
Table of Contents

| | |
|----------------------------------|-------|
| Introduction | 1.1 |
| Setup | 1.2 |
| Functional JavaScript | 1.3 |
| ES6 constructs | 1.4 |
| Default Params | 1.4.1 |
| Template Literals | 1.4.2 |
| Destructuring | 1.4.3 |
| Arrow Functions | 1.4.4 |
| Promises | 1.4.5 |
| let and const | 1.4.6 |
| Modules | 1.4.7 |
| Thinking in Components | 1.5 |
| Atomic Design | 1.5.1 |
| Atomic Component Principles | 1.5.2 |
| Benefits of This Approach | 1.5.3 |
| The Process | 1.5.4 |
| Task #1 | 1.5.5 |
| React Components | 1.6 |
| Stateless Components | 1.6.1 |
| Stateful Components | 1.6.2 |
| Stateful vs Stateless Components | 1.6.3 |
| Composition | 1.6.4 |
| Task #2 | 1.6.5 |
| Task #3 | 1.6.6 |
| Task #4 | 1.6.7 |
| Task #5 | 1.6.8 |
| Immutable | 1.7 |
| What Is Immutability? | 1.7.1 |
| The Case for Immutability | 1.7.2 |
| JavaScript Solutions | 1.7.3 |

| | |
|---|-----------|
| Object.assign | 1.7.3.1 |
| Object.freeze | 1.7.3.2 |
| Immutable.js Basics | 1.7.4 |
| Immutable.Map | 1.7.4.1 |
| Map.merge | 1.7.4.1.1 |
| Nested Objects | 1.7.4.2 |
| Deleting Keys | 1.7.4.2.1 |
| Maps are Iterable | 1.7.4.2.2 |
| Immutable.List | 1.7.4.3 |
| Performance | 1.7.4.4 |
| Persistent and Transient Data Structures | 1.7.4.5 |
| Official Documentation | 1.7.4.6 |
| Exercises | 1.7.5 |
| Task #1 | 1.7.5.1 |
| Task #2 | 1.7.5.2 |
| Task #3 | 1.7.5.3 |
| Task #4 | 1.7.5.4 |
| Task #5 | 1.7.5.5 |
| Task #6 | 1.7.5.6 |
| Task #7 | 1.7.5.7 |
| Redux | 1.8 |
| Review of Reducers and Pure Functions | 1.8.1 |
| Redux Reducers | 1.8.2 |
| Redux Actions | 1.8.3 |
| Configuring your Application to use Redux | 1.8.4 |
| Using Redux with Components | 1.8.5 |
| Redux and Component Architecture | 1.8.6 |
| Routing | 1.9 |
| React Router | 1.9.1 |
| Router Redux | 1.9.2 |
| Forms | 1.10 |
| Redux Form | 1.10.1 |
| Testing | 1.11 |
| Setup | 1.11.1 |

| | |
|------------|--------|
| Components | 1.11.2 |
| Reducers | 1.11.3 |
| Actions | 1.11.4 |

React.js Training



As a dedicated JavaScript design and development firm, we work extensively with clients on React projects. We've developed this training so that now you can jump start your React projects using the best practices we've developed.

This book walks the reader through everything they need to know from getting started with the React toolchain to writing applications with scalable front end architectures.

What you'll learn

- How to build a web application with ReactJS, ImmutableJS, and Redux.
- The process used to build an app using component based architecture.
- The benefits of having a single global application state and how to leverage the power of Redux's flux-inspired architecture.

Prerequisites

Students should have the following tools installed:

Command Line

The React toolchain is quite command-line oriented, so we suggest installing a good command-line terminal program.

For **Mac OS X**, we recommend [iTerm2](#).

For **Windows**, we recommend using [Git Bash](#) (also comes with git – see below).

Git

We'll be using Git from the command line: <http://git-scm.com/download/>

Tips for Mac OS X users

If you have [homebrew](#) installed, you may install Git by simply issuing: `brew install git` .

Tips for Windows users

- When in doubt, select the default options in the installation wizard.
- The rest of this training assumes you are using the terminal that comes with Git (Git Bash) to run all command-line examples.

Notes: The official Git download provides several GUI tools, for both OS X and Windows. If you are proficient with these interfaces, feel free to use them. This training, however, will only provide instructions from the command-line.

Node.js

<http://nodejs.org/download/>

The download above should install two commands: `node` and `npm` .

`npm` may require some extra configuration to set permissions properly on your system.

On **Mac OS X**, do the following:

```
npm config set prefix ~/.npm
echo 'export PATH="$PATH:~/.npm/bin"' >> ~/.bashrc
~/.bashrc
```

On **Windows**, fire up `Git Bash` and type the following:

```
mkdir /c/Users/$USER/AppData/Roaming/npm
echo 'export PATH="$PATH:/c/Program Files/nodejs"' >> ~/.bashrc
~/.bashrc
```

Note: if you installed the 32-bit Windows version of node, the second line above should instead read:

```
echo 'export PATH="$PATH:/c/Program Files (x86)/nodejs"' >> ~/.bashrc
```

A Code Editor

Any text editor will work. At Rangle.io, the most popular editors/IDEs are:

- [Vim](#)
- [Sublime Text](#)
- [Atom](#)
- [WebStorm](#)

Google Chrome

<https://www.google.com/chrome/browser/desktop/index.html>

Setup

```
git clone https://github.com/rangle/react-training

npm install -g jspm
npm install
jspm install

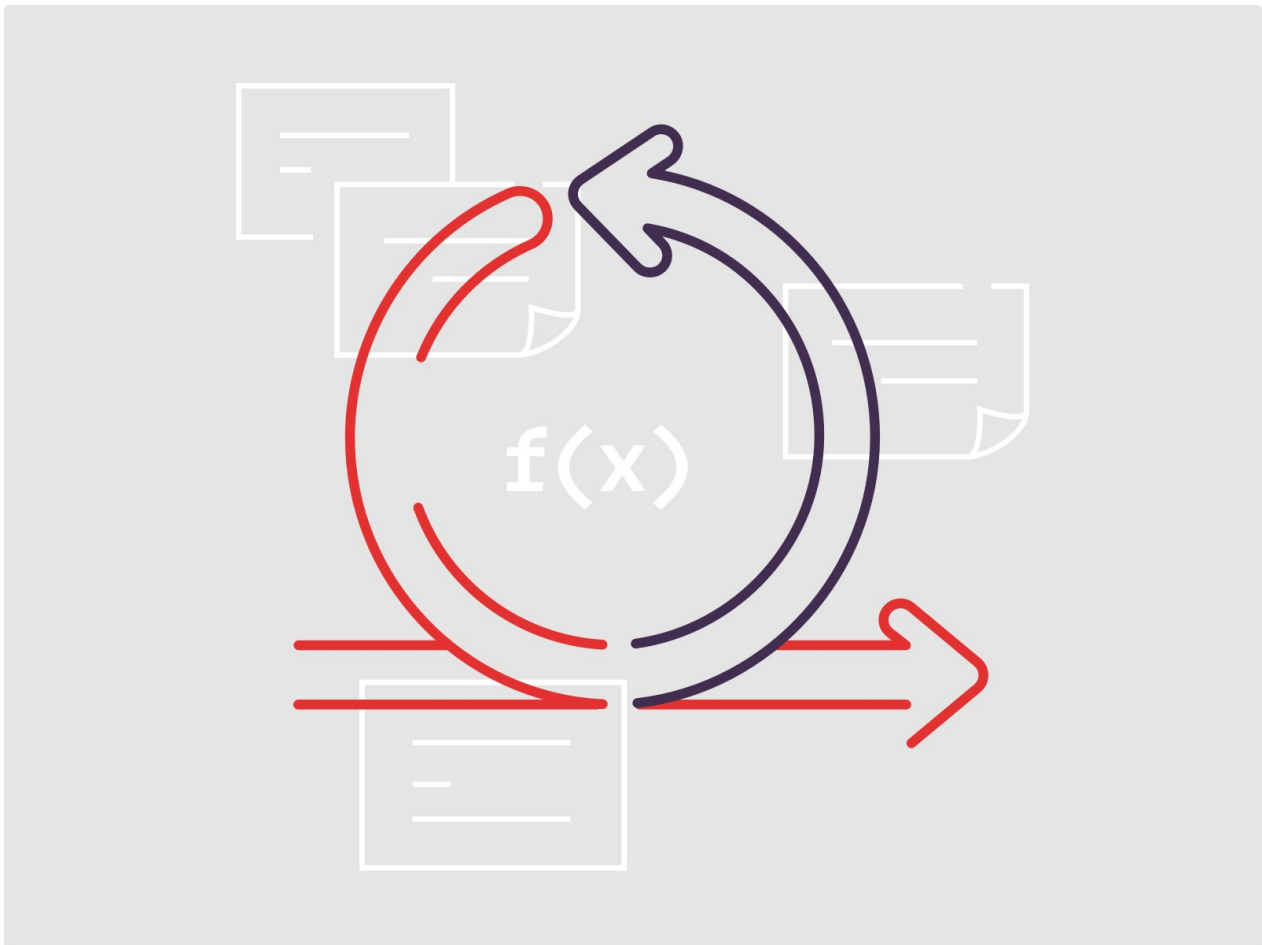
npm run dev
```

Troubleshooting

jspm complains that `github rate limit reached`

1. Go to github.com, login and click `settings`
2. Click Personal access tokens and then `Generate new token` (make sure to enable all the options you want)
3. Copy the token and start command line inside the project folder
4. Run `jspm registry config github`

Functional JavaScript



Functional programming approaches computer programs as consisting of data transformations, much like functions in math. Your program is a collection of functions. Each function takes some inputs and produces some outputs. Simple functions are used as building blocks for more complex ones. This is one of the earliest approaches to programming, dating back to the 1960s and based on math developed in the 1930s.

JavaScript allows for different styles of programming: object oriented, imperative, and functional.

React brings these functional programming concepts to the UI. It provides abstractions such as components which are (mostly) pure functions. Given some state (as props) they return some DOM. This allows us to get away from imperatively touching the DOM.

The functional style is more flexible, and makes testing and debugging easier. Two core concepts of functional programming are immutability and stateless, both of which are covered in later sections.

Functional Concepts

Pure Functions (Slice vs. Splice)

A pure function is a function where the return value is only determined by its input values, without observable side effects.

For example, `slice` and `splice` complete the same functionality, however, `splice` has the undesired impact of mutating the origin input.

```
let ltrs = ["a", "b", "c"]

ltrs.slice(1) // returns ["b", "c"], where ltrs is still ["a", "b", "c"]

ltrs.splice(1) // returns ["b", "c"], where ltrs is now ["a"]
```

Map, Filter, Reduce

Array helper methods `map`, `filter`, and `reduce` are examples of functional programming, which take in functions and do not mutate the original array.

```
var list = [1, 2, 3, 4, 5];

var double = list.map(function (x) {
  return x * 2;
});

var gt3 = list.filter(function (x) {
  return x > 3;
});

var sum = list.reduce((result, x) => {
  console.log(`result in: ${result }, x: ${x }, result out: ${result + x }`);
  return result + x;
}, 0);
```

Currying

Currying is the technique of translating the evaluation of a function that takes multiple arguments (or a tuple of arguments) into evaluating a sequence of functions, each with a single argument.

```
function add (a, b, c) {  
  return a + b + c;  
}  
  
add(1, 2, 3); // returns 6  
  
// Currying add()  
function add (a) {  
  return function(b) {  
    return function(c) {  
      return a + b + c;  
    }  
  }  
}  
  
add1 = add(1); // returns a function where a = 1  
add2 = add1(2); // returns a function where a = 1, b = 2  
add3 = add2(3); // returns 6  
  
// Below is a short form of currying using ES6 array functions  
add => a => b => c => a + b + c  
  
result = add(1)(2)(3); // returns 6
```

Resources

This section covers the basics, check out this [gitbook](#) for an in-depth on Functional Programming

ES6

JavaScript was written by Brendan Eich during a ten day stretch in the nineteen ninties.

More than twenty years later, the language is thriving. There are subsets, supersets, current versions, and an upcoming version. ES6 is that upcoming version, and it brings a lot of new features. Some of the highlights:

- Classes
- Default Params
- Template Literals
- Destructuring
- Arrow Functions
- Promises
- let and const
- Modules

Default Params

Old ES5 way to set default params

Approach 1: easy way.

```
// This is what people normally do in ES5 to set default params
function link(height, color, callbackFn) {
  var height = height || 50;
  var color = color || 'red';
  var callbackFn = callbackFn || function() {};

  // function content...
}
```

It works well, but in the above implementation we didn't account for falsy values. For example: `0`, `''`, `null`, `NaN`, `false` are falsy values.

Approach 2: Better way.

```
// So there is a better way to do this, it checks param is actually undefined or not:
function link(height, color, callbackFn) {
  var height = typeof height !== 'undefined' ? height : 50;
  var color = typeof color !== 'undefined' ? color : 'red';
  var callbackFn = typeof callbackFn !== 'undefined' ? callbackFn : function() {};

  // function content...
}
```

ES6 way to write default params

Approach 3: ES6 way, it gets just so much better.

```
function link(height = 50, color = 'red', callbackFn = () => {}) {
  // function content...
}

// or using ES6 const and let
const noop = () => {};
const link = (height = 50, color = 'red', callbackFn = noop) => {
  // function content...
};
```

Further reading:

- https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/default_parameters
- <http://tc39wiki.calculist.org/es6/default-parameter-values/>
- http://exploringjs.com/es6/ch_parameter-handling.html
- <https://dorey.github.io/JavaScript-Equality-Table>

Template Literals

In traditional JavaScript, text that is enclosed within matching `"` marks, or `'` marks is considered a string. Text within double or single quotes can only be on one line. There was also no way to insert data into these strings. This resulted in a lot of ugly concatenation code that looked like:

```
var name = 'Sam';
var age = 42;

console.log('hello my name is ' + name + ' I am ' + age + ' years old');
//= 'hello my name is Sam I am 42 years old'
```

ES6 introduces a new type of string literal that is marked with back ticks (```). These string literals *can* include newlines, and there is a new mechanism for inserting variables into strings:

```
var name = 'Sam';
var age = 42;

console.log(`hello my name is ${name}, and I am ${age} years old`);
//= 'hello my name is Sam, and I am 42 years old'
```

The `${}` works fine with any kind of expression, including member expressions and method calls.

```
var name = 'Sam';
var age = 42;

console.log(`hello my name is ${name.toUpperCase()}, and I am ${age / 2} years old`);
//= 'hello my name is SAM, and I am 21 years old'
```

There are all sorts of places where these kind of strings can come in handy, and front end web development is one of them.

Further reading

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals
- <https://developers.google.com/web/updates/2015/01/ES6-Template-Strings?hl=en>
- <https://ponyfoo.com/articles/es6-template-strings-in-depth>

- http://exploringjs.com/es6/ch_template-literals.html

Destructuring

Destructuring is a way to quickly extract data out of an `{}` or `[]` without having to write much code.

To [borrow from the MDN][mdnDest], destructuring can be used to turn the following:

```
let foo = ['one', 'two', 'three'];

let one  = foo[0];
let two  = foo[1];
let three = foo[2];
```

into

```
let foo = ['one', 'two', 'three'];
let [one, two, three] = foo;
console.log(one); // 'one'
```

This is pretty interesting, but at first it might be hard to see the use case. ES6 also supports Object destructuring, which might make uses more obvious:

```
let myModule = {
  drawSquare: function drawSquare(length) { /* implementation */ },
  drawCircle: function drawCircle(radius) { /* implementation */ },
  drawText: function drawText(text) { /* implementation */ },
};

let {drawSquare, drawText} = myModule;

drawSquare(5);
drawText('hello');
```

Destructuring can also be used for passing objects into a function, allowing you to pull specific properties out of an object in a concise manner. It is also possible to assign default values to destructured arguments, which can be a useful pattern if passing in a configuration object.


```
let jane = { firstName: 'Jane', lastName: 'Doe'};
let john = { firstName: 'John', lastName: 'Doe', middleName: 'Smith' }
function sayName({firstName, lastName, middleName = 'N/A'}) {
  console.log(`Hello ${firstName} ${middleName} ${lastName}`)
}

sayName(jane) // -> Hello Jane N/A Doe
sayName(john) // -> Helo John Smith Doe
```

The spread operator allows an expression to be expanded in places where multiple arguments (for function calls) or multiple elements (for array literals) or multiple variables (for destructuring assignment) are expected.

For Arrays:

```
const fruits = ['apple', 'banana'];
const veggies = ['cucumber', 'potato'];

const food = ['grapes', ...fruits, ...veggies];
// -> ["grapes", "apple", "banana", "cucumber", "potato"]

const [fav, ...others] = food;
console.log(fav); // -> "grapes"
console.log(others); // -> ["apple", "banana", "cucumber", "potato"]
```

For Objects:

```
const member = {
  name: 'Ben',
  title: 'software developer',
  skills: ['javascript', 'react', 'redux'],
};

const memberWithMetadata = {
  ...member,
  previousProjects: ['BlueSky', 'RedOcean'];
};

// behind the scenes:
const memberWithMetadata = Object.assign(member, {previousProjects: ['BlueSky', 'RedOcean']});

console.log(memberWithMetadata.name); // -> "Ben"
console.log(Object.keys(memberWithMetadata)); // -> ["name", "title", "skills", "previousProjects"]
```

For function calls:

```
const food = ['grapes', 'apple', 'banana', 'cucumber', 'potato'];
function eat() {
  console.log(...arguments);
}

eat(...food)
// -> grapes apple banana cucumber potato
```

Further reading

- [MDN Destructuring](#)
- [ES6 In Depth: Destructuring](#)
- [Destructuring Assignment in ECMAScript 6](#)
- [Object.assign\(\)](#)

Arrow Functions

ES6 offers some new syntax for dealing with `this` : "arrow functions". Arrow function also make working with "higher order" functions (functions that take functions as parameters) much easier to work with.

The new "fat arrow" notation can be used to define anonymous functions in a simpler way.

Consider the following example:

```
items.forEach(function(x) {  
  console.log(x);  
  incrementedItems.push(x+1);  
});
```

This can be rewritten as an "arrow function" using the following syntax:

```
items.forEach((x) => {  
  console.log(x);  
  incrementedItems.push(x+1);  
});
```

Functions that calculate a single expression and return its values can be defined even simpler:

```
incrementedItems = items.map((x) => x+1);
```

The latter is *almost* equivalent to the following:

```
incrementedItems = items.map(function (x) {  
  return x+1;  
});
```

There is one important difference, however: arrow functions do not set a local copy of `this` , `arguments` , `super` , or `new.target` . When `this` is used inside an arrow function JavaScript uses the `this` from the outer scope. Consider the following example:

```
class Toppings {
  constructor(toppings) {
    this.toppings = Array.isArray(toppings) ? toppings : [];
  }
  outputList() {
    this.toppings.forEach(function(topping, i) {
      console.log(topping, i + '/' + this.toppings.length); // no this
    })
  }
}

var ctrl = new Toppings(['cheese', 'lettuce']);

ctrl.outputList();
```

Let's try this code on ES6 Fiddle (<http://www.es6fiddle.net/>). As we see, this gives us an error, since `this` is undefined inside the anonymous function.

Now, let's change the method to use the arrow function:

```
class Toppings {
  constructor(toppings) {
    this.toppings = Array.isArray(toppings) ? toppings : [];
  }
  outputList() {
    this.toppings
      .forEach((topping, i) => console
        .log(topping, i + '/' + this.toppings.length); // `this` works!
      )
  }
}

var ctrl = new Toppings(['cheese', 'lettuce']);
```

Here `this` inside the arrow function refers to the instance variable.

Warning arrow functions do *not* have their own `arguments` variable, this can be confusing to veteran JavaScript programmers. `super`, and `new.target` are also scoped from the outer enclosure.

Promises

Promises are built-in ES6.

```
const wait (ms) => {  
  return new Promise((resolve, reject) => {  
    setTimeout(resolve, ms);  
  });  
}  
  
wait(1000).then(() => console.log('tick'));
```

Promises vs Callbacks

For HTTP Requests, our existing solution is to use callbacks:

```
request(url, (error, response) => {  
  // handle success or error.  
});  
doSomethingElse();
```

A few problems exist with callbacks. One is known as "[Callback Hell](#)". A larger problem is decomposition.

The callback pattern requires us to specify the task and the callback at the same time. In contrast, promises allow us to specify and dispatch the request in one place:

```
promise = fetch(url); //fetch is a replacement for XMLHttpRequest
```

and then to add the callback later, and in a different place:

```
promise.then(response => {  
  // handle the response.  
});
```

This also allows us to attach multiple handlers to the same task:

```
promise.then(response => {  
  // handle the response.  
});  
promise.then(response => {  
  // do something else with the response.  
});
```

More on Promises

`.then()` always returns a promise. Always.

```
p1 = getDataAsync(query);  
  
p2 = p1.then(  
  results => transformData(results));
```

`p2` is now a promise regardless of what `transformData()` returned. Even if something fails.

If the callback function returns a value, the promise resolves to that value:

```
p2 = p1.then(results => 1);
```

`p2` will resolve to “1”.

If the callback function returns a promise, the promise resolves to a functionally equivalent promise:

```
p2 = p1.then(results => {  
  let newPromise = getSomePromise();  
  return newPromise;  
});
```

`p2` is now functionally equivalent to `newPromise`.

```
p2 = p1.then(  
  results => throw Error('Oops'));  
  
p2.then(results => {  
  // You will be wondering why this is never  
  // called.  
});
```

`p2` is still a promise, but now it will be rejected with the thrown error.

Why won't the second callback ever be called?

Catching Rejections

The function passed to `then` takes a second argument, i.e. `error`, which represents error catching within the promise chain.

```
fetch('http://ngcourse.herokuapp.com/api/v1/tasks')
  .then(response => response.data)
  .then(tasks => filterTasksAsynchronously(tasks))
  .then(tasks => {
    $log.info(tasks);
    vm.tasks = tasks;
  })
  .then(
    null,
    error => log.error(error)
  );
```

Note that one catch at the end is often enough.

let and const

ES6 introduces the concept of block scoping. Block scoping will be familiar to programmers from other languages like C, Java, or even PHP. In ES5 JavaScript, and earlier, `var` s are scoped to `function` s, and they can "see" outside their functions to the outer context.

```
var five = 5;
var threeAlso = three; // error

function scope1() {
  var three = 3;
  var fiveAlso = five; // == 5
  var sevenAlso = seven; // error
}

function scope2() {
  var seven = 7;
  var fiveAlso = five; // == 5
  var threeAlso = three; // error
}
```

In ES5 functions were essentially containers that could be "seen" out of, but not into.

In ES6 `var` still works that way, using functions as containers, but there are two new ways to declare variables: `const`, and `let`. `const`, and `let` use `{`, and `}` blocks as containers, hence "block scope".

Block scoping is most useful during loops. Consider the following:

```
var i;
for (i = 0; i < 10; i += 1) {
  var j = i;
  let k = i;
}
console.log(j); // 9
console.log(k); // undefined
```

Despite the introduction of block scoping, functions are still *the* preferred mechanism for dealing with most loops.

`let` works like `var` in the sense that its data is read/write. `let` is also useful when used in a for loop. For example, without `let`:


```
for(var x=0;x<5;x++) {  
  setTimeout(()=>console.log(x), 0)  
}
```

Would output 5,5,5,5,5 . However, when using `let` instead of `var` , the value would be scoped in a way that people would expect.

```
for(let x=0;x<5;x++) {  
  setTimeout(()=>console.log(x), 0)  
}
```

Alternatively, `const` is read only. Once `const` has been assigned, the identifier can not be re-assigned, the value is [not immutable](#).

For example:

```
const myName = 'pat';  
let yourName = 'jo';  
  
yourName = 'sam'; // assigns  
myName = 'jan';   // error
```

The read only nature can be demonstrated with any object:

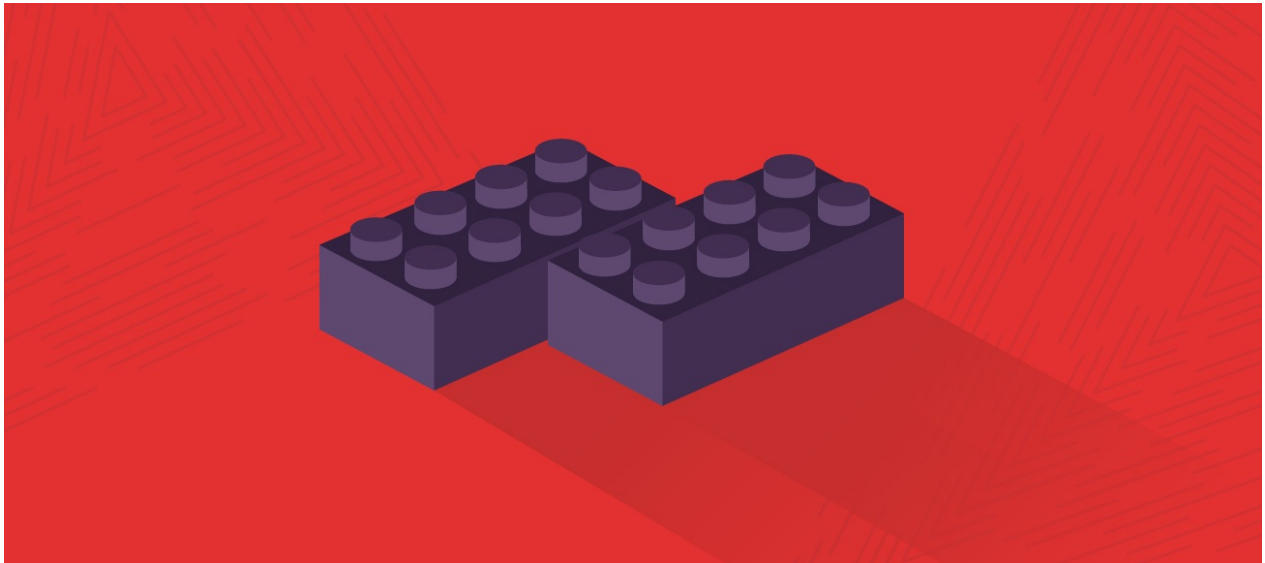
```
const literal = {};  
  
literal.attribute = 'test'; // fine  
literal = []; // error;
```

Modules

ES6 also introduces the concept of a module, which works in a similar way to other languages. Defining an ES6 module is quite easy: each file is assumed to define a module and we'll specify its exported values using the export keyword.

```
import Parse from 'parse';  
import { myFunc } from './myModule';  
import * as myModule from './myOtherModule';
```

Thinking in Components

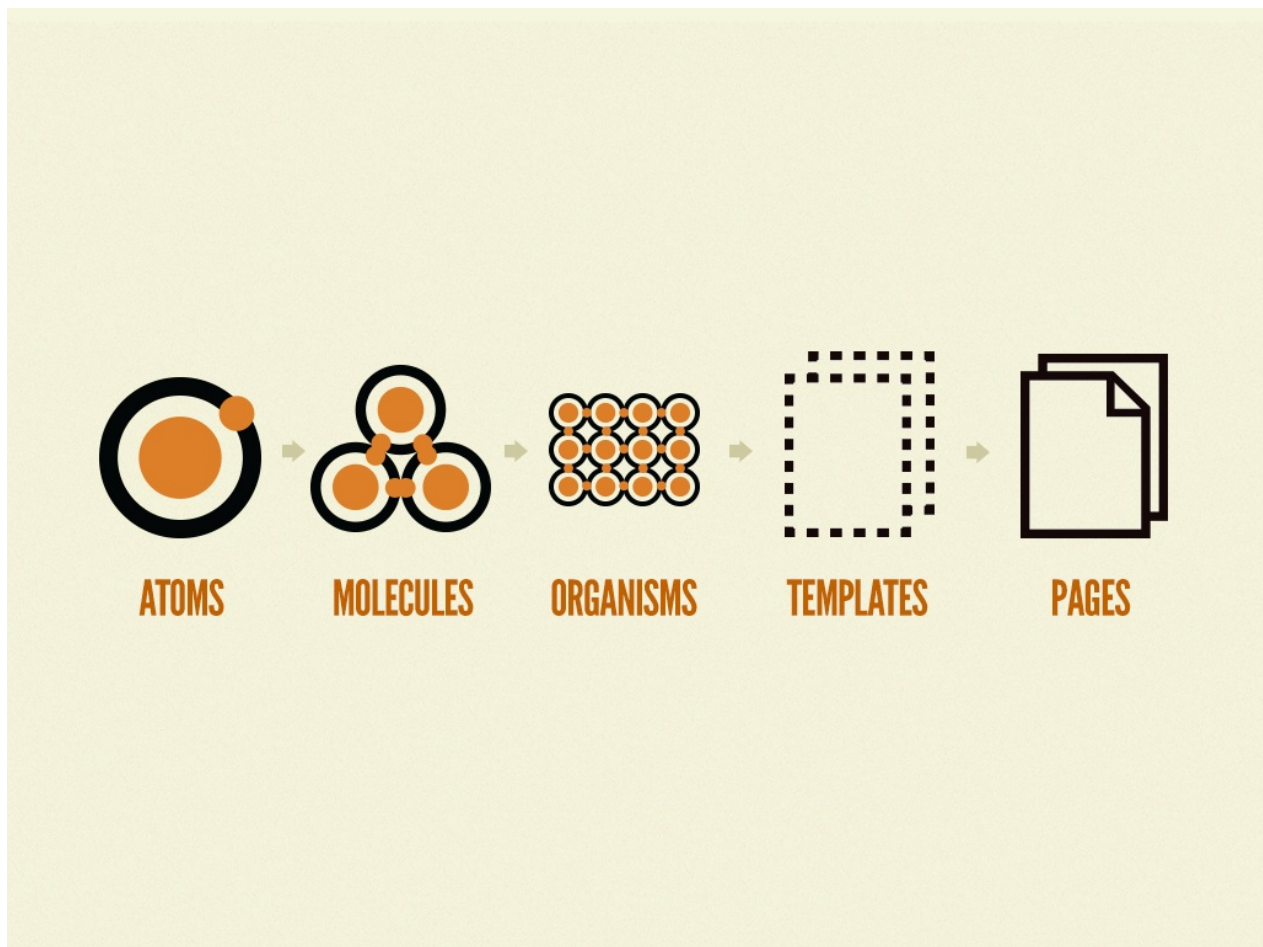


Atomic Design

<http://bradfrost.com/blog/post/atomic-web-design/>

| *We're not designing pages, we're designing systems of components* -- Stephen Hay

Atomic Design Principles



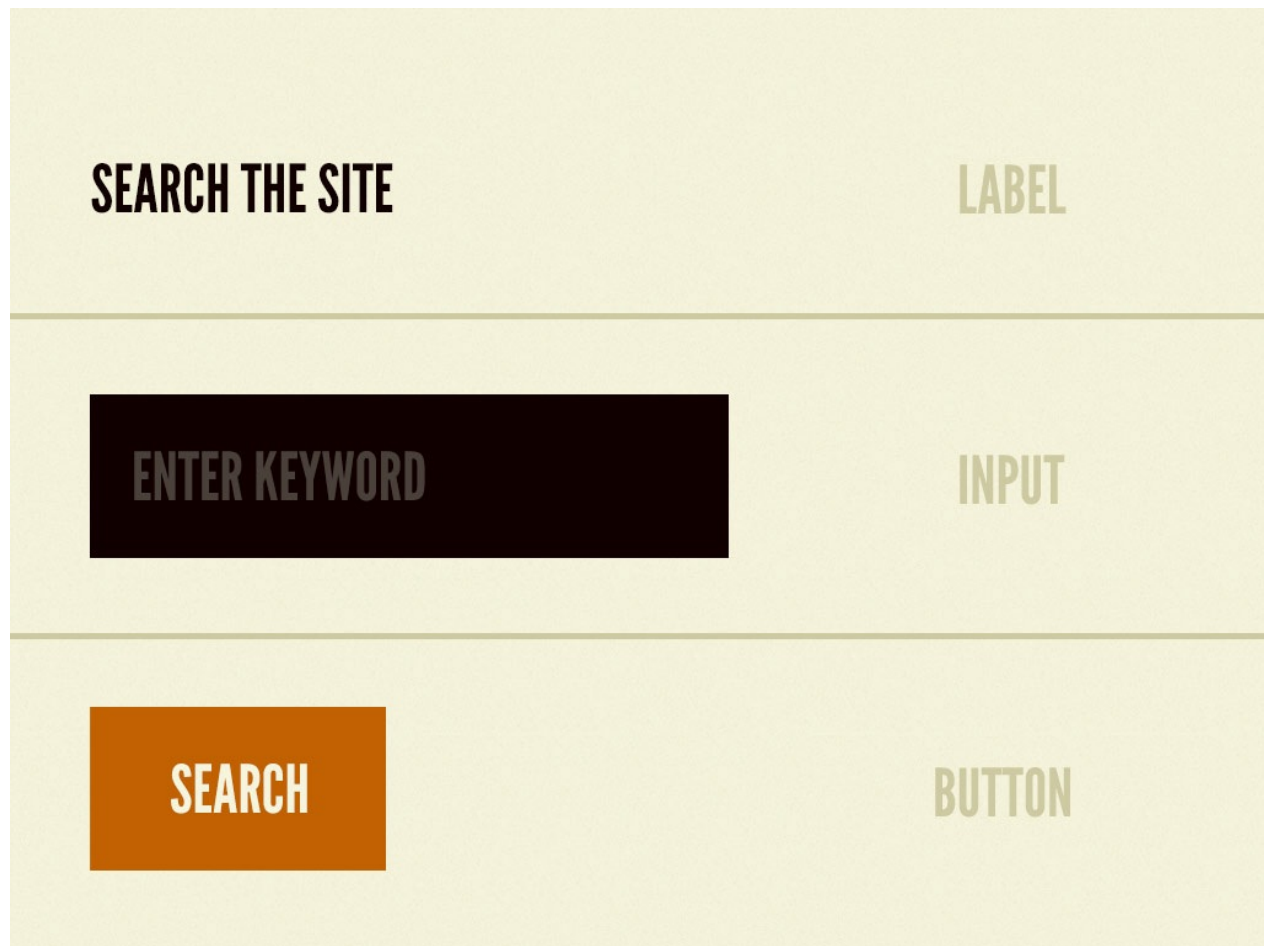
But we are designing a *single-page application*. We should never think of our interfaces as *Pages*.

Instead we should think in terms of *stateless, dynamic abstractions*, which are rendered based on the state provided

Atomic Component Principles



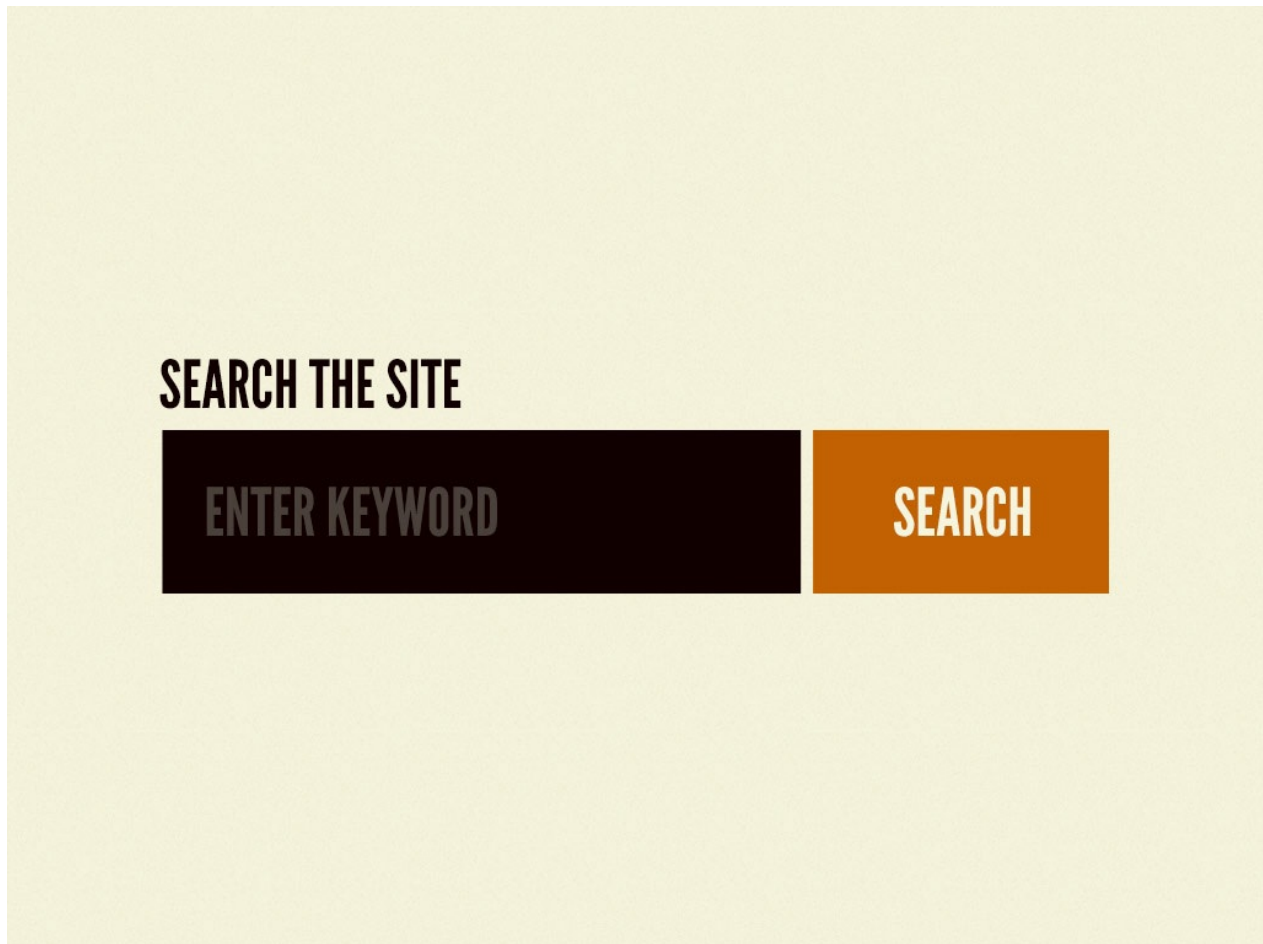
Atoms



- The simplest building block
- HTML tags
- Not very useful on their own
- Easily styled
- Very reusable
- Foundation of building a brand

```
<Form>
  <Label>Search</Label>
  <Input />
</Form>
```

Molecules

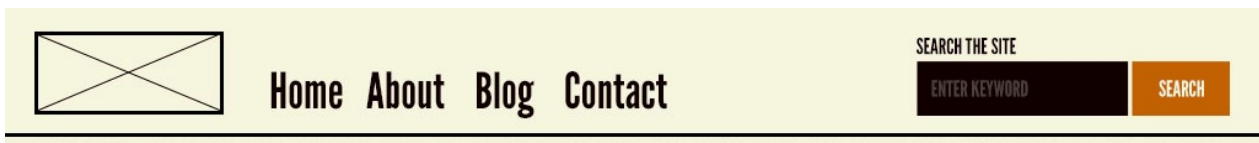


- Groups of Atoms bonded together
- Serve as backbone of design system
- For example, a Search Form
- Do one thing, do it well

```
<Form onSubmit={ onSubmit }>
  <Label>Search</Label>
  <Input type="text" value={ search } />

  <Button type="submit">Search</Button>
</Form>
```

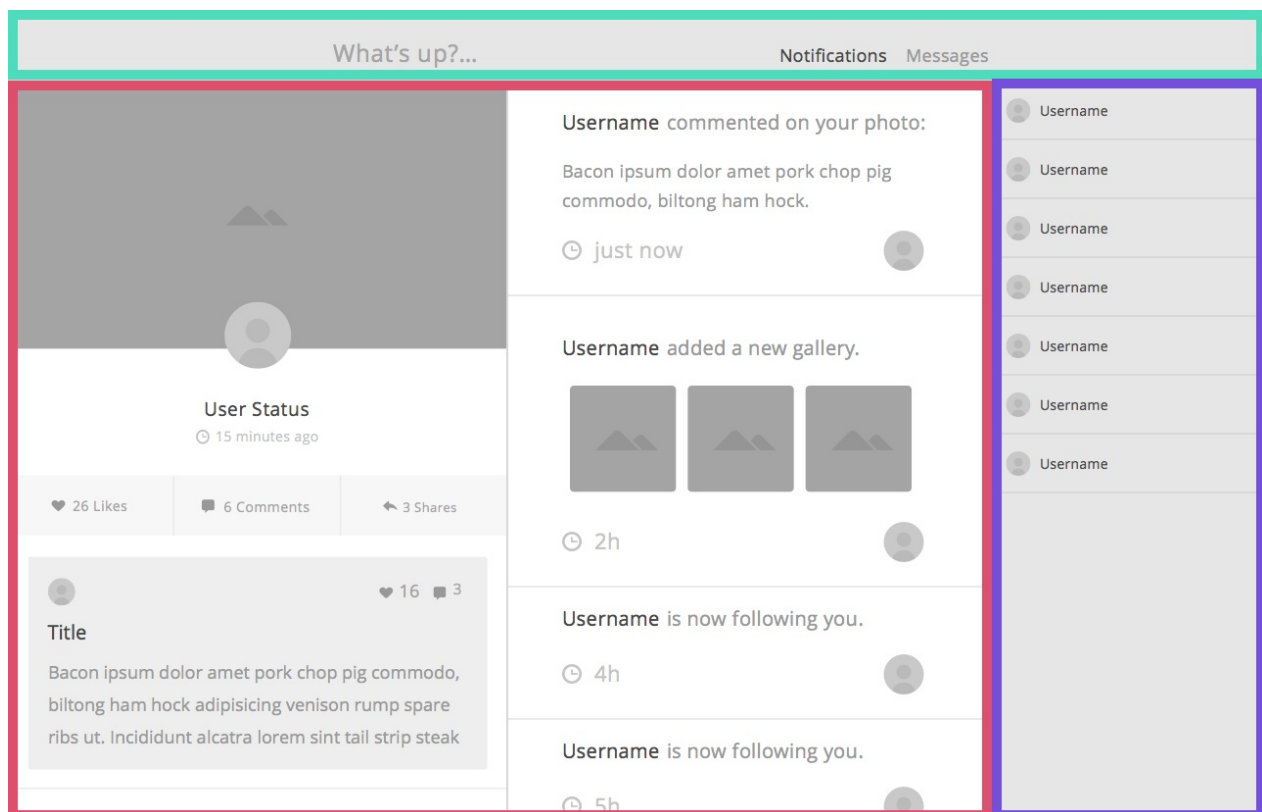
Organisms



- Groups of molecules
- Distinct section of an interface
- Portable, easily modified

```
<Header>
  <Navigator>
    <Brand />
    <NavItem to="home">Home</NavItem>
    <NavItem to="about">About</NavItem>
    <NavItem to="blog">Blog</NavItem>
  </Navigator>
  <SearchForm />
</Header>
```

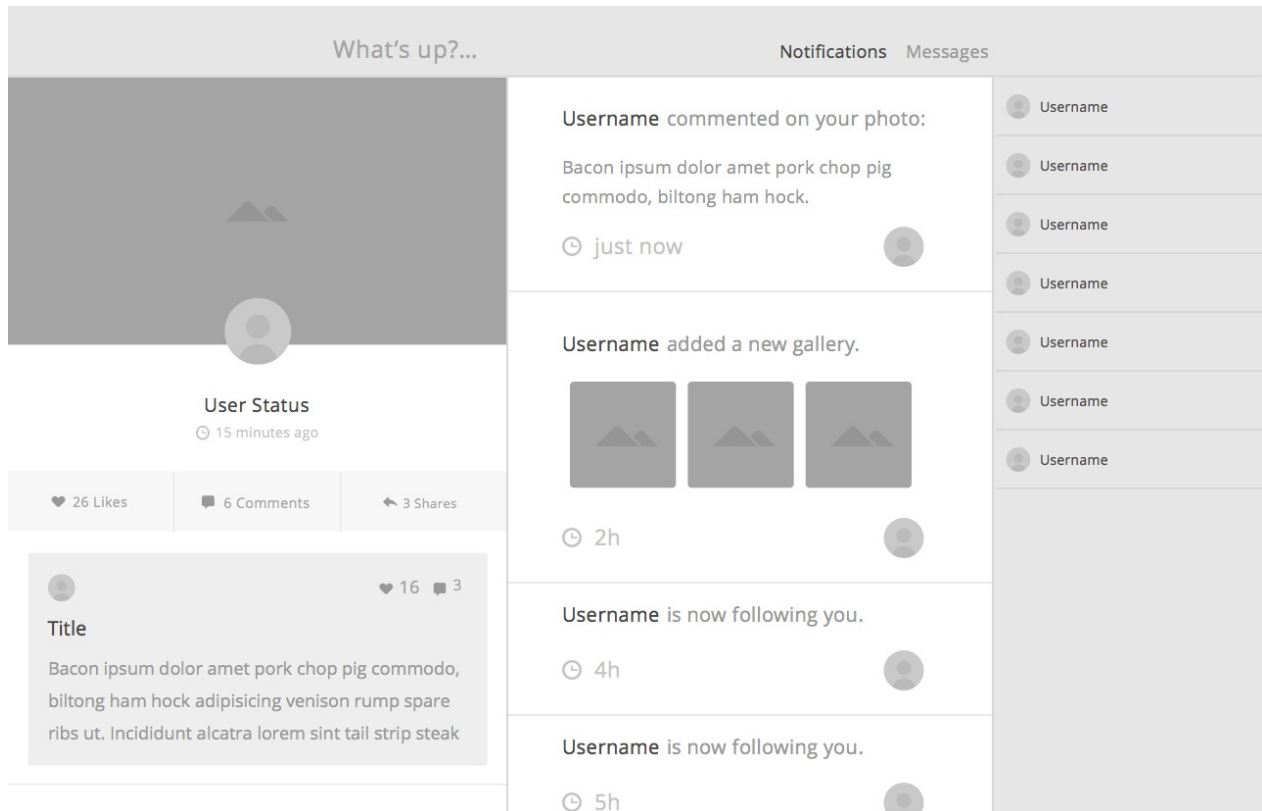
Ecosystem



- What the client will see

- Connected containers
- Many components that make up a view

Environment



- Root Component
- Typically the `<App />` component
- Represents everything packaged together as an application

Images from:

- Brad Frosts's article, [Atomic Design](#)
- Joey Di Nardo's article, [Atomic Components: Managing Dynamic React Components using Atomic Design—Part 1.](#)

Benefits of this Approach

- Single Responsibility Principle
- Modular components
- Build a Brand
 - Walmart Electrode

The Process

☐ Only show products in stock






| Name | Price |
|-----------------------|----------|
| Sporting Goods | |
| Football | \$49.99 |
| Baseball | \$9.99 |
| Basketball | \$29.99 |
| Electronics | |
| iPod Touch | \$99.99 |
| iPhone 5 | \$399.99 |
| Nexus 7 | \$199.99 |

Start with a Mock

- Break the UI into a hierarchy
- Single Responsibility Principle
- Think in terms of Information Architecture
- Atoms first, Molecules second

| Name | Price |
|-----------------------|----------|
| Sporting Goods | |
| Football | \$49.99 |
| Baseball | \$9.99 |
| Basketball | \$29.99 |
| Electronics | |
| iPod Touch | \$99.99 |
| iPhone 5 | \$399.99 |
| Nexus 7 | \$199.99 |

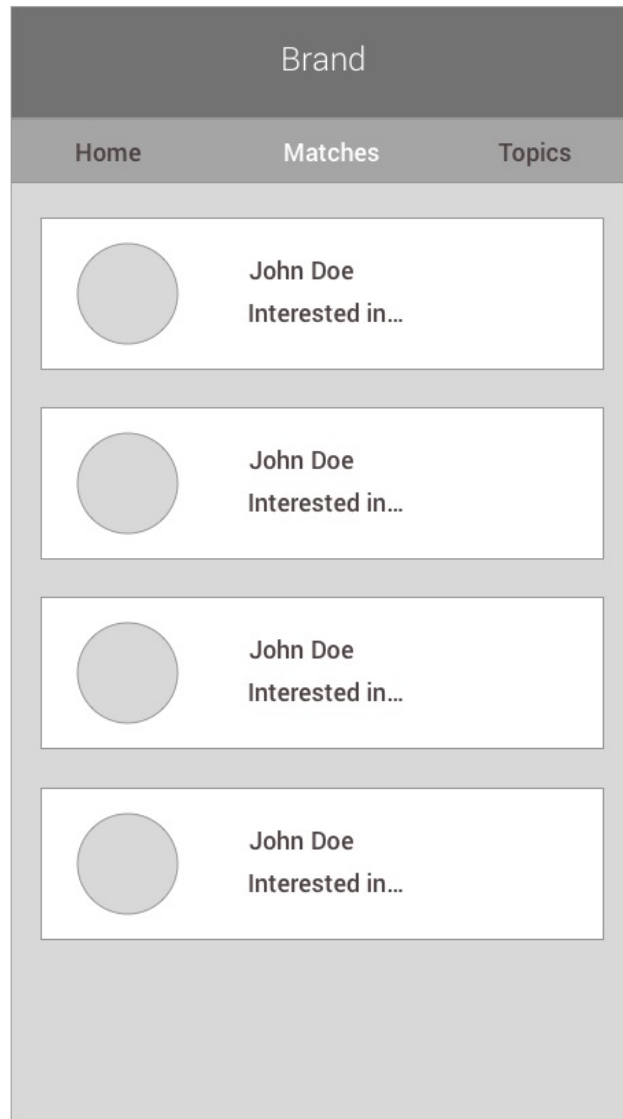
Component Hierarchy

1.  `FilterableProductTable` : contains the entirety of the example
2.  `SearchBar` : receives all user input
3.  `ProductTable` : displays and filters the data collection based on user input
4.  `ProductCategoryRow` : displays a heading for each category
5.  `ProductRow` : displays a row for each product

Task #1

Break Down the Mockup

Matches Page



Answer #1

Hover to see the answer.

- Header
- Navigator
- NavigatorLink

- Content
- ProfileCard
- ProfileImage

React Components

```
git checkout 02-component  
jspm install  
  
npm run dev
```

Open `src/index.js`

Stateless Components

- Super simple
- Given some state (as props)... return some DOM (or additional components)
- Pure

```
import React from 'react';

function HelloMessage({ name }) {
  return(
    <div>Hello {name}</div>
  );
};

HelloMessage.propTypes = { name: React.PropTypes.string };
HelloMessage.defaultProps = { name: 'World' };

ReactDOM.render(<HelloMessage name="Alice" />, mountNode);
```

Useful 95% of the time

Stateful Component

- Internal State
- Component Lifecycle Hooks

```
import React, { Component } from 'react';

class Profile extends Component {
  constructor(props) {
    super(props);
    this.state = { isLiked: false };
  }

  componentDidMount() {
    console.log('Stateful component successfully mounted.');
```

```
  }

  _toggleLikeState = () => {
    this.setState({
      isLiked: this.state.isLiked
    });
  }

  render() {
    const { name } = this.props;
    const { isLiked } = this.state;

    return (
      <div>
        <h3>{ name }</h3>
        <span>{ isLiked ? ' : ' } </span>
        <button onClick={ this._toggleLikeState }>
          Toggle Like
        </button>
      </div>
    );
  }
}

HelloMessage.propTypes = { name: React.PropTypes.string };
HelloMessage.defaultProps = { name: 'World' };

ReactDOM.render(<Profile name="Alice" />, mountNode);
```

Useful when...

- We need to have an internal state
- We need to perform an action when the component is mounted

Stateful vs Stateless Components

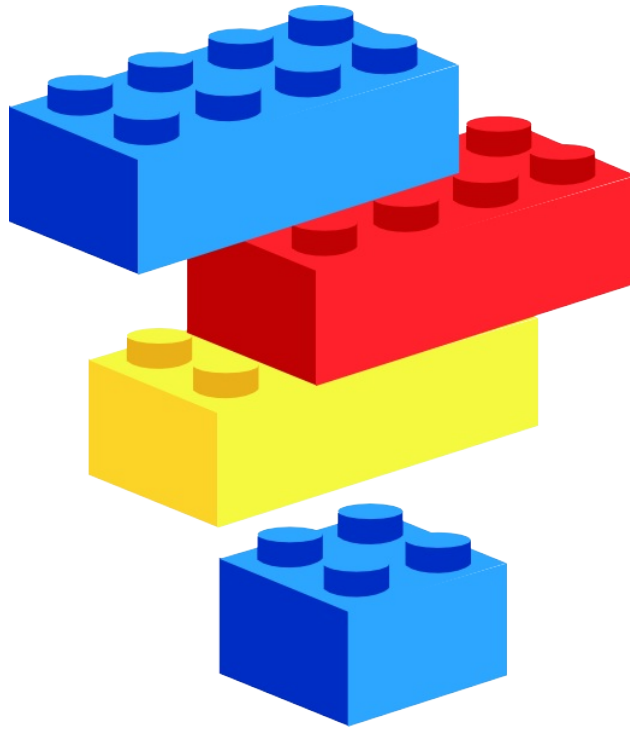
Stateful

- When you need internal state
 - D3 graph
- When you need to utilize a Component Lifecycle hook
 - Ajax request on mount
 - Setup animations
 - Access the raw DOM node for a 3rd party library

Stateless

95% of the time all you need is a stateless component.

Components are Bricks



- Think of them as Lego bricks
- Build small components that together build larger components, screens
- Composition is key!

Composition

We compose these bricks to build larger things

```
<!-- LoginForm.js -->

<Form>
  <FormGroup>
    <Label>Username</Label>
    <Input name="username" />

    <Label>Password</Label>
    <Input name="password" />
  </FormGroup>

  <Button>Submit</Button>
</Form>
```


Task #2

```
git checkout 02-component  
jspm install  
npm run dev
```

Create a component that renders a media object



- Should be made up of separate components:
 - `<ProfileImage />`
 - `<Card />`
- Content for the `<Card></Card>` should be provided as `children`

Task #3

Build a component which renders a list of profiles

- Use your previously created `<Card />` component
- Create a new stateful `<CardList />` component
- Store the array of profiles in state

```
[
  {
    id: 1,
    name: 'Jane React',
    description: 'Coolest person alive'
  },
  ...
]
```

Bonus task

Add a counter that lists the amount of people in the list

Task #4

Add a delete button to each profile

- The delete button should remove an item from the array
- Please use `Array().filter`

Bonus task

Add a checkbox to each item in the list, allow a user to clear all selected items.

Task #5

Add the ability to add a profile to the list

- Will require moving the state out of `<ProfileList />`
- Create a new container component called `<App />` which stores the state
- Don't forget to breakdown components!
 - `Label` , `Input` , `Form` , `ProfileForm`

Bonus task

Add ability to inline-edit each item.

Immutable.js

[Immutable.js](#) is a library that provides immutable generic collections.



What Is Immutability?

Immutability is a design pattern where something can't be modified after being instantiated. If we want to change the value of that thing, we must recreate it with the new value instead. Some JavaScript types are immutable and some are mutable, meaning their value can change without having to recreate it. Let's explain this difference with some examples:

```
let movie = {
  name: 'Star Wars',
  episode: 7
};

let myEp = movie.episode;

movie.episode = 8;

console.log(myEp); // outputs 7
```

As you can see in this case, although we changed the value of `movie.episode`, the value of `myEp` didn't change. That's because `movie.episode`'s type, `number`, is immutable.

```
let movie1 = {
  name: 'Star Wars',
  episode: 7
};

let movie2 = movie1;

movie2.episode = 8;

console.log(movie1.episode); // outputs 8
```

In this case however, changing the value of episode on one object also changed the value of the other. That's because `movie1` and `movie2` are of the **Object** type, and Objects are mutable.

Of the JavaScript built-in types, these are immutable:

- Boolean
- Number
- String
- Symbol
- Null

- Undefined

These are mutable:

- Object
- Array
- Function

String's an unusual case, since it can be iterated over using `for...of` and provides numeric indexers just like an array, but doing something like:

```
let message = 'Hello world';  
message[5] = '-';  
console.log(message); // writes Hello world
```

Note: This will throw an error in strict mode and fail silently in non-strict mode.

The Case for Immutability

One of the more difficult things to manage when structuring an application is managing its state. This is especially true when your application can execute code asynchronously. Let's say you execute some piece of code, but something causes it to wait (such as an http request or user input). After its completed, you notice the state its expecting changed because some other piece of code executed asynchronously and changed its value.

Dealing with that kind of behaviour on a small scale might be manageable, but this can show up all over an application and can be a real headache as the application gets bigger with more interactions and more complex logic.

Immutability attempts to solve this by making sure that any object that's been referenced in one part of the code can't all of a sudden be changed by another part of the code unless they have the ability to rebind it directly.

JavaScript solutions

Some new features have been added in ES6 that allow for easier implementation of immutable data patterns.

Object.assign

`Object.assign` lets us merge one object's properties into another one, replacing values of properties with matching names. We can use this to copy an object's values without altering the existing one.

```
let movie1 = {
  name: 'Star Wars',
  episode: 7
};

let movie2 = Object.assign({}, movie1);

movie2.episode = 8;

console.log(movie1.episode); // writes 7
console.log(movie2.episode); // writes 8
```

As you can see, although we have some way of copying an object, we haven't made it immutable, since we were able to set the episode's property to 8. Also, how do we modify the episode property in this case? We do that through the assign call:

```
let movie1 = {
  name: 'Star Wars',
  episode: 7
};

let movie2 = Object.assign({}, movie1, { episode: 8 });

console.log(movie1.episode); // writes 7
console.log(movie2.episode); // writes 8
```

Object.freeze

`Object.freeze` allows us to disable object mutation.

```
let movie1 = {
  name: 'Star Wars',
  episode: 7
};

let movie2 = Object.freeze(Object.assign({}, movie1));

movie2.episode = 8; // fails silently in non-strict mode,
                   // throws error in strict mode

console.log(movie1.episode); // writes 7
console.log(movie2.episode); // writes 7
```

One problem with this pattern however, is how much more verbose our code is and how difficult it is to read and understand what's actually going on with our data with all of the boilerplate calls to `Object.freeze` and `Object.assign`. We need some more sensible interface to create and interact with immutable data, and that's where `Immutable.js` fits in.

Note: `Object.freeze` is also very slow and should not be done with large arrays.

Immutable.js basics

To solve our mutability problem, Immutable.js needs to provide immutable versions of the two core mutable types, **Object** and **Array**.

Immutable.Map

`Map` is the immutable version of JavaScript's object structure. Due to JavaScript objects having the concise object literal syntax, it's often used as a key-value store with `key` being type `string`. This pattern closely follows the map data structure. Let's revisit the previous example, but use `Immutable.Map` instead.

```
import * as Immutable from 'immutable';

let movie1 = Immutable.Map({
  name: 'Star Wars',
  episode: 7
});

let movie2 = movie1;

movie2 = movie2.set('episode', 8);

console.log(movie1.get('episode')); // writes 7
console.log(movie2.get('episode')); // writes 8
```

Instead of binding the object literal directly to `movie1`, we pass it as an argument to `Immutable.Map`. This changes how we interact with `movie1`'s properties.

To *get* the value of a property, we call the `get` method, passing the property name we want, like how we'd use an object's string indexer.

To *set* the value of a property, we call the `set` method, passing the property name and the new value. Note that it *won't* mutate the existing Map object. It returns a new object with the updated property so we need to rebind the `movie2` variable to the new object.

Map.merge

Sometimes we want to update multiple properties. We can do this using the `merge` method.

```
let baseButton = Immutable.Map({
  text: 'Click me!',
  state: 'inactive',
  width: 200,
  height: 30
});

let submitButton = baseButton.merge({
  text: 'Submit',
  state: 'active'
});

console.log(submitButton);
// writes { text: 'Submit', state: 'active', width: 200, height: 30 }
```

Nested Objects

`Immutable.Map` wraps objects shallowly, meaning if you have an object with properties bound to mutable types then those properties can be mutated.

```
let movie = Immutable.Map({
  name: 'Star Wars',
  episode: 7,
  actors: [
    { name: 'Daisy Ridley', character: 'Rey'},
    { name: 'Harrison Ford', character: 'Han Solo' }
  ],
  mpaa: {
    rating: 'PG-13',
    reason: 'sci-fi action violence'
  }
});

movie.get('actors').pop();
movie.get('mpaa').rating = 'PG';

console.log(movie.toObject());

/* writes
{ name: 'Star Wars',
  episode: 7,
  actors: [ { name: 'Daisy Ridley', character: 'Rey' } ],
  mpaa: { rating: 'PG', reason: 'sci-fi action violence' } }
*/
```

To avoid this issue, use `Immutable.fromJS` instead.

```
let movie = Immutable.fromJS({
  name: 'Star Wars',
  episode: 7,
  actors: [
    { name: 'Daisy Ridley', character: 'Rey'},
    { name: 'Harrison Ford', character: 'Han Solo' }
  ],
  mpaa: {
    rating: 'PG-13',
    reason: 'sci-fi action violence'
  }
});

movie.get('actors').pop();
movie.get('mpaa').rating = 'PG';

console.log(movie.toObject());

/* writes
{ name: 'Star Wars',
  episode: 7,
  actors: List [ Map { "name": "Daisy Ridley", "character": "Rey" }, Map { "name": "Harrison Ford", "character": "Han Solo" } ],
  mpaa: Map { "rating": "PG-13", "reason": "sci-fi action violence" } }
*/
```

So let's say you want to modify `movie.mpaa.rating`, you might think of doing something like this: `movie = movie.get('mpaa').set('rating', 'PG')`. However, `set` will always return the calling Map instance which in this case returns the Map bound to the `mpaa` key rather than the movie you wanted. We need to use the `setIn` method to update nested properties.

```
let movie = Immutable.fromJS({
  name: 'Star Wars',
  episode: 7,
  actors: [
    { name: 'Daisy Ridley', character: 'Rey'},
    { name: 'Harrison Ford', character: 'Han Solo' }
  ],
  mpaa: {
    rating: 'PG-13',
    reason: 'sci-fi action violence'
  }
});

movie = movie
  .update('actors', actors => actors.pop())
  .setIn(['mpaa', 'rating'], 'PG')
  .update('actors', actors => actors.push({
    name: 'John Boyega',
    character: 'Finn'
  }));

console.log(movie.toObject());

/* writes
{ name: 'Star Wars',
  episode: 7,
  actors: List [ Map { "name": "Daisy Ridley", "character": "Rey" } , Map { "name": "John Boyega", "character": "Finn" }],
  mpaa: Map { "rating": "PG", "reason": "sci-fi action violence" } }
*/
```

We also added in a call to `Map.update` which, unlike `set`, accepts a function as the second argument instead of a value. This function accepts the existing value at that key and must return the new value of that key.

Deleting keys

Keys can be deleted from maps using the `Map.delete` and `Map.deleteIn` methods.

```
let movie = Immutable.fromJS({
  name: 'Star Wars',
  episode: 7,
  actors: [
    { name: 'Daisy Ridley', character: 'Rey'},
    { name: 'Harrison Ford', character: 'Han Solo' }
  ],
  mpaa: {
    rating: 'PG-13',
    reason: 'sci-fi action violence'
  }
});

movie = movie.delete('mpaa');

console.log(movie.toObject());

/* writes
{ name: 'Star Wars',
  episode: 7,
  actors: List [ Map { "name": "Daisy Ridley", "character": "Rey" }, Map { "name": "Harrison Ford", "character": "Han Solo" } ] }
*/
```

Maps are iterable

Maps in Immutable.js are *iterable*, meaning that you can `map`, `filter`, `reduce`, etc. each key-value pair in the map.

```
let features = Immutable.Map({
  'send-links': true,
  'send-files': true,
  'local-storage': true,
  'mirror-notifications': false,
  'api-access': false
});

let myFeatures = features.reduce((providedFeatures, provided, feature) => {
  if(provided)
    providedFeatures.push(feature);

  return providedFeatures;
}, []);

console.log(myFeatures); // [ 'send-links', 'send-files', 'local-storage' ]
```

```
const mapMap = Immutable.Map({ a: 0, b: 1, c: 2 });
mapMap.map(i => i * 30);

const mapFilter = Immutable.Map({ a: 0, b: 1, c: 2 });

mapFilter.filter(i => i % 2);

const mapReduce = Immutable.Map({ a: 10, b: 20, c: 30 });

mapReduce.reduce((acc, i) => acc + i, 0);
```

Immutable.List

`List` is the immutable version of JavaScript's array structure.

```
let movies = Immutable.fromJS([ // again use fromJS for deep immutability
  {
    name: 'The Fellowship of the Ring',
    released: 2001,
    rating: 8.8
  },
  {
    name: 'The Two Towers',
    released: 2002,
    rating: 8.7
  }
]);

movies = movies.push(Immutable.Map({
  name: 'The Return of the King',
  released: 2003
}));

movies = movies.update(2, movie => movie.set('rating', 8.9)); // 0 based

movies = movies.zipWith(
  (movie, seriesNumber) => movie.set('episode', seriesNumber),
  Immutable.Range(1, movies.size + 1) // size property instead of length
);

console.log(movies);
/* writes
List [
  Map { "name": "The Fellowship of the Ring", "released": 2001, "rating": 8.8, "episode": 1 },
  Map { "name": "The Two Towers", "released": 2002, "rating": 8.7, "episode": 2 },
  Map { "name": "The Return of the King", "released": 2003, "rating": 8.9, "episode": 3 } ]
*/
```

Here we use the `Immutable.fromJS` call again since we have objects stored in the array. We call `push` to add items to the list, just like we would call it on an array but since we're creating a new copy we need to rebind the variable. We have the same `set` and `update` calls when we want to update items at specific indexes. We also have access to array functions like `map`, `reduce` with support for extras like the one we're using here, `zipWith`.

Performance

Due to having to allocate memory and having to copy the data structure whenever any change is made, this can potentially lead to a large number of extra operations having to be performed depending on what type of changes are made and how many of them. To demonstrate the difference, here is a [test run](#). Doing memory allocation and copy on large strings can be expensive even on a shallow object.

Persistent and transient data structures

Immutable data structures are also sometimes referred to as **persistent data structures**, since its values persist for its lifetime. Immutable.js provides the option for **transient data structures**, which is a mutable version of a persistent data structure during a transformation stage and returning a new immutable version upon completion. This is one approach to solving the performance issues we encountered earlier. Let's revisit the immutable case outlined in the performance example, but using a transient data structure this time:

```
import * as Immutable from 'immutable';

let list = Immutable.List();

list = list.withMutations(mutableList => {
  let val = "";

  return Immutable.Range(0, 1000000)
    .forEach(() => {
      val += "concatenation";
      mutableList.push(val);
    });
});

console.log(list.size); // writes 1000000
list.push('');
console.log(list.size); // writes 1000000
```

As we can see in [this performance test](#), the transient list builder was still a lot slower than the fully mutable version, but much faster than the fully immutable version. Also, if you pass the mutable array to `Immutable.List` or `Immutable.fromJS`, you'll find the transient version closes the performance gap. The test also shows how slow `Object.freeze` can be compared to the other 3.

Official documentation

For more information on Immutable.js, visit the official documentation at <https://facebook.github.io/immutable-js/>.

Let's Try It Out

Immutable Repl

neilff.github.io/immutable-repl

Dataset

For all the tasks in this section use this as the initial dataset.

```
[
  {
    "_id": "56e18ce608c0a0190da963f8",
    "index": 0,
    "guid": "5e0dbf88-33f1-4b84-bdca-ac21719bf0e8",
    "isActive": false,
    "balance": "$1,284.82",
    "picture": "http://placeholder.it/32x32",
    "age": 36,
    "eyeColor": "blue",
    "name": {
      "first": "Lauren",
      "last": "Stanley"
    },
    "company": "HAIRPORT",
    "email": "lauren.stanley@hairport.name",
    "phone": "+1 (876) 425-2958",
    "address": "456 Front Street, Wacissa, Virginia, 9236",
    "about": "Dolor aliqua enim irure mollit. Sunt ullamco laborum reprehenderit labor e. Eu consequat laborum consectetur voluptate laborum fugiat quis tempor amet nulla. Irure duis reprehenderit irure officia sit magna deserunt. Incididunt eu aliquip proident id amet enim dolor reprehenderit ut ipsum est elit ea.",
    "registered": "Friday, August 8, 2014 4:08 PM",
    "latitude": "41.628375",
    "longitude": "104.950835",
    "tags": [
      7,
      "veniam"
    ],
    "range": [
      0,
      1,
      2,
      3,
      4,
      5,
      6,

```

```
    7,  
    8,  
    9  
  ],  
  "friends": [  
    3,  
    {  
      "id": 1,  
      "name": "Mccall Petersen"  
    }  
  ],  
  "greeting": "Hello, Lauren! You have 9 unread messages.",  
  "favoriteFruit": "banana"  
},  
{  
  "_id": "56e18ce6dc7d5ade1e3c7889",  
  "index": 1,  
  "guid": "7ceca65c-cc8d-4f88-ab00-b5d00b72e27f",  
  "isActive": true,  
  "balance": "$1,423.68",  
  "picture": "http://placeholder.it/32x32",  
  "age": 35,  
  "eyeColor": "brown",  
  "name": {  
    "first": "Schmidt",  
    "last": "Floyd"  
  },  
  "company": "ANIXANG",  
  "email": "schmidt.floyd@anixang.org",  
  "phone": "+1 (913) 595-3119",  
  "address": "274 Norfolk Street, Freeburn, Nevada, 1869",  
  "about": "Exercitation deserunt quis commodo ad qui aliqua proident mollit labore  
mollit. Deserunt occaecat in pariatum mollit aute consequat reprehenderit in deserunt  
magna ad. Aliquip labore do mollit officia laboris in aliquip magna aliqua. Sunt occae  
cat eiusmod ea amet dolore consectetur aute consequat adipisicing et nisi fugiat. Aute  
eiusmod quis dui ipsum occaecat culpa eiusmod Lorem amet laborum occaecat adipisicin  
g minim. Labore exercitation laborum sint enim veniam labore officia. Aliquip do esse  
consectetur amet.",  
  "registered": "Sunday, October 12, 2014 8:17 AM",  
  "latitude": "-3.271053",  
  "longitude": "-124.321634",  
  "tags": [  
    7,  
    "veniam"  
  ],  
  "range": [  
    0,  
    1,  
    2,  
    3,  
    4,  
    5,  
    6,
```

```

    7,
    8,
    9
  ],
  "friends": [
    3,
    {
      "id": 1,
      "name": "Mccall Petersen"
    }
  ],
  "greeting": "Hello, Schmidt! You have 9 unread messages.",
  "favoriteFruit": "apple"
},
{
  "_id": "56e18ce603784459df38b06c",
  "index": 2,
  "guid": "b19ffa1d-ca97-4e94-809e-3bf82df7fd40",
  "isActive": true,
  "balance": "$2,420.16",
  "picture": "http://placeholder.it/32x32",
  "age": 30,
  "eyeColor": "blue",
  "name": {
    "first": "Jane",
    "last": "Wheeler"
  },
  "company": "DIGINETIC",
  "email": "jane.wheeler@diginetic.co.uk",
  "phone": "+1 (826) 545-3381",
  "address": "385 Morgan Avenue, Manila, Puerto Rico, 8503",
  "about": "Dolore velit dolor exercitation non voluptate cillum aliquip excepteur. Eiusmod mollit et nostrud pariatur amet reprehenderit deserunt elit ex. Do adipisicing qui pariatur cupidatat ut sint proident incidunt ipsum. Reprehenderit aliquip elit labore mollit consequat ipsum est sunt culpa. Est incidunt qui ea incidunt. Exercitation pariatur laborum sit occaecat sint ea eiusmod et Lorem amet in magna elit. Eu veniam eu qui laborum eiusmod esse ullamco ipsum proident exercitation et exercitation officia.",
  "registered": "Saturday, July 4, 2015 9:47 PM",
  "latitude": "-5.955075",
  "longitude": "37.129517",
  "tags": [
    7,
    "veniam"
  ],
  "range": [
    0,
    1,
    2,
    3,
    4,
    5,
    6,

```

```
7,  
8,  
9  
],  
"friends": [  
  3,  
  {  
    "id": 1,  
    "name": "Mccall Petersen"  
  }  
],  
"greeting": "Hello, Jane! You have 10 unread messages.",  
"favoriteFruit": "apple"  
},  
{  
  "_id": "56e18ce6adf25f0905c47a64",  
  "index": 3,  
  "guid": "d9547c25-8437-48d3-b3d6-ef890343b843",  
  "isActive": false,  
  "balance": "$2,059.14",  
  "picture": "http://placeholder.it/32x32",  
  "age": 29,  
  "eyeColor": "green",  
  "name": {  
    "first": "Brennan",  
    "last": "Santos"  
  },  
  "company": "SPEEDBOLT",  
  "email": "brennan.santos@speedbolt.com",  
  "phone": "+1 (964) 417-3448",  
  "address": "327 Bills Place, Strong, Maryland, 4414",  
  "about": "Et dolor sit eiusmod eu labore velit. Laboris veniam consequat eiusmod a  
liqua ex in adipisicing deserunt quis eiusmod ullamco ut reprehenderit. Velit reprehen  
derit elit cupidatat laborum consequat ipsum quis consequat dolor magna sit nostrud. L  
aborum et minim irure ad elit dolore eu amet. Esse elit ex officia sit culpa pariatur  
nostrud anim sint nostrud culpa eiusmod non qui. Cupidatat ea dolor dolor ea pariatur  
et deserunt consequat est incididunt sit voluptate ipsum nostrud. Elit quis deserunt e  
st in qui sunt nulla ut.",  
  "registered": "Thursday, June 5, 2014 2:35 AM",  
  "latitude": "22.827405",  
  "longitude": "-50.704291",  
  "tags": [  
    7,  
    "veniam"  
  ],  
  "range": [  
    0,  
    1,  
    2,  
    3,  
    4,  
    5,  
    6,
```



```
    7,  
    8,  
    9  
  ],  
  "friends": [  
    3,  
    {  
      "id": 1,  
      "name": "Mccall Petersen"  
    }  
  ],  
  "greeting": "Hello, Brennan! You have 9 unread messages.",  
  "favoriteFruit": "strawberry"  
},  
{  
  "_id": "56e18ce671021dc16753b56d",  
  "index": 4,  
  "guid": "725fb6f9-d900-4764-8f41-7fe2779b2dc9",  
  "isActive": false,  
  "balance": "$2,399.10",  
  "picture": "http://placeholder.it/32x32",  
  "age": 37,  
  "eyeColor": "blue",  
  "name": {  
    "first": "Perez",  
    "last": "Turner"  
  },  
  "company": "CENTICE",  
  "email": "perez.turner@centice.us",  
  "phone": "+1 (855) 446-3306",  
  "address": "596 Varick Street, Genoa, Arkansas, 6957",  
  "about": "Veniam est dolor laboris eiusmod. Nostrud dui est nostrud aliquip in la  
borum qui culpa. Sunt mollit adipisicing amet laboris esse.",  
  "registered": "Monday, September 8, 2014 11:25 PM",  
  "latitude": "23.65985",  
  "longitude": "-65.321713",  
  "tags": [  
    7,  
    "veniam"  
  ],  
  "range": [  
    0,  
    1,  
    2,  
    3,  
    4,  
    5,  
    6,  
    7,  
    8,  
    9  
  ],  
  "friends": [  

```

```
    3,  
    {  
      "id": 1,  
      "name": "Mccall Petersen"  
    }  
  ],  
  "greeting": "Hello, Perez! You have 5 unread messages.",  
  "favoriteFruit": "banana"  
}  
]
```

Task #1

Generate a list of the names of people

Expected Result

```
[
  {
    first: 'Lauren',
    last: 'Stanley'
  },
  {
    first: 'Schmidt',
    last: 'Floyd'
  },
  {
    first: 'Jane',
    last: 'Wheeler'
  },
  {
    first: 'Brennan',
    last: 'Santos'
  },
  {
    first: 'Perez',
    last: 'Turner'
  }
]
```

Answer

Map over the Immutable list and then use the *get* method to access the *name* prop.

Task #2

Generate a list of last names of people

Expected Result

```
[  
  'Stanley',  
  'Floyd',  
  'Wheeler',  
  'Santos',  
  'Turner',  
]
```

Answer

Map over the Immutable list and then use the *getIn* method to access the *name.last* prop.

Task #3

Get the last name of the last person

Expected Result

```
'Turner'
```

Answer

Use *last* to get the last item in an Immutable list. Then use the *getIn* method to access the *name.last* prop.

Task #4

Generate a list of users with brown eyes

Expected Result

```
[
  {
    "_id": "56e18ce6dc7d5ade1e3c7889",
    "index": 1,
    "guid": "7ceca65c-cc8d-4f88-ab00-b5d00b72e27f",
    "isActive": true,
    "balance": "$1,423.68",
    "picture": "http://placeholder.it/32x32",
    "age": 35,
    "eyeColor": "brown",
    "name": {
      "first": "Schmidt",
      "last": "Floyd"
    },
    "company": "ANIXANG",
    "email": "schmidt.floyd@anixang.org",
    "phone": "+1 (913) 595-3119",
    "address": "274 Norfolk Street, Freeburn, Nevada, 1869",
    "about": "Exercitation deserunt quis commodo ad qui aliqua proident mollit labore mollit. Deserunt occaecat in pariatum mollit aute consequat reprehenderit in deserunt magna ad. Aliquip labore do mollit officia laboris in aliquip magna aliqua. Sunt occaecat eiusmod ea amet dolore consectetur aute consequat adipisicing et nisi fugiat. Aute eiusmod quis dui ipsum occaecat culpa eiusmod Lorem amet laborum occaecat adipisicing minim. Labore exercitation laborum sint enim veniam labore officia. Aliquip do esse consectetur amet.",
    "registered": "Sunday, October 12, 2014 8:17 AM",
    "latitude": "-3.271053",
    "longitude": "-124.321634",
    "tags": [
      7,
      "veniam"
    ],
    "range": [
      0,
      1,
      2,
      3,
      4,
      5,
      6,
      7,
      8,
    ]
  }
]
```

```
    9
  ],
  "friends": [
    3,
    {
      "id": 1,
      "name": "Mccall Petersen"
    }
  ],
  "greeting": "Hello, Schmidt! You have 9 unread messages.",
  "favoriteFruit": "apple"
}
]
```

Answer

Use Immutable list's *filter* method.

Task #5

Generate a **Map** of the eye colors to count of people with that color

Expected Result

```
{  
  blue: 3,  
  brown: 1,  
  green: 1  
}
```

Answer

Use the Immutable list's *groupBy* method to generate groups of people by eye color. Then use *map* to generate counts. Or just use the List's *reduce* method.

Task #6

Add your own profile to the end of the list

Answer

Use the Immutable list's *push* method.

Task #7

Change the name of the first person to Marty Robbins

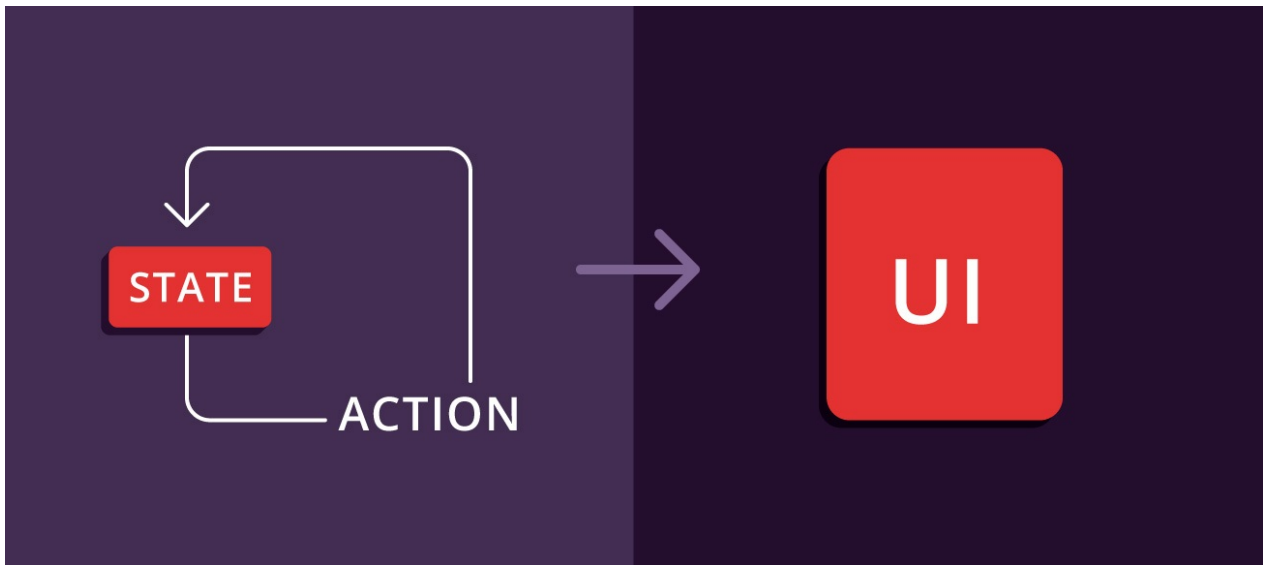
Answer

Use the Immutable list's `set()` method.

Redux

Redux is an application state manager for JavaScript applications, and keeps with the core principals of flux-architecture by having a unidirectional data flow in your application.

How it differs from traditional Flux though, is that instead of multiple stores, you have one global application state. The state is calculated and returned in the reducer. The state management is held else where.



Three Principles of Redux

1. Single Source of Truth

- Entire state is stored in an object tree
 - Easy to debug
 - Easy to inspect application
 - Easy to hydrate initial state

2. State Is Read Only

- Only way to mutate state is to emit an action
 - Actions describe what happened
 - Views, network callbacks, etc. will *never* mutate state
 - Mutations are centralized and happen in strict order
 - No race conditions
 - Actions are objects, they can be logged, serialized, stored, and replayed

3. Changes Are Made With Pure Functions

- Reducers are responsible for modifying the state tree
 - Pure functions
 - Take in previous state, action, and return new state
 - Can be split out into smaller reducers to manage specific parts of state tree

Resources

- [Redux Documentation](#)
- [React-Redux - React Bindings for Redux](#)
- [React Redux Starter Kit](#)
- [Getting Started with Redux - Egghead.io](#)
- [Idiomatic Redux - Egghead.io](#)

Quick Review of Reducers and Pure Functions

One of the core concepts of Redux is the reducer. A reducer is simply a pure function that iterates over a collection of values, and returns a new single value at the end of it.

The simplest examples of a reducer, is a sum function:

```
const x = [1,2,3].reduce((value,state) => value + state, 0)
// x == 6
```

Redux Reducers

While a very simple idea, it is very powerful. With Redux, you replay a series of events into the reducer - and get your new application state as a result.

Reducers in a Redux application should not mutate the state, but return a copy of it, and be side-effect free. Lets take a look at a simple counter reducer.

Simple Reducer

```
const INCREMENT_COUNTER = 'INCREMENT_COUNTER';
const DECREMENT_COUNTER = 'DECREMENT_COUNTER';

export default function counter(state = 0, action) {

  switch (action.type) {
    case INCREMENT_COUNTER:
      return state + 1;
    case DECREMENT_COUNTER:
      return state - 1;
    default:
      return state;
  }
}
```

We can see here that we are passing in an initial state, and an action. To handle each action - we have setup a switch statement. Instead of each reducer needing to explicitly subscribe to the dispatcher - **every action gets passed into every reducer, which handles the action it is interested in or otherwise returns the state along to the next reducer.**

Reducers in Redux should be side-effect free, that means that they should not modify things outside of the application state. Instead, they should reflect the state of the application. This is why side-effect causing operations, such as updating a record in a database, generating an id, etc should be handled elsewhere in the application - such as in the action creators, or middleware.

Another consideration when creating your reducers, is to ensure that they are immutable and not modifying the state of your application. If you mutate your application state, it can cause unexpected behavior. There are a few ways to help maintain immutability in your reducers. One, is using new ES6 features such as the spread operator for objects and arrays.

```
let immutableObjectReducer = (state = { someValue: 'value' } , action) => {
  switch(action.payload) {
    case SOME_ACTION:
      return Object.assign({}, state, { someValue: action.payload.value });
    default:
      return state;
  }
}

let immutableArrayReducer = (state = [1,2,3], action) => {
  switch(action.payload) {
    case ADD_ITEM:
      return [...state, action.payload.value]
    break;
    default:
      return state;
  }
}
```

However, if dealing with complex or deeply nested objects - it can be difficult to maintain immutability in your application using this syntax. This is where a library like ImmutableJS can help.

Redux Actions

Redux actions, generally should return a simple JSON object. This is because they should be serialized and replayable into the application state. Even if your actions need to return promises, the final dispatched action should remain a plain JSON object.

Redux actions are generally where side-effects should happen, such as making API calls, or generating ID's. This is because when the final action gets dispatched to the reducers, we want to update the application state to reflect what has already happened.

Lets take a look at the actions that are used in this example. For now, lets just focus on the synchronous actions.

Synchronous Actions

```
const INCREMENT_COUNTER = 'INCREMENT_COUNTER';
const DECREMENT_COUNTER = 'DECREMENT_COUNTER';

export function increment() {
  return {
    type: INCREMENT_COUNTER
  };
}

export function decrement() {
  return {
    type: DECREMENT_COUNTER
  };
}
```

As you can see, the actions creators are simple functions that take parameters, and return a JSON object containing more information. Actions follows the [Flux Standard Action](#) and contain the properties:

- **type**: a string/enum representing the action
- **payload**: the data that you want to pass into the reducer if applicable
- **meta?** : optional - any extra information

When using Redux, libraries like react-redux will take care of wrapping your actions into the dispatch so they will get passed off to the store appropriately.

Asynchronous Actions

To do async operations, or have actions that return something other than a plain JSON object, you need to register a middleware with redux. For our examples, we can use the `thunk` middleware, and setting this up is covered later in the training. For now, all you need to know is that once you register a middleware with redux, you can make `dispatch` and `getState` available to your actions. To show how these are used, lets take a look at the `incrementIfOdd` and `increaseAsync` actions.

```
// ...
export function incrementIfOdd() {
  return (dispatch, getState) => {
    const { counter } = getState();

    if (counter % 2 === 0) {
      return;
    }

    dispatch(increment());
  };
}

const delay = (timeInMs) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve() , timeInMs);
  });
}

export function incrementAsync(timeInMs = 1000) {
  return dispatch => {
    delay(timeInMs).then(() => dispatch(increment()));
  };
}
```

In the `incrementIfOdd` action, we are making use of the `getState` function to get the current state of the application.

In the `incrementAsync` action, we are making use of `dispatch`. For example, we have created a `Promise` that will resolve after the delay. Once the `Promise` resolves, we can then do a `dispatch` with the `increase` action. However, this promise could also be an API call, with the dispatched action containing the result of the API call.

Configuring Your Application to Use Redux

Once you have the reducers and actions created, it is time to configure your application to make use of Redux. For this, we will need to:

- Create our application reducer
- Create and configure a store
- Subscribe to the store and update the view

Create Our Application Reducer

```
import { createStore, combineReducers } from 'redux';

export const INCREASE = '@@reactTraining/INCREASE';
export const DECREASE = '@@reactTraining/DECREASE';

const INITIAL_STATE = 0;

function counterReducer(state = INITIAL_STATE, action = {}) {
  switch (action.type) {
    case INCREASE:
      return state + 1;

    case DECREASE:
      return state - 1;

    default:
      return state;
  }
}

const rootReducer = combineReducers({ counter: counterReducer });
```

What `combineReducers` does, is allows us to break out our application into smaller reducers with a single area of concern. **Each reducer that you pass into it, will become a property on the state. So when we are subscribing to our state changes, we will be passed in a state object with a property counter, and any other reducers you have provided.**

Create and Configure a Store

When creating a store in redux, this is where you provide the middleware you want to use, and the reducer that you want to have for your application.

```
import { createStore, applyMiddleware, compose } from 'redux';
import thunk from 'redux-thunk';

//...

const store = compose(
  applyMiddleware(
    thunk
  ),
)(rootReducer);
```

In this example, we are creating a store that is using the `thunk` middleware, which will allow our actions to return non-JSON objects such as promises. We could also use other middlewares such as `redux-logger`, which will provides some logging functionality to the application.

Subscribe to State Changes

Now that we have created our state reducer, and created a store. We now need to subscribe to the store and update our view with the latest state.

```
//...
store.subscribe(() => {
  ReactDOM.render(
    <div>
      <pre>{ JSON.stringify(store.getState(), null, 2) }</pre>
      <button onClick={ () => store.dispatch(increase()) }>Increase</button>
      <button onClick={ () => store.dispatch(decrease()) }>Decrease</button>
    </div>,
    document.getElementById('root')
  );
});

store.dispatch({ type: 'INIT' });
```

Full Example

```
git checkout 04-redux
jspm install
npm run dev
```


Using Redux With Components

Instead of having to manage the store subscriptions manually we can use react-redux to connect our store to React components. To demonstrate how this works, let's take a look at the counter example.

Counter Example

We start by building out a counter component. The component will be responsible for keeping track of how many times it was clicked, and displaying the amount.

```
import React from 'react';
import { connect } from 'react-redux';
import { increase, decrease } from '../reducers/counter';

function mapStateToProps(state) {
  return {
    counter: state.counter,
  };
}

function mapDispatchToProps(dispatch) {
  return {
    onIncrease: () => dispatch(increase()),
    onDecrease: () => dispatch(decrease()),
  };
}

const Counter = ({ onIncrease, onDecrease, counter }) => {
  return (
    <div>
      <pre>{ counter }</pre>
      <button onClick={ onIncrease }>Increase</button>
      <button onClick={ onDecrease }>Decrease</button>
    </div>
  );
};

const App = connect(
  mapStateToProps,
  mapDispatchToProps,
)(Counter);

ReactDOM.render(
  App,
  document.getElementById('root')
);
```

The template syntax should be familiar by now, displaying a counter value, and handling some click events. Lets take a look at the use of `connect` .

- `mapStateToProps` : `connect` will subscribe to Redux store updates. Any time it updates, `mapStateToProps` will be called. It's result must be a plain object. Which will then be passed to the component as props.
- `mapDispatchToProps` : Optional. The store's `dispatch` method is passed in as an argument here. You can then use that to wrap your actions or pass `dispatch` into them. The result of this function is also passed into the component as props. *Tip*: you could use the `bindActionCreators()` helper from Redux to simplify this.

Full Example

```
git checkout 04a-redux  
jspm install  
npm run dev
```

Redux and Component Architecture

In the previous example, our `counter` component is a smart component. It knows about redux, the structure of the state, and the actions it needs to call. While in theory you can drop this component into any area of your application and have it just work. But, it will be tightly bound to that specific slice of state, and those specific actions. For example, what if we wanted to have multiple counters tracking different things on the page? For example, counting the number of red clicks vs blue clicks.

To help make components more generic and reusable, it is worth considering smart component, or container components - and dumb components.

| | Container Components | Presentational Components |
|-----------------------|---------------------------|-----------------------------|
| Location | Top level, route handlers | Middle and leaf components |
| Aware of Redux | Yes | No |
| To read data | Subscribe to Redux state | Read data from props |
| To change data | Dispatch Redux actions | Invoke callbacks from props |

[Redux Docs](#)

Routing in React

This section will look at setting up [React Router](#).



React Router

Setup

```
npm install --save react-router
```

Routing

[React router](#) is the root component rendered, which will render the app components as children based on the url path.

React router has configuration components: `Router` , `Route` , `IndexRoute` , and `Redirect` . Routing can be defined declarative using `Route` tags or via a config object passed in the `Router` component. `IndexRoute` nested within a `Route` specifies the default nested component to mount (i.e. Home in the example below)

The `route` accepts `onLeave` and `onEnter` hooks that can be used to complete tasks, such as authentication or data persistence, before the the route unmounts or mounts .

For maintaining browser history, bring in `browserHistory` or `hashHistory` . For cleaner urls, it is recommended to use `browserHistory` , however, this requires [server configuration for production](#).

```
import { Router, Route, browserHistory, IndexRoute } from 'react-router'
... // import components and React dependencies

// declarative definition
render((
  <Router history={browserHistory}>
    <Route path="/" component={App}>
      <IndexRoute component={Home} />
      <Route path="about" component={About} />
      <Route path="/products" component={Products}>
        <Route path="products/:id" component={Product} />
      </Route>
    </Route>
  </Router>
), document.body)
```

```
// configuration definition
const routes = {
  path: '/',
  component: App,
  indexRoute: { component: Home },
  childRoutes: [
    { path: 'about', component: About },
    {
      component: Product,
      childRoutes: [{
        path: 'product/:id', component: Product
      }]
    }
  ]
}

render(<Router routes={routes} />, document.body)
```

Params, Nested Components, Links

Any url (i.e. `product/:id`) parameters can be accessed from the rendered component via `this.props.params` . Use `this.props.children` to render nested components. A simple example is maintaining the navigation across the app where the pages are rendered below the navigation.

The `link` component is to create links that use the router configuration (like `<a/>` tags). To determine the active link, you can pass `activeClassName` or `activeStyle` props to `link` .

git st

```
import { IndexLink, Link } from 'react-router'
... // import components and React dependencies

class App extends Component {
  render() {
    return (
      <div>
        <h1>App</h1>
        <ul>
          <li><IndexLink to="/" activeStyle={{ color: green }}>Home</Link></li>
          <li><Link to="/about" activeClassName="active">About</Link></li>
        </ul>
        { /* nested components, i.e. <Home/>, rendered below */ }
        { this.props.children }
      </div>
    )
  }
}
```

Redux and React Router

React Router and Redux work fine together, however, debugging features such as replaying actions with [Redux DevTools](#) will not work.

Since React Router is controlling the components being rendered via the url, we need to store the history within the application state. We can use [react-router-redux](#) to do this.

Setup

```
npm install --save react-router-redux
```

Example

```
import { createStore, combineReducers } from 'redux'
import { Provider } from 'react-redux'
import { Router, Route, browserHistory, IndexRoute } from 'react-router'
import { syncHistoryWithStore, routerReducer } from 'react-router-redux'
... // import components, reducers and React dependencies

const store = createStore(
  combineReducers({
    ...reducers,
    // add routerReducer on `routing` key
    routing: routerReducer
  })
)

// sync the history with the store
const history = syncHistoryWithStore(browserHistory, store);

render((
  <Provider store={store}>
    <Router history={history}>
      <Route path="/" component={App}>
        <IndexRoute component={Home} />
        <Route path="about" component={About} />
        <Route path="/products" component={Products}>
          <Route path="products/:id" component={Product} />
        </Route>
      </Route>
    </Router>
  </Provider>
), document.body)
```

Forms in React

This section will look at setting up [Redux Form](#).



Setup

```
npm install redux-form
```

Reducer

Redux-form provides a reducer that manages the form's state. Add this to the

`combineReducers` . It is important to specify the state reducer key as `form` for **redux-form to work**

```
import {reducer as formReducer} from 'redux-form'

const reducer = combineReducers(Object.assign({}, reducers, {
  ...
  routing: routeReducer,
  form: formReducer,
}))
```

Wrapper

Redux-form provides a `redux-form` wrapper to pass your component props as callbacks (`resetForm` , `handleSubmit`) and form data (`error` , `dirty` , `fields` , `submitting`). View the [full list of props](#).

```
export default reduxForm({
  form: 'formKey', // form key must be unique
  fields: ['name', 'email'] // form fields
  ...
  validate // custom form validation function
})(Form)
```

Form

The `fields` props contains the field values (i.e. name, email) and several event listeners for each field, so these must be passed into the input tag for the specific field via `{...name}` .


```
import React, {Component} from 'react';
import {reduxForm} from 'redux-form';

const submit = (formValues, dispatch) => {
  ...
}

class Form extends Component {
  render() {
    const {fields: {name, email}, handleSubmit} = this.props;
    return (
      <form onSubmit={handleSubmit(submit)}>
        <label>First Name</label>
        <input type="text" placeholder="Name" {...name}/>
        <label>Last Name</label>
        <input type="text" placeholder="Email" {...email}/>
        <button type="submit">Submit</button>
      </form>
    )
  }
}
```

For submission of form data, the `handleSubmit` prop is passed to the `onSubmit` or `onClick` and will complete any validations before calling `this.props.onSubmit(data)`. You can also pass a function into `handleSubmit` to be called, so `handleSubmit(submit)`.

Check out [Redux Form](#) for more form examples.

Validation

The `validate` function will be called on each render cycle and will be passed the a form values object where the function must return an errors object with the specific error message for each field.

```
const validate = fields => {
  const errors = {}
  if (!fields.name) {
    errors.name = 'Required'
  }
  if (!fields.email) {
    errors.email = 'Required'
  } else if (!/^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$/i.test(fields.email)) {
    errors.email = 'Invalid email address'
  }
  return errors
}
```

The keys for the input field values and output errors objects must match the form `fields` specified (i.e. `name`, and `email`).

Full Example

```
import React, {Component} from 'react';

const validate = values => {
  const errors = {};
  if (!values.name) {
    errors.name = 'Required';
  }
  if (!values.email) {
    errors.email = 'Required';
  } else if (!/^([A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4})$/i.test(values.email)) {
    errors.email = 'Invalid email';
  }
  return errors;
}

class Form extends Component {
  render() {
    const {fields: {name, email}, handleSubmit} = this.props;
    return (
      <form onSubmit={handleSubmit}>
        <label>First Name</label>
        <input type="text" placeholder="Name" {...name}/>
        <label>Last Name</label>
        <input type="text" placeholder="Email" {...email}/>
        <button type="submit">Submit</button>
      </form>
    )
  }
}

export default reduxForm({
  form: 'formKey', // form key must be unique
  fields: ['name', 'email'] // form fields
  ...
  validate // custom form validation function
})(Form)
```

Testing

We will be using [Enzyme](#) to render and test React components.

Setup

To get started, we need to do a little bit of setup. Install the following dependencies.

- babel-eslint
- babel-preset-es2015
- babel-preset-react
- babel-preset-stage-0
- babel-register
- chai
- enzyme
- jsdom
- mock-localstorage
- react-addons-test-utils
- redux-mock-store
- sinon
- mocha
- nock

Enzyme

[Enzyme](#) allows for rendering of React Components in three different ways: `render` (static component), `shallow` (isolated component), `mount` (full DOM). Most components can be `shallow` rendered and tested in isolation. Enzyme allows for searching of the rendered component for testing (i.e. `.find`).

We will be using Mocha as the test runner. View the [API Documentation](#) for further details.

JSDOM

When using Enzyme to render component via `mount` (full DOM), you can use [jsdom](#) as a headless browser since `mount` requires a DOM environment.

Components

Enzyme is used to output React components and manipulate or transverse them. Using the `chai` assertion library, we can make assertions on the component.

Example

```
// ./counter/index.js
import React from 'react';
import Button from '../button';

function Counter({ counter, increment, decrement, ...props }) {
  return (
    <div className="flex" data-testid={ props.testid }>
      <Button data-ref="decrementButton" className="bg-black col-2"
        onClick={ decrement }>
        -
      </Button>

      <div data-ref="result" className="flex-auto center h1">
        { counter }
      </div>

      <Button data-ref="incrementButton" className="col-2"
        onClick={ increment }>
        +
      </Button>
    </div>
  );
}

Counter.propTypes = {
  counter: React.PropTypes.number,
  increment: React.PropTypes.func,
  decrement: React.PropTypes.func,
  testid: React.PropTypes.func,
};

export default Counter;
```

```
// ./counter/index.test.js
import { assert } from 'chai';
import React from 'react';
import { shallow, render } from 'enzyme';
import sinon from 'sinon';
import Counter from './index';

describe('counter', () => {
  it('should create a counter', () => {
    const wrapper = render(<Counter counter={5} />);

    assert.isOk(wrapper.children().length,
      'Counter not found');
    assert.strictEqual(wrapper.find('[data-ref="result"]').text(), '5',
      'Counter not showing its value');
  });

  it('should respond to click events', () => {
    const onIncrement = sinon.spy();
    const onDecrement = sinon.spy();
    const wrapper = shallow(
      <Counter increment={onIncrement} decrement={onDecrement} />
    );

    wrapper.find('[data-ref="incrementButton"]').simulate('click');
    assert.isTrue(onIncrement.calledOnce, 'increment not called');

    wrapper.find('[data-ref="decrementButton"]').simulate('click');
    assert.isTrue(onIncrement.calledOnce, 'decrement not called');
  });
});
```

Reducers

Since reducers are pure functions, they can be tested like a JavaScript function (i.e. given an action, confirm the expected result)

Example

```
// ./reducers/counter.js
import { INCREMENT_COUNTER, DECREMENT_COUNTER } from '../constants';
import { fromJS } from 'immutable';

const INITIAL_STATE = fromJS({
  count: 0,
});

function counterReducer(state = INITIAL_STATE, action = {}) {
  switch (action.type) {

    case INCREMENT_COUNTER:
      return state.update('count', (value) => value + 1);

    case DECREMENT_COUNTER:
      return state.update('count', (value) => value - 1);

    default:
      return state;
  }
}

export default counterReducer;
```

```
// ./reducers/counter.test.js
import { assert } from 'chai';
import counterReducer from './counter';
import { INCREMENT_COUNTER, DECREMENT_COUNTER } from '../constants';
import { Map } from 'immutable';

//Used for testing to fire actions against a reducer.
function fireAction(reducer, currentState, type, payload = {}) {
  return reducer(currentState, {
    type,
    payload,
  });
}

let state = counterReducer(undefined, {});

describe('counter reducer', () => {
  describe('initial state', () => {
    it('should be a Map', () => {
      assert.strictEqual(Map.isMap(state), true);
    });
  });

  describe('on INCREMENT_COUNTER', () => {
    it('should increment state.count', () => {
      const previousValue = state.get('count');
      state = fireAction(counterReducer, state, INCREMENT_COUNTER);
      assert.strictEqual(state.get('count'), previousValue + 1);
    });
  });

  describe('on DECREMENT_COUNTER', () => {
    it('should decrement state.count', () => {
      const previousValue = state.get('count');
      state = fireAction(counterReducer, state, DECREMENT_COUNTER);
      assert.strictEqual(state.get('count'), previousValue - 1);
    });
  });
});
```


Actions

Simple actions can be tested like reducers, as pure functions (i.e. given params, confirm the expected object/response). The [Redux gitbook](#) contains more examples, including handling tricky async actions.

Example

```
// ./actions/counter.js
import { INCREMENT_COUNTER, DECREMENT_COUNTER } from '../constants';

export function increment() {
  return {
    type: INCREMENT_COUNTER,
  };
}

...
```

```
// ./actions/counter.test.js

import { assert } from 'chai';
import { INCREMENT_COUNTER, DECREMENT_COUNTER } from '../constants';
import { increment } from '../actions/counter';

describe('Counter Actions', () => {
  describe('increment', () => {
    let obj;
    beforeEach(() => {
      obj = increment();
    });

    it('should return an object', () => {
      assert.isObject(obj)
    });

    it('should return an object with correct type property', () => {
      assert.deepEqual(obj, {type: INCREMENT_COUNTER});
    });
  })
});
```

