JSX and the Virtual DOM

React Uses a Virtual DOM

React works differently than many earlier front-end JavaScript frameworks in that instead of working with the **browser's DOM**, it builds a **virtual representation** of the DOM. By **virtual**, we mean a tree of JavaScript objects that *represent* the "actual DOM". More on this in a minute.

In React, we *do not directly manipulate the actual DOM*. Instead, we must manipulate the virtual representation and let React take care of changing the browser's DOM.

As we'll see in this chapter, this is a very powerful feature but it requires us to think differently about how we build web apps.

Why Not Modify the Actual DOM?

It's worth asking: why do we need a Virtual DOM? Can't we just use the "actual-DOM"?

When we do "classic-" (e.g. jQuery-) style web development, we would typically:

1. locate an element (using `document.querySelector` or `document.getElementById`) and then
2. modify that element directly (say, by calling `.innerHTML()` on the element).

This style of development is problematic in that:

- **It's hard to keep track of changes** - it can become difficult keep track of current (and prior) state of the DOM to manipulate it into the form we need
- **It can be slow** - modifying the actual-DOM is a costly operation, and modifying the DOM on every change can cause poor performance

What is a Virtual DOM?

The Virtual DOM was created to deal with these issues. But *what is the Virtual DOM anyway*?

The Virtual DOM is a tree of JavaScript objects that represents the actual DOM.

One of the interesting reasons to use the Virtual DOM is the API it gives us. When using the Virtual DOM we code as if we're **recreating the entire DOM on every update**.

This idea of re-creating the entire DOM results in an easy-to-comprehend development model: instead of the developer keeping track of all DOM state changes, the developer simply returns the DOM *they wish to see*. React takes care of the transformation behind the scenes.

This idea of re-creating the Virtual DOM every update might sound like a bad idea: isn't it going to be slow? In fact, React's Virtual DOM implementation comes with important performance optimizations that make it very fast.

The Virtual DOM will:

- use **efficient diffing algorithms**, in order to know what changed
- **update subtrees** of the DOM simultaneously
- **batch updates** to the DOM

All of this results in an easy-to-use and optimized way to build web apps.

Virtual DOM Pieces

Again, when building a web app in React, we're not working directly with the browser's "actual DOM" directly, but instead a *virtual representation* of it. Our job is to provide React with enough information to build a **JavaScript object** that *represents* what the browser will render.

But what does this Virtual DOM JavaScript object actually consist of?

React's Virtual DOM is a tree of `ReactElements`.

Understanding the Virtual DOM, `ReactElements`, and how they interact with the "actual DOM" is a lot easier to understand by working through some examples, which we'll do below.

Q: Virtual DOM vs. Shadow DOM, are they the same thing? (A: No)

Maybe you've heard of the "Shadow DOM" and you're wondering, is the Shadow DOM the same thing as the Virtual DOM? The answer is **no**.

The *Virtual DOM* is a tree of JavaScript objects that represent the real DOM elements.

The *Shadow DOM* is a form of encapsulation on our elements. Think about using the `<video>` tag in your browser. In a `video` tag, your browser will create a set of video controls such as a play button, a timecode number, a scrubber progress bar etc. These elements aren't part of your "regular DOM", but instead, part of the "Shadow DOM".

Talking about the Shadow DOM is outside the scope of this chapter. But if you want to learn more about the Shadow DOM checkout this article: Introduction to Shadow DOM[43]

[43] http://webcomponents.org/articles/introduction-to-shadow-dom/

ReactElement

A `ReactElement` is a representation of a DOM element in the Virtual DOM.

React will take these `ReactElements` and place them into the "actual DOM" for us.

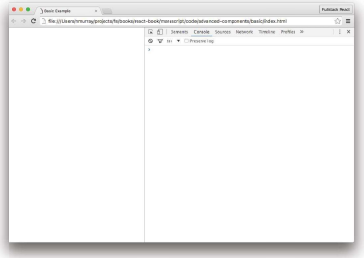
One of the best ways to get an intuition about `ReactElement` is to play around with it in our browser, so let's do that now.

Experimenting with `ReactElement`

Try this in your browser

For this section, open up the file `code/jsx/basic/index.html` (from the code download) in your browser.

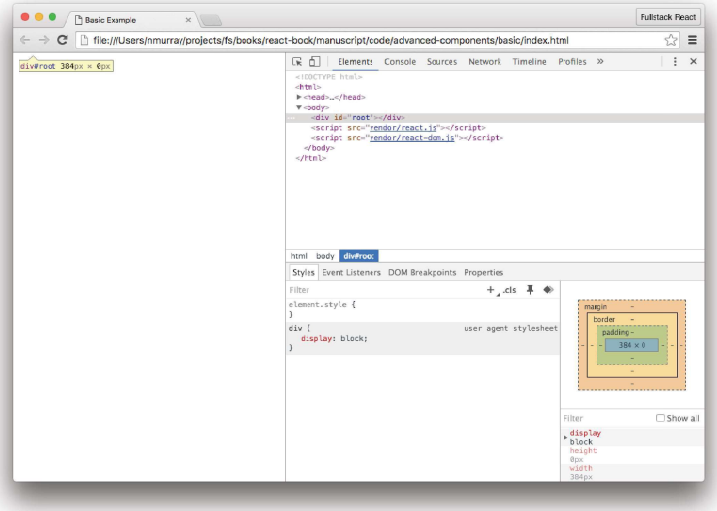
Then open up your developer console and type commands into the console. You can access the console in Chrome by right-clicking and picking "Inspect" and then clicking on "Console" in the inspector.



Basic Console

We'll start by using a simple HTML template that includes one `<div>` element with an id tag:

```
1  <div id='root' />
```

Root Element

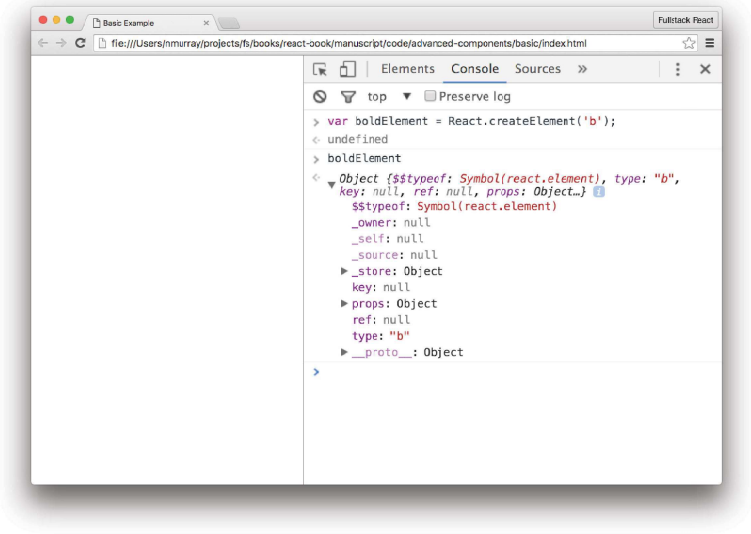
Let's walk through how we render a `<b></b>` tag in our (actual) DOM using React. Of course, we **are not** going to create a `<b>` tag directly in the DOM (like we might if we were using jQuery).

Instead, React expects us to provide a **Virtual DOM tree**. That is, we're going to give React a set of JavaScript objects which **React will turn into a real DOM tree**.

The objects that make up the tree will be `ReactElements`. To create a `ReactElement`, we use the `createElement` method provided by React.

For instance, to create a `ReactElement` that represents a `<b>` (bold) element in React, type the following in the browser console:

```
1  var boldElement = React.createElement('b');
```



`boldElement` is a `ReactElement`

Our `boldElement` above is an instance of a `ReactElement`. Now, we have this `boldElement`, but it's not visible without giving it to React to render in the actual DOM tree.

Rendering Our `ReactElement`

In order to render this element to the actual DOM tree we need to use `ReactDOM.render()` (which we cover in more detail a bit later in this chapter. `ReactDOM.render()` requires two things:

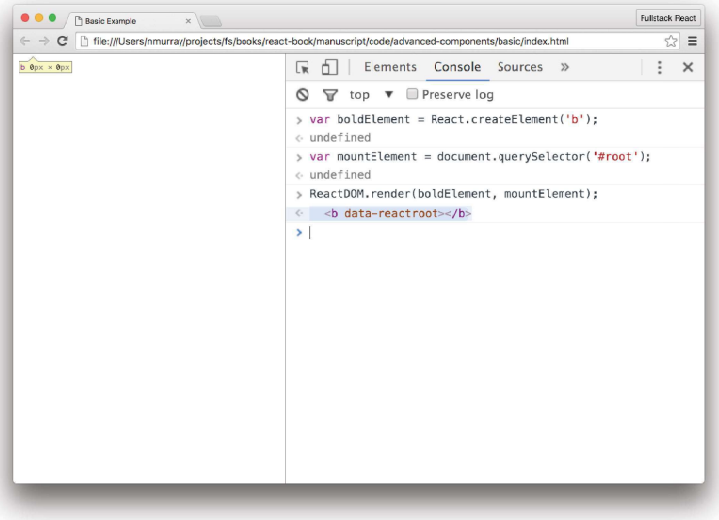
1. The *root* of our virtual tree
2. the *mount location* where we want React write to the *actual* browser DOM

In our simple template we want to get access to the `div` tag with an `id` of `root`. To get a reference to our actual DOM `root` element, we use one of the following:


```
// Either of these will work
var mountElement = document.getElementById('root');
var mountElement = document.querySelector('#root');

// if we were using jQuery this would work too
var mountElement = $('#root')
```

With our `mountElement` retrieved from the DOM, we can give React a point to insert it's own rendered DOM.

```
var boldElement = React.createElement('b');
var mountElement = document.querySelector('#root');
// Render the boldElement in the DOM tree
ReactDOM.render(boldElement, mountElement);
```



Despite the fact that nothing appears in the DOM, a new empty element has been inserted into the document as a child of the `mountElement`.

If we click the "Elements" tab in the Chrome inspector, we can see that a `b` tag was created in the actual DOM.

Adding Text (with children)

Although we now have a `b` tag in our DOM, it would be nice if we could add some text in the tag. Because text is in-between the opening and closing `b` tags, adding text is a matter of creating a *child* of the element.

Above, we used `React.createElement` with only a single argument (`'b'` for the `b` tag), however the `React.createElement()` function accepts three arguments:

1. The DOM element type
2. The element `props`
3. The children of the element

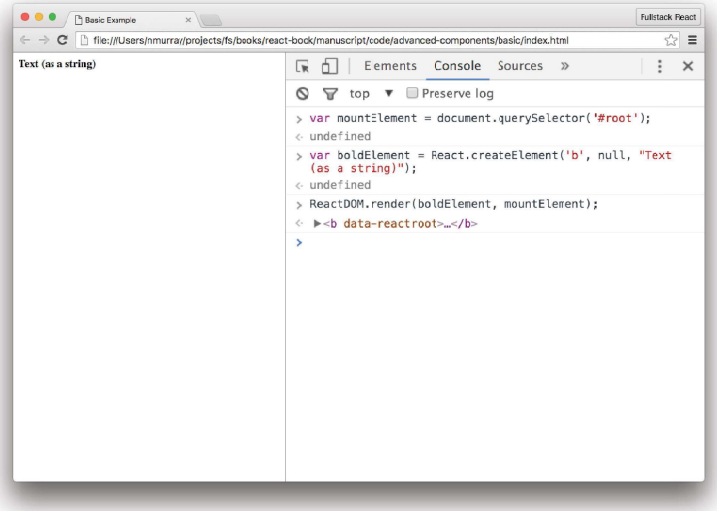
We'll walk through `props` in detail later in this section, so for now we'll set this parameter to `null`.

The `children` of the DOM element must be a `ReactNode` object, which is any of the following:

1. `ReactElement`
2. A string or a number (a `ReactText` object)
3. An array of `ReactNodes`

For example, to place text in our `boldElement`, we can pass a string as the third argument in the `createElement()` function from above:

```
1 var mountElement = document.querySelector('#root');
2 // Third argument is the inner text
3 var boldElement = React.createElement('b', null, "Text (as a string)");
4 ReactDOM.render(boldElement, mountElement);
```

`ReactDOM.render()`

As we've seen, we use a React renderer places the virtual tree into a "hard" browser view (the "actual" DOM).

But there's a neat side effect of React using it's own virtual representation of the view-tree: it can render this tree in **multiple types of canvases**.

That is, not only can React render into the browser's DOM, but it can also be used to **render views in other frameworks such as mobile apps**. In React Native (which we talk about later in this book), this tree is rendered into *native mobile views*.

That said, in this section we'll spend most of our time in the DOM, so we'll use the `ReactDOM` renderer to manage elements in the browser DOM.

As we've seen `ReactDOM.render()` is the way we get our React app into the DOM:

```
// ...
const component = ReactDOM.render(boldElement, mountElement);
```

We can call the `ReactDOM.render()` function multiple times and it will only perform updates (mutations) to the DOM as necessary.

The `ReactDOM.render` function accepts a 3rd argument: a callback argument that is executed after the component is rendered/updated. We can use this callback as a way to run functions after our app has started:

```
ReactDOM.render(boldElement, mountElement, function() {
  // The React app has been rendered/updated
});
```

JSX

JSX Creates Elements

When we created our `ReactElement` earlier, we used `React.createElement` like this:

```
var boldElement = React.createElement('b', null, "Text (as a string)");
```

This works fine as we had a small component, but if we had many nested components the syntax could get messy very quickly. Our DOM is hierarchical and our React component tree is hierarchical as well.

We can think of it this way: to describe pages to our browser we write HTML; the HTML is parsed by the browser to create HTML Elements which become the DOM.

HTML works very well for specifying tag hierarchies. It would be nice to represent our React component tree using markup, much like we do for HTML.

This is the idea behind `JSX`.

When using JSX, creating the `ReactElement` objects are handled for us. Instead of calling `React.createElement` for each element, the equivalent structure in JSX is:

```
var boldElement = <b>Text (as a string)</b>;
// => boldElement is now a ReactElement
```

The JSX parser will read that string and call `React.createElement` *for us*.

JSX stands for **JavaScript Syntax Extension**, and it is a syntax React provides that looks a lot like HTML/XML. Rather than building our component trees using normal JavaScript directly, we write our components almost as if we were writing HTML.

JSX provides a syntax that is similar to HTML. However, in JSX we can create our own tags (which encapsulate functionality of other components).

Although it has a scary-sounding name, writing JSX is not much more difficult than writing HTML. For instance, here is a JSX component:

```
const element = <div>Hello world</div>;
```

One difference between React components and HTML tags is in the naming. HTML tags start with a lowercase letter, while React components start with an uppercase. For example:

```
// html tag
const htmlElement = (<div>Hello world</div>);

// React component
const Message = props => (<div>{props.text}</div>)

// Use our React component with a `Message` tag
const reactComponent = (<Message text="Hello world" />);
```

We often surround JSX with parenthesis `()`. Although this is not always technically required, it helps us set apart JSX from JavaScript.

Our browser doesn't know how to read JSX, so how is JSX possible?

JSX is transformed into JavaScript by using a pre-processor build-tool before we load it with the browser.

When we write JSX, we pass it through a "compiler" (sometimes we say the code is *transpiled*) that converts the JSX to JavaScript. The most common tool for this is a plugin to `babel`, which we'll cover later.

Besides being able to write HTML-like component trees, JSX provides another advantage: we can mix JavaScript with our JSX markup. This lets us add logic inline with our views.

We've seen basic examples of JSX several times in this book already. What is different in this section is that we're going to take a more structured look at the different ways we can use JSX. We'll cover tips for using JSX and then talk about how to handle some tricky cases.

Let's look at:

- attribute expressions
- child expressions
- boolean attributes
- and comments

JSX Attribute Expressions

In order to use a JavaScript expression in a component's attribute, we wrap it in curly braces `{}` instead of quotes `""`.

```
// ...
const warningLevel = 'debug';
const component = (<Alert
                    color={warningLevel === 'debug' ? 'gray' : 'red'}
                    log={true} />)
```

This example uses the ternary operator[44] on the `color` prop.

If the `warningLevel` variable is set to `debug`, then the `color` prop will be `'gray'`, otherwise it will be `'red'`.

JSX Conditional Child Expressions

Another common pattern is to use a boolean checking expression and then render another element conditionally.

For instance, if we're building a menu that shows options for an `admin` user, we might write:

```
// ...
const renderAdminMenu = function() {
  return (<MenuLink to="/users">User accounts</MenuLink>)
}
// ...
const userLevel = this.props.userLevel;
return (
  <ul>
    <li>Menu</li>
    {userLevel === 'admin' && renderAdminMenu()}
  </ul>
)
```

We can also use the ternary operator to render one component or another.

For instance, if we want to show a `<UserMenu>` component for a logged in user and a `<LoginLink>` for an anonymous user, we can use this expression:

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


```
const Menu = (<ul>{loggedInUser ? <UserMenu /> : <LoginLink />}</ul>)
```

JSX Boolean Attributes

In HTML, the presence of some attributes sets the attribute to true. For instance, a disabled `<input>` HTML element can be defined:

```
<input name='Name' disabled />
```

In React we need to set these as booleans. That is, we need to pass a `true` or `false` explicitly as an attribute:

```
// Set the boolean in brackets directly
<input name='Name' disabled={true} />

// ... or use JavaScript variables
let formDisabled = true;
<input name='Name' disabled={formDisabled} />
```

If we ever need to *enable* the `input` above, then we set `formDisabled` to `false`.

JSX Comments

We can define comments inside of JSX by using the curly braces (`{}`) with comment delimiters (`/* */`):

```
let userLevel = 'admin';
{/*
  Show the admin menu if the userLevel is 'admin'
*/}
{userLevel === 'admin' && <AdminMenu />}
```

JSX Spread Syntax

Sometimes when we have many props to pass to a component, it can be cumbersome to list each one individually. Thankfully, JSX has a shortcut syntax that makes this easier.

For instance, if we have a `props` object that has two keys:

```
const props = {msg: "Hello", recipient: "World"}
```

We *could* pass each prop individually like this:

```
<Component msg={"Hello"} recipient={"World"} />
```

But by using the JSX *spread syntax* we can do it like this instead:

```
<Component {...props} />
<!-- essentially the same as this: -->
<Component msg={"Hello"} recipient={"World"} />
```

JSX Gotchas

Although JSX mimics HTML, there are a few important differences to pay attention to.

Here's a few things to keep in mind:

JSX Gotcha: `class` and `className`

When we want to set the CSS class of an HTML element, we normally use the `class` attribute in the tag:

```
<div class='box'></div>
```

Since JSX is so closely tied to JavaScript, we cannot use identifiers that JavaScript uses in our tag attributes. Attributes such as `for` and `class` conflict with the JavaScript keywords `for` and `class`.

Instead of using `class` to identify a class, JSX uses `className`:

```
<!-- Same as <div class='box'></div> -->
<div className='box'></div>
```

The `className` attributes works similarly to the `class` attribute in HTML. It expects to receive a string that identifies the class (or classes) associated with a CSS class.

To pass multiple classes in JSX, we can join an array to convert it to a string:


```
var cssNames = ['box', 'alert']
// and use the array of cssNames in JSX
(<div className={cssNames.join(' ')}></div>)
```

Tip: Managing `className` with `classnames`

The `classnames` npm package[45] is a great extension that we use to help manage css classes. It can take a list of strings or objects and allows us to conditionally apply classes to an element.

The `classnames` package takes the arguments, converts them to an object and conditionally applies a CSS class if the value is truthy.

code/jsx/basic/app.js

```
class App extends React.Component {
  render() {
    const klass = classnames({
      box: true, // always apply the box class
      alert: this.props.isAlert, // if a prop is set
      severity: this.state.onHighAlert, // with a state
      timed: false // never apply this class
    });
    return (<div className={klass} />)
  }
}
```

The package readme[46] provides alternate examples for more complex environments.

JSX Gotcha: `for` and `htmlFor`

For the same reason we cannot use the `class` attribute, we cannot apply the `for` attribute to a `<label>` element. Instead, we must use the attribute `htmlFor`. The property is a pass-through property in that it applies the attribute as `for`:

```
<!-- ... -->
<label htmlFor='email'>Email</label>
<input name='email' type='email' />
<!-- ... -->
```

JSX Gotcha: HTML Entities and Emoji

Entities are reserved characters in HTML which include characters such as less-than `<`, greater-than `>`, the copyright symbol, etc. In order to display entities, we can just place the entity code in literal text.

[45] https://www.npmjs.com/package/classnames
[46] https://github.com/JedWatson/classnames/blob/master/README.md

```
<ul>
  <li>phone: &phone;</li>
  <li>star: &star;</li>
</ul>
```

In order to display entities in dynamic data, we need to surround them in a string inside of curly braces ({}). Using unicode directly in JS works as expected. Just as we can send our JS to the browser as UTF-8 text directly. Our browser knows how to display UTF-8 code natively.

Alternatively, instead of using the entity character code, we can use unicode version instead.

```
return (
  <ul>
    <li>phone: {'\u0260e'}</li>
    <li>star: {'\u2606'}</li>
  </ul>
)
```

Emoji are just unicode character sequences, so we can add emoji the same way:

```
return(
  <ul>
    <li>dolphin: {'\uD83D\uDC2C'}</li>
    <li>dolphin: {'\uD83D\uDC2C'}</li>
    <li>dolphin: {'\uD83D\uDC2C'}</li>
  </ul>
)
```

Everyone needs more dolphins

JSX Gotcha: `data-anything`

If we want to apply our own attributes that the HTML spec does not cover, we have to prefix the attribute key with the string `data-`.


```
<div className='box' data-dismissible={true} />
<span data-highlight={true} />
```

This requirements *only* applies to DOM components that are native to HTML and does not mean custom components cannot accept arbitrary keys as props. That is, we can accept *any* attribute on a custom component:

```
<Message dismissible={true} />
<Note highlight={true} />
```

There are a standard set of web accessibility[47] attributes[48] and its a good idea to use them because there are many people who will have a hard time using our site without them. We can use any of these attribute on an element with the key prepended with the string `aria-`. For instance, to set the `hidden` attribute:

```
<div aria-hidden={true} />
```

JSX Summary

JSX isn't magic. The key thing to keep in mind is that JSX is syntactic sugar to call `React.createElement`.

JSX is going to parse the tags we write and then create JavaScript objects. JSX is a convenience syntax to help build the component tree.

As we saw earlier, when we use JSX tags in our code, it gets converted to a `ReactElement`:

```
var boldElement = <b>Text (as a string)</b>;
// => boldElement is now a ReactElement
```

We can pass that `ReactElement` to `ReactDOM.render` and see our code rendered on the page.

There's one problem though: `ReactElement` is stateless and immutable. If we want to add interactivity (with state) into our app, we need another piece of the puzzle: `ReactComponent`.

In the next chapter, we'll talk about `ReactComponents` in depth.

[47] https://www.w3.org/WAI/intro/aria
[48] https://www.w3.org/TR/wai-aria/

References

Here are some places to read more about JSX and the Virtual DOM:

- JSX in Depth[49] - (Facebook)
- If-Else in JSX[50] - (Facebook)
- React (Virtual) DOM Terminology[51] - (Facebook)
- What is Virtual DOM[52] - (Jack Bishop)

[49] https://facebook.github.io/react/docs/jsx-in-depth.html
[50] https://facebook.github.io/react/tips/if-else-in-JSX.html
[51] https://facebook.github.io/react/docs/glossary.html
[52] http://jbi.sh/what-is-virtual-dom/

Advanced Component Configuration with `props`, `state`, and `children`

> Unlike the rest of the chapters in this book, this chapter is intended on being read as an in-depth, deep dive into the different features of React, from an encylopedia-like perspective. For this reason, we did not include a step-by-step follow-along style project in this section of the book.

Intro

In this chapter we're going to dig deep into the configuration of components.

A `ReactComponent` is a JavaScript object that, at a minimum, has a `render()` function. `render()` is expected to **return a `ReactElement`**.

Recall that `ReactElement` is a representation of a DOM element in the Virtual DOM.

In the chapter on JSX and the Virtual DOM we talked about `ReactElement` extensively. Checkout that chapter if you want to understand `ReactElement` better.

The goal of a `ReactComponent` is to

- `render()` a `ReactElement` (which will eventually become the real DOM) and
- attach functionality to this section of the page

"Attaching functionality" is a bit ambiguous; it includes attaching event handlers, managing state, interacting with children, etc. In this chapter we're going to cover:

- `render()` - the one required function on every `ReactComponent`
- `props` - the "input parameters" to our components
- `context` - a "global variable" for our components
- `state` - a way to hold data that is local to a component (that affects rendering)
- Stateless components - a simplified way to write reusable components
- `children` - how to interact and manipulate child components
- `statics` - how to create "class methods" on our components

Let's get started!

How to use this chapter

This chapter is built using a specific tool called `styleguidist`. In the code included with this course is a section called `components-cookbook`, which accompanies this chapter with the `styleguidist` tool bundled in with it. To use the `styleguidist` tool, which allows introspection into the components themselves, we can boot up the section through the chapter.

In order to get it started, change into the directory in terminal and issue the following commands. First, we'll need to get the dependencies for the project using `npm install`:

```
1  npm install
```

To start the application, issue the `npm start` command:

```
1  npm start
```

Once the server is running, we can navigate to our browser and head to the URL of `http://localhost:3000`. We'll see the styleguide running with all the components exposed by this chapter, where we can navigate through running examples of the components executing in real-time.

ReactComponent

Creating `ReactComponent`s - `createClass` or ES6 Classes

As discussed in the first chapter, there are two ways to define a `ReactComponent` instance:

1. `React.createClass()` or
2. ES6 classes

As we've seen, the two methods of creating components are roughly equivalent:

advanced-components/components-cookbook/src/components/Component/CreateClassApp.js

```
import React from 'react'

// React.createClass
const App = React.createClass({
  render: function() {} // required method
});

export default App
```

and

advanced-components/components-cookbook/src/components/Component/Components.js

```
import React from 'react'

// ES6 class-style
class App extends React.Component {
  render() {} // required
}

export default App
```

Regardless of the method we used to define the `ReactComponent`, React expects us to define the `render()` function.

`render()` Returns a `ReactElement` Tree

The `render()` method is the only required method to be defined on a `ReactComponent`.

After the component is **mounted** and **initialized**, `render()` will be called. The `render()` function's job is to provide React a *virtual representation* of a native DOM component.

An example of using `React.createClass` with the `render` function might look like this

advanced-components/components-cookbook/src/components/Component/CreateClassHeading.js

```
const CreateClassHeading = React.createClass({
  render: function() {
    return (
      <h1>Hello</h1>
    )
  }
});
```

Or with ES6 class-style components:

advanced-components/components-cookbook/src/components/Component/Header.js

```
class Heading extends React.Component {
  render() {
    return (
      <h1>Hello</h1>
    )
  }
};
```

The above code should look familiar. It describes a `Heading` component class with a single `render()` method that returns a simple, single Virtual DOM representation of a `<h1>` tag.

Remember that this `render()` method returns a `ReactElement` which isn't part of the "actual DOM", but instead a description of the Virtual DOM.

React expects the method to return a *single* child element. It can be a virtual representation of a DOM component or can return the falsy value of `null` or `false`. React handles the falsy value by rendering an empty element (a `<noscript />` tag). This is used to remove the tag from the page.

Keeping the `render()` method side-effect free provides an important optimization and makes our code easier to understand.

Getting Data into `render()`

Of course, while `render` is the only required method, it isn't very interesting if the only data we can render is known at compile time. That is, we need a way to:

- input "arguments" into our components and
- maintain state within a component.

React provides ways to do both of these things, with `props` and `state`, respectively.

Understanding these are crucial to making our components dynamic and *useable* within a larger app.

In React, `props` are immutable pieces of **data that are passed into** child components from parents (if we think of our component as the "function" we can think of `props` as our component's "arguments").

Component `state` is where we hold data, local to a component. Typically, when our component's `state` changes, the component needs to be re-rendered. Unlike props, `state` is private to a component and is mutable.

We'll look at both `props` and `state` in detail below. Along the way we'll also talk about `context`, a sort of "implicit `props`" that gets passed through the whole component tree.

Let's look at each of these in more detail.

`props` are the parameters

`props` are the inputs to your components. If we think of our component as a "function", we can think of the props as the "parameters".

Let's look at an example:

```
<div>
  <Header headerText="Hello world" />
</div>
```

In the example code, we're creating both a `<div>` and a `<Header>` element, where the `<div>` is a usual DOM element, while `<Header>` is an instance of our `Header` component.

In this example, we're passing data from the component (the string `"Hello world"`) through the attribute `headerText` to the component.

Passing data through attributes to the component is often called *passing props*.

When we pass data to a component through an attribute it becomes available to the component through the `this.props` property. So in this case, we can access our `headerText` through the property `this.props.headerText`:

```
import React from 'react';

export class Header extends React.Component {
  render() {
    return (
      <h1>{this.props.headerText}</h1>
    );
  }
}
```

While we can access the `headerText` property, we *cannot* change it.

By using `props` we've taken our static component and allowed it to dynamically render whatever `headerText` is passed into it. The `<Header>` component cannot change the `headerText`, but it can use the `headerText` itself or pass it on to its children.

We can pass any JavaScript object through `props`. We can pass primitives, simple JavaScript objects, atoms, functions etc. We can even pass other React elements and Virtual DOM nodes.

We can document the functionality of our components using `props` and we can specify the *type* of each prop by using `PropTypes`.

PropTypes

`PropTypes` are a way to validate the values that are passed in through our `props`. Well-defined interfaces provide us with a layer of safety at the run time of our apps. They also provide a layer of documentation to the consumer of our components.

We include the `prop-types` package in our `package.json`.

We define `PropTypes` by setting a *static* (class) property `propTypes`. This object should be a map of prop-name keys to `PropTypes` values:

```
class Map extends React.Component {
  static propTypes = {
    lat: PropTypes.number,
    lng: PropTypes.number,
    zoom: PropTypes.number,
    place: PropTypes.object,
    markers: PropTypes.array,
  };
```


If using `createClass`, we define `PropTypes` by passing them as an option to `createClass()`:

```
const Map = React.createClass({
  propTypes: {
    lat: PropTypes.number,
    lng: PropTypes.number
    // ...
  },
})
```

In the example above, our component will validate that `name` is a `string` and that `totalCount` is a `number`.

There are a number of built-in `PropTypes`, and we can define our own.

We've written a code example for many of the `PropTypes` validators here in the appendix on `PropTypes`. For more details on `PropTypes`, check out that appendix.

For now, we need to know that there are validators for scalar types:

- `string`
- `number`
- `boolean`

We can also validate complex types such as:

- `function`
- `object`
- `array`
- `arrayOf` - expects an array of a particular type
- `node`
- `element`

We can also validate a particular `shape` of an input object, or validate that it is an `instanceOf` a particular class.

Checkout the appendix on `PropTypes` for more details and code examples on `PropTypes`

Default `props` with `getDefaultProps()`

Sometimes we want our `props` to have defaults. We can use the static property `defaultProps` to do this.

For instance, create a `Counter` component definition and tell the component that if no `initialValue` is set in the `props` to set it to 1 using `defaultProps`:

```
class Counter extends React.Component {
  static defaultProps = {
    initialValue: 1
  };
  // ...
};
```

Now the component can be used without setting the `initialValue` prop. The two usages of the component are functionally equivalent:

```
<Counter />
<Counter initialValue={1} />
```

context

Sometimes we might find that we have a prop which we want to expose "globally". In this case, we might find it cumbersome to pass this particular prop down from the root, to every leaf, through every intermediate component.

Instead, specifying `context` allows us to automatically pass down variables from component to component, rather than needing to pass down our props at every level,

The `context` feature is experimental and it's similar to using a global variable to handle state in an application - i.e. minimize the use of `context` as relying on it too frequently is a code smell.

That is, `context` works best for things that truly are global, such as the central store in Redux.

When we specify a `context`, React will take care of passing down `context` from component to component so that at any point in the tree hierarchy, any component can reach up to the "global" context where it's defined and get access to the parent's variables.

In order to tell React we want to pass context from a parent component to the rest of its children we need to define two attributes in the parent class:

- `childContextTypes` and
- `getChildContext`

To retrieve the context inside a child component, we need to define the `contextTypes` in the child.

To illustrate, let's look at a possible message reader implementation:


```
class Messages extends React.Component {
  static propTypes = {
    users: PropTypes.array.isRequired,
    messages: PropTypes.array.isRequired
  };

  render() {
    return (
      <div>
        <ThreadList />
        <ChatWindow />
      </div>
    )
  }
});

// ThreadList and ChatWindow are also React.Components
```

Without context, our `Messages` will have to pass the users along with the messages to the two child components (which in turn pass them to their children). Let's set up our hierarchy to accept `context` instead of needing to pass down `this.props.users` and `this.props.messages` along with every component.

In the `Messages` component, we'll define the two required properties. First, we need to tell React what the *types* of our context.

We define this with the `childContextTypes` key. Similar to `propTypes`, the `childContextTypes` is a key-value object that lists the keys as the name of a context item and the value is a `PropType`.

Implementing `childContextTypes` in our `Messages` component looks like the following:

advanced-components/components-cookbook/src/components/Messages/Messages.js

```
class Messages extends React.Component {
  static propTypes = {
    users: PropTypes.array.isRequired,
    initialActiveChatIdx: PropTypes.number,
    messages: PropTypes.array.isRequired,
  };

  static childContextTypes = {
    users: PropTypes.array,
    userMap: PropTypes.object,
  };
```

Just like `propTypes`, the `childContextTypes` doesn't populate the context, it just defines it. In order to fill data the `this.context` object, we need to define the second required function: `getChildContext()`.

With `getChildContext()` we can set the initial value of our context with the return value of the function. Back in our `Messages` component, we will set our `users` context object to the value of the `this.props.users` given to the component.

advanced-components/components-cookbook/src/components/Messages/Messages.js

```
class Messages extends React.Component {
  // ...
  static childContextTypes = {
    users: PropTypes.array,
    userMap: PropTypes.object,
  };
  // ...
  getChildContext() {
    return {
      users: this.getUsers(),
      userMap: this.getUserMap(),
    };
  }
  // ...
}
```

Since the `state` and `props` of a component can change, the `context` can change as well. The `getChildContext()` method in the parent component gets called every time the `state` or `props` change on the parent component. If the `context` is updated, then the children will receive the updated context and will subsequently be re-rendered.

With the two required properties set on the parent component, React *automatically* passes the object down it's subtree where any component can reach into it. In order to grab the context in a child component, we need to tell React we want access to it. We communicate this to React using the `contextTypes` definition in the child.

Without the `contextTypes` property on the child React component, React won't know what to send our component. Let's give our child components access to the context of our `Messages`.

advanced-components/components-cookbook/src/components/Messages/ThreadList.js

```
class ThreadList extends React.Component {
  // ...
  static contextTypes = {
    users: PropTypes.array,
  };
  // ...
}
```

advanced-components/components-cookbook/src/components/Messages/ChatWindow.js

```
class ChatWindow extends React.Component {
  // ...
  static contextTypes = {
    userMap: PropTypes.object,
  };
  // ...
}
```

advanced-components/components-cookbook/src/components/Messages/ChatMessage.js

```
class ChatMessage extends React.Component {
  // ...
  static contextTypes = {
    userMap: PropTypes.object,
  };
  // ...
}
```

Now anywhere in any one of our child components (that have `contextTypes` defined), we can reach into the parent and grab the `users` without needing to pass them along manually via `props`. The `context` data is set on the `this.context` object of the component with `contextTypes` defined.

For instance, our complete `ThreadList` might look something like:

advanced-components/components-cookbook/src/components/Messages/ThreadList.js

```
class ThreadList extends React.Component {
  // ...
  render() {
    return (
      <div className={styles.threadList}>
        <ul className={styles.list}>
          {this.context.users.map((u, idx) => {
            return (
              <UserListing
                onClick={this.props.onClick}
                key={idx}
                index={idx}
                user={u}
              />
            );
          })}
        </ul>
      </div>
    );
  }
```

If `contextTypes` is defined on a component, then several of it's lifecycle methods will get passed an additional argument of `nextContext`:

advanced-components/components-cookbook/src/components/Messages/ThreadList.js

```
class ThreadList extends React.Component {
  // ...
  static contextTypes = {
    users: PropTypes.array,
  };
  // ...
  componentWillReceiveProps(nextProps, nextContext) {
    // ...
  }
  // ...
  shouldComponentUpdate(nextProps, nextState, nextContext) {
    // ...
  }
  // ...
  componentWillUpdate(nextProps, nextState, nextContext) {
```



```
    // ...
  }
  // ...
  componentDidUpdate(prevProps, prevState, prevContext) {
    // ...
  }
```

In a functional stateless component, `context` will get passed as the second argument:

We talk about stateless components below

advanced-components/components-cookbook/src/components/Messages/ChatHeader.js

```
const ChatHeader = (props, context) => {
  // ...
};
```

Using global variables in JavaScript is usually a never good idea and `context` is usually best reserved for limited situations where a global variable needs to be retrieved, such as a logged-in user. We err on the side of caution in terms of using `context` in our production apps and tend to prefer `props`.

`state`

The second type of data we'll deal with in our components is `state`. To know when to apply state, we need to understand the concept of *stateful* components. Any time a component needs to *hold on to a dynamic piece of data*, that component can be considered stateful.

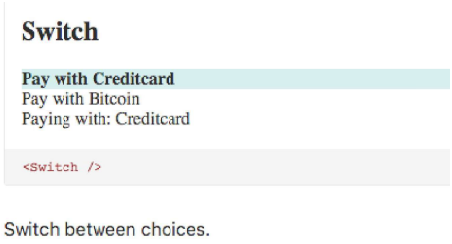
For instance, when a light switch is turned on, that light switch is holding the state of "on." Turning a light off can be described as flipping the state of the light to "off."

In building our apps, we might have a switch that describe a particular setting, such as an input that requires validation, or a presence value for a particular user in a chat application. These are all cases for keeping state about a component within it.

We'll refer to components that hold local-mutable data as *stateful* components. We'll talk a bit more below about when we should use component state. For now, know that it's a good idea to have as **few stateful components as possible**. This is because state introduces complexity and makes composing components more difficult. That said, sometimes we need component-local state, so let's look at how to implement it, and we'll discuss *when* to use it later..

Using `state`: Building a Custom Radio Button

In this example, we're going to use internal state to build a radio button to switch between payment methods. Here's what the form will look like when we're done:



Simple Switch

Let's look at how to make a component stateful:

advanced-components/components-cookbook/src/components/Switch/steps/Switch1.js

```
3  class Switch extends React.Component {
4    state = {};
5
6    render() {
7      return <div><em>Template will be here</em></div>;
8    }
9  }
10
11 module.exports = Switch;
```

That's it! Of course, just setting state on the component isn't all that interesting. To *use* the state on our component, we'll reference it using `this.state`.

advanced-components/components-cookbook/src/components/Switch/steps/Switch2.js

```
3  const CREDITCARD = 'Creditcard';
4  const BTC = 'Bitcoin';
5
6  class Switch extends React.Component {
7    state = {
8      payMethod: BTC,
9    };
10
11   render() {
12     return (
13       <div className='switch'>
14         <div className='choice'>Creditcard</div>
15         <div className='choice'>Bitcoin</div>
16         Pay with: {this.state.payMethod}
17       </div>
18     );
19   }
20 }
21
22 module.exports = Switch;
```

In our render function, we can see the choices our users can pick from (although we can't change a method of payment yet) and their current choice stored in the component's `state`. This `Switch` component is now stateful as it's keeping track of the user's preferred method of payment.

Our payment switch isn't yet interactive; we cannot change the state of the component. Let's hook up our first bit of interactivity by adding an event handler to run when our user selects a different payment method.

In order to add interaction, we'll want to respond to a click event. To add a callback handler to *any* component, we can use the `onClick` attribute on a component. The `onClick` handler will be fired anytime the component it's defined on is clicked.

advanced-components/components-cookbook/src/components/Switch/steps/Switch3.js

```
21      return (
22        <div className='switch'>
23          <div
24            className='choice'
25            onClick={this.select(CREDITCARD)} // add this
26          >Creditcard</div>
27          <div
28            className='choice'
29            onClick={this.select(BTC)} // ... and this
30          >Bitcoin</div>
31          Pay with: {this.state.payMethod}
32        </div>
33      );
```

Using the `onClick` attribute, we've attached a callback handler that will be called every time either one of the `<div>` elements are clicked.

The `onClick` handler expects to receive a *function* that it will call when the click event occurs. Let's look at the `select` function:

advanced-components/components-cookbook/src/components/Switch/steps/Switch3.js

```
6   class Switch extends React.Component {
7     state = {
8       payMethod: BTC,
9     };
10
11    select = (choice) => {
12      return (evt) => {
13        // <-- handler starts here
14        this.setState({
15          payMethod: choice,
16        });
17      };
18    };
```

Notice two things about `select`:

1. It returns a function
2. It uses `setState`

Returning a New Function

Notice something interesting about `select` and `onClick`: the attribute `onClick` expects a *function* to be passed in, but we're *calling* a function first. That's because the `select` function will *return* a function itself.

This is a common pattern for passing arguments to handlers. We *close over* the `choice` argument when we call `select`. `select` returns a new function that will call `setState` with the appropriate `choice`.

When one of the child `<div>` elements are clicked, the handler function will be called. Note that `select` is actually called during `render`, and it's the *return value* of `select` that gets called `onClick`.

Updating the State

When the handler function is called, the component will call `setState` on itself. Calling `setState` triggers a refresh, which means the `render` function will be called again, and we'll be able to see the current `state.payMethod` in our view.

`setState` has performance implications

Since the `setState` method triggers a refresh, we want to be careful about how often we call it.

Modifying the actual-DOM is slow so we don't want to cause a cascade of `setStates` to be called, as that could result it poor performance for our user.

Viewing the Choice

In our component we don't (yet) have a way to indicate which choice has been selected other than the accompanying text.

It would be nice if the choice itself had a visual indication of being the selected one. We usually do this with CSS by applying an `active` class. In our example, we use the `className` attribute.

In order to do this, we'll need to add some logic around which CSS classes to add depending upon the current state of the component.

But before we add too much logic around the CSS, let's refactor component to use a function to render each choice:

advanced-components/components-cookbook/src/components/Switch/steps/Switch4.js

```
      <div className='choice' onClick={this.select(choice)}>
        {choice}
      </div>
    );
  };

  render() {
    return (
      <div className='switch'>
        {this.renderChoice(CREDITCARD)}
        {this.renderChoice(BTC)}
        Pay with: {this.state.payMethod}
      </div>
    );
  }
}

module.exports = Switch;
```

Now, instead of putting all render code into `render()` function, we isolate the choice rendering into it's own function.

Lastly, let's add the `.active` class to the `<div>` choice component.

advanced-components/components-cookbook/src/components/Switch/steps/Switch5.js

```
    const cssClasses = [];

    if (this.state.payMethod === choice) {
      cssClasses.push(styles.active); // add .active class
    }

    return (
      <div
        className='choice'
        onClick={this.select(choice)}
        className={cssClasses}
      >
        {choice}
      </div>
    );
```



```
  };

  render() {
    return (
      <div className='switch'>
        {this.renderChoice(CREDITCARD)}
        {this.renderChoice(BTC)}
        Pay with: {this.state.payMethod}
      </div>
    );
  }
```

Notice that we push the style `styles.active` onto the `cssClassses` array. Where did `styles` come from?

For this code example, we're using a webpack loader to import the CSS. Diving in to how webpack works is beyond the scope of this chapter. But just so you know how we're using it, there are two things to know:

1. We're importing the styles like this: `import styles from '../Switch.css';`
2. This means all of the styles in that file are accessible like an object - e.g. `styles.active` gives us a reference to the `.active` class from `Switch.css`

We do it this way because it's a form of *CSS encapsulation*. That is, the *actual* CSS class won't actually be `.active`, which means we won't conflict with other components that might use the same class name.

Stateful components

Defining state on our component requires us to set an instance variable called `this.state` in the object prototype class. In order to do this, it requires us to set the state in one of two places, either as a property of the class or in the constructor.

Setting up a stateful component in this way allows us to:

1. It allows us to define the *initial* state of our component.
2. It tells React that our component will be stateful. Without this method defined, our component will be considered to be stateless.

For a component, this looks like:

advanced-components/components-cookbook/src/components/InitialState/Component.js

```
class InitialStateComponent extends React.Component {
  // ...
  constructor(props) {
    super(props)

    this.state = {
      currentValue: 1,
      currentUser: {
        name: 'Ari'
      }
    }
  }
  // ...
}
```

In this example, the `state` object is just a JavaScript object, but we can return anything in this function. For instance, we may want to set it to a single value:

advanced-components/components-cookbook/src/components/InitialState/Component.js

```
class Counter extends React.Component {
  constructor(props) {
    super(props)

    this.state = 0
  }
}
```

Setting `props` inside of our component is usually always a bad idea. Setting the initial value of the `state` property is the *only* time we should ever use props when dealing with a component's state. That is, if we ever want to set the value of a prop to the state, we should do it here.

For instance, if we have a component where the `prop` indicates a `value` of the component, we should apply that value to the `state` in the `constructor()` method. A better name for the value as a prop is `initialValue`, indicating that the initial state of the value will be set.

For example, consider a `Counter` component that displays some count and contains an increment and decrement button. We can set the initial value of the counter like this:

advanced-components/components-cookbook/src/components/Counter/CounterWrapper.js

```
const CounterWrapper = props => (
  <div>
    <Counter initialValue={125} />
  </div>
)
```

From the usage of the `<Counter>` component, we know that the value of the `Counter` will change simply by the name `initialValue`. The `Counter` component can use this prop in `constructor()`, like so:

advanced-components/components-cookbook/src/components/Counter/Counter.js

```
class Counter extends Component {
  constructor(props) {
    super(props);

    this.state = {
      value: this.props.initialValue
    };
  }
  // ...
}
```

Since the constructor is run once and only once before the component itself is mounted, we can use it to establish our initial state.

State updates that depend on the current state

`Counter` has buttons for incrementing and decrementing the count:

The `Counter` component

When the "-" button is pressed, React will invoke `decrement()`. `decrement()` will subtract `1` from state's `value`. Something like this would appear to be sufficient:

advanced-components/components-cookbook/src/components/Counter/Counter1.js

```
  decrement = () => {
    // Appears correct, but there is a better way
    const nextValue = this.state.value - 1;
    this.setState({
      value: nextValue,
    });
  }
```

However, **whenever a state update depends on the current state, it is preferable to pass a function to `setState()`**. We can do so like this:

advanced-components/components-cookbook/src/components/Counter/Counter.js

```
  decrement = () => {
    this.setState(prevState => {
      return {
        value: prevState.value - 1,
      };
    });
  }
```

`setState()` will invoke this function with the previous version of the state as the first argument.

Why is setting state this way necessary? Because **`setState()` is asynchronous**.

Here's an example. Let's say we're using the first `decrement()` method where we pass an object to `setState()`. When we invoke `decrement()` for the first time, `value` is `125`. We'd then invoke `setState()`, passing an object with a `value` of `124`.

However, **the state will not necessarily be updated immediately**. Instead, React will add our requested state update to its queue.

Let's say that the user is particularly fast with her mouse and her computer is particularly slow with its processing. The user manages to click the decrement button *again* before React gets around to our previous state update. Responding to user interactions are high-priority, so React invokes `decrement()`. The `value` in state is *still* `125`. So, we enqueue *another* state update, again setting `value` to `124`.

React then commits both state updates. To the dismay of our astute and quick-fingered user, instead of the correct `value` of `123` the app shows a count of `124`.

In our simple example, there's a thin chance this bug would occur. But as a React app grows in complexity, React might encounter periods where it is overloaded with high-priority work, like

animations. And it is conceivable that state updates might be queued for consequential lengths of time.

Whenever a state transition depends on the current state, using a function to set the state helps to avoid the chance for such enigmatic bugs to materialize.

For further reading on this topic, see our own Sophia Shoemaker's post Using a function in `setState` instead of an object[53].

Thinking About State

Spreading state throughout our app can make it difficult to reason about. When building stateful components, we should be mindful about **what** we put in state and **why** we're using state.

Generally, we want to minimize the number of components in our apps that keep component-local state.

If we have a component that has UI states which:

1. cannot be "fetched" from outside or
2. cannot be passed into the component,

that's usually a case for building state into the component.

However, any data that can be passed in through `props` or by other components is usually best to leave untouched. The *only* information we should ever put in state are values that are not computed and do not need to be *sync'd* across the app.

> The decision to put state in our components or not is deeply related to the tension between "object-oriented programming" and "functional programming".
>
> In functional programming, if you have a pure function, then calling the same function, with the same arguments, will always return the same value for a given set of inputs. This makes the behavior of a pure function easy to reason about, because the output is consistent at all times, with respect to the inputs.
>
> In object-oriented programming you have objects which hold on to state within that object. The object state then becomes implicit parameters to the methods on the object. The state can change and so calling the same function, with the same arguments, at different times in your program can return different answers.
>
> This is related to `props` and `state` in React components because you can think of `props` as "arguments" to our components and `state` as "instance variables" to an object.
>
> If our component uses only `props` for configuring a component (and it does not use `state` or any

[53] https://medium.com/@shopsifter/using-a-function-in-setstate-instead-of-an-object-1f5cfd6e55d1

other outside variables) then we can easily predict how a particular component will render.

However, if we use mutable, component-local state then it becomes more difficult to understand what a component will render at a particular time.

So while carrying "implicit arguments" through state can be convenient, it can also make the system difficult to reason about.

That said, state can't be avoided entirely. The real world has state: when you flip a light switch the world has now changed - our programs have to be able to deal with state in order to operate in the real world.

The good news is that there are a variety of tools and patterns that have emerged for dealing with state in React (notably Flux and its variants), which we talk about elsewhere in the book. The rule of thumb you should work with is to **minimize the number of components with `state`**.

Keeping state is usually good to enforce and maintain consistent UI that wouldn't otherwise be updated. Additionally, one more thing to keep in mind is that we should try to minimize the amount of information we put into our state. The smaller and more serializable we can keep it (i.e. can we easily turn it into JSON), the better. Not only will our app be the faster, but it will be easier to reason about. It's usually a red-flag when our state gets large and/or unmanageable.

One way that we can mitigate and minimize the complex states is by building our apps with a single stateful component composed of stateless components: components that do not keep state.

Stateless Components

An alternative approach to building *stateful* components would be to use *stateless* components. Stateless components are intended to be lightweight components that do not need any special handling around the component.

Stateless components are React's lightweight way of building components that only need the `render()` method.

Let's look an example of a stateless component:

advanced-components/components-cookbook/src/components/Header/StatelessHeader.js

```
const Header = function(props) {
  return (<h1>{props.headerText}</h1>)
}
```

Notice that we don't reference `this` when accessing our props as they are simply passed into the function. In fact, the stateless component here isn't actually a class in the sense that it isn't a `ReactElement`.

Functional, stateless components do *not* have a `this` property to reference. In fact, when we use a stateless component, the React rendering processes does *not* introduce a new `ReactComponent` instance, but instead it is null. They are just functions and do not have a backing instance. These components *cannot* contain state and do not get called with the normal component lifecycle methods. We cannot use `refs` (described below), cannot reference the DOM, etc.

React **does** allow us to use `propTypes` and `defaultProps` on stateless components

With so many constraints, why would we want to use stateless components? There are two reasons:

- Minimizing stateful components and
- Performance

As we discussed above, stateful components often spread complexity throughout a system. A component being stateless, when used properly, can help contain the state in fewer locations, which can make our programs easier to reason about.

Also, since React doesn't have to keep track of component instance in memory, do any dirty checking etc., we can get a significant performance increase.

A good rule of thumb is to use stateless components as much as we can. If we don't need any lifecycle methods and can get away with only a rendering function, using a stateless component is a great choice.

`Switch`ing to Stateless

Can we convert our `Switch` component above to a stateless component? Well, the currently selected payment choice *is* state and so it has to be kept somewhere.

While we can't remove state completely, we could at least isolate it. This is a common pattern in React apps: try to pull the state into a few parent components.

In our `Switch` component we pulled each choice out into the `renderChoice` function. This indicates that this is a good candidate for pulling into it's own stateless component. There's one problem: `renderChoice` is the function that calls `select`, which means that it indirectly is the function that calls `setState`. Let's take a look at how to handle this issue:

advanced-components/components-cookbook/src/components/Switch/steps/Switch6.js

```
const Choice = function (props) {
  const cssClasses = [];

  if (props.active) {
    // <-- check props, not state
    cssClasses.push(styles.active);
  }

  return (
    <div
      className='choice'
      onClick={props.onClick}
      className={cssClasses}
    >
      {props.label} {/* <-- allow any label */}
```

Here we've created a `Choice` function which is *a stateless component*. But we have a problem: if our component is stateless then we can't read from `state`. What do we do instead? **Pass the arguments down through `props`.**

In `Choice` we make three changes (which is marked by comments in the code above):

1. We determine if this choice is the active one by reading `props.active`
2. When a `Choice` is clicked, we call whatever function that is on `props.onClick`
3. The label is determined by `props.label`

All of these changes mean that `Choice` is *decoupled* from the `Switch` statement. We could now conceivably use `Choice` anywhere, as long as we pass `active`, `onClick`, and `label` through the props.

Let's look at how this changes `Switch`:

advanced-components/components-cookbook/src/components/Switch/steps/Switch6.js

```
  render() {
    return (
      <div className='switch'>
        <Choice
          onClick={this.select(CREDITCARD)}
          active={this.state.payMethod === CREDITCARD}
          label='Pay with Creditcard'
        />
```



```

        <Choice
          onClick={this.select(BTC)}
          active={this.state.payMethod === BTC}
          label='Pay with Bitcoin'
        />

        Paying with: {this.state.payMethod}
```

Here we're using our `Choice` component and passing the three props (parameters) `active`, `onClick`, and `label`. What's neat about this is that we could easily:

1. Change what happens when we click this choice by changing the input to `onClick`
2. Change the condition by which a particular choice is considered active by changing the `active` prop
3. Change what the label is to any arbitrary string

By creating a stateless component `Choice` we're able to make `Choice` reusable and not be tied to any particular `state`.

Stateless Encourages Reuse

Stateless components are a great way to create reusable components. Because stateless components need to have all of their configuration passed from the outside, we can often reuse stateless components in nearly any project, provided that we supply the right hooks.

Now that we've covered both `props`, `context`, and `state` we're going to cover a couple more advanced features we can use with components.

Our components exist in a hierarchy and sometimes we need to communicate (or manipulate) the children components. The the next section, we're going to discuss how to do this.

Talking to Children Components with `props.children`

While we generally specify `props` ourselves, React provides provides some special `props` for us. In our components, we can refer to child components in the tree using `this.props.children`.

For instance, say we have a `Newspaper` component that holds an `Article`:

advanced-components/components-cookbook/src/components/Article/Newspaper.js

```
const Newspaper = props => {
  return (
    <Container>
      <Article headline="An interesting Article">
        Content Here
      </Article>
    </Container>
  )
}
```

The container component above contains a single child, the `Article` component. How many children does the `Article` component contain? It contains a single child, the text `Content Here`.

In the `Container` component, say that we want to add markup *around* whatever the `Article` component renders. To do this, we write our JSX in the `Container` component, and then place `this.props.children`:

advanced-components/components-cookbook/src/components/Article/Container.js

```
class Container extends React.Component {
  // ...
  render() {
    return (
      <div className='container'>
        {this.props.children}
      </div>
    )
  }
```

The `Container` component above will create a `div` with `class='container'` and the children of this React tree will render within that `div`.

Generally, React will pass the `this.props.children` prop as a list of components if there are multiple children, whereas it will pass a single element if there is only one component.

Now that we know how `this.props.children` works, we should rewrite the previous `Container` component to use `propTypes` to document the API of our component. We can expect that our `Container` is likely to contain multiple `Article` components, but it might also contain only a single `Article`. So let's specify that the `children` prop can be either an element or an array.

If `PropTypes.oneOfType` seems unfamiliar, checkout the appendix on `PropTypes` which explains how it works

advanced-components/components-cookbook/src/components/Article/DocumentedContainer.js

```
class Container extends React.Component {
  static propTypes = {
    children: PropTypes.oneOf([
      PropTypes.element,
      PropTypes.array
    ])
  }
  // ...
  render() {
    return (
      <div className="container">
        {this.props.children}
      </div>
    )
  }
}
```

It can become cumbersome to check what type our `children` prop is every time we want to use `children` in a component. We can handle this a few different ways:

1. Require `children` to be a single child every time (e.g., wrap our children in their own element).
2. Use the `Children` helper provided by React.

The first method of requiring a child to be a single element is straightforward. Rather than defining the children above as `oneOfType()`, we can set the children to a be a single element.

advanced-components/components-cookbook/src/components/Article/SingleChildContainer.js

```
class Container extends React.Component {
  static propTypes = {
    children: PropTypes.element.isRequired,
  }
  // ...
  render() {
    return (
      <div className="container">
        {this.props.children}
      </div>
    )
  }
}
```

Inside the `Container` component we can deal with the children *always* being able to be rendered as a single leaf of the hierarchy.

The second method of is to use the `React.Children` utility helper for dealing with the child components. There are a number of helper methods for handling children, let's look at them now.

`React.Children.map()` & `React.Children.forEach()`

The most common operation we'll use on children is mapping over the list of them. We'll often use a `map` to call `React.cloneElement()` or `React.createElement()` along the children.

`map()` and `forEach()`

Both the `map()` and `forEach()` function execute a provided function once per each element in an iterable (either an object or array).

```
[1, 2, 3].forEach(function(n) {
  console.log("The number is: " + n);
  return n; // we won't see this
})
[1, 2, 3].map(function(n) {
  console.log("The number is: " + n);
  return n; // we will get these
})
```

The difference between `map()` and `forEach()` is that the *return* value of `map()` is an array of the result of the callback function, whereas `forEach()` does not collect results.

So in this case, while both `map()` and `forEach()` will print the `console.log` statements, `map()` will return the array `[1, 2, 3]` whereas `forEach()` will not.

Let's rewrite the previous `Container` to allow a **configurable wrapper component for each child**. The idea here is that this component takes:

1. A prop `component` which is going to wrap each child
2. A prop `children` which is the list of children we're going to wrap

To do this, we call `React.createElement()` to generate a new `ReactElement` for each child:

advanced-components/components-cookbook/src/components/Article/MultiChildContainer.js

```
class Container extends React.Component {
  static propTypes = {
    component: PropTypes.element.isRequired,
    children: PropTypes.element.isRequired
  }
  // ...
  renderChild = (childData, index) => {
    return React.createElement(
      this.props.component,
      {}, // <~ child props
      childData // <~ child's children
    )
  }
  // ...
  render() {
    return (
      <div className='container'>
        {React.Children.map(
          this.props.children,
          this.renderChild
        )}
      </div>
    )
  }
}
```

Again, the difference between `React.Children.map()` and `React.Children.forEach()` is that the former creates an array and returns the result of each function and the latter does not. We'll mostly use `.map()` when we render a child collection.

`React.Children.toArray()`

`props.children` returns a data structure that can be tricky to work with. Often when dealing with children, we'll want to convert our `props.children` object into a regular array, for instance when we want to re-sort the ordering of the children elements. `React.Children.toArray()` converts the `props.children` data structure into an array of the children.

advanced-components/components-cookbook/src/components/Article/ArrayContainer.js

```
class Container extends React.Component {
  static propTypes = {
    component: PropTypes.element.isRequired,
    children: PropTypes.element.isRequired
  }
  // ...
  render() {
    const arr =
      React.Children.toArray(this.props.children);

    return (
      <div className="container">
        {arr.sort((a, b) => a.id < b.id )}
      </div>
    )
  }
}
```

Summary

By using `props` and `context` we get data in to our components and by using `PropTypes` we can specify clear expectations about what we require that data to be.

By using `state` we hold on to component-local data, and we tell our components to re-render whenever that state changes. However state can be tricky! One technique to minimize stateful components is to use stateless, functional components.

Using these tools we can create powerful interactive components. However there is one important set of configurations that we did not cover here: lifecycle methods.

Lifecycle methods like `componentDidMount` and `componentWillUpdate` provide us with powerful hooks into the application process. In the next chapter, we're going to dig deep into component lifecycle and show how we can use those hooks to validate forms, hook in to external APIs, and build sophisticated components.

References

- React Top-Level API Docs[54]
- React Component API Docs[55]

[54] https://facebook.github.io/react/docs/top-level-api.html
[55] https://facebook.github.io/react/docs/component-api.html

Forms

Forms 101

Forms are one of the most crucial parts of our application. While we get some interaction through clicks and mouse moves, it's really through forms where we'll get the majority of our rich input from our users.

In a sense, it's where the rubber meets the road. It's through a form that a user can add their payment info, search for results, edit their profile, upload a photo, or send a message. Forms transform your web **site** into a web **app**.

Forms can be deceptively simple. All you really need are some `input` tags and a `submit` tag wrapped up in a `form` tag. However, creating a rich, interactive, easy to use form can often involve a significant amount of programming:

- Form inputs modify data, both on the page and the server.
- Changes often have to be kept in sync with data elsewhere on the page.
- Users can enter unpredictable values, some that we'll want to modify or reject outright.
- The UI needs to clearly state expectations and errors in the case of validation failures.
- Fields can depend on each other and have complex logic.
- Data collected in forms is often sent asynchronously to a back-end server, and we need to keep the user informed of what's happening.
- We want to be able to test our forms.

If this sounds daunting, don't worry! This is exactly why React was created: to handle the complicated forms that needed to be built at Facebook.

We're going to explore how to handle these challenges with React by building a sign up app. We'll start simple and add more functionality in each step.

Preparation

Inside the code download that came with this book, navigate to `forms`:

```
$ cd forms
```

This folder contains all the code examples for this chapter. To view them in your browser install the dependencies by running `npm install` (or `npm i` for short):

```
$ npm i
```

Once that's finished, you can start the app with `npm start`:

```
$ npm start
```

You should expect to see the following in your terminal:

```
$ npm start

Compiled successfully!

The app is running at:

  http://localhost:3000/
```

You should now be able to see the app in your browser if you go to `http://localhost:3000`.

This app is powered by Create React App, which we cover in the next chapter.

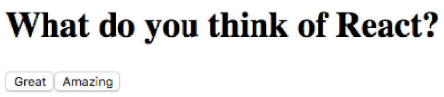
The Basic Button

At their core, forms are a conversation with the user. Fields are the app's questions, and the values that the user inputs are the answers.

Let's ask the user what they think of React.

We could present the user with a text box, but we'll start even simpler. In this example, we'll constrain the response to just one of two possible answers. We want to know whether the user thinks React is either "great" or "amazing", and the simplest way to do that is to give them two buttons to choose from.

Here's what the first example looks like:



Basic Buttons

To get our app to this stage we create a component with a `render()` method that returns a `div` with three child elements: an `h1` with the question, and two `button` elements for the answers. This will look like the following:

forms/src/01-basic-button.js

```
17  render() {
18    return (
19      <div>
20        <h1>What do you think of React?</h1>
21
22        <button
23          name='button-1'
24          value='great'
25          onClick={this.onGreatClick}
26        >
27          Great
28        </button>
29
30        <button
31          name='button-2'
32          value='amazing'
33          onClick={this.onAmazingClick}
34        >
35          Amazing
36        </button>
37      </div>
38    );
39  }
```

So far this looks a lot like how you'd handle a form with vanilla HTML. The important part to pay attention to is the `onClick` prop of the `button` elements. When a `button` is clicked, if it has a function set as its `onClick` prop, that function will be called. We'll use this behavior to know what our user's answer is.

To know what our user's answer is, we pass a different function to each button. Specifically, we'll create function `onGreatClick()` and provide it to the "Great" button and create function `onAmazingClick()` and provide it to the "Amazing" button.

Here's what those functions look like:

forms/src/01-basic-button.js

```
9   onGreatClick = (evt) => {
10    console.log('The user clicked button-1: great', evt);
11  };
12
13  onAmazingClick = (evt) => {
14    console.log('The user clicked button-2: amazing', evt);
15  };
```

When the user clicks on the "Amazing" button, the associated `onClick` function will run (`onAmazingClick()` in this case). If, instead, the user clicked the "Great" button, `onGreatClick()` would be run instead.

Notice that in the `onClick` handler we pass `this.onGreatClick` and *not* `this.onGreatClick()`.

What's the difference?

In the first case (without parens), we're passing the *function* `onGreatClick`, whereas in the second case we're passing the *result of calling the function* `onGreatClick` (which isn't what we want right now).

This becomes the foundation of our app's ability to respond to a user's input. Our app can do different things depending on the user's response. In this case, we log different messages to the `console`.

Events and Event Handlers

Note that our `onClick` functions (`onAmazingClick()` and `onGreatClick()`) accept an argument, `evt`. This is because these functions are *event handlers*.

Handling events is central to working with forms in React. When we provide a function to an element's `onClick prop`, that function becomes an event handler. The function will be called when that event occurs, and it will receive **an event object as its argument**.

In the above example, when the `button` element is clicked, the corresponding event handler function is called (`onAmazingClick()` or `onGreatClick()`) and it is provided with a mouse click event object (`evt` in this case). This object is a `SyntheticMouseEvent`. This `SyntheticMouseEvent` is just a cross-browser wrapper around the browser's native `MouseEvent`, and you'll be able to use it the same way you would a native DOM event. In addition, if you need the original native event you can access it via the `nativeEvent` attribute (e.g. `evt.nativeEvent`).

Event objects contain lots of useful information about the action that occurred. A `MouseEvent` for example, will let you see the x and y coordinates of the mouse at the time of the click, whether or not the shift key was pressed, and (most useful for this example) a reference to the element that was clicked. We'll use this information to simplify things in the next section.

Instead, if we were interested in mouse movement, we could have created an event handler and provided it to the `onMouseMove prop`. In fact, there are many such element `props` that you can provide mouse event handlers to:, `onClick`, `onContextMenu`, `onDoubleClick`, `onDrag`, `onDragEnd`, `onDragEnter`, `onDragExit`, `onDragLeave`, `onDragOver`, `onDragStart`, `onDrop`, `onMouseDown`, `onMouseEnter`, `onMouseLeave`, `onMouseMove`, `onMouseOut`, `onMouseOver`, and `onMouseUp`.

And those are only the mouse events. There are also clipboard, composition, keyboard, focus, form, selection, touch, ui, wheel, media, image, animation, and transition event groups. Each group has its own types of events, and not all events are appropriate for all elements. For example, here we will mainly work with the form events, `onChange` and `onSubmit`, which are related to `form` and `input` elements.

For more information on events in React, see React's documentation on the Event System[56].

Back to the Button

In the previous section, we were able to perform different actions (log different messages) depending on the action of the user. However, the way that we set it up, we'd need to create a separate function for each action. Instead, it would be much cleaner if we provided the same event handler to both buttons, and used information from the event itself to determine our response.

To do this, we replace the two event handlers `onGreatClick()` and `onAmazingClick()` with a new single event handler, `onButtonClick()`.

forms/src/02-basic-button.js

```
9   onButtonClick = (evt) => {
10    const btn = evt.target;
11    console.log(`The user clicked ${btn.name}: ${btn.value}`);
12  };
```

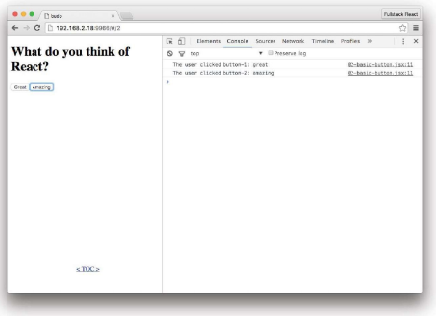
Our click handler function receives an event object, `evt`. `evt` has an attribute `target` that is a reference to the button that the user clicked. This way we can access the button that the user clicked without creating a function for each button. We can then log out different messages for different user behavior.

Next we update our `render()` function so that our `button` elements both use the same event handler, our new `onButtonClick()` function.

[56] https://facebook.github.io/react/docs/events.html

forms/src/02-basic-button.js

```
14   render() {
15     return (
16       <div>
17         <h1>What do you think of React?</h1>
18
19         <button
20           name='button-1'
21           value='great'
22           onClick={this.onButtonClick}
23         >
24           Great
25         </button>
26
27         <button
28           name='button-2'
29           value='amazing'
30           onClick={this.onButtonClick}
31         >
32           Amazing
33         </button>
34       </div>
35     );
36   }
```



One Event Handler for Both Buttons

By taking advantage of the event object and using a shared event handler, we could add 100 new buttons, and we wouldn't have to make any other changes to our app.

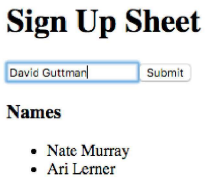
Text Input

In the previous example, we constrained our user's response to only one of two possibilities. Now that we know how to take advantage of event objects and handlers in React, we're going to accept a much wider range of responses and move on to a more typical use of forms: text input.

To showcase text input we'll create a "sign-up sheet" app. The purpose of this app is to allow a user to record a list of names of people who want to sign up for an event.

The app presents the user a text field where they can input a name and hit "Submit". When they enter a name, it is added to a list, that list is displayed, and the text box is cleared so they can enter a new name.

Here's what it will look like:



Sign-Up Adding to a List

Accessing User Input With `refs`

We want to be able to read the contents of the text field when the user submits the form. A simple way to do this is to wait until the user submits the form, find the text field in the DOM, and finally grab its `value`.

To begin we'll start by creating a form element with two child elements: a text input field and a submit button:

forms/src/03-basic-input.js

```
14  render() {
15    return (
16      <div>
17        <h1>Sign Up Sheet</h1>
18
19        <form onSubmit={this.onFormSubmit}>
20          <input
21            placeholder='Name'
22            ref='name'
23          />
24
25          <input type='submit' />
26        </form>
27      </div>
28    );
29  }
```

This is very similar to the previous example, but instead of two `button` elements, we now have a `form` element with two child elements: a text field and a submit button.

There are two things to notice. First, we've added an `onSubmit` event handler to the `form` element. Second, we've given the text field a `ref` prop of `'name'`.

By using an `onSubmit` event handler on the `form` element this example will behave a little differently than before. One change is that the handler will be called either by clicking the "Submit" button, or by pressing "enter"/"return" while the `form` has focus. This is more user-friendly than forcing the user to click the "Submit" button.

However, because our event handler is tied to the `form`, the event object argument to the handler is less useful than it was in the previous example. Before, we were able to use the `target` prop of the event to reference the `button` and get its value. This time, we're interested in the text field's value. One option would be to use the event's `target` to reference the `form` and from there we could find the child `input` we're interested in, but there's a simpler way.

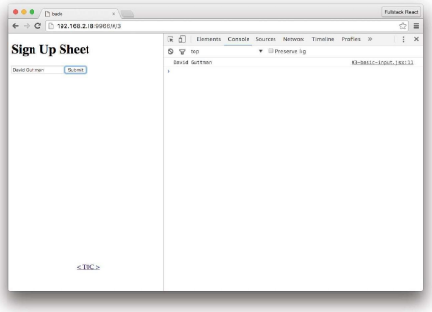
In React, if we want to easily access a DOM element in a component we can use `refs` (references). Above, we gave our text field a `ref` property of `'name'`. Later when the `onSubmit` handler is called, we have the ability to access `this.refs.name` to get a reference to that text field. Here's what that looks like in our `onFormSubmit()` event handler:

forms/src/03-basic-input.js

```
9   onFormSubmit = (evt) => {
10    evt.preventDefault();
11    console.log(this.refs.name.value);
12  };
```

Use `preventDefault()` with the `onSubmit` handler to prevent the browser's default action of submitting the form.

As you can see, by using `this.refs.name` we gain a reference to our text field element and we can access its `value` property. That `value` property contains the text that was entered into the field.



Logging The Name

With just the two functions `render()` and `onFormSubmit()`, we should now be able to see the `value` of the text field in our console when we click "Submit". In the next step we'll take that `value` and display it on the page.

Using User Input

Now that we've shown that we can get user submitted names, we can begin to use this information to change the app's state and UI.

The goal of this example is to show a list with all of the names that the user has entered. React makes this easy. We will have an array in our state to hold the names, and in `render()` we will use that array to populate a list.

When our app loads, the array will be empty, and each time the user submits a new name, we will add it to the array. To do this, we'll make a few additions to our component.

First, we'll create a `names` array in our state. In React, when we're using ES6 component classes we can set the initial value of our `state` object by defining a property of `state`.

Here's what that looks like:

forms/src/04-basic-input.js

```
6  module.exports = class extends React.Component {
7    static displayName = "04-basic-input";
8    state = { names: [] }; // <-- initial state
```

`static` belongs to the class

Notice in this component we have the line:

```
1  static displayName = "04-basic-input";
```

This means that this component class has a *static* property `displayName`. When a property is static, that means it is a class property (instead of an instance property). In this case, we're going to use this `displayName` when we show the list of examples on the demo listing page.

Next, we'll modify `render()` to show this list. Below our `form` element, we'll create a new `div`. This new container `div` will hold a heading, `h3`, and our names list, a `ul` parent with a `li` child for each name. Here's our updated `render()` method:

forms/src/04-basic-input.js

```
18   render() {
19     return (
20       <div>
21         <h1>Sign Up Sheet</h1>
22
23         <form onSubmit={this.onFormSubmit}>
24           <input
25             placeholder='Name'
26             ref='name'
27           />
28
29           <input type='submit' />
30         </form>
31
```



```
32           <div>
33             <h3>Names</h3>
34             <ul>
35               { this.state.names.map((name, i) => <li key={i}>{name}</li>) }
36             </ul>
37           </div>
38         </div>
39       );
40     }
```

ES2015 gives us a compact way to insert `li` children. Since `this.state.names` is an array, we can take advantage of its `map()` method to return a `li` child element for each name in the array. Also, by using "arrow" syntax, for our iterator function in `map()`, the `li` element is returned without us explicitly using `return`.

One other thing to note here is that we provide a `key prop` to the `li` element. React will complain when we have children in an array or iterator (like we do here) and they don't have a `key prop`. React wants this information to keep track of the child and make sure that it can be reused between render passes.

We won't be removing or reordering the list here, so it is sufficient to identify each child by its index. If we wanted to optimize rendering for a more complex use-case, we could assign an immutable id to each name that was not tied to its value or order in the array. This would allow React to reuse the element even if its position or value was changed.

See React's documentation on Multiple Components and Dynamic Children[57] for more information.

Now that `render()` is updated, the `onFormSubmit()` method needs to update the `state` with the new name. To add a name to the `names` array in our `state` we might be tempted to try to do something like `this.state.names.push(name)`. However, React relies on `this.setState()` to mutate our `state` object, which will then trigger a new call to `render()`.

The way to do this properly is to:

1. create a new variable that copies our current `names`
2. add our new name to that array, and finally
3. use that variable in a call to `this.setState()`.

We also want to clear the text field so that it's ready to accept additional user input. It would not be very user friendly to require the user to delete their input before adding a new name. Since we already have access to the text field via `refs`, we can set its `value` to an empty string to clear it.

This is what `onFormSubmit()` should look like now:

[57] https://facebook.github.io/react/docs/multiple-components.html#dynamic-children

forms/src/04-basic-input.js

```
  onFormSubmit = (evt) => {
    const name = this.refs.name.value;
    const names = [ ...this.state.names, name ];
    this.setState({ names: names });
    this.refs.name.value = '';
    evt.preventDefault();
  };
```

At this point, our sign-up app is functional. Here's a rundown of the application flow:

1. User enters a name and clicks "Submit".
2. `onFormSubmit` is called.
3. `this.refs.name` is used to access the value of the text field (a name).
4. The name is added to our `names` list in the `state`.
5. The text field is cleared so that it is ready for more input.
6. `render` is called and displays the updated list of names.

So far so good! In the next sections we'll improve it further.

Uncontrolled vs. Controlled Components

In the previous sections we took advantage of `refs` to access the user's input. When we created our `render()` method we added an `input` field with a `ref` attribute. We later used that attribute to get a reference to the rendered `input` so that we could access and modify its `value`.

We covered using `refs` with forms because it is conceptually similar to how one might deal with forms without React. However, by using `refs` this way, we opt out of a primary advantage of using React.

In the previous example we access the DOM directly to retrieve the name from the text field, as well as manipulate the DOM directly by resetting the field after a name has been submitted.

With React we shouldn't have to worry about modifying the DOM to match application state. We should concentrate only on altering `state` and rely on React's ability to efficiently manipulate the DOM to match. This provides us with the certainty that for any given value of `state`, we can predict what `render()` will return and therefore know what our app will look like.

In the previous example, our text field is what we would call an "uncontrolled component". This is another way of saying that React does not "control" how it is rendered – specifically its value. In other words, React is hands-off, and allows it to be freely influenced by user interaction. This means that knowing the application state is not enough to predict what the page (and specifically the `input` field) looks like. Because the user could have typed (or not typed) input into the field, the only way to know what the `input` field looks like is to access it via `refs` and check its `value`.

There is another way. By converting this field to a "controlled component", we give React control over it. It's `value` will always be specified by `render()` and our application state. When we do this, we can predict how our application will look by examining our `state` object.

By directly tying our view to our application state we get certain features for very little work. For example, imagine a long `form` where the user must answer many questions by filling out lots of `input` fields. If the user is halfway through and accidentally reloads the page that would ordinarily clear out all the fields. If these were controlled components and our application state was persisted to `localStorage`, we would be able to come back exactly where they left off. Later, we'll get to another important feature that controlled components pave the way for: validation.

Accessing User Input With `state`

Converting an uncontrolled `input` component to a controlled one requires three things. First, we need a place in `state` to store its value. Second, we provide that location in `state` as its `value` `prop`. Finally, we add an `onChange` handler that will update its value in `state`. The flow for a controlled component looks like this:

1. The user enters/changes the input.
2. The `onChange` handler is called with the "change" event.
3. Using `event.target.value` we update the `input` element's `value` in `state`.
4. `render()` is called and the `input` is updated with the new value in `state`.

Here's what our `render()` looks like after converting the `input` to a controlled component:

forms/src/05-state-input.js

```
  render() {
    return (
      <div>
        <h1>Sign Up Sheet</h1>

        <form onSubmit={this.onFormSubmit}>
          <input
            placeholder='Name'
            value={this.state.name}
            onChange={this.onNameChange}
          />

          <input type='submit' />
```

```
        </form>

        <div>
          <h3>Names</h3>
          <ul>
            { this.state.names.map((name, i) => <li key={i}>{name}</li>) }
          </ul>
        </div>
      </div>
    );
  }
```

The only difference in our `input` is that we've removed the `ref` prop and replaced it with both a `value` and an `onChange` prop.

Now that the `input` is "controlled", its `value` will always be set equal to a property of our `state`. In this case, that property is `name`, so the `value` of the `input` is `this.state.name`.

While not strictly necessary, it's a good habit to provide sane defaults for any properties of `state` that will be used in our component. Because we now use `state.name` for the `value` of our `input`, we'll want to choose what value it will have before the user has had a chance to provide one. In our case, we want the field to be empty, so the default value will be an empty string, `''`:

forms/src/05-state-input.js

```
  state = {
    name: '',
    names: [],
  };
```

If we had just stopped there, the `input` would effectively be disabled. No matter what the user types into it, its `value` wouldn't change. In fact, if we left it like this, React would give us a warning in our console.

To make our `input` operational, we'll need to listen to its `onChange` events and use those to update the `state`. To achieve this, we've created an event handler for `onChange`. This handler is responsible for updating our `state` so that `state.name` will always be updated with what the user has typed into the field. We've created the method `onNameChange()` for that purpose.

Here's what `onNameChange()` looks like now:

forms/src/05-state-input.js

```
  onNameChange = (evt) => {
    this.setState({ name: evt.target.value });
  };
```

`onNameChange()` is a very simple function. Like we did in a previous section, we use the event passed to the handler to reference the field and get its `value`. We then update `state.name` with that `value`.

Now the controlled component cycle is complete. The user interacts with the field. This triggers an `onChange` event which calls our `onNameChange()` handler. Our `onNameChange()` handler updates the `state`, and this in turn triggers `render()` to update the field with the new `value`.

Our app still needs one more change, however. When the user submits the form, `onFormSubmit()` is called, and we need that method to add the entered name (`state.name`) to the names list (`state.names`). When we last saw `onFormSubmit()` it did this using `this.refs`. Since we're no longer using `refs`, we've updated it to the following:

forms/src/05-state-input.js

```
  onFormSubmit = (evt) => {
    const names = [ ...this.state.names, this.state.name ];
    this.setState({ names: names, name: '' });
    evt.preventDefault();
  };
```

Notice that to get the current entered name, we simply access `this.state.name` because it will be continually updated by our `onNameChange()` handler. We then append that to our names list, `this.state.names` and update the `state`. We also clear `this.state.name` so that the field is empty and ready for a new name.

While our app didn't gain any new features in this section, we've both paved the way for better functionality (like validation and persistence) while also taking greater advantage of the React paradigm.

Multiple Fields

Our sign-up sheet is looking good, but what would happen if we wanted to add more fields? If our sign-up sheet is like most projects, it's only a matter of time before we want to add to it. With forms, we'll often want to add inputs.

If we continue our current approach and create more controlled components, each with a corresponding state property and an `onChange` handler, our component will become quite verbose. Having a one-to-one-to-one relationship between our inputs, state, and handlers is not ideal.

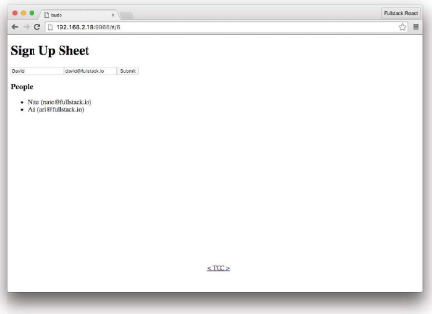
Let's explore how we can modify our app to allow for additional inputs in a clean, maintainable way. To illustrate this, let's add email address to our sign-up sheet.

In the previous section our `input` field has a dedicated property on the root of our `state` object. If we were to do that here, we would add another property, `email`. To avoid adding a property for each input on `state`, let's instead add a `fields` object to store the values for all of our fields in one place. Here's our new initial `state`:

forms/src/06-state-input-multi.js

```
9   state = {
10    fields: {
11      name: '',
12      email: ''
13    },
14    people: [],
15  };
```

This `fields` object can store state for as many inputs as we'd like. Here we've specified that we want to store fields for `name` and `email`. Now, we will find those values at `state.fields.name` and `state.fields.email` instead of `state.name` and `state.email`.



Name and Email Fields

Of course, those values will need to be updated by an event handler. We *could* create an event handler for each field we have in the form, but that would involve a lot of copy/paste and needlessly bloat our component. Also it would make maintainability more difficult, as any change to a form would need to be made in multiple places.

Instead of creating an `onChange` handler for each input, we can create a single method that accepts change events from **all** of our inputs. The trick is to write this method in such a way that it updates

the correct property in `state` depending on the `input` field that triggered the event. To pull this off, the method uses the `event` argument to determine which input was changed and update our `state.fields` object accordingly. For example, if we have an `input` field and we were to give it a `name` prop of `"email"`, when that field triggers an event, we would be able to know that it was email field, because `event.target.name` would be `"email"`.

To see what this looks like in practice, here's the updated `render()`:

forms/src/06-state-input-multi.js

```
38  render() {
39    return (
40      <div>
41        <h1>Sign Up Sheet</h1>
42
43        <form onSubmit={this.onFormSubmit}>
44          <input
45            placeholder='Name'
46            name='name'
47            value={this.state.fields.name}
48            onChange={this.onInputChange}
49          />
50
51          <input
52            placeholder='Email'
53            name='email'
54            value={this.state.fields.email}
55            onChange={this.onInputChange}
56          />
57
58          <input type='submit' />
59        </form>
60
61        <div>
62          <h3>People</h3>
63          <ul>
64            { this.state.people.map(({ name, email }, i) =>
65              <li key={i}>{name} ({ email })</li>
66            ) }
67          </ul>
68        </div>
69      </div>
70    );
71  }
```

There are a several things to note: first, we've added a second `input` to handle email addresses.

Second, we've changed the `value` prop of the `input` fields so that they don't access attributes on the root of the `state` object. Instead they access the attributes of `state.fields`. Looking at the code above, the `input` for name now has its `value` set to `this.state.fields.name`.

Third, both `input` fields have their `onChange` prop set to the same event handler, `onInputChange()`. We'll see below how we modified `onNameChange()` to be a more general event handler that can accept events from any field, not just "name".

Fourth, our `input` fields now have a `name` prop. This is related to the last point. To allow our general event handler, `onInputChange()`, to be able to tell where the change event came from and where we should store it in our state (e.g. if the change comes from the "email" `input` the new value should be stored at `state.fields.email`), we provide that `name` prop so that it can be pulled off of the event via its `target` attribute.

Finally, we modify how our people list is rendered. Because it's no longer just a list of names, we modify our `li` element to display both the previous `name` attribute as well as the new `email` data we plan to have.

To make sure that all the data winds up in the right place, we'll need to make sure that our event handlers are properly modified. Here's what the `onInputChange()` event handler (that gets called when any field's input changes) should look like:

forms/src/06-state-input-multi.js

```
32  onInputChange = (evt) => {
33    const fields = this.state.fields;
34    fields[evt.target.name] = evt.target.value;
35    this.setState({ fields });
36  };
```

At its core this is similar to what we did before in `onNameChange()` in the last section. The two key differences are that:

1. we are updating a value nested in the `state` (e.g. updating `state.fields.email` instead of `state.email`), and
2. we're using `evt.target.name` to inform which attribute of `state.fields` needs to be updated.

To properly update our state, we first grab a local reference to `state.fields`. Then, we use information from the event (`evt.target.name` and `evt.target.value`) to update the local reference. Lastly, we `setState()` with the modified local reference.

To get concrete, let's go through what would happen if the user enters "someone@somewhere.com" into the "email" field.

First, `onInputChange()` would be called with the `evt` object as an argument. `evt.target.name` would be `"email"` (because `"email"` is set as its `name` prop in `render()`) and `evt.target.value` would be `"someone@somewhere.com"` (because that's what they entered into the field).

Next, `onInputChange()` would grab a local reference to `state.fields`. If this is the first time there was input, `state.fields` and our local reference will be the default `fields` in `state`, `{ name: '', email: '' }`. Then, the local reference would be modified so that it becomes `{ name: '', email: "someone@somewhere.com" }`.

And finally, `setState()` is called with that change.

At this point, `this.state.fields` will always be in sync with any text in the `input` fields. However, `onFormSubmit()` will need to be changed to get that information into the list of people who have signed up. Here's what the updated `onFormSubmit()` looks like:

forms/src/06-state-input-multi.js

```
17  onFormSubmit = (evt) => {
18    const people = [
19      ...this.state.people,
20      this.state.fields,
21    ];
22    this.setState({
23      people,
24      fields: {
25        name: '',
26        email: ''
27      }
28    });
29    evt.preventDefault();
30  };
```

In `onFormSubmit()` we first obtain a local reference to the list of people who have signed up, `this.state.people`. Then, we add our `this.state.fields` object (an object representing the name and email currently entered into the fields) onto the `people` list. Finally, we use `this.setState()` to simultaneously update our list with the new information and clear all the fields by returning `state.fields` to the empty defaults, `{ name: '', email: '' }`.

The great thing about this is that we can easily add as many input fields as we want with very minimal changes. In fact, only the `render()` method would need to change. For each new field, all we would have to do is add another `input` field and change how the list is rendered to display the new field.

For example, if we wanted to add a field for phone number, we would add a new `input` with appropriate `name` and `value` props: `name` would be `phone` and `value` would be `this.state.fields.phone`. `onChange`, like the others, would be our existing `onInputChange()` handler.

After doing that our `state` will automatically keep track of the phone field and will add it to the `state.people` array and we could change how the view (e.g. the `li`) displays the information.

At this point we have a functional app that's well situated to be extended and modified as requirements evolve. However, it is missing one crucial aspect that almost all forms need: validation.

On Validation

Validation is so central to building forms that it's rare to have a form without it. Validation can be both on the level of the **individual field** and on the **form as a whole**.

When you validate on an individual field, you're making sure that the user has entered data that conforms to your application's expectations and constraints as it relates to that piece of data.

For example, if we want a user to enter an email address, we expect their input to look like a valid email address. If the input does not look like an email address, they might have made a mistake and we're likely to run into trouble down the line (e.g. they won't be able to activate their account). Other common examples are making sure that a zip code has exactly five (or nine) numerical characters and enforcing a password length of at least some minimum length.

Validation on the form as a whole is slightly different. Here is where you'll make sure that all required fields have been entered. This is also a good place to check for internal consistency. For example you might have an order form where specific options are required for specific products.

Additionally, there are trade-offs for "how" and "when" we validate. On some fields we might want to give validation feedback in real-time. For example, we might want to show password strength (by looking at length and characters used) while the user is typing. However, if we want to validate the availability of a username, we might want to wait until the user has finished typing before we make a request to the server/database to find out.

We also have options for how we display validation errors. We might style the field differently (e.g. a red outline), show text near the field (e.g. "Please enter a valid email."), and/or disable the form's submit button to prevent the user from progressing with invalid information.

As for our app, we can begin with validation of the form as a whole and

1. make sure that we have both a name and email and
2. make sure that the email is a valid address.

Adding Validation to Our App

To add validation to our sign-up app we've made some changes. At a high level these changes are

1. add a place in `state` to store validation errors if they exist,
2. change our `render()` method will show validation error messages (if they exist) with red text next to each field,

3. add a new `validate()` method that takes our `fields` object as an argument and returns a `fieldErrors` object, and
4. `onFormSubmit()` will call the new `validate()` method to get the `fieldErrors` object, and if there are errors it will add them to the state (so that they can be shown in `render()`) and return early without adding the "person" to the list, `state.people`.

First, we've changed our initial `state`:

forms/src/07-basic-validation.js

```
10  state = {
11    fields: {
12      name: '',
13      email: '',
14    },
15    fieldErrors: {},
16    people: [],
17  };
```

The only change here is that we've created a default value for the `fieldErrors` property. This is where we'll store errors for each of the field if they exist.

Next, here's what the updated `render()` method looks like:

forms/src/07-basic-validation.js

```
51  render() {
52    return (
53      <div>
54        <h1>Sign Up Sheet</h1>
55
56        <form onSubmit={this.onFormSubmit}>
57
58          <input
59            placeholder='Name'
60            name='name'
61            value={this.state.fields.name}
62            onChange={this.onInputChange}
63          />
64
65          <span style={{ color: 'red' }}>{ this.state.fieldErrors.name }</span>
66
67          <br />
68
```

```
          <input
            placeholder='Email'
            name='email'
            value={this.state.fields.email}
            onChange={this.onInputChange}
          />

          <span style={{ color: 'red' }}>{ this.state.fieldErrors.email }</span>

          <br />

          <input type='submit' />
        </form>

        <div>
          <h3>People</h3>
          <ul>
            { this.state.people.map(({ name, email }, i) =>
              <li key={i}>{name} ({ email })</li>
            ) }
          </ul>
        </div>
      </div>
    );
  }
```

The only differences here are two new `span` elements, one for each field. Each `span` will look in the appropriate place in `state.fieldErrors` for an error message. If one is found it will be displayed in red next to the field. Next up, we'll see how those error messages can get into the `state`.

It is after the user submits the form that we will check the validity of their input. So the appropriate place to begin validation is in the `onFormSubmit()` method. However, we'll want to create a standalone function for that method to call. We've created the pure function, `validate()` method for this:

forms/src/07-basic-validation.js

```
  validate = (person) => {
    const errors = {};
    if (!person.name) errors.name = 'Name Required';
    if (!person.email) errors.email = 'Email Required';
    if (person.email && !isEmail(person.email)) errors.email = 'Invalid Email';
    return errors;
  };
```

Our `validate()` method is pretty simple and has two goals. First, we want to make sure that both name and email are present. By checking their truthiness we can know that they are defined and not empty strings. Second, we want to know that the provided email address looks valid. This is actually a bit of a thorny issue, so we rely on `validator`[58] to let us know. If any of these conditions are not met, we add a corresponding key to our errors object and set an error message as the value.

Afterwards, we've updated our `onFormSubmit()` to use this new `validate()` method and act on the returned error object:

forms/src/07-basic-validation.js

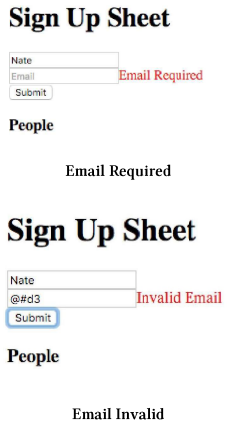
```
  onFormSubmit = (evt) => {
    const people = [ ...this.state.people ];
    const person = this.state.fields;
    const fieldErrors = this.validate(person);
    this.setState({ fieldErrors });
    evt.preventDefault();

    if (Object.keys(fieldErrors).length) return;

    this.setState({
      people: people.concat(person),
      fields: {
        name: '',
        email: '',
      },
    });
  };
```

To use the `validate()` method, we get the current values of our fields from `this.state.fields` and provide it as the argument. `validate()` will either return an empty object if there are no issues, or

[58] http://npm.im/validator

if there are issues, it will return an object with keys corresponding to each field name and values corresponding to each error message. In either case, we want to update our `state.fieldErrors` object so that `render()` can display or hide the messages as necessary.

If the validation errors object has any keys (`Object.keys(fieldErrors).length > 0`) we know there are issues. If there are no validation issues, the logic is the same as in previous sections – we add the new information and clear the fields. However, if there are any errors, we return early. This prevents the new information from being added to the list.



At this point we've covered the fundamentals of creating a form with validation in React. In the next section we'll take things a bit further and show how we can validate in real-time at the field level and we'll create a `Field` component to improve the maintainability when an app has multiple fields with different validation requirements.

Creating the Field Component

In the last section we added validation to our form. However, our form component is responsible for running the validations on the form as a whole as well as **the individual validation rules for each field**.

It would be ideal if *each field* was responsible for identifying validation errors on *its own input*, and the parent form was only responsible for identifying issues at the form-level. This comes with several advantages:

1. an email field created in this way could check the format of its input while the user types in real-time.

2. the field could incorporate its validation error message, freeing the parent form from having to keep track of it.

To do this we're first going to create a new separate `Field` component, and we will use it instead of `input` elements in the form. This will let us combine a normal `input` with both validation logic and error messaging.

Before we get into the creation of this new component, it will be useful to think of it at a high level in terms of inputs and outputs. In other words, "what information do we need to provide this component?", and "what kinds of things would we expect in return?"

These inputs are going to become this component's `props` and the output will be used by any event handlers we pass into it.

Because our `Field` component will contain a child `input`, we'll need to provide the same baseline information so that it can be passed on. For example, if we want a `Field` component rendered with a specific `placeholder prop` on its child `input`, we'll have to provide it as a `prop` when we create the `Field` component in our form's `render()` method.

Two other `props` we'll want to provide are `name`, and `value`. `name` will allow us to share an event handler between components like we've done before, and `value` allows the parent form to pre-populate `Field` as well as keep it updated.

Additionally, this new `Field` component is responsible for its own validation. Therefore we'll need to provide it rules specific to data it contains. For example, if this is the "email" `Field`, we'll provide it a function as its `validate prop`. Internally it will run this function to determine if its input is a valid email address.

Lastly, we'll provide an event handler for `onChange` events. The function we provide as the `onChange prop` will be called every time the input in the `Field` changes, and it will be called with an event argument that we get to define. This event argument should have three properties that we're interested in: (1) the name of the `Field`, (2) the current `value` of the `input`, and (3) the current validation error (if present).

To quickly review, for the new `Field` component to do its job it will need the following:

- `placeholder`: This will be passed straight through to the `input` child element. Similar to a label, this tells the user what data to the `Field` expects.
- `name`: We want this for the same reason we provide `name` to `input` elements: we'll use this in the event handler decide where to store input data and validation errors.
- `value`: This is how our parent form can initialize the `Field` with a value, or it can use this to update the `Field` with a new value. This is similar to how the `value prop` is used on an `input`.
- `validate`: A function that returns validation errors (if any) when run.
- `onChange`: An event handler to be run when the `Field` changes. This function will accept an event object as an argument.

Following this, we're able to set up `propTypes` on our new `Field` component:

forms/src/08-field-component-field.js

```
  static propTypes = {
    placeholder: PropTypes.string,
    name: PropTypes.string.isRequired,
    value: PropTypes.string,
    validate: PropTypes.func,
    onChange: PropTypes.func.isRequired,
  };
```

Next, we can think about the `state` that `Field` will need to keep track of. There are only two pieces of data that `Field` will need, the current `value` and `error`. Like in previous sections where our form component needed that data for its `render()` method, so too does our `Field` component. Here's how we'll set up our initial `state`:

forms/src/08-field-component-field.js

```
  state = {
    value: this.props.value,
    error: false,
  };
```

One key difference is that our `Field` has a parent, and sometimes this parent will want to update the `value` prop of our `Field`. To allow this, we'll need to create a new lifecycle method, `componentWillReceiveProps()` to accept the new `value` and update the `state`. Here's what that looks like:

forms/src/08-field-component-field.js

```
  componentWillReceiveProps(update) {
    this.setState({ value: update.value });
  }
```

The `render()` method of `Field` should be pretty simple. It's just the `input` and the corresponding `span` that will hold the error message:

forms/src/08-field-component-field.js

```
  render() {
    return (
      <div>
        <input
          placeholder={this.props.placeholder}
          value={this.state.value}
          onChange={this.onChange}
        />
        <span style={{ color: 'red' }}>{ this.state.error }</span>
      </div>
    );
  }
```

For the `input` element, the `placeholder` will be passed in from the parent and is available from `this.props.placeholder`. As mentioned above, the `value` of the `input` and the error message in the `span` will both be stored in the `state`. `value` comes from `this.state.value` and the error message is at `this.state.error`. And lastly, we'll set an `onChange` event handler that will be responsible for accepting user input, validating, updating state, *and calling the parent's event handler as well.* The method that will take care of that is `this.onChange`:

```
  onChange (evt) {
    const name = this.props.name;
    const value = evt.target.value;
    const error = this.props.validate ? this.props.validate(value) : false;

    this.setState({value, error});

    this.props.onChange({name, value, error});
  }
```

`this.onChange` is a pretty efficient function. It handles four different responsibilities in as many lines. As in previous sections, the event object gives us the current text in the `input` via its `target.value` property. Once we have that, we see if it passes validation. If `Field` was given a function for its `validate` prop, we use it here. If one was not given, we don't have to validate the input and `error` sets to `false`. Once we have both the `value` and `error` we can update our `state` so that they both appear in `render()`. However, it's not just the `Field` component that needs to be updated with this information.

When `Field` is used by a parent component, it passes in its own event handler in as the `onChange` prop. We call this function so that we can pass information up the parent. Here in `this.onChange()`,

it is available as `this.props.onChange()`, and we call it with three pieces of information: the `name`, `value`, and `error` of the `Field`.

This might be a little confusing since "onChange" appears in multiple places. You can think of it as carrying information in a chain of event handlers. The form contains the `Field` which contains an `input`. Events occur on the `input` and the information passes first to the `Field` and finally to the form.

At this point our `Field` component is ready to go, and can be used in place of the `input` and error message combos in our app.

Using our new Field Component

Now that we're ready to use our brand new `Field` component, we can make some changes to our app. The most obvious change is that `Field` will take the place of both the `input` and error message `span` elements in our `render()` method. This is great because `Field` can take care of validation at the field level. But what about at the form level?

If you remember, we can employ two different levels of validation, one at the field level, and one at the form level. Our new `Field` component will let us validate the format of each field in real-time. What they won't do, however, is validate the entire form to make sure we have all the data we need. For that, we also want form-level validation.

Another nice feature we'll add here is disabling/enabling the form submit button in real-time as the form passes/fails validation. This is a nice bit of feedback that can improve a form's UX and make it feel more responsive.

Here's how our update `render()` looks:

forms/src/08-field-component-form.js

```
60  render() {
61    return (
62      <div>
63        <h1>Sign Up Sheet</h1>
64
65        <form onSubmit={this.onFormSubmit}>
66
67          <Field
68            placeholder='Name'
69            name='name'
70            value={this.state.fields.name}
71            onChange={this.onInputChange}
72            validate={(val) => (val ? false : 'Name Required')}
73          />
74
```



```
75          <br />
76
77          <Field
78            placeholder='Email'
79            name='email'
80            value={this.state.fields.email}
81            onChange={this.onInputChange}
82            validate={(val) => (isEmail(val) ? false : 'Invalid Email')}
83          />
84
85          <br />
86
87          <input type='submit' disabled={this.validate()} />
88        </form>
89
90        <div>
91          <h3>People</h3>
92          <ul>
93            { this.state.people.map(({ name, email }, i) =>
94              <li key={i}>{name} ({email})</li>
95            ) }
96          </ul>
97        </div>
98      </div>
99    );
100 }
```

You can see that `Field` is a drop-in replacement for `input`. All the `props` are the same as they were on `input`, except we have one additional `prop` this time: `validate`.

Above in the `Field` component's `onChange()` method, we make a call to the `this.props.validate()` function. What we provide as the `validate prop` to `Field`, will be that function. Its goal is to take user provided input as its argument and give a return value that corresponds to the validity of that input. If the input is not valid, `validate` should return an error message. Otherwise, it should return `false`.

For the "name" `Field` the `validate prop` is pretty simple. We're just checking for a truthy value. As long as there are characters in the box, validation will pass, otherwise we return the `'Name Required'` error message.

For the "email" `Field`, we're going to use the `isEmail()` function that we imported from the `validator` module. If that function returns `true`, we know it's a valid-looking email and validation passes. If not, we return the `'Invalid Email'` message.

Notice that we left their `onChange` prop alone, it is still set to `this.onInputChange`. However, since `Field` uses the function differently than `input`, we must update `onInputChange()`.

Before we move on, notice the only other change that we've made to `render()`: we conditionally disable the submit button. To do this, we set the value of the `disabled` prop to the return value of `this.validate()`. Because `this.validate()` will have a truthy return value if there are validation errors, the button will be disabled if the form is not valid. We'll show what the `this.validate()` function looks like in a bit.

Disabled Submit Button

As mentioned, both `Field` components have their `onChange` props set to `this.onInputChange`. We've had to make some changes to match the difference between `input` and our `Field`. Here's the updated version:

forms/src/08-field-component-form.js

```
38  onInputChange = ({ name, value, error }) => {
39    const fields = this.state.fields;
40    const fieldErrors = this.state.fieldErrors;
41
42    fields[name] = value;
43    fieldErrors[name] = error;
44
45    this.setState({ fields, fieldErrors });
46  };
```

Previously, the job of `onInputChange()` was to update `this.state.fields` with the current user input values. In other words, when an a text field was edited, `onInputChange()` would be called with an event object. That event object had a `target` property that referenced the `input` element. Using that reference, we could get the `name` and `value` of the `input`, and with those, we would update `state.fields`.

This time around `onInputChange()` has the same responsibility, but it is our `Field` component that calls this function, not `input`. In the previous section, we show the `onChange()` method of

`Field`, and that's where `this.props.onChange()` is called. When it is called, it's called like this: ‘ this.props.onChange({name, value, error})‘.

This means that instead of using `evt.target.name` or `evt.target.value` as we did before, we get `name` and `value` directly from the argument object. In addition, we also get the validation error for each field. This is necessary – for our form component to prevent submission, it will need to know about field-level validation errors.

Once we have the `name`, `value`, and `error`, we can update two objects in our `state`, the `state.fields` object we used before, and a new object, `state.fieldErrors`. Soon, we will show how `state.fieldErrors` will be used to either prevent or allow the form submit depending on the presence or absence of field-level validation errors.

With both `render()` and `onInputChange()` updated, we again have a nice feedback loop set up for our `Field` components:

- First, the user types into the `Field`.
- Then, the event handler of the `Field` is called, `onInputChange()`.
- Next, `onInputChange()` updates the `state`.
- After, the form is rendered again, and the `Field` passed an updated `value` prop.
- Then, `componentWillReceiveProps()` is called in `Field` with the new `value`, and its state is updated.
- Finally, `Field.render()` is called again, and the text field shows the appropriate input and (if applicable) validation error.

At this point, our form's `state` and appearance are in sync. Next, we need to change how we handle the submit event. Here's the updated event handler for the form, `onFormSubmit()`:

forms/src/08-field-component-form.js

```
21  onFormSubmit = (evt) => {
22    const people = this.state.people;
23    const person = this.state.fields;
24
25    evt.preventDefault();
26
27    if (this.validate()) return;
28
29    this.setState({
30      people: people.concat(person),
31      fields: {
32        name: '',
33        email: '',
34      },
```