `TodosListItem.scss` and update our import statement:

src/components/TodosListItem/TodosListItem.js

```
// dependencies
...

// local dependencies
...
import './TodosListItem.css';

import './TodosListItem.scss';

...
```

Then, make the following style changes:

src/components/TodosListItem/TodosListItem.scss

```
@import '../../styles/theme';

##{$APP} .todos-list-item {
  background: $light-gray;
  border-radius: 5px;
  border: 1px solid #E1E1E1;
  box-shadow: 0 2px 1px 0 rgba(0, 0, 0, 0.2);
  color: $dark-gray;
  cursor: pointer;
  font-size: 2rem;
  padding: 15px 20px;
  position: relative;
  transition: all 0.2s ease;

  &:hover {
```

```
16      opacity: 0.8;
17    }
18
19    &.completed {
20      background: darken($light-gray, 10);
21      box-shadow: inset 0 1px 2px 0 rgba(0, 0, 0, 0.3);
22      color: #AAAAAA;
23      text-decoration: line-through;
24      top: 3px;
25    }
26  }
```

All of our `.todos-list-item` styling are nested and we've made use of the `&` operator to be able to nest additional selectors tied to `.todos-list-item`.

We also use the `darken` function. Instead of dealing with another css color, we can leverage useful Sass operations, such as `darken`, to modify existing colors.

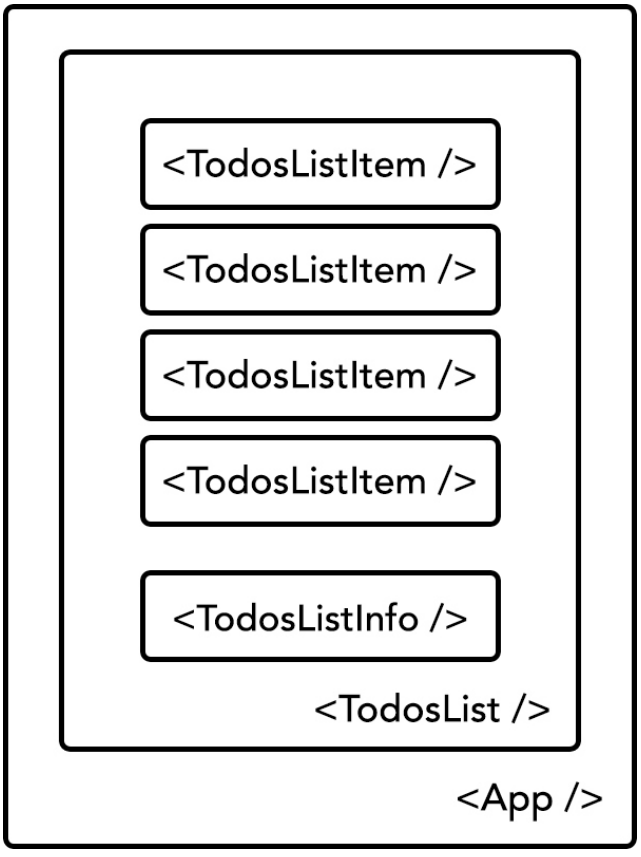
CSS preprocessors, like Sass, have many useful operators that can make working with colors a breeze. Be sure to leverage them in your projects.

Enhancements

Now that we have a good flow for creating components, we're going to introduce one additional component, as a means to show the ease of our existing architecture.

On <TodosListInfo />

We introduce a new component that sits inside of our `TodosList`, providing some basic information about the state of our todos.



Components Layout

Building <TodosListInfo />

Start by creating our component directory and files:

```
src/
  components/
    TodosListInfo/
      TodosListInfo.scss
      TodosListInfo.js
      TodosListInfo.spec.js
```

Next, we'll make a component rendering the number of todos remaining over the total number of todos.

```
// dependencies
import _ from 'lodash';
import React, { PropTypes } from 'react';

// local dependencies
import TodosListItem from '../TodosListItem/TodosListItem';
import './TodosListInfo.scss';

const TodosListInfo = ({ todos }) => {
  const todosCount = _.size(todos);
  const completedTodosCount = _(todos).filter('completed').size();
  return (
    <li className="todos-list-info">
      {completedTodosCount}/{todosCount} completed
    </li>
  );
};

TodosListInfo.defaultProps = {
  todos: []
};

TodosListInfo.propTypes = {
  todos: PropTypes.objectOf(TodosListItem.propTypes.todo)
};
```

```
26
27 export default TodosListInfo;
```

Of course we need some styles:

```
1 @import '../../styles/theme';
2
3 ##{$APP} .todos-list-info {
4   color: $dark-gray;
5   font-size: 14px;
6   text-align: right;
7 }
```

We use a few useful Lodash functions to filter and count the number of todos. We also import `TodosListItem` to reuse the same prop types validation definitions. This may suggest a deep dependency between these two different components, but a single source of truth for these validations is good practice.

If these dependencies become unmaintainable, there are several patterns that can help decouple components from one another:

1. Create an entry file within your `components` directory that exports all components from a single source.
2. Extract all component `propTypes` into a shared resource as a single source of truth.

We'll keep things simple in our project, but it's good to be aware of your application's dependency management as the number of U&I components grow.

Let's use our new component:

```
// dependencies
...

// local dependencies
...
import TodosListInfo from '../TodosListInfo/TodosListInfo';

class App extends Component {

  constructor(...args) {
    ...
  }

  componentDidMount() {
    ...
  }

  toggleTodo = (id) => (e) => {
    ...
  };

  generateTodosListItem = (todo, id) => (
    ...
  );

  render() {
    const { todos } = this.state;
    return (
      <div className="app">
        <TodosList>
          {_.map(todos, this.generateTodosListItem)}
          <TodosListInfo todos={todos} />
        </TodosList>
      </div>
    );
```

```
36    }
37  }
38
39  export default App;
```

We import our `TodosListInfo` component and render it inside of `TodosList` with the appropriate props. By now, the advantages of decoupling components and ensuring a clear separation of roles should be clear.

Open your browser and you should see our new component rendering real-time todo updates.

Summary

In this chapter, we introduced Sass, a CSS preprocessor, and leveraged some of common features to help maintain some of the practices we laid out in earlier chapters. We also added a new component as a means to test the simplicity of adding a new component to our U&I suite.

Where did we end up with our U&I checklist?

	CSS	SCSS
No global namespace	*	*
Unidirectional styles	✓	✓
Dead code elimination		
Minification		
Shareable constants		
Deterministic resolution	*	*
Isolation	*	*
Extendable	*	*
Documentable	NA	NA
Presentable	NA	NA

[✓ Fulfilled] [* Pseudo fulfilled]

We didn't fulfill any additional criteria, but if the previous contract state was satisfactory, adding a CSS preprocessor to your workflow is a great choice for overall quality.

Let's move onto our next chapter where we go beyond naming conventions and clever techniques to avoid styles clashing.

Chapter 7: Exploring CSS Modules

In the previous chapter we introduced Sass, a powerful CSS preprocessor, into our development pipeline. We leveraged a few useful Sass features to help maintain some of our core U&I requirements. Even though we've improved the quality of our U&I components, CSS styles can still clash in more complex applications.

If this is your concern, you're in luck! In this chapter, we will explore CSS modules[43], a new and interesting technology, that offers a better way to mitigate CSS clashing.

What are CSS Modules?

According to the CSS Modules repository[44]:

> CSS files in which all class names and animation names are scoped locally by default. CSS Modules is a step in the build process that changes class names and selectors to be locally scoped.

For example:

[43] https://github.com/css-modules/css-modules
[44] https://github.com/css-modules/css-modules

```
import styles from "./styles.css";

const element = () => (
  <h1 className={styles.title}>
    Hello, world!
  </h1>
);
```

During our build process, the CSS modules loader searches through `styles.css` and makes the `.title` class accessible via `styles.title`. Behind the scenes, however, our template and styles are generated with new characters replacing both the HTML class and the CSS selector class.

An example of what that may look like:

```
<h1 class="_styles__title_3095">
  Hello, world!
</h1>


._styles__title_3095 {
  background-color: red;
}
```

The class attribute and selector `.title` are replaced by this entirely new **unique** string. In short, classes are dynamically generated, unique and mapped to the correct styles.

Why use CSS Modules?

It's a guarantee that all the styles for a single component live in one place and are locally scoped. **This approach is designed to fix the problem of the global scope in CSS**. CSS modules allow for you to name your CSS selectors in any shape or form without needing to worry about name clashes. With CSS Modules, and the concept of **local scope by default**, this problem is avoided.

CSS Modules in Action

Fortunately, with our current Webpack configuration we can enable CSS modules with ease. Go ahead and update the webpack config file to enable CSS modules:

config/webpack.config.dev.js

```
...

      {
        test: /\.(css|scss)$/,
        use: [
          require.resolve('style-loader'),
          {
            loader: require.resolve('css-loader'),
            options: {
              importLoaders: 1,
              modules: true,
              localIdentName: '[name]_[local]_[hash:base64:5]'
            },
          },
          {
            loader: require.resolve('postcss-loader'),
            options: {
              // Necessary for external CSS imports to work
              // https://github.com/facebookincubator/create-react-a\
pp/issues/2677
              ident: 'postcss',
              plugins: () => [
                require('postcss-flexbugs-fixes'),
                autoprefixer({
                  browsers: [
                    '>1%',
                    'last 4 versions',
                    'Firefox ESR',
                    'not ie < 9', // React doesn't support IE8 anyway
```

```
                ],
                flexbox: 'no-2009',
              }),
            ],
          },
        },
        require.resolve('sass-loader')
      ],
    },

...
```

Our `css-loader` already has CSS modules functionality — we just had to enable it. You may be wondering what the `localIdentName`, `[name]`, `[local]` and `[hash:base64:5]` represent. We're simply defining the name of the CSS selector. There are other options that can be found in the `css-loader` documentation. This combination is intuitive for `development` mode as it's easy to reason about and debug. We're generating our css selectors as a combination of their path, component name and a unique 5 digit base64 encoded string tied together with _. In `production`, however, you can omit many of these variables to derive the shortest CSS selector.

After updating the Webpack config, none of the styles will work. That's because `css-loader` is expecting for us to be working with CSS modules.

Refactoring `styles/*.scss`

Since all of our general styles are global by nature, i.e. there is nothing specific to app as a CSS class, we don't need to do anything.

Refactoring <App />

Let's start with our `import` statement:

src/components/App/App.js

```
// dependencies
...

// local dependencies
...
import styles from './App.scss';

class App extends Component {

  constructor(...args) {
    ...
  }

  componentDidMount() {
    ...
  }

  toggleTodo = (id) => (e) => {
    ...
  };

  generateTodosListItem = (todo, id) => (
    <TodosListItem key={id}
                   todo={todo}
                   pStyles={styles}
                   handleClick={this.toggleTodo} />
  );

  render() {
    const { todos } = this.state;
    return (
      <div className={styles.app}>
        <TodosList pStyles={styles}>
          {_.map(todos, this.generateTodosListItem)}
```

```
35            <TodosListInfo todos={todos} />
36          </TodosList>
37        </div>
38      );
39    }
40  }
41
42  export default App;
```

We introduce a new prop, `pStyles` short for propStyles, to be able to pass our styles down.

src/components/App/App.scss

```
1  @import '../../styles/theme';
2
3  .app {}
4
5  .todosList {
6    margin: 50px auto;
7    max-width: 800px;
8  }
9
10 .todosListItem {
11   margin: 10px 0;
12 }
```

Since all of our styles are locally scoped, we can no longer have our `TodosList` cascade its style downwards to its children. We bubbled those styles one level higher to `App`, so that we can pass them down as needed. This is not the most elegant solution, since:

- We can `import` the styles associated with `todosListItem` within our `TodosListItem.scss` and have CSS modules create the local scope for them.
- We can use the `composes` keyword and retrieve those styles.

We won't pursue these options, as they create a deep dependency between components. **This may be perfectly appropriate in your applications**, however, we'll opt in for the `prop` interface for passing the styles down, since we'll be using this as a foundation for future chapters.

The app ID has been removed and css classes have been updated to be in camelCase format, so that we can reference them appropriately within our templates. We could continue to keep our class names in `kebab-case` format and reference them via `styles['todos-list']`, but I prefer the Javascript `camelCase` convention, given that we're writing most of our application in JS.

Refactoring <TodosList />

Let's start by updating how we import and apply our styles:

src/components/TodosList/TodosList.js

```
// dependencies
import classNames from 'classnames';
import React, { PropTypes } from 'react';

// local dependencies
import styles from './TodosList.scss';

const TodosList = ({ children, pStyles }) => (
  <ul className={classNames(
    styles.todosList,
    pStyles.todosList
  )}>
    {children}
  </ul>
);

TodosList.defaultProps = {
  children: [],
  pStyles: {
    todosList: ''
```

```
  }
};

TodosList.propTypes = {
  children: PropTypes.oneOfType([
    PropTypes.array,
    PropTypes.element
  ]),
  pStyles: PropTypes.shape({
    todosList: PropTypes.string
  })
};

export default TodosList;
```

We import our styles with a name, `styles`, so that we can reference its keys. We use `classNames` to add base and prop styles to our element. Lastly, we updated our prop definitions to reflect these changes.

Now our updated styles:

src/components/TodosList/TodosList.scss

```
@import '../../styles/theme';

.todosList {
  list-style: none;
  padding: 10px 15px;
}
```

Refactoring <TodosListInfo />

Rinse and repeat! Let's update our template:

src/components/TodosListInfo/TodosListInfo.js

```
// dependencies
...

// local dependencies
...
import styles from './TodosListInfo.scss';

const TodosListInfo = ({ todos }) => {
  const todosCount = _.size(todos);
  const completedTodosCount = _(todos).filter('completed').size();
  return (
    <li className={styles.todosListInfo}>
      {completedTodosCount}/{todosCount} completed
    </li>
  );
};

...
```

We reference our styles and apply the `todosListInfo` class to our component.

src/components/TodosListInfo/TodosListInfo.scss

```
@import '../../styles/theme';

.todosListInfo {
  color: $dark-gray;
  font-size: 14px;
  text-align: right;
}
```

We could provide a `pStyles` interface, but we'll keep this component locked in its current format. Why? Because with CSS Modules, we can...

Refactoring <TodosListItem />

Rinse and repeat! Let's update our template:

src/components/TodosListItem/TodosListItem.js

```
// dependencies
import _ from 'lodash';
import classNames from 'classnames';
import PropTypes from 'prop-types';
import React from 'react';

// local dependencies
import styles from './TodosListItem.scss';

const TodosListItem = ({ pStyles, todo, handleClick }) => (
  <li
    className={classNames(
      styles.todosListItem,
      pStyles.todosListItem,
      {
        [styles.completedTodosListItem]: todo.completed,
        [pStyles.completedTodosListItem]: todo.completed
      }
    )}
    onClick={handleClick(todo.id)}>
    {todo.description}
  </li>
);

TodosListItem.defaultProps = {
  todo: {},
  toggleTodo: _.noop,
  pStyles: {
    todosListItem: '',
    completedTodosListItem: ''
```

```
  }
};

TodosListItem.propTypes = {
  handleClick: PropTypes.func,
  todo: PropTypes.shape({
    completed: PropTypes.boolean,
    description: PropTypes.string,
    id: PropTypes.number
  }),
  pStyles: {
    todosListItem: PropTypes.string,
    completedTodosListItem: PropTypes.string
  }
};

export default TodosListItem;
```

We name our `styles` and we reference our `todosListItem` as a base style. If the item is `completed`, we apply the `completedTodosListItem` styles. We update our prop type definitions for the various style definitions this component supports.

The ES6 `{ [variable]: value }` syntax allows to interpolate variables in an object's key.

src/components/TodosListItem/TodosListItem.scss

```
@import '../../styles/theme';

.todosListItem {
  background: $light-gray;
  border-radius: 5px;
  border: 1px solid #E1E1E1;
  box-shadow: 0 2px 1px 0 rgba(0, 0, 0, 0.2);
  color: $dark-gray;
  cursor: pointer;
  font-size: 2rem;
  padding: 15px 20px;
  position: relative;
  transition: all 0.2s ease;

  &:hover {
    opacity: 0.8;
  }
}

.completedTodosListItem {
  background: darken($light-gray, 10);
  box-shadow: inset 0 1px 2px 0 rgba(0, 0, 0, 0.3);
  color: #AAAAAA;
  text-decoration: line-through;
  top: 3px;
}
```

We ensure our classnames match and we're good to go.

If you visit the browser, everything should work as expected. All of the necessary styles have been applied and our app works flawlessly, thanks to CSS modules. This may look trivial, but it's quite amazing! We no longer have global selectors.

Suggested Exercise

CSS modules are extremely powerful and I encourage you to explore these suggestions before moving on:

- Inspect the `DOM` and look at the applied CSS selectors. Experiment with the Webpack configuration and explore the results.
- Throw a `debugger` or `console` state within your component and observe the `styles` file. This will assist you in understanding how CSS classes are generated.
- Try the different strategies for passing styles around.
- Experiment with different ways of applying styles to components: flat structure versus the nested structure, functional specific versus component specific names (e.g. `todosListItem` versus `todo`).

Summary

In this chapter, we converted our entire app to use CSS Modules. Let's see how this strategy fares against the others we've explored in previous chapters:

	CSS	SCSS	CSS Modules
No global namespace	*	*	✓
Unidirectional styles	✓	✓	✓
Dead code elimination			✓
Minification			✓
Shareable constants			
Deterministic resolution	*	*	✓
Isolation	*	*	✓
Extendable	*	*	✓
Documentable	NA	NA	NA
Presentable	NA	NA	NA

[✓ Fulfilled] [* Pseudo fulfilled]

That is a major improvement! We not only were able to refactor our entire application with minimal amount of changes, but we were also able to fulfill many of our U&I criteria. We are now able to build predicable components, minify our CSS classes and remove any styles that are not referenced in our application. Amazing!

In the next chapter, we're going to turn everything we've learned on its head by exploring inline styles.

Chapter 8: Exploring Inline Styles

In the previous chapter we introduced CSS Modules and uncovered some clever techniques for mitigating many common UI development issues. We've drastically improved the state of our U&I components, however, we technically haven't fulfilled all of our contract.

If this is a requirement in your application, then inline styles may be the solution. Inline styles? Really? Yes!

In November of 2014, Christopher Chedeau[45], of Facebook, gave a talk entitled React: CSS in JS[46]. He examined seven problems with writing CSS at scale. Using the task of building a simple button as an example, he walked through some fairly complex approaches to solving the first five of them and leaving that last two unsolved. He then dropped a pretty big bomb, declaring that all seven problems could be solved by using inline styles defined in the local react component.

Although he stated in his slides that his goal was *not* to convince developers to drop CSS and use JS instead, that's exactly what began to happen. The talk spurred a flurry of activity.

What are Inline Styles

If you've been following along, you've already been introduced to inline styles. In short, it's the methodology of applying styles directly on the element with JavaScript. Why may this be useful? Here are a few quick reasons that will come to light as we progress through the next several chapters:

- **Cascade-less**: The scary "global" nature of CSS is neutered.

[45] https://twitter.com/Vjeux

[46] https://speakerdeck.com/vjeux/react-css-in-js

- **All JavaScript**: One sense I get is that some people just like and prefer working in all JavaScript.
- **Dynamic Styles**: "State" is largely a JavaScript concern. If you want/need style to change based on dynamic conditions (states) on your site, it may make sense to handle the styling related to the state change along with everything else.

Inline Styles in Action

Unlike before, where we needed to install additional dependencies, we already have inline styles provided to us. All that's needed is to use the `style` prop on the appropriate elements.

Due to the modular nature of our components, it has been effortless to switch between these various technologies. In fact, you can have both CSS modules and inline styles work together across a U&I library.

Configuring our Styles

Let's start with our `styles` directory, since we use these across our application.

src/styles/theme.js

```
export default {
  $APP: 'todos-app',

  $mainBackgroundColor: '#F1F1F1',
  $mainFontFamily: 'Helvetica Neue, Arial, sans-serif',

  $lightGray: '#FAFAFA',
  $darkGray: '#888888'
}
```

This should look familiar; we've simply converted our old `theme.scss` to its JS equivalent, where variables and their values are the `keys` and `values` in a JS object. We continue to adhere to the JS `camelCase` naming convention.

We no longer need `main.scss`, so delete the file and remove its reference inside of `index.js`. For the time being, we're going to ignore `globals.scss` as we'll apply global styles later in the chapter.

Refactoring <App />

Visit the `App` styles:

src/components/App/App.styles.js

```
export default {
  app: {},
  todosList: {
    margin: '50px auto',
    maxWidth: 800
  },
  todosListItem: {
    margin: '10px 0'
  }
}
```

The file type is updated to JS and it's good practice to denote that the file is of a `style` type; hence, the `ModuleName.styles.js`. The styles are converted to their JS equivalents and are then exported for consumption.

src/components/App/App.js

```
// dependencies
...

// local dependencies
...
import styles from './App.styles';

class App extends Component {

  constructor(...args) {
    ...
  }

  componentDidMount() {
    ...
  }

  toggleTodo = (id) => (e) => {
    ...
  };

  generateTodosListItem = (todo, id) => (
    ...
  );

  render() {
    const { todos } = this.state;
    return (
      <div style={styles.app}>
        <TodosList pStyles={styles}>
          {_.map(todos, this.generateTodosListItem)}
          <TodosListInfo todos={todos} />
        </TodosList>
      </div>
```

```
    );
  }
}

export default App;
```

Similar to the CSS modules chapter earlier, we import our `styles` and reference the `key` on the appropriate elements. A major difference here is that we're no longer passing CSS names down as props, but the actual styles, hence the reason we opted for the `prop` interface for passing down our styles.

Refactoring <TodosList />

Let's start with our styles first:

src/components/TodosList/TodosList.styles.js

```
export default {
  todosList: {
    listStyle: 'none',
    padding: '10px 15px'
  }
}
```

Styles are converted to their equivalent JS format.

src/components/TodosList/TodosList.js

```
// dependencies
import _ from 'lodash';
import React from 'react';
import PropTypes from 'prop-types';

// local dependencies
import styles from './TodosList.styles';

const TodosList = ({ children, pStyles }) => (
  <ul style={_.assign({}, styles.todosList, pStyles.todosList)}>
    {children}
  </ul>
);

TodosList.defaultProps = {
  children: [],
  pStyles: {
    todosList: {}
  }
};

TodosList.propTypes = {
  children: PropTypes.oneOfType([
    PropTypes.array,
    PropTypes.element
  ]),
  pStyles: PropTypes.shape({
    todosList: PropTypes.object
  })
};

export default TodosList;
```

Update the dependency, merged the inner and prop styles and, lastly, update the prop

definitions.

You should apply the internal styles first and then `pStyles`, so that custom styles passed down via props take priority in the merge operation.

Refactoring <TodosListInfo />

First, we'll update our styles file:

src/components/TodosListInfo/TodosListInfo.styles.js

```
import theme from '../../styles/theme';

export default {
  todosListInfo: {
    color: theme.$darkGray,
    fontSize: 14,
    textAlign: 'right'
  }
}
```

Styles are converted to their JS equivalent and we reference our `theme`'s variables, as needed.

src/components/TodosListInfo/TodosListInfo.js

```
// dependencies
...

// local dependencies
...
import styles from './TodosListInfo.styles';

const TodosListInfo = ({ todos }) => {
  ...
```

```
  return (
    <li style={styles.todosListInfo}>
      {completedTodosCount}/{todosCount} completed
    </li>
  );
};

...
```

We import `styles` and reference the appropriate keys.

Refactoring <TodosListItems />

Let's start with our styles:

src/components/TodosListItem/TodosListItem.styles.js

```
import theme from '../../styles/theme';

export default {
  todosListItem: {
    background: theme.$lightGray,
    borderRadius: 5,
    border: '1px solid #E1E1E1',
    boxShadow: '0 2px 1px 0 rgba(0, 0, 0, 0.2)',
    color: theme.$darkGray,
    cursor: 'pointer',
    fontSize: '2rem',
    padding: '15px 20px',
    position: 'relative',
    transition: 'all 0.2s ease',

    // inline styles do not support complex CSS pseudo-selectors,
    // key frame animations, media queries, etc.
    // we will later explore how we can get this sort of
```

```
    // functionality with useful open source libraries

    // &:hover {
    //   opacity: 0.8
    // }
  },

  completedTodosListItem: {
    background: theme.$lightGray,
    boxShadow: 'inset 0 1px 2px 0 rgba(0, 0, 0, 0.3)',
    color: '#AAAAAA',
    textDecoration: 'line-through',
    top: 3
  }
}
```

We make the necessary changes to update our styles file, but we've commented out some of the more complex CSS properties as they are not provided with base inline styles. We will take care of making inline styles fully-functional very soon.

src/components/TodosListItem/TodosListItem.js

```
// dependencies
import _ from 'lodash';
import PropTypes from 'prop-types';
import React from 'react';

// local dependencies
import styles from './TodosListItem.styles';

const TodosListItem = ({ pStyles, todo, handleClick }) => (
  <li
    style={_.assign({},
      styles.todosListItem,
      pStyles.todosListItem,
      todo.completed && styles.completedTodosListItem,
```

```
15        todo.completed && pStyles.completedTodosListItem
16      )}
17      onClick={handleClick(todo.id)}>
18      {todo.description}
19    </li>
20  );
21
22  TodosListItem.defaultProps = {
23    todo: {},
24    toggleTodo: _.noop,
25    pStyles: {
26      todosListItem: {},
27      completedTodosListItem: {}
28    }
29  };
30
31  TodosListItem.propTypes = {
32    handleClick: PropTypes.func,
33    todo: PropTypes.shape({
34      completed: PropTypes.boolean,
35      description: PropTypes.string,
36      id: PropTypes.number
37    }),
38    pStyles: PropTypes.shape({
39      todosListItem: PropTypes.object,
40      completedTodosListItem: PropTypes.object
41    })
42  };
43
44  export default TodosListItem;
```

We import our styles and apply them appropriately to the correct elements. We continue to follow the same pattern for giving higher priority to our `pStyles`. Finally, we made sure to update our `defaultProps` and `propTypes` definitions to reflect our changes.

If you visit the app, you will find that the app is nearly on par with where we were in our previous chapters:

There are, however, a few things to consider:

- We no longer have our global CSS styles. We could include a CSS file to apply these different styles, however, we'll opt for a better all-in-JS solution.
- With inline styles, we don't have many of the advanced CSS features, such as pseudo-selectors, media queries, key frame animations, etc. however, there are a few useful libraries that we'll later explore for this functionality.
- We no longer have the `composes` or `extends` functionality, so we merge our two different objects with either the ES6 ... spread operator or `assign/merge`.

Not bad for something provided to us out of the box. Go ahead and inspect the DOM elements. You can see all of the styles applied directly on the HTML elements, just as you'd suspect.

Inline Styles Enhanced

Inline styles are neat, but we can't build anything substantial if we're not able to use some of the more advanced CSS features. This is why we require a better solution

to help us achieve important functionality like global styles, advanced CSS features, etc.

What is Radium?

Meet Radium[47]! A well documented and supported library, made by Formidable Labs[48], to supplement inline styles with all the desirable functionality we need. There are other libraries out there, however, Radium is robust with an active community, so we'll use it to demonstrate inline styles in all of its glory.

Radium in Action

First, we'll need to install two dependencies:

```
$ npm i radium radium-normalize --save
```

We're using `radium-normalize` to help reset browser styles. This is similar to what you'd see in a `reset.css` or a `normalize.css` to achieve common behavior across different browsers.

Once these dependencies are installed, adding Radium to our project is easy. We'll use a few of Radium's most popular features, but I recommend looking through their documentation to familiarize yourself with many of the other provided features.

Configuring our Styles

Update the `globals` styles file.

[47] http://formidable.com/open-source/radium/
[48] https://formidable.com/open-source/

src/styles/globals.styles.js

```
import v from './theme';

export default {
  '*, *:before, *:after': {
    boxSizing: 'inherit'
  },
  html: {
    boxSizing: 'border-box'
  },
  body: {
    background: v.$mainBackgroundColor,
    fontSize: 10,
    fontFamily: v.$mainFontFamily,
    margin: 0,
    padding: 0
  }
}
```

The styles were converted to their JS equivalent. We are ready to apply these global styles.

Refactoring <App />

Since we're applying all of our main styles inside of `App.js`, let's go ahead and start with this file.

src/components/App/App.js

```
// dependencies
...
import { Style, StyleRoot } from 'radium';
import normalize from 'radium-normalize';

// local dependencies
...
import globalStyles from '../../styles/globals.styles';

class App extends Component {

  constructor(...args) {
    ...
  }

  componentDidMount() {
    ...
  }

  toggleTodo = (id) => (e) => {
    ...
  };

  generateTodosListItem = (todo, id) => (
    ...
  );

  render() {
    const { todos } = this.state;
    return (
      <StyleRoot>
        <Style rules={normalize} />
        <Style rules={globalStyles} />
        <div style={styles.app}>
```

```
35            <TodosList pStyles={styles}>
36              {_.map(todos, this.generateTodosListItem)}
37              <TodosListInfo todos={todos} />
38            </TodosList>
39          </div>
40        </StyleRoot>
41      );
42    }
43  }
44
45  ...
```

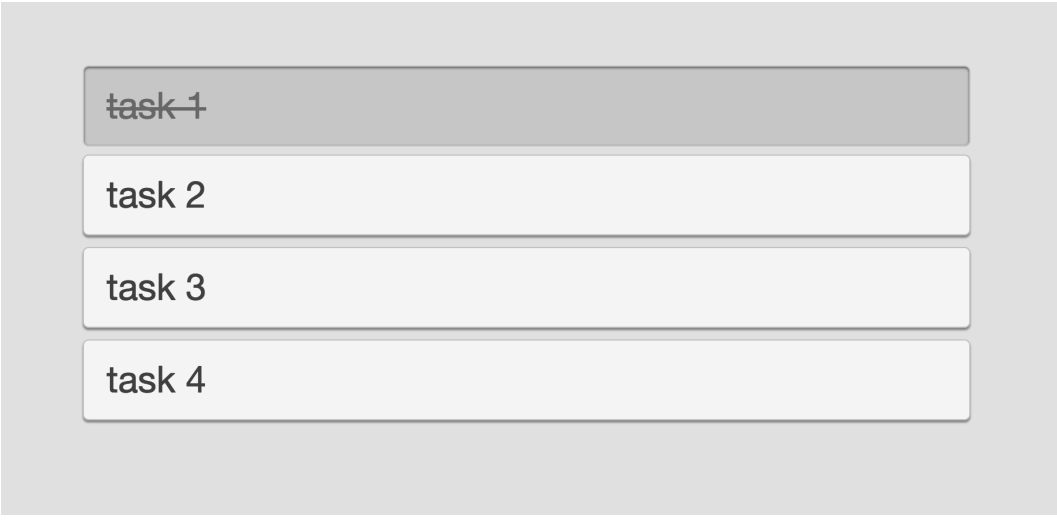
We import `Style` and `StyleRoot` from `radium` and, of course, `radium-normalize`. `Style` allows for us to inject CSS into our app, which we'll use for our `radium-normalize`. The `StyleRoot` module wraps the entire app as a way to keep all of its state in the React `context`. If you're not familiar with React's `context`[49], it's how libraries like `react-redux` are able to pass data from `redux` into your app. Radium uses `context` to maintain styles state.

We wrap our entire app with `<StyleRoot>` and pass `normalize` to the first instance of `<Style>` component and our `gloablStyles` to the second instance. The `<Style>` component is what allows for us to inject CSS into our application and since we're not defining any particular scope in its props, it defaults to being global.

That's all! We now have Radium running with our app and it's already supplementing all of our inline styles.

You can already see that some of the global browser resets have been applied.

[49] https://facebook.github.io/react/docs/context.html



Refactoring <TodosList />

We were not using any advanced CSS features in this component, so nothing to change other than wrap our component with `Radium`.

src/components/TodosList/TodosList.js

```
// dependencies
...

// local dependencies
...

const TodosList = ({ children, pStyles }) => (
  <ul style={[styles.todosList, pStyles.todosList]}>
    {children}
  </ul>
);

...

export default Radium(TodosList);
```

The style `array` syntax is another feature that Radium provides. Instead of `merge/assign`, we can use the `array` syntax and have styles applied in the order they are defined. That's clean!

Refactoring <TodosListInfo />

Very similar as before:

src/components/TodosListInfo/TodosListInfo.js

```
// dependencies
...
import Radium from 'radium';

// local dependencies
...

const TodosListInfo = ({ todos }) => {
  ...
};

...

export default Radium(TodosListInfo);
```

We import Radium and wrapped our component so that it's enhanced with Radium's features.

Refactoring <TodosListItems />

Let's start with our styles:

src/components/TodosListItem/TodosListItem.styles.js

```
import theme from '../../styles/theme';

export default {
  todosListItem: {
    background: theme.$lightGray,
    borderRadius: 5,
    border: '1px solid #E1E1E1',
    boxShadow: '0 2px 1px 0 rgba(0, 0, 0, 0.2)',
    color: theme.$darkGray,
    cursor: 'pointer',
    fontSize: '2rem',
    padding: '15px 20px',
    position: 'relative',
    transition: 'all 0.2s ease',
    ':hover': {
      opacity: 0.8
    }
  },

  completedTodosListItem: {
    background: theme.$lightGray,
    boxShadow: 'inset 0 1px 2px 0 rgba(0, 0, 0, 0.3)',
    color: '#AAAAAA',
    textDecoration: 'line-through',
    top: 3
  }
}
```

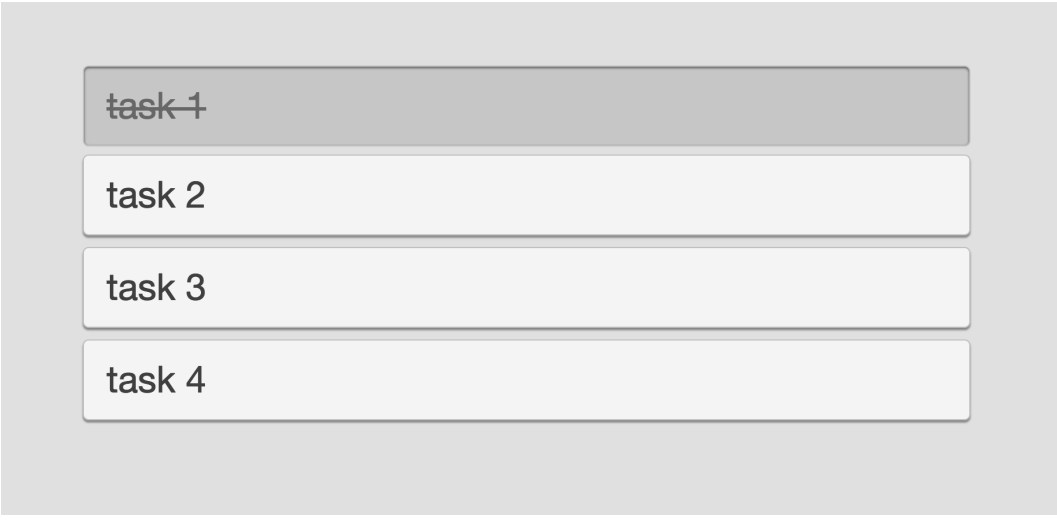
We uncomment our code and enable `:hover` by defining it as an explicit `string` in the styles object.

src/components/TodosListItem/TodosListItem.js

```
// dependencies
...
import Radium from 'radium';

// local dependencies
...

const TodosListItem = ({ pStyles, todo, handleClick }) => (
  <li
    style={[
      styles.todosListItem,
      pStyles.todosListItem,
      todo.completed && styles.completedTodosListItem,
      todo.completed && pStyles.completedTodosListItem
    ]}
    onClick={handleClick(todo.id)}>
    {todo.description}
  </li>
);

...

export default Radium(TodosListItem);
```

We require `Radium` and update our export, just like before, and continue to use the array syntax to merge our styles.

Visit your browser and you'll find that the app has all of the features you'd expect. This all worked with a few additional lines of code. This is just the beginning, I tell ya!



Suggested Exercise

To better understand Radium and inline styles, I recommend experimenting with the following:

- Add other global styles to your app. For example, add a default `background-color` to all `div` elements within the app.
- Experiment with media queries.
- Experiment with CSS key frame animations.
- CSS preprocessors provide us many useful functions like color manipulations. How would you do this with JS? You may want to look into a small npm module called `color`[50].
- … and anything else you'd like, of course.

Summary

In this chapter, we converted our entire application to consume inline styles with a few minor changes. Where do we stand in our U&I checklist?

[50] https://github.com/brehaut/color-js

	CSS	SCSS	CSS Modules	Inline Styles
No global namespace	*	*	✓	✓
Unidirectional styles	✓	✓	✓	✓
Dead code elimination			✓	✓
Minification			✓	✓
Shareable constants				✓
Deterministic resolution	*	*	✓	✓
Isolation	*	*	✓	✓
Extendable	*	*	✓	✓
Documentable	NA	NA	NA	NA
Presentable	NA	NA	NA	NA

[✓ Fulfilled] [* Pseudo fulfilled]

Whoora! We have finally fulfilled all of our U&I specs outlined in Chapter 2, excluding a few that we intentionally are leaving until later chapters. With inline styles, we no longer have to worry about name clashes and dependency management. We continue to provide an intuitive API for custom styles and are able to share code between our styles and templates.

In the next chapter, we're going to see how we can leverage inline styles to achieve real-time capabilities. What does that mean? You'll soon find out…

Chapter 9: Adding Real Time Capabilities

In the previous chapter we uncovered how we can use an old paradigm, such as inline styles, in modern applications to fulfill many of the U&I criteria outlined in Chapter 2.

So, what are we going to do in this chapter?

In this chapter, we're going to take inline styles to an entirely new level. Say you wanted to build an app that you could swap variables, themes and styles on the fly. How would you do that with CSS, CSS preprocessors, CSS Modules, etc.? You need to rely on some clever build tools to help you achieve pseudo real-time capabilities. However, since we've doubled-downed on JS-everything, we can leverage many of its powerful qualities to add real-time behaviors.

What is Theme Wrap?

In this chapter, we introduce `theme-wrap`[51], an open-source library that I built to help manage inline styles, provide theme management, offer style mixing support, and, ultimately, real-time capabilities.

You can think of `theme-wrap` like a gift wrap for your React apps. Clever, I know! :)

Theme Wrap in Action

Start by installing the library:

[51] https://github.com/FarhadG/theme-wrap

```
$ npm i theme-wrap --save
```

Once installed, we can start exploring all of its different functionality.

Refactoring <App />

Let's start with the root of our components:

src/components/App/App.js

```
// dependencies
...
import { ThemeWrapProvider } from 'theme-wrap';

// local dependencies
...
import theme from '../../styles/theme';

class App extends Component {

  constructor(...args) {
    ...
  }

  componentDidMount() {
    ...
  }

  toggleTodo = (id) => (e) => {
    .
  };

  generateTodosListItem = (todo, id) => (
    ...
  );

```

```
  render() {
    const { todos } = this.state;
    return (
      <StyleRoot>
        <Style rules={normalize} />
        <Style rules={globalStyles} />
        <ThemeWrapProvider theme={theme}>
          ...
        </ThemeWrapProvider>
      </StyleRoot>
    );
  }
}

...
```

We wrap our application with `ThemeWrapProvider` and pass our `theme` via the `theme` prop. We now have `theme` in React's `context`.

Refactoring <TodosList />

Let's update styles:

src/components/TodosList/TodosList.styles.js

```
export default () => ({
  todosList: {
    listStyle: 'none',
    padding: '10px 15px'
  }
});
```

We export a `function` where `theme` is provided as the first argument in our styles function *(not used here)*. Note the implicit ES6 return statement where we're returning an object with a single key, `todosList`.

src/components/TodosList/TodosList.js

```
// dependencies
...
import { applyThemeWrap } from 'theme-wrap';

// local dependencies
...

export const TodosList = ({ children, pStyles, twStyles }) => (
  <ul style={[twStyles.todosList, pStyles.todosList]}>
    {children}
  </ul>
);

...

export default _.flow(
  Radium,
  applyThemeWrap(styles)
)(TodosList);
```

We import `applyThemeWrap` from `theme-wrap` to *enhance* our component by applying our styles with the provided `theme`. We'll also need a function composition utility to help us wrap our component with several higher order functions. If you've used `Redux`, then you've most likely used `compose`, allowing components to be wrapped with several higher order functions. If you're not familiar with this concept, I'd highly recommend looking into the many articles explaining function currying, higher order functions, etc. It's a worthy pursuit to understand its benefits and how to use it, in addition to it being quite prevalent in React applications. That said, `flow` is lodash's utility with an added benefit that it applies the function arguments in the order as they're read, as opposed to `compose` being right to left.

We pass `styles` to `applyThemeWrap` so that we have `theme` provided to our `styles` for consumption. Secondly, we update the references to our styles to `twStyles` *(short for theme wrapped styles)* passed down via `props` by `applyThemeWrap`.

Lastly, we `export` our base component without all of the enhancements, in cases where we want to test or reference its prop definitions.

Let's do a quick recap of the necessary changes:

- Our styles `styles` become a `function` with `theme` as its first argument (if needed).
- Wrap components with `applyThemeWrap` and pass the component's `styles` as the first argument.
- Reference styles from `twStyles` passed down via `props`.
- Export the base component.

Refactoring <TodosListInfo />

Let's start with the styles:

src/components/TodosListInfo/TodosListInfo.styles.js

```
export default (theme) => ({
  todosListInfo: {
    color: theme.$darkGray,
    fontSize: 14,
    textAlign: 'right'
  }
});
```

Our styles become a function with `theme` provided as the first argument. We no longer need to access our `theme` directly.

src/components/TodosListInfo/TodosListInfo.js

```
// dependencies
...
import { applyThemeWrap } from 'theme-wrap';

// local dependencies
...
import { TodosListItem } from '../TodosListItem/TodosListItem';

export const TodosListInfo = ({ todos, twStyles }) => {
  ...
  return (
    <li style={twStyles.todosListInfo}>
      {completedTodosCount}/{todosCount} completed
    </li>
  );
};

...

export default _.flow(
  Radium,
  applyThemeWrap(styles)
)(TodosListInfo);
```

Wrap the component with `applyThemeWrap` and pass in our `styles`. Then, reference styles via `twStyles` passed down via `props`. Don't forget to `export` the base component and update how to import `TodosListItem` for prop reference.

Refactoring <TodosListItem />

As before, it's easier to start with our `styles`:

src/components/TodosListItem/TodosListItem.styles.js

```
export default (theme) => ({
  todosListItem: {
    background: theme.$lightGray,
    borderRadius: 5,
    border: '1px solid #E1E1E1',
    boxShadow: '0 2px 1px 0 rgba(0, 0, 0, 0.2)',
    color: theme.$darkGray,
    cursor: 'pointer',
    fontSize: '2rem',
    padding: '15px 20px',
    position: 'relative',
    transition: 'all 0.2s ease',
    ':hover': {
      opacity: 0.8
    }
  },

  completedTodosListItem: {
    background: theme.$lightGray,
    boxShadow: 'inset 0 1px 2px 0 rgba(0, 0, 0, 0.3)',
    color: '#AAAAAA',
    textDecoration: 'line-through',
    top: 3
  }
});
```

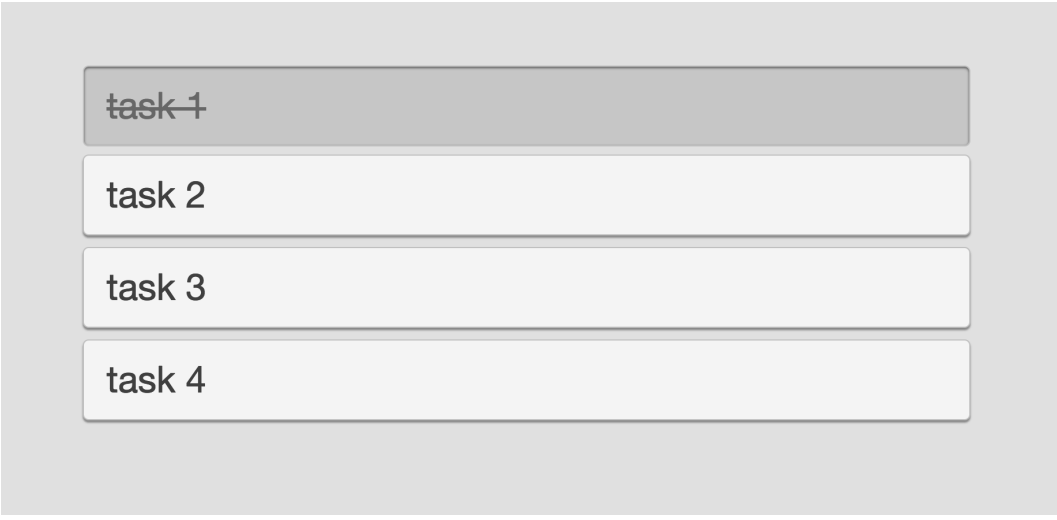
Convert styles into a function and reference `theme`, as needed.

src/components/TodosListItem/TodosListItem.js

```
// dependencies
...
import { applyThemeWrap } from 'theme-wrap';

// local dependencies
...

export const TodosListItem = ({
  pStyles, todo, twStyles, handleClick
}) => (
  <li
    style={[
      twStyles.todosListItem,
      pStyles.todosListItem,
      todo.completed && twStyles.completedTodosListItem,
      todo.completed && pStyles.completedTodosListItem
    ]}
    onClick={handleClick(todo.id)}>
    {todo.description}
  </li>
);

...

export default _.flow(
  Radium,
  applyThemeWrap(styles)
)(TodosListItem);
```

Did you expect anything different? I hope not, because we went through the same steps as mentioned before.

Go ahead and visit your browser and ensure everything works:



Why did we make all of these changes? Here are some of the benefits to consider, which we'll leverage shortly:

- `ThemeWrapProvider` now manages our `theme`. That is, we provide our `theme` once and all of our components get access to this single source of truth.
- `ThemeWrapProvider` manages our `theme` with real-time capabilities. That is, if we update our theme variables, all of our components become aware of this change and react instantly. This can be useful if you want to build an app that allows dynamic styling, live editing, theme swapping, etc.
- Since `theme` is treated as `data`, we can follow similar patterns to how `redux` and `react-redux` treat data by exposing what is essential to connected components.

Let's start to have some fun…

Dynamic Theme

Let's have a little bit of fun. To really drive the point home, we need to introduce a few more variables to our `theme`:

src/styles/theme.js

```
export default {
  $APP: 'todos-app',

  $mainBackgroundColor: '#F1F1F1',
  $mainFontFamily: 'Helvetica Neue, Arial, sans-serif',

  $lightGray: '#FAFAFA',
  $darkGray: '#888888',

  $primaryColor: 'red',
  $secondaryColor: 'green',
  $tertiaryColor: 'blue',
};
```

We added three new variables to our `theme`. Feel free to get creative by adding any color palette you'd like:

src/components/TodosList/TodosList.styles.js

```
export default (theme) => ({
  todosList: {
    listStyle: 'none',
    padding: '10px 15px',
    border: `10px solid ${theme.$secondaryColor}`,
    background: theme.$primaryColor,
    transition: 'all 0.5s ease',
    borderRadius: 5
  }
});
```

Yeah, this is not going to look very pretty :)

src/components/TodosListInfo/TodosListInfo.styles.js

```
export default (theme) => ({
  todosListInfo: {
    color: theme.$secondaryColor,
    fontSize: 14,
    textAlign: 'right'
  }
});
```

One more...

src/components/TodosListItem/TodosListItem.styles.js

```
export default (theme) => ({
  todosListItem: {
    background: theme.$lightGray,
    borderRadius: 5,
    border: `5px solid ${theme.$tertiaryColor}`,
    boxShadow: '0 2px 1px 0 rgba(0, 0, 0, 0.2)',
    color: theme.$darkGray,
    cursor: 'pointer',
    fontSize: '2rem',
    padding: '15px 20px',
    position: 'relative',
    transition: 'all 0.2s ease',
    ':hover': {
      opacity: 0.8
    }
  },

  completedTodosListItem: {
    background: theme.$primaryColor,
    border: `5px solid ${theme.$secondaryColor}`,
    boxShadow: 'inset 0 1px 2px 0 rgba(0, 0, 0, 0.3)',
    color: '#AAAAAA',
```

```
    textDecoration: 'line-through',
    top: 3
  }
});
```

Visit your browser and you'll see that we've officially made our todos list a designer's worst nightmare.

Hey! You didn't pick up this book for design advice, so take it easy or feel free to build an appropriate color palette of your choice. One of my favorite resources is Adobe Color[52], when I need inspiration to choose a color palette.

Moving on…

Currently, when we toggle our todo items, they animate to their updated values. These values are dynamic and can be changed on the fly, so let's go ahead and see how that would work:

[52] https://color.adobe.com/

src/components/App/App.js

```
// dependencies
...

// local dependencies
...

class App extends Component {

  constructor(...args) {
    super(...args);
    this.state = {
      todos: {},
      dynamicTheme: theme
    };
  }

  componentDidMount() {
    ...
  }

  randomizeTheme = () => {
    const colorPalette = [
      'red', 'green', 'blue', 'yellow',
      'orange', 'purple', 'cyan'
    ];
    const dynamicTheme = {
      ...this.state.dynamicTheme,
      $primaryColor: _.sample(colorPalette),
      $secondaryColor: _.sample(colorPalette),
      $tertiaryColor: _.sample(colorPalette)
    };
    this.setState({ dynamicTheme });
  };

```

```
35    toggleTodo = (id) => (e) => {
36      ...
37      this.randomizeTheme();
38    };
39
40    generateTodosListItem = (todo, id) => (
41      ...
42    );
43
44    render() {
45      ...
46    }
47  }
48
49  ...
```

Since we update our `theme`, we start to track it inside of our component state as `dynamicTheme`. We pass this theme via our component's state to the `ThemeWrapProvider`.

It's ideal to manage our `theme` inside of a state manager, like `Redux`, as a single source of truth.

We add a new function, `randomizeTheme`, that gets called every time we `toggleTodo`. This randomizer spreads over all `theme` variables and updates `$primaryColor`, `$secondaryColor` and `$tertiaryColor`.

Go ahead and check out your browser! Every time you toggle a todo, our component's react to our updated theme in real-time, without any sort of refresh. Awesome!

How is this useful? Imagine scenarios where users can change their profile by changing fonts, font sizes, entire app color schemes, etc. and we can have the entire app react accordingly, in real time, without any sort of refresh. This are the sorts of apps we can come to expect in the future.

I recommend stopping here and experimenting with different styles and elements. It can be fun and rewarding!

Mixins

Mixins are another useful functionality provided by `theme-wrap`. These mixins are similar to what you'd get from Sass mixins, but in JS.

For example, if we had a list of lists, such as a `TodosList`, `NamesList`, etc., that had common styles, mixins provide an elegant solution for managing these common styles. Let's see how that looks:

src/styles/mixins.js

```
import { ThemeWrapMixin } from 'theme-wrap';

export default new ThemeWrapMixin()
.set('list', (theme) => ({
  listStyle: 'none',
  padding: '10px 15px',
  borderRadius: 5
}));
```

Import `ThemeWrapMixin` from `theme-wrap` and create a new instance. There may be situations where you have many different mixin instances, so by defining individual themes and mixins, we have a good amount of flexibility. In many cases, a single instance is fine.

After creating an instance, we simply `set` our mixin by defining a unique name, e.g. `list`, and a function that is accepts `theme` as the first argument. This is the same `theme` managed by `ThemeWrapProvider`. Neat!

As you can see, these styles resemble our `TodosList.styles.js`. That's because we want to extract these and have them shared across all list components. Finally, we export our `mixins` instance.

How do `mixins` get consumed by `ThemeWrapProvider`? In the same way that it consumes `theme`. Let's see how we'd do that:

```
// dependencies
...

// local dependencies
...
import mixins from '../../styles/mixins';

class App extends Component {

  constructor(...args) {
```

```
    ...
  }

  componentDidMount() {
    ...
  }

  randomizeTheme = () => {
    ...
  };

  toggleTodo = (id) => (e) => {
    ...
  };

  generateTodosListItem = (todo, id) => (
    ...
  );

  render() {
    const { dynamicTheme, todos } = this.state;
    return (
      <StyleRoot>
        <Style rules={normalize} />
        <Style rules={globalStyles} />
        <ThemeWrapProvider theme={dynamicTheme}
                           mixins={mixins}>
          ...
        </ThemeWrapProvider>
      </StyleRoot>
    );
  }
}

export default App;
```

Import `mixins` and pass it as a prop to `ThemeWrapProvider`. That's it!

Now, we just need to use our mixins in our `TodosList` component:

src/components/TodosList/TodosList.styles.js

```
export default (theme, mixins) => ({
  todosList: {
    ...mixins.get('list'),
    border: `10px solid ${theme.$secondaryColor}`,
    background: theme.$primaryColor,
    transition: 'all 0.5s ease',
  }
});
```

Our `styles` file is actually provided two values: `theme` and `mixins`. We remove any common styles and `get` them by calling the appropriate `mixins` value. The spread operator is used to merge the styles together.

Let's try another example:

src/styles/mixins.js

```
import { ThemeWrapMixin } from 'theme-wrap';

export default new ThemeWrapMixin()
.set('list', (theme) => ({
  listStyle: 'none',
  padding: '10px 15px',
  transition: 'all 0.5s ease',
  borderRadius: 5
}))

.set('listItem', (theme) => ({
  background: theme.$lightGray,
  borderRadius: 5,
  boxShadow: '0 2px 1px 0 rgba(0, 0, 0, 0.2)',
  color: theme.$darkGray,
```

```
  cursor: 'pointer',
  fontSize: '2rem',
  padding: '15px 20px',
  position: 'relative',
  transition: 'all 0.2s ease',
  ':hover': {
    opacity: 0.8
  }
}))
```

We chain another mixin definition onto our `mixin` instance for common `listItem` styles. Let's put it to use:

src/components/TodosListItem/TodosListItem.styles.js

```
export default (theme, mixins) => ({
  todosListItem: {
    ...mixins.get('listItem'),
    border: `5px solid ${theme.$tertiaryColor}`,
  },

  completedTodosListItem: {
    ...mixins.get('listItem'),
    background: theme.$primaryColor,
    border: `5px solid ${theme.$secondaryColor}`,
    boxShadow: 'inset 0 1px 2px 0 rgba(0, 0, 0, 0.3)',
    color: '#AAAAAA',
    textDecoration: 'line-through',
    top: 3
  }
});
```

Visit your browser and see the application in action:

There is a lot more functionality contained in `theme-wrap`. For example, you can provide multiple `ThemeWrapProviders` in your app. That is, you can have multiple themes within your app by having different parts of your app contained under different `ThemeWrapProviders`. You could even nest your `ThemeWrapProviders`. Why would you want multiple themes in your app? Imagine cases in large applications, e.g. a News site, where each section of the application has different themes for the same components.

Summary

In this chapter, we took our app to the next level by providing live-reloadable styles, powerful mixins functionality, theme encapsulation, and more…

Since we've only added functionality to our inline styles, you shouldn't be surprised that we continue to meet all of the U&I specs as before:

	CSS	SCSS	CSS Modules	Inline Styles
No global namespace	*	*	✓	✓
Unidirectional styles	✓	✓	✓	✓
Dead code elimination			✓	✓
Minification			✓	✓
Shareable constants				✓
Deterministic resolution	*	*	✓	✓
Isolation	*	*	✓	✓
Extendable	*	*	✓	✓
Documentable	✓	✓	✓	✓
Presentable	✓	✓	✓	✓

[✓ Fulfilled] [* Pseudo fulfilled]

Before you continue to next chapters, experiment with `theme-wrap` and start thinking about different ways of leverage this sort of functionality. One thing to note is that all of this is possible due to the power and flexibility of inline styles.

Chapter 10: Showcasing

In the previous chapter we unlocked some powerful capabilities by offering real-time theme support to our application. Even though we only have a handful of components, it's starting to become complex in understanding the various APIs they provide. This issue becomes increasingly apparent in larger U&I libraries with hundreds of components. How do we, as developers, communicate our component's interfaces to other developers, designers, etc.?

There's a little secret that I've been holding out on the entire book! I'm sorry! It was for your own good!

Doesn't it make sense to build and document your U&I components in complete isolation? I believe so! Not only does this strategy provide the ability to build, review and test in isolation but it also provides the appropriate context to ensure you're building exactly what's needed — nothing more, nothing less. Meet Storybook[53]!

What is Storybook?

Storybook, as described on their site, is the "UI Development Environment You'll love to use." That captures my experience of Storybook. It's a handy tool that provides an environment to build, document and showcase components in isolation. Instead of reading a component's source code to try and figure out the various APIs provided, we can visit Storybook, where you get the component and all of its documentation in one simple and intuitive environment.

React Storybook in Action

Adding React Storybook to your app is as easy as a few simple commands:

First, you need the global command installed:

[53] https://storybook.js.org/

```
$ npm i -g getstorybook
```

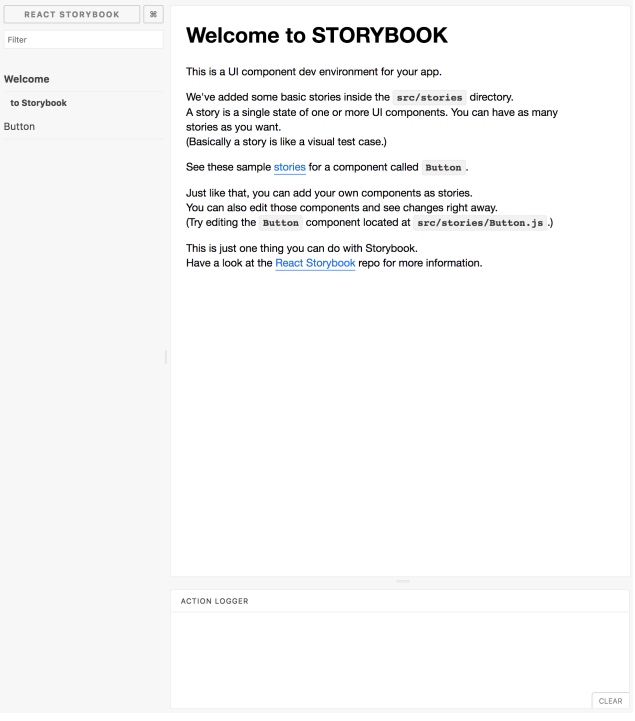
Then, at the root of our directory, we execute the following command for Storybook to configure our application.

```
$ getstorybook
```

Once this is done, you'll see a `.storybook` directory at the root of your project. The beauty about Storybook, similar to `create-react-app`, is that it works out-of-the-box with some wonderful functionality.

```
$ npm run storybook
```

If you the run the following command, you will see Storybook in action by visit `http://localhost:6006` or whatever address displayed.



Storybook UI

Documenting <TodosList />

Let's create our first story! In the same spirit of keeping our component files co-located, we will house the story file in the same directory.

src/components/TodosList/TodosList.story.js

```
// dependencies
import React from 'react';
import { storiesOf } from '@kadira/storybook';

// local dependencies
import TodosList from './TodosList';

export default storiesOf('TodosList', module)
.add('default view', () => (
  <TodosList>

  </TodosList>
));
```

We import `storiesOf` from `storybook` to be able to create a new story. We add a story called `default view` where we don't render anything.

Now, we need to import this story in our storybook configuration file:

.storybook/config.js

```
// dependencies
import { configure } from '@kadira/storybook';

function loadStories() {
  require('../src/components/TodosList/TodosList.story');
}

configure(loadStories, module);
```

If you visit the Storybook UI, you should see an error:

```
Error: mixins.get is not a function
```

Can you guess why that is?

Our component relies on `mixins` and since we're building our component in isolation without `theme-wrap`, we get that breaking error. Let's see how we can resolve this:

src/components/TodosList/TodosList.story.js

```
// dependencies
import React from 'react';
import { storiesOf } from '@kadira/storybook';
import { Style, StyleRoot } from 'radium';
import normalize from 'radium-normalize';
import { ThemeWrapProvider } from 'theme-wrap';

// local dependencies
import TodosList from './TodosList';
import globalStyles from '../../styles/globals.styles';
import mixins from '../../styles/mixins';
import theme from '../../styles/theme';

export default storiesOf('TodosList', module)
.add('default view', () => (
  <StyleRoot>
    <Style rules={normalize} />
    <Style rules={globalStyles} />
    <ThemeWrapProvider theme={theme}
                       mixins={mixins}>
      <TodosList>

      </TodosList>
    </ThemeWrapProvider>
  </StyleRoot>
));
```

We replicate the same functionality in `App.js` by wrapping our component with all of the necessary dependencies. Go ahead and check out Storybook UI.

It's not very exciting, since we're not displaying any children, so let's document that functionality:

src/components/TodosList/TodosList.story.js

```
// dependencies
...

// local dependencies
...

export default storiesOf('TodosList', module)
.add('default view', () => (
  ...
))

.add('renders children', () => (
  <StyleRoot>
    <Style rules={normalize} />
    <Style rules={globalStyles} />
    <ThemeWrapProvider theme={theme}
                       mixins={mixins}>
      <TodosList>
        {_.map([1, 2, 3], (i) => <li>{i} - Item</li>)}
      </TodosList>
    </ThemeWrapProvider>
  </StyleRoot>
))
```

Visit Storybook UI and you should see a second story, labeled as `renders children`, where it display our component containing 3 `li` elements.

There's a good amount of repeated code! How do we solve this? We can create another component or use Storybook's `decorator`:

src/components/TodosList/TodosList.story.js

```
// dependencies
...
import { addDecorator, storiesOf } from '@kadira/storybook';

// local dependencies
...

const Root = (story) => (
  <StyleRoot>
    <Style rules={normalize} />
    <Style rules={globalStyles} />
    <ThemeWrapProvider theme={theme}
                       mixins={mixins}>
      {story()}
    </ThemeWrapProvider>
  </StyleRoot>
);

addDecorator(Root);

// helper function that generates a set of children
const generateChildren = (num = 5) => _(num)
  .range()
  .map((i) => <li key={i}>{i} - Item</li>)
  .value();

export default storiesOf('TodosList', module)
.add('default view', () => (
  <TodosList>

  </TodosList>
))

.add('renders children', () => (
```

```
35    <TodosList>
36      {generateChildren(5)}
37    </TodosList>
38  ))
```

Very nice! We create a `Root` component that gets `story` via `props` and use it to decorate all of our stories with `addDecorator`. Let's continue:

src/components/TodosList/TodosList.story.js

```
1   // dependencies
2   ...
3
4   // local dependencies
5   ...
6
7   export default storiesOf('TodosList', module)
8   .add('default view', () => ...)
9
10  .add('renders children', () => (...)
11
12  .add('accepts custom styles', () => {
13    const customStyles = {
14      todosList: {
15        background: 'orange',
16        border: '10px solid purple',
17        color: 'purple',
18        fontSize: 40
19      }
20    };
21    return (
22      <TodosList pStyles={customStyles}>
23        {generateChildren(10)}
24      </TodosList>
25    );
26  });
```

We document that `TodosList` accepts custom styles, which can be seen in Storybook UI.

This is great but how would others know how to use this components without needing to look at its source code? Storybook provides many useful `addons` for developers! Let's use one that does exactly what we need:

```
$ npm i @storybook/addon-info --save-dev
```

We need to update our configuration file to use this addon:

.storybook/config

```
1  // dependencies
2  import { configure, setAddon } from '@kadira/storybook';
3  import infoAddon from '@storybook/addon-info';
4
5  setAddon(infoAddon);
6
7  function loadStories() {
8    require('../src/components/TodosList/TodosList.story');
9  }
10
11 configure(loadStories, module);
```

Import `infoAddon` and use it with `setAddon`.

If you encounter an error that `react-addons-create-fragment` is `undefined`, you can install it with the following command: `npm i react-addons-create-fragment --save`.

If we want component stories to include their API documentation, we need to update the `add` method to `addWithInfo`:

src/components/TodosList/TodosList.story.js

```
// dependencies
...

// local dependencies
...

export default storiesOf('TodosList', module)
.addWithInfo('default view', () => ...)

.addWithInfo('renders children', () => (...)

.addWithInfo('accepts custom styles', () => { ... });
```

Visit Storybook and you should see a little icon, where you can toggle the component's info and interface. There are many other options with this addon, so I recommend experimenting to see what works for you.

Before we move onto our next component, let's refactor our configuration file so that we can `decorate` all of our stories with `theme-wrap` and `radium`:

.storybook/config.js

```
// dependencies
import React from 'react';
import normalize from 'radium-normalize';
import { Style, StyleRoot } from 'radium';
import infoAddon from '@storybook/addon-info';
import { ThemeWrapProvider } from 'theme-wrap';
import { addDecorator, configure, setAddon } from '@kadira/storybook\
';

// local dependencies
import mixins from '../src/styles/mixins';
import theme from '../src/styles/theme';
```

```
import globalStyles from '../src/styles/globals.styles';

const Root = story => (
  <StyleRoot>
    <Style rules={normalize} />
    <Style rules={globalStyles} />
    <ThemeWrapProvider theme={theme}
                       mixins={mixins}>
      {story()}
    </ThemeWrapProvider>
  </StyleRoot>
);

// attach our decorator
addDecorator(Root);

// enable component info
setAddon(infoAddon);

configure(() => {
  require('../src/components/TodosList/TodosList.story');
}, module);
```

Then update `TodosList.story.js`:

src/components/TodosList/TodosList.story.js

```
// dependencies
import _ from 'lodash';
import React from 'react';
import { storiesOf } from '@kadira/storybook';

// local dependencies
import TodosList from './TodosList';

// helper function that generates a set of children
```

```
const generateChildren = (num = 5) => _(num)
.range()
.map((i) => <li key={i}>Item #{i}</li>)
.value();

export default storiesOf('TodosList', module)
.addWithInfo('default view', () => (
  <TodosList>
    <li>Default</li>
  </TodosList>
))

.addWithInfo('renders children', () => (
  <TodosList>
    {generateChildren(5)}
  </TodosList>
))

.addWithInfo('accepts custom styles', () => {
  const customStyles = {
    todosList: {
      background: 'orange',
      border: '10px solid purple',
      color: 'purple',
      fontSize: 40
    }
  };
  return (
    <TodosList pStyles={customStyles}>
      {generateChildren(10)}
    </TodosList>
  );
});
```

Visit Storybook UI and everything should work as expected.

Documenting <TodosListInfo />

Let's create our second story for `TodosListInfo`.

First update the Storybook config file:

.storybook/config.js

```
...

configure(() => {
  require('../src/components/TodosList/TodosList.story');
  require('../src/components/TodosListInfo/TodosListInfo.story');
}, module);
```

We need to require each story to be displayed inside of Storybook UI.

Now, onto our story:

src/components/TodosListInfo/TodosListInfo.story.js

```
// dependencies
import _ from 'lodash';
import React from 'react';
import { storiesOf } from '@kadira/storybook';

// local dependencies
import TodosListInfo from './TodosListInfo';

const generateTodos = (num = 5) => {
  const todos = {};
  _.times(num, id => {
    todos[id] = {
      id,
      completed: Math.random() > 0.5,
      description: `Task #${id}`
    };
  });
```

```
  return todos;
};

export default storiesOf('TodosListInfo', module)
.addWithInfo('default view', () => (
  <TodosListInfo todos={generateTodos()} />
));
```

Simple enough! Since we've locked this component's styling to the outside world, it's nearly impossible to style/move it, as seen in the Storybook UI. Feel free to extend this component's functionality, if needed.

Documenting <TodosListItem />

Let's document our last remaining component, `TodosListItem`.

.storybook/config.js

```
...

configure(() => {
  require('../src/components/TodosList/TodosList.story');
  require('../src/components/TodosListInfo/TodosListInfo.story');
  require('../src/components/TodosListItem/TodosListItem.story');
}, module);
```

Onto the fun parts:

src/components/TodosListItem/TodosListItem.story.js

```
// dependencies
import React from 'react';
import { storiesOf } from '@kadira/storybook';

// local dependencies
import TodosListItem from './TodosListItem';

const sampleTodo = {
  id: 1,
  completed: false,
  description: 'Sample Todo'
};

const sampleCompletedTodo = {
  id: 2,
  completed: true,
  description: 'Sample Completed Todo'
};

export default storiesOf('TodosListItem', module)
.addWithInfo('default view', () => (
  <TodosListItem />
))

.addWithInfo('accepts todo via props', () => (
  <TodosListItem todo={sampleTodo} />
))

.addWithInfo('accepts completed todo via props', () => (
  <TodosListItem todo={sampleCompletedTodo} />
))

.addWithInfo('accepts custom styles via props', () => {
  const customStyles = {
```

```
    todosListItem: {
      border: '20px dashed orange',
      color: 'orange',
      fontSize: 40,
      margin: '0 auto',
      padding: 10,
      width: 500
    }
  };
  return (
    <TodosListItem todo={sampleTodo}
                   pStyles={customStyles} />
  );
})

.addWithInfo('accepts custom completed styles via props', () => {
  const customStyles = {
    completed: {
      backgroundColor: 'black',
      color: 'orange'
    }
  };
  return (
    <TodosListItem todo={sampleCompletedTodo}
                   pStyles={customStyles} />
  );
});
```

Nothing out of the ordinary! We create a few stories, outlining the various APIs provided by `TodosListItem`. However, we have one functionality that hasn't been documented: `handleClick`. Storybook provides another useful feature, `action`, that helps document our event handlers:

src/components/TodosListItem/TodosListItem.story.js

```
// dependencies
...
import { action, storiesOf } from '@kadira/storybook';

...

.addWithInfo('accepts toggleTodo callback via props', () => (
  <TodosListItem todo={sampleTodo}
                 handleClick={(id) => (e) => action('toggle todo')(i\
d)} />
));
```

Since we opted for higher order functions to handle our event handlers, our event handler for `handleClick` mimics what we originally had in `App.js`. If you now click the item, you'll see a message in the `action logger` section of Storybook UI.

Summary

In this chapter, we introduced Storybook into our tool chain. You should now have the foundation to be able to experiment with and create other Storybook addons, decorators and configurations. Storybook is both useful and powerful in providing an environment to rapidly create, document and showcase components.

Lastly, we can fulfill the last two U&I criteria that we ignored in previous chapters.

Documenting and presenting a U&I library doesn't necessarily rely on any particular styling strategy, so we could've used Storybook all along.

	CSS	SCSS	CSS Modules	Inline Styles
No global namespace	*	*	✓	✓
Unidirectional styles	✓	✓	✓	✓
Dead code elimination			✓	✓
Minification			✓	✓
Shareable constants				✓
Deterministic resolution	*	*	✓	✓
Isolation	*	*	✓	✓
Extendable	*	*	✓	✓
Documentable	✓	✓	✓	✓
Presentable	✓	✓	✓	✓

[✓ Fulfilled] [* Pseudo fulfilled]

Before moving on I recommend creating other components using Storybook and experiment with other add-ons.

In the following chapter, we will cover other promising technologies around the corner.

Chapter 11: Looking Ahead

Throughout this book, we covered several powerful U&I strategies. Are there other strategies not covered in this book? What does the current landscape look like?

There are many, many more!

Explorations

We've covered many different strategies throughout this book, but the ecosystem for building scalable UI is new and continues to evolve quickly. Here are a few technologies to explore:

CSS Next

CSS Next[54] allows for you to write "tomorrow's CSS syntax, today." CSS Next offers many of the same features of a traditional CSS preprocessors, along with a few additional powerful capabilities. For example, variables, in CSS Next, can be changed via JavaScript? ***WHAAaaaAATT?*** That's right! We can change our CSS variables in JavaScript. This example[55], by Wes Bos[56], does a great job showcasing this feature.

This is an exciting feature, because we can share constants between JavaScript and CSS. I look forward to CSS Next becoming widely supported so that I can deprecate `theme-wrap` :)

CSS in JS

Writing your CSS in JavaScript seems to be the most promising strategy and it continues to be one of the quickest evolving amongst styling strategies. Instead of

[54] http://cssnext.io/

[55] https://codepen.io/wesbos/pen/adQjoY

[56] https://twitter.com/wesbos

outlining all the various CSS in JS implementations, I recommend investigating them in this informative repo[57].

Hardware Accelerated UI

Lastly, we don't necessarily need to tie all of our components to DOM. We could leverage `canvas` and, better yet, leverage the GPU with WebGL. If you're looking to build a highly interactive and immersive interface, you may want to look at these libraries: A-Frame[58], React Three Renderer[59], etc.

Conclusion

Thank you for taking the time to go through this book. I hope that it has been both informative and useful for the next time that you decide to build an application. Lastly, we've covered many strategies in this book, but haven't had the chance to step back and determine their practical applications. As a rule of thumb, when I approach UI projects, I ask the following set of questions:

- Are the other front end developers on the team mostly comfortable with CSS or a CSS preprocessor?
- How large is the web application?
- Is SEO important? If so, does squeezing every bit of improvement necessary?
- Does the application need to be highly interactive and customizable?
- Do we serve up different applications for different interfaces or are we relying on a single responsive application?

Whatever strategy you decide to use, whether it's Sass, BEM, CSS Modules, Inline Styles, CSS in JS, etc., there is no substitute for a well defined architecture.

If you have any questions or feedback, please refer to the repository[60] or contact me directly.

Thank you!

[57] https://github.com/MicheleBertoli/css-in-js
[58] https://aframe.io/
[59] https://github.com/toxicFork/react-three-renderer
[60] https://github.com/FarhadG/ui-react