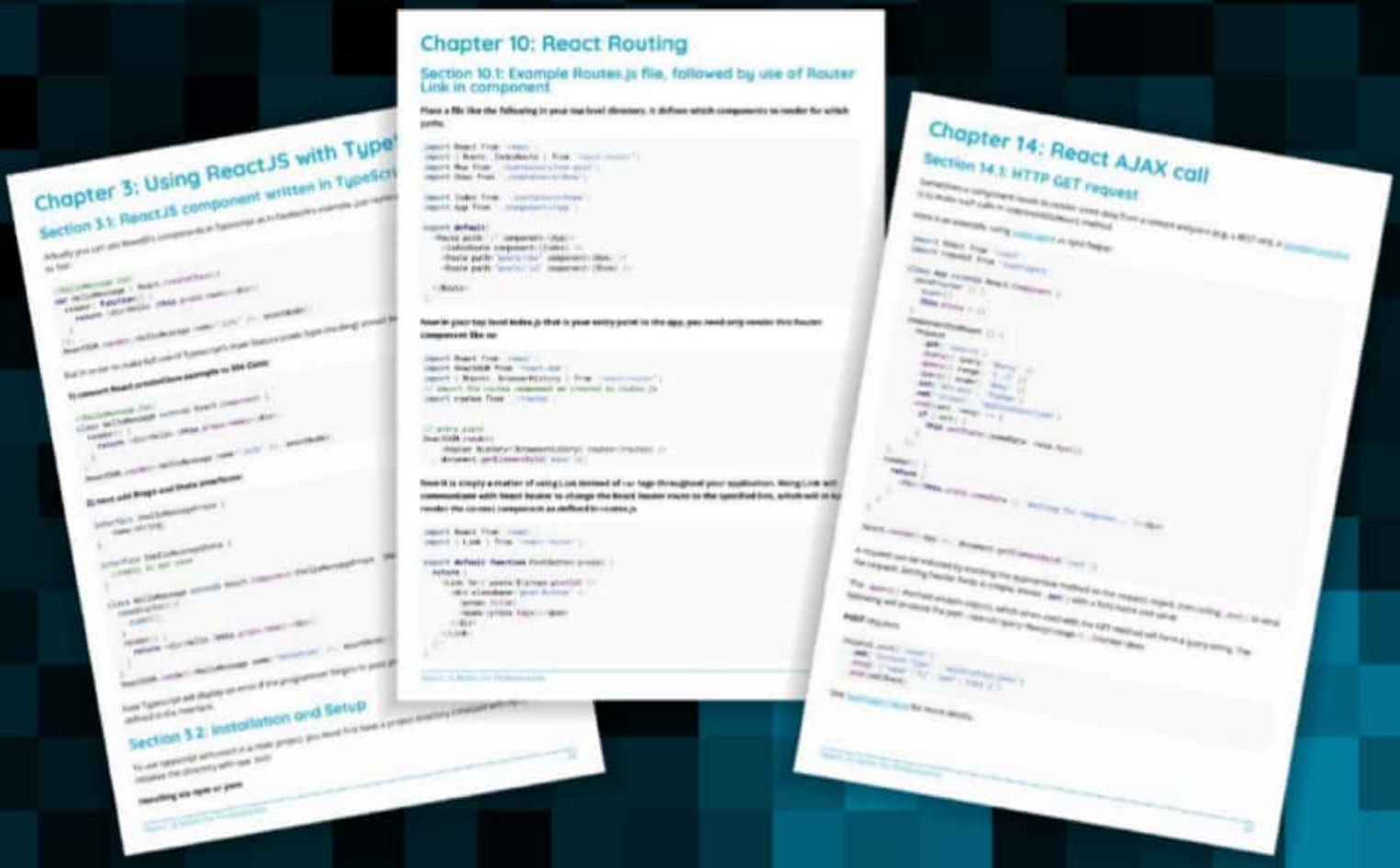
React JS

Notes for Professionals



Contents

React allows us to write components using a domain-specific language called JSX. JSX allows us to write our components using HTML, whilst mixing in JavaScript events. React will internally convert this into a virtual DOM, and will ultimately output our HTML for us.

React "*reacts*" to state changes in your components quickly and automatically to rerender the components in the HTML DOM by utilizing the virtual DOM. The virtual DOM is an in-memory representation of an actual DOM. By doing most of the processing inside the virtual DOM rather than directly in the browser's DOM, React can act quickly and only add, update, and remove components which have changed since the last render cycle occurred.

Section 1.2: Installation or Setup

ReactJS is a JavaScript library contained in a single file `react-<version>.js` that can be included in any HTML page. People also commonly install the React DOM library `react-dom-<version>.js` along with the main React file:

Basic Inclusion

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
    <script type="text/javascript" src="/path/to/react.js"></script>
    <script type="text/javascript" src="/path/to/react-dom.js"></script>
    <script type="text/javascript">
        // Use react JavaScript code here or in a separate file
    </script>
    </body>
</html>
```

To get the JavaScript files, go to the installation page of the official React documentation.

React also supports JSX syntax. JSX is an extension created by Facebook that adds XML syntax to JavaScript. In order to use JSX you need to include the Babel library and change `<script type="text/javascript">` to `<script type="text/babel">` in order to translate JSX to Javascript code.

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
    <script type="text/javascript" src="/path/to/react.js"></script>
    <script type="text/javascript" src="/path/to/react-dom.js"></script>
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>
    <script type="text/babel">
        // Use react JSX code here or in a separate file
    </script>
    </body>
</html>
```

Installing via npm

You can also install React using npm by doing the following:

```
npm install --save react react-dom
```

To use React in your JavaScript project, you can do the following:

```
var React = require('react');
var ReactDOM = require('react-dom');
```

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

Installing via Yarn

Facebook released its own package manager named Yarn, which can also be used to install React. After installing Yarn you just need to run this command:

yarn add react react-dom

You can then use React in your project in exactly the same way as if you had installed React via npm.

Section 1.3: Hello World with Stateless Functions

Stateless components are getting their philosophy from functional programming. Which implies that: A function returns all time the same thing exactly on what is given to it.

For example:

```
const statelessSum = (a, b) => a + b;

let a = 0;
const statefulSum = () => a++;
```

As you can see from the above example that, statelessSum is always will return the same values given a and b. However, statefulSum function will not return the same values given even no parameters. This type of function's behaviour is also called as a *side-effect*. Since, the component affects somethings beyond.

So, it is advised to use stateless components more often, since they are *side-effect free* and will create the same behaviour always. That is what you want to be after in your apps because fluctuating state is the worst case scenario for a maintainable program.

The most basic type of react component is one without state. React components that are pure functions of their props and do not require any internal state management can be written as simple JavaScript functions. These are said to be `Stateless Functional Components` because they are a function only of `props`, without having any `state` to keep track of.

Here is a simple example to illustrate the concept of a `Stateless Functional Component`:

```
// In HTML
<div id="element"></div>

// In React
const MyComponent = props => {
    return <h1>Hello, {props.name}!</h1>;
};

ReactDOM.render(<MyComponent name="Arun" />, element);
// Will render <h1>Hello, Arun!</h1>
```

Note that all that this component does is render an `h1` element containing the `name` prop. This component doesn't keep track of any state. Here's an ES6 example as well:

```
import React from 'react'

const HelloWorld = props => (
    <h1>Hello, {props.name}!</h1>
)
```

```
HelloWorld.propTypes = {
    name: React.PropTypes.string.isRequired
}

export default HelloWorld
```

Since these components do not require a backing instance to manage the state, React has more room for optimizations. The implementation is clean, but as of yet no such optimizations for stateless components have been implemented.

Section 1.4: Absolute Basics of Creating Reusable Components

Components and Props

As React concerns itself only with an application's view, the bulk of development in React will be the creation of components. A component represents a portion of the view of your application. "Props" are simply the attributes used on a JSX node (e.g. `<SomeComponent someProp="some prop's value" />`), and are the primary way our application interacts with our components. In the snippet above, inside of SomeComponent, we would have access to `this.props`, whose value would be the object `{someProp: "some prop's value"}`.

It can be useful to think of React components as simple functions - they take input in the form of "props", and produce output as markup. Many simple components take this a step further, making themselves "Pure Functions", meaning they do not issue side effects, and are idempotent (given a set of inputs, the component will always produce the same output). This goal can be formally enforced by actually creating components as functions, rather than "classes". There are three ways of creating a React component:

- **Functional ("Stateless") Components**

```
const FirstComponent = props => (
    <div>{props.content}</div>
);
```

- **React.createClass()**

```
const SecondComponent = React.createClass({
    render: function () {
        return (
            <div>{this.props.content}</div>
        );
    }
});
```

- **ES2015 Classes**

```
class ThirdComponent extends React.Component {
    render() {
        return (
            <div>{this.props.content}</div>
        );
    }
}
```

These components are used in exactly the same way:

```
const ParentComponent = function (props) {
    const someText = "FooBar";
```

```
    return (
        <FirstComponent content={someText} />
        <SecondComponent content={someText} />
        <ThirdComponent content={someText} />
    );
}
```

The above examples will all produce identical markup.

Functional components cannot have "state" within them. So if your component needs to have a state, then go for class based components. Refer Creating Components for more information.

As a final note, react props are immutable once they have been passed in, meaning they cannot be modified from within a component. If the parent of a component changes the value of a prop, React handles replacing the old props with the new, the component will rerender itself using the new values.

See Thinking In React and Reusable Components for deeper dives into the relationship of props to components.

Section 1.5: Create React App

create-react-app is a React app boilerplate generator created by Facebook. It provides a development environment configured for ease-of-use with minimal setup, including:

- ES6 and JSX transpilation
- Dev server with hot module reloading
- Code linting
- CSS auto-prefixing
- Build script with JS, CSS and image bundling, and sourcemaps
- Jest testing framework

Installation

First, install create-react-app globally with node package manager (npm).

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

Then run the generator in your chosen directory.

```
create-react-app my-app
```

Navigate to the newly created directory and run the start script.

```
cd my-app/
npm start
```

Configuration

create-react-app is intentionally non-configurable by default. If non-default usage is required, for example, to use a compiled CSS language such as Sass, then the eject command can be used.

```
npm run eject
```

This allows editing of all configuration files. N.B. this is an irreversible process.

Alternatives

Alternative React boilerplates include:

- enclave
- nwb
- motion
- rackt-cli
- budō
- rwb
- quik
- sagui
- roc

Build React App

To build your app for production ready, run following command

```
npm run build
```

Section 1.6: Hello World

Without JSX

Here's a basic example that uses React's main API to create a React element and the React DOM API to render the React element in the browser.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>

    <!-- Include the React and ReactDOM libraries -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>

  </head>
  <body>
    <div id="example"></div>

    <script type="text/javascript">

      // create a React element rElement
      var rElement = React.createElement('h1', null, 'Hello, world!');

      // dElement is a DOM container
      var dElement = document.getElementById('example');

      // render the React element in the DOM container
      ReactDOM.render(rElement, dElement);

    </script>

  </body>
</html>
```

With JSX

Instead of creating a React element from strings one can use JSX (a Javascript extension created by Facebook for adding XML syntax to JavaScript), which allows to write

```
var rElement = React.createElement('h1', null, 'Hello, world!');
```

as the equivalent (and easier to read for someone familiar with HTML)

```
var rElement = <h1>Hello, world!</h1>;
```

The code containing JSX needs to be enclosed in a **<script type="text/babel">** tag. Everything within this tag will be transformed to plain Javascript using the Babel library (that needs to be included in addition to the React libraries).

So finally the above example becomes:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>

    <!-- Include the React and ReactDOM libraries -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>
    <!-- Include the Babel library -->
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>

  </head>
  <body>
   <div id="example"></div>

    <script type="text/babel">

      // create a React element rElement using JSX
      var rElement = <h1>Hello, world!</h1>;

      // dElement is a DOM container
      var dElement = document.getElementById('example');

      // render the React element in the DOM container
      ReactDOM.render(rElement, dElement);

    </script>

  </body>
</html>
```

Section 1.7: Hello World Component

A React component can be defined as an ES6 class that extends the base `React.Component` class. In its minimal form, a component *must* define a `render` method that specifies how the component renders to the DOM. The `render` method returns React nodes, which can be defined using JSX syntax as HTML-like tags. The following example shows how to define a minimal Component:

```
import React from 'react'

class HelloWorld extends React.Component {
    render() {
```

```
        return <h1>Hello, World!</h1>
    }
}

export default HelloWorld
```

A Component can also receive `props`. These are properties passed by its parent in order to specify some values the component cannot know by itself; a property can also contain a function that can be called by the component after certain events occur - for example, a button could receive a function for its `onClick` property and call it whenever it is clicked. When writing a component, its `props` can be accessed through the `props` object on the Component itself:

```
import React from 'react'

class Hello extends React.Component {
    render() {
        return <h1>Hello, {this.props.name}!</h1>
    }
}

export default Hello
```

The example above shows how the component can render an arbitrary string passed into the `name` prop by its parent. Note that a component cannot modify the props it receives.

A component can be rendered within any other component, or directly into the DOM if it's the topmost component, using `ReactDOM.render` and providing it with both the component and the DOM Node where you want the React tree to be rendered:

```
import React from 'react'
import ReactDOM from 'react-dom'
import Hello from './Hello'

ReactDOM.render(<Hello name="Billy James" />, document.getElementById('main'))
```

By now you know how to make a basic component and accept `props`. Lets take this a step further and introduce `state`.

For demo sake, let's make our Hello World app, display only the first name if a full name is given.

```
import React from 'react'

class Hello extends React.Component {

    constructor(props){

        //Since we are extending the default constructor,
        //handle default activities first.
        super(props);

        //Extract the first-name from the prop
        let firstName = this.props.name.split(" ")[0];

        //In the constructor, feel free to modify the
        //state property on the current context.
        this.state = {
            name: firstName
        }
```

```
    } //Look maa, no comma required in JSX based class defs!

    render() {
        return <h1>Hello, {this.state.name}!</h1>
    }
}

export default Hello
```

Note: Each component can have it's own state or accept it's parent's state as a prop.

Codepen Link to Example.

Chapter 2: Components

Section 2.1: Creating Components

This is an extension of Basic Example:

Basic Structure

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class FirstComponent extends Component {
    render() {
        return (
            <div>
                Hello, {this.props.name}! I am a FirstComponent.
            </div>
        );
    }
}

render(
    <FirstComponent name={ 'User' } />,
    document.getElementById('content')
);
```

The above example is called a **stateless** component as it does not contain state (in the React sense of the word).

In such a case, some people find it preferable to use Stateless Functional Components, which are based on ES6 arrow functions.

Stateless Functional Components

In many applications there are smart components that hold state but render dumb components that simply receive props and return HTML as JSX. Stateless functional components are much more reusable and have a positive performance impact on your application.

They have 2 main characteristics:

1. When rendered they receive an object with all the props that were passed down
2. They must return the JSX to be rendered

```
// When using JSX inside a module you must import React
import React from 'react';
import PropTypes from 'prop-types';

const FirstComponent = props => (
    <div>
        Hello, {props.name}! I am a FirstComponent.
    </div>
);

//arrow components also may have props validation
FirstComponent.propTypes = {
    name: PropTypes.string.isRequired,
}

// To use FirstComponent in another file it must be exposed through an export call:
```

```
export default FirstComponent;
```

Stateful Components

In contrast to the 'stateless' components shown above, 'stateful' components have a state object that can be updated with the `setState` method. The state must be initialized in the `constructor` before it can be set:

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

class SecondComponent extends Component {
    constructor(props) {
        super(props);

        this.state = {
            toggle: true
        };

        // This is to bind context when passing onClick as a callback
        this.onClick = this.onClick.bind(this);
    }

    onClick() {
        this.setState((prevState, props) => ({
            toggle: !prevState.toggle
        }));
    }

    render() {
        return (
            <div onClick={this.onClick}>
                Hello, {this.props.name}! I am a SecondComponent.
                <br />
                Toggle is: {this.state.toggle}
            </div>
        );
    }
}
```

Extending a component with `PureComponent` instead of `Component` will automatically implement the `shouldComponentUpdate()` lifecycle method with shallow prop and state comparison. This keeps your application more performant by reducing the amount of un-necessary renders that occur. This assumes your components are 'Pure' and always render the same output with the same state and props input.

Higher Order Components

Higher order components (HOC) allow to share component functionality.

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

const PrintHello = ComposedComponent => class extends Component {
    onClick() {
        console.log('hello');
    }

    /* The higher order component takes another component as a parameter
    and then renders it with additional props */
    render() {
        return <ComposedComponent {...this.props } onClick={this.onClick} />
    }
}
```

```
const FirstComponent = props => (
    <div onClick={ props.onClick }>
        Hello, {props.name}! I am a FirstComponent.
    </div>
);

const ExtendedComponent = PrintHello(FirstComponent);
```

Higher order components are used when you want to share logic across several components regardless of how different they render.

Section 2.2: Basic Component

Given the following HTML file:

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Tutorial</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.34/browser.min.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/babel" src="scripts/example.js"></script>
  </body>
</html>
```

You can create a basic component using the following code in a separate file:

scripts/example.js

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';

class FirstComponent extends Component {
  render() {
    return (
      <div className="firstComponent">
        Hello, world! I am a FirstComponent.
      </div>
    );
  }
}
ReactDOM.render(
  <FirstComponent />, // Note that this is the same as the variable you stored above
  document.getElementById('content')
);
```

You will get the following result (note what is inside of `div#content`):

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
```

```
    <title>React Tutorial</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.34/browser.min.js"></script>
  </head>
  <body>
    <div id="content">
        <div className="firstComponent">
            Hello, world! I am a FirstComponent.
        </div>
    </div>
    <script type="text/babel" src="scripts/example.js"></script>
  </body>
</html>
```

Section 2.3: Nesting Components

A lot of the power of ReactJS is its ability to allow nesting of components. Take the following two components:

```
var React = require('react');
var createReactClass = require('create-react-class');

var CommentList = reactCreateClass({
  render: function() {
    return (
      <div className="commentList">
        Hello, world! I am a CommentList.
      </div>
    );
  }
});

var CommentForm = reactCreateClass({
  render: function() {
    return (
      <div className="commentForm">
        Hello, world! I am a CommentForm.
      </div>
    );
  }
});
```

You can nest and refer to those components in the definition of a different component:

```
var React = require('react');
var createReactClass = require('create-react-class');

var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList /> // Which was defined above and can be reused
        <CommentForm /> // Same here
      </div>
    );
  }
});
```

Further nesting can be done in three ways, which all have their own places to be used.

1. Nesting without using children

(continued from above)

```
var CommentList = reactCreateClass({
  render: function() {
    return (
      <div className="commentList">
        <ListTitle/>
        Hello, world! I am a CommentList.
      </div>
    );
  }
});
```

This is the style where A composes B and B composes C.

Pros

- Easy and fast to separate UI elements
- Easy to inject props down to children based on the parent component's state

Cons

- Less visibility into the composition architecture
- Less reusability

Good if

- B and C are just presentational components
- B should be responsible for C's lifecycle

2. Nesting using children

(continued from above)

```
var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList>
            <ListTitle/> // child
        </CommentList>
        <CommentForm />
      </div>
    );
  }
});
```

This is the style where A composes B and A tells B to compose C. More power to parent components.

Pros

- Better components lifecycle management
- Better visibility into the composition architecture
- Better reusuability

Cons

- Injecting props can become a little expensive
- Less flexibility and power in child components

Good if

- B should accept to compose something different than C in the future or somewhere else
- A should control the lifecycle of C

B would render C using `this.props.children`, and there isn't a structured way for B to know what those children are for. So, B may enrich the child components by giving additional props down, but if B needs to know exactly what they are, #3 might be a better option.

3. Nesting using props

(continued from above)

```
var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList title={ListTitle}/> //prop
        <CommentForm />
      </div>
    );
  }
});
```

This is the style where A composes B and B provides an option for A to pass something to compose for a specific purpose. More structured composition.

Pros

- Composition as a feature
- Easy validation
- Better composaiblility

Cons

- Injecting props can become a little expensive
- Less flexibility and power in child components

Good if

- B has specific features defined to compose something
- B should only know how to render not what to render

#3 is usually a must for making a public library of components but also a good practice in general to make composable components and clearly define the composition features. #1 is the easiest and fastest to make something that works, but #2 and #3 should provide certain benefits in various use cases.

Section 2.4: Props

Props are a way to pass information into a React component, they can have any type including functions - sometimes referred to as callbacks.

In JSX props are passed with the attribute syntax

```
<MyComponent userID={123} />
```

Inside the definition for MyComponent userID will now be accessible from the props object

```
// The render function inside MyComponent
render() {
    return (
        <span>The user's ID is {this.props.userID}</span>
    )
}
```

It's important to define all `props`, their types, and where applicable, their default value:

```
// defined at the bottom of MyComponent
MyComponent.propTypes = {
    someObject: React.PropTypes.object,
    userID: React.PropTypes.number.isRequired,
    title: React.PropTypes.string
};

MyComponent.defaultProps = {
    someObject: {},
    title: 'My Default Title'
}
```

In this example the prop `someObject` is optional, but the prop `userID` is required. If you fail to provide `userID` to `MyComponent`, at runtime the React engine will show a console warning you that the required prop was not provided. Beware though, this warning is only shown in the development version of the React library, the production version will not log any warnings.

Using `defaultProps` allows you to simplify

```
const { title = 'My Default Title' } = this.props;
console.log(title);
```

to

```
console.log(this.props.title);
```

It's also a safeguard for use of `object` `array` and `functions`. If you do not provide a default prop for an object, the following will throw an error if the prop is not passed:

```
if (this.props.someObject.someKey)
```

In example above, `this.props.someObject` is **`undefined`** and therefore the check of `someKey` will throw an error and the code will break. With the use of `defaultProps` you can safely use the above check.

Section 2.5: Component states - Dynamic user-interface

Suppose we want to have the following behaviour - We have a heading (say h3 element) and on clicking it, we want it to become an input box so that we can modify heading name. React makes this highly simple and intuitive using component states and if else statements. (Code explanation below)

```
// I have used ReactBootstrap elements. But the code works with regular html elements also
var Button = ReactBootstrap.Button;
var Form = ReactBootstrap.Form;
```

```
var FormGroup = ReactBootstrap.FormGroup;
var FormControl = ReactBootstrap.FormControl;

var Comment = reactCreateClass({
  getInitialState: function() {
    return {show: false, newTitle: ''};
  },

  handleTitleSubmit: function() {
     //code to handle input box submit - for example, issue an ajax request to change name in
database
  },

  handleTitleChange: function(e) {
    //code to change the name in form input box. newTitle is initialized as empty string. We need to
update it with the string currently entered by user in the form
    this.setState({newTitle: e.target.value});
  },

  changeComponent: function() {
    // this toggles the show variable which is used  for dynamic UI
    this.setState({show: !this.state.show)};
  },

  render: function() {

    var clickableTitle;

    if(this.state.show) {
        clickableTitle = <Form inline onSubmit={this.handleTitleSubmit}>
                            <FormGroup controlId="formInlineTitle">
                                <FormControl type="text" onChange={this.handleTitleChange}>
                            </FormGroup>
                        </Form>;
    } else {
        clickabletitle = <div>
                           <Button bsStyle="link" onClick={this.changeComponent}>
                               <h3> Default Text </h3>
                           </Button>
                        </div>;
    }

    return (
        <div className="comment">
            {clickableTitle}
        </div>
    );
  }
});

ReactDOM.render(
    <Comment />, document.getElementById('content')
);
```

The main part of the code is the **clickableTitle** variable. Based on the state variable **show**, it can be either be a Form element or a Button element. React allows nesting of components.

So we can add a {clickableTitle} element in the render function. It looks for the clickableTitle variable. Based on the value 'this.state.show', it displays the corresponding element.

Section 2.6: Variations of Stateless Functional Components

```
const languages = [
  'JavaScript',
  'Python',
  'Java',
  'Elm',
  'TypeScript',
  'C#',
  'F#'
]
```

```
// one liner
const Language = ({language}) => <li>{language}</li>

Language.propTypes = {
  message: React.PropTypes.string.isRequired
}
```

```
/**
* If there are more than one line.
* Please notice that round brackets are optional here,
* However it's better to use them for readability
*/
const LanguagesList = ({languages}) => {
  <ul>
    {languages.map(language => <Language language={language} />)}
  </ul>
}

LanguagesList.PropTypes = {
  languages: React.PropTypes.array.isRequired
}
```

```
/**
 * This syntax is used if there are more work beside just JSX presentation
 * For instance some data manipulations needs to be done.
 * Please notice that round brackets after return are required,
 * Otherwise return will return nothing (undefined)
 */
const LanguageSection = ({header, languages}) => {
  // do some work
  const formattedLanguages = languages.map(language => language.toUpperCase())
  return (
    <fieldset>
      <legend>{header}</legend>
      <LanguagesList languages={formattedLanguages} />
    </fieldset>
  )
}

LanguageSection.PropTypes = {
  header: React.PropTypes.string.isRequired,
  languages: React.PropTypes.array.isRequired
}
```

```
ReactDOM.render(
  <LanguageSection
```

```
    header="Languages"
    languages={languages} />,
  document.getElementById('app')
)
```

Here you can find working example of it.

Section 2.7: setState pitfalls

You should use caution when using `setState` in an asynchronous context. For example, you might try to call `setState` in the callback of a get request:

```
class MyClass extends React.Component {
    constructor() {
        super();

        this.state = {
            user: {}
        };
    }

    componentDidMount() {
        this.fetchUser();
    }

    fetchUser() {
        $.get('/api/users/self')
            .then((user) => {
                this.setState({user: user});
            });
    }

    render() {
        return <h1>{this.state.user}</h1>;
    }
}
```

This could call problems - if the callback is called after the `Component` is dismounted, then `this.setState` won't be a function. Whenever this is the case, you should be careful to ensure your usage of `setState` is cancellable.

In this example, you might wish to cancel the XHR request when the component dismounts:

```
class MyClass extends React.Component {
    constructor() {
        super();

        this.state = {
            user: {},
            xhr: null
        };
    }

    componentWillUnmount() {
        let xhr = this.state.xhr;

        // Cancel the xhr request, so the callback is never called
        if (xhr && xhr.readyState != 4) {
            xhr.abort();
        }
    }
```

```
    componentDidMount() {
        this.fetchUser();
    }

    fetchUser() {
        let xhr = $.get('/api/users/self')
            .then((user) => {
                this.setState({user: user});
            });

        this.setState({xhr: xhr});
    }
}
```

The async method is saved as a state. In the `componentWillUnmount` you perform all your cleanup - including canceling the XHR request.

You could also do something more complex. In this example, I'm creating a 'stateSetter' function that accepts the this object as an argument and prevents `this.setState` when the function `cancel` has been called:

```
function stateSetter(context) {
    var cancelled = false;
    return {
        cancel: function () {
            cancelled = true;
        },
        setState(newState) {
            if (!cancelled) {
                context.setState(newState);
            }
        }
    }
}

class Component extends React.Component {
    constructor(props) {
        super(props);
        this.setter = stateSetter(this);
        this.state = {
            user: 'loading'
        };
    }
    componentWillUnmount() {
        this.setter.cancel();
    }
    componentDidMount() {
        this.fetchUser();
    }
    fetchUser() {
        $.get('/api/users/self')
            .then((user) => {
                this.setter.setState({user: user});
            });
    }
    render() {
        return <h1>{this.state.user}</h1>
    }
}
```

This works because the `cancelled` variable is visible in the `setState` closure we created.

Chapter 3: Using ReactJS with TypeScript

Section 3.1: ReactJS component written in TypeScript

Actually you can use ReactJS's components in Typescript as in facebook's example. Just replace 'jsx' file's extension to 'tsx':

```
//helloMessage.tsx:
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

But in order to make full use of Typescript's main feature (static type checking) should be done couple things:

1) convert React.createClass example to ES6 Class:

```
//helloMessage.tsx:
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

2) next add Props and State interfaces:

```
interface IHelloMessageProps {
    name:string;
}

interface IHelloMessageState {
  //empty in our case
}

class HelloMessage extends React.Component<IHelloMessageProps, IHelloMessageState> {
  constructor(){
    super();
  }
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
```

Now Typescript will display an error if the programmer forgets to pass props. Or if they added props that are not defined in the interface.

Section 3.2: Installation and Setup

To use typescript with react in a node project, you must first have a project directory initialized with npm. To initialize the directory with `npm init`

Installing via npm or yarn

You can install React using npm by doing the following:

```
npm install --save react react-dom
```

Facebook released its own package manager named Yarn, which can also be used to install React. After installing Yarn you just need to run this command:

```
yarn add react react-dom
```

You can then use React in your project in exactly the same way as if you had installed React via npm.

Installing react type definitions in Typescript 2.0+

To compile your code using typescript, add/install type definition files using npm or yarn.

```
npm install --save-dev @types/react @types/react-dom
```

or, using yarn

```
yarn add --dev @types/react @types/react-dom
```

Installing react type definitions in older versions of Typescript

You have to use a separate package called tsd

```
tsd install react react-dom --save
```

Adding or Changing the Typescript configuration

To use JSX, a language mixing javascript with html/xml, you have to change the typescript compiler configuration. In the project's typescript configuration file (usually named `tsconfig.json`), you will need to add the JSX option as:

"compilerOptions": { "jsx": "react" },

That compiler option basically tells the typescript compiler to translate the JSX tags in code to javascript function calls.

To avoid typescript compiler converting JSX to plain javascript function calls, use

```
"compilerOptions": {
    "jsx": "preserve"
},
```

Section 3.3: Stateless React Components in TypeScript

React components that are pure functions of their props and do not require any internal state can be written as JavaScript functions instead of using the standard class syntax, as:

```
import React from 'react'

const HelloWorld = (props) => (
    <h1>Hello, {props.name}!</h1>
);
```

The same can be achieved in Typescript using the `React.SFC` class:

```
import * as React from 'react';

class GreeterProps {
   name: string
}

const Greeter : React.SFC<GreeterProps> = props =>
    <h1>Hello, {props.name}!</h1>;
```

Note that, the name `React.SFC` is an alias for `React.StatelessComponent` So, either can be used.

Section 3.4: Stateless and property-less Components

The simplest react component without a state and no properties can be written as:

```
import * as React from 'react';

const Greeter = () => <span>Hello, World!</span>
```

That component, however, can't access `this.props` since typescript can't tell if it is a react component. To access its props, use:

```
import * as React from 'react';

const Greeter: React.SFC<{}> = props => () => <span>Hello, World!</span>
```

Even if the component doesn't have explicitly defined properties, it can now access **`props.children`** since all components inherently have children.

Another similar good use of stateless and property-less components is in simple page templating. The following is an examplinary simple `Page` component, assuming there are hypothetical `Container`, `NavTop` and `NavBottom` components already in the project:

```
import * as React from 'react';

const Page: React.SFC<{}> = props => () =>
    <Container>
        <NavTop />
        {props.children}
        <NavBottom />
    </Container>

const LoginPage: React.SFC<{}> = props => () =>
    <Page>
        Login Pass: <input type="password" />
    </Page>
```

In this example, the `Page` component can later be used by any other actual page as a base template.

Chapter 4: State in React

Section 4.1: Basic State

State in React components is essential to manage and communicate data in your application. It is represented as a JavaScript object and has *component level* scope, it can be thought of as the private data of your component.

In the example below we are defining some initial state in the `constructor` function of our component and make use of it in the `render` function.

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

    // Set-up our initial state
    this.state = {
      greeting: 'Hiya Buddy!'
    };
  }

  render() {
    // We can access the greeting property through this.state
    return(
      <div>{this.state.greeting}</div>
    );
  }
}
```

Section 4.2: Common Antipattern

You should not save `props` into `state`. It is considered an anti-pattern. For example:

```
export default class MyComponent extends React.Component {
    constructor() {
        super();

        this.state = {
            url: ''
        }

        this.onChange = this.onChange.bind(this);
    }

    onChange(e) {
        this.setState({
            url: this.props.url + '/days=?' + e.target.value
        });
    }

    componentWillMount() {
        this.setState({url: this.props.url});
    }

    render() {
        return (
            <div>
                <input defaultValue={2} onChange={this.onChange} />
```

```
                URL: {this.state.url}
            </div>
        )
    }
}
```

The prop `url` is saved on `state` and then modified. Instead, choose to save the changes to a state, and then build the full path using both `state` and `props`:

```
export default class MyComponent extends React.Component {
    constructor() {
        super();

        this.state = {
            days: ''
        }

        this.onChange = this.onChange.bind(this);
    }

    onChange(e) {
        this.setState({
            days: e.target.value
        });
    }

    render() {
        return (
            <div>
                <input defaultValue={2} onChange={this.onChange} />

                URL: {this.props.url + '/days?=' + this.state.days}
            </div>
        )
    }
}
```

This is because in a React application we want to have a single source of truth - i.e. all data is the responsibility of one single component, and only one component. It is the responsibility of this component to store the data within its state, and distribute the data to other components via props.

In the first example, both the MyComponent class and its parent are maintaining 'url' within their state. If we update state.url in MyComponent, these changes are not reflected in the parent. We have lost our single source of truth, and it becomes increasingly difficult to track the flow of data through our application. Contrast this with the second example - url is only maintained in the state of the parent component, and utilised as a prop in MyComponent - we therefore maintain a single source of truth.

Section 4.3: setState()

The primary way that you make UI updates to your React applications is through a call to the `setState()` function. This function will perform a *shallow merge* between the new state that you provide and the previous state, and will trigger a re-render of your component and all decedents.

Parameters

1. `updater`: It can be an object with a number of key-value pairs that should be merged into the state or a function that returns such an object.

2. `callback (optional)`: a function which will be executed after `setState()` has been executed successfully. Due to the fact that calls to `setState()` are not guaranteed by React to be atomic, this can sometimes be useful if you want to perform some action after you are positive that `setState()` has been executed successfully.

Usage:

The `setState` method accepts an `updater` argument that can either be an object with a number of key-value-pairs that should be merged into the state, or a function that returns such an object computed from `prevState` and `props`.

Using `setState()` with an Object as `updater`

```
//
// An example ES6 style component, updating the state on a simple button click.
// Also demonstrates where the state can be set directly and where setState should be used.
//
class Greeting extends React.Component {
    constructor(props) {
        super(props);
        this.click = this.click.bind(this);
        // Set initial state (ONLY ALLOWED IN CONSTRUCTOR)
        this.state = {
            greeting: 'Hello!'
        };
    }
    click(e) {
        this.setState({
              greeting: 'Hello World!'
        });
    }
    render() {
        return(
            <div>
                <p>{this.state.greeting}</p>
                <button onClick={this.click}>Click me</button>
            </div>
        );
    }

}
```

Using `setState()` with a Function as `updater`

```
//
// This is most often used when you want to check or make use
// of previous state before updating any values.
//

this.setState(function(previousState, currentProps) {
  return {
    counter: previousState.counter + 1
  };
});
```

This can be safer than using an object argument where multiple calls to `setState()` are used, as multiple calls may be batched together by React and executed at once, and is the preferred approach when using current props to set state.

```
this.setState({ counter: this.state.counter + 1 });
this.setState({ counter: this.state.counter + 1 });
```

```
this.setState({ counter: this.state.counter + 1 });
```

These calls may be batched together by React using `Object.assign()`, resulting in the counter being incremented by 1 rather than 3.

The functional approach can also be used to move state setting logic outside of components. This allows for isolation and re-use of state logic.

```
// Outside of component class, potentially in another file/module

function incrementCounter(previousState, currentProps) {
    return {
        counter: previousState.counter + 1
    };
}

// Within component

this.setState(incrementCounter);
```

Calling `setState()` with an Object and a callback function

```
//
// 'Hi There' will be logged to the console after setState completes
//

this.setState({ name: 'John Doe' }, console.log('Hi there'));
```

Section 4.4: State, Events And Managed Controls

Here's an example of a React component with a "managed" input field. Whenever the value of the input field changes, an event handler is called which updates the state of the component with the new value of the input field. The call to `setState` in the event handler will trigger a call to `render` updating the component in the dom.

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


class ManagedControlDemo extends React.Component {

  constructor(props){
    super(props);
    this.state = {message: ""};
  }

  handleChange(e){
    this.setState({message: e.target.value});
  }

  render() {
    return (
      <div>
        <legend>Type something here</legend>
          <input
            onChange={this.handleChange.bind(this)}
            value={this.state.message}
            autoFocus />
        <h1>{this.state.message}</h1>
      </div>
   );
```

```
  }
}


render(<ManagedControlDemo/>, document.querySelector('#app'));
```

Its very important to note the runtime behavior. Every time a user changes the value in the input field

- `handleChange` will be called and so
- `setState` will be called and so
- `render` will be called

Pop quiz, after you type a character in the input field, which DOM elements change

1. all of these - the top level div, legend, input, h1
2. only the input and h1
3. nothing
4. whats a DOM?

You can experiment with this more here to find the answer

Chapter 5: Props in React

Section 5.1: Introduction

`props` are used to pass data and methods from a parent component to a child component.

Interesting things about `props`

1. They are immutable.
2. They allow us to create reusable components.

Basic example

```
class Parent extends React.Component{
  doSomething(){
     console.log("Parent component");
  }
  render() {
    return <div>
         <Child
           text="This is the child number 1"
           title="Title 1"
           onClick={this.doSomething} />
         <Child
           text="This is the child number 2"
           title="Title 2"
           onClick={this.doSomething} />
       </div>
  }
}

class Child extends React.Component{
  render() {
    return <div>
       <h1>{this.props.title}</h1>
       <h2>{this.props.text}</h2>
      </div>
  }
}
```

As you can see in the example, thanks to `props` we can create reusable components.

Section 5.2: Default props

`defaultProps` allows you to set default, or fallback, values for your component `props`. `defaultProps` are useful when you call components from different views with fixed props, but in some views you need to pass different value.

Syntax

ES5

```
var MyClass = React.createClass({
  getDefaultProps: function() {
    return {
      randomObject: {},
```

```
      ...
    };
  }
}
```

ES6

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

MyClass.defaultProps = {
    randomObject: {},
    ...
}
```

ES7

```
class MyClass extends React.Component {
    static defaultProps = {
        randomObject: {},
        ...
    };
}
```

The result of `getDefaultProps()` or `defaultProps` will be cached and used to ensure that `this.props.randomObject` will have a value if it was not specified by the parent component.

Section 5.3: PropTypes

`propTypes` allows you to specify what `props` your component needs and the type they should be. Your component will work without setting `propTypes`, but it is good practice to define these as it will make your component more readable, act as documentation to other developers who are reading your component, and during development, React will warn you if you you try to set a prop which is a different type to the definition you have set for it.

Some primitive `propTypes` and commonly useable `propTypes` are -

```
  optionalArray: React.PropTypes.array,
  optionalBool: React.PropTypes.bool,
  optionalFunc: React.PropTypes.func,
  optionalNumber: React.PropTypes.number,
  optionalObject: React.PropTypes.object,
  optionalString: React.PropTypes.string,
  optionalSymbol: React.PropTypes.symbol
```

If you attach `isRequired` to any `propType` then that prop must be supplied while creating the instance of that component. If you don't provide the **required** `propTypes` then component instance can not be created.

Syntax

ES5

```
var MyClass = React.createClass({
  propTypes: {
    randomObject: React.PropTypes.object,
```

```
    callback: React.PropTypes.func.isRequired,
    ...
  }
}
```

ES6

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

MyClass.propTypes = {
    randomObject: React.PropTypes.object,
    callback: React.PropTypes.func.isRequired,
    ...
};
```

ES7

```
class MyClass extends React.Component {
     static propTypes = {
        randomObject: React.PropTypes.object,
        callback: React.PropTypes.func.isRequired,
        ...
    };
}
```

More complex props validation

In the same way, PropTypes allows you to specify more complex validation

Validating an object

```
...
    randomObject: React.PropTypes.shape({
        id: React.PropTypes.number.isRequired,
        text: React.PropTypes.string,
    }).isRequired,
...
```

Validating on array of objects

```
...
    arrayOfObjects: React.PropTypes.arrayOf(React.PropTypes.shape({
        id: React.PropTypes.number.isRequired,
        text: React.PropTypes.string,
    })).isRequired,
...
```

Section 5.4: Passing down props using spread operator

Instead of

```
var component = <Component foo={this.props.x} bar={this.props.y} />;
```

Where each property needs to be passed as a single prop value you could use the spread operator ... supported for arrays in ES6 to pass down all your values. The component will now look like this.

```
var component = <Component {...props} />;
```

Remember that the properties of the object that you pass in are copied onto the component's props.

The order is important. Later attributes override previous ones.

```
var props = { foo: 'default' };
var component = <Component {...props} foo={'override'} />;
console.log(component.props.foo); // 'override'
```

Another case is that you also can use spread operator to pass only parts of props to children components, then you can use destructuring syntax from props again.

It's very useful when children components need lots of props but not want pass them one by one.

```
const { foo, bar, other } = this.props // { foo: 'foo', bar: 'bar', other: 'other' };
var component = <Component {...{foo, bar}} />;
const { foo, bar } = component.props
console.log(foo, bar); // 'foo bar'
```

Section 5.5: Props.children and component composition

The "child" components of a component are available on a special prop, `props.children`. This prop is very useful for "Compositing" components together, and can make JSX markup more intuitive or reflective of the intended final structure of the DOM:

```
var SomeComponent = function () {
    return (
        <article className="textBox">
            <header>{this.props.heading}</header>
            <div className="paragraphs">
                {this.props.children}
            </div>
        </article>
    );
}
```

Which allows us to include an arbitrary number of sub-elements when using the component later:

```
var ParentComponent = function () {
    return (
        <SomeComponent heading="Amazing Article Box" >
            <p className="first"> Lots of content </p>
            <p> Or not </p>
        </SomeComponent>
    );
}
```

Props.children can also be manipulated by the component. Because props.children may or may not be an array, React provides utility functions for them as React.Children. Consider in the previous example if we had wanted to wrap each paragraph in its own **<section>** element:

```
var SomeComponent = function () {
    return (
        <article className="textBox">
            <header>{this.props.heading}</header>
            <div className="paragraphs">
```

```
                {React.Children.map(this.props.children, function (child) {
                    return (
                        <section className={child.props.className}>
                            React.cloneElement(child)
                        </section>
                    );
                })}
            </div>
        </article>
    );
}
```

Note the use of React.cloneElement to remove the props from the child **<p>** tag - because props are immutable, these values cannot be changed directly. Instead, a clone without these props must be used.

Additionally, when adding elements in loops, be aware of how React reconciles children during a rerender, and strongly consider including a globally unique `key` prop on child elements added in a loop.

Section 5.6: Detecting the type of Children components

Sometimes it's really useful to know the type of child component when iterating through them. In order to iterate through the children components you can use React `Children.map` util function:

```
React.Children.map(this.props.children, (child) => {
  if (child.type === MyComponentType) {
    ...
  }
});
```

The child object exposes the `type` property which you can compare to a specific component.

Chapter 6: React Component Lifecycle

Lifecycle methods are to be used to run code and interact with your component at different points in the components life. These methods are based around a component Mounting, Updating, and Unmounting.

Section 6.1: Component Creation

When a React component is created, a number of functions are called:

- If you are using `React.createClass` (ES5), 5 user defined functions are called
- If you are using `class Component extends React.Component` (ES6), 3 user defined functions are called

`getDefaultProps()` (ES5 only)

This is the **first** method called.

Prop values returned by this function will be used as defaults if they are not defined when the component is instantiated.

In the following example, `this.props.name` will be defaulted to `Bob` if not specified otherwise:

```
getDefaultProps() {
  return {
    initialCount: 0,
    name: 'Bob'
  };
}
```

`getInitialState()` (ES5 only)

This is the **second** method called.

The return value of `getInitialState()` defines the initial state of the React component. The React framework will call this function and assign the return value to `this.state`.

In the following example, `this.state.count` will be intialized with the value of `this.props.initialCount`:

```
getInitialState() {
  return {
    count : this.props.initialCount
  };
}
```

`componentWillMount()` (ES5 and ES6)

This is the **third** method called.

This function can be used to make final changes to the component before it will be added to the DOM.

```
componentWillMount() {
  ...
}
```

`render()` (ES5 and ES6)

This is the **fourth** method called.

The `render()` function should be a pure function of the component's state and props. It returns a single element

which represents the component during the rendering process and should either be a representation of a native DOM component (e.g. `<p />`) or a composite component. If nothing should be rendered, it can return `null` or `undefined`.

This function will be recalled after any change to the component's props or state.

```
render() {
  return (
    <div>
      Hello, {this.props.name}!
    </div>
  );
}
```

`componentDidMount()` (ES5 and ES6)

This is the **fifth** method called.

The component has been mounted and you are now able to access the component's DOM nodes, e.g. via `refs`.

This method should be used for:

- Preparing timers
- Fetching data
- Adding event listeners
- Manipulating DOM elements

```
componentDidMount() {
  ...
}
```

ES6 Syntax

If the component is defined using ES6 class syntax, the functions `getDefaultProps()` and `getInitialState()` cannot be used.

Instead, we declare our `defaultProps` as a static property on the class, and declare the state shape and initial state in the constructor of our class. These are both set on the instance of the class at construction time, before any other React lifecycle function is called.

The following example demonstrates this alternative approach:

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

    this.state = {
      count: this.props.initialCount
    };
  }

  upCount() {
    this.setState((prevState) => ({
      count: prevState.count + 1
    }));
  }

  render() {
    return (
      <div>
```

```
        Hello, {this.props.name}!<br />
        You clicked the button {this.state.count} times.<br />
        <button onClick={this.upCount}>Click here!</button>
      </div>
    );
  }
}

MyReactClass.defaultProps = {
  name: 'Bob',
  initialCount: 0
};
```

Replacing `getDefaultProps()`

Default values for the component props are specified by setting the `defaultProps` property of the class:

```
MyReactClass.defaultProps = {
  name: 'Bob',
  initialCount: 0
};
```

Replacing `getInitialState()`

The idiomatic way to set up the initial state of the component is to set `this.state` in the constructor:

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

  this.state = {
    count: this.props.initialCount
  };
}
```

Section 6.2: Component Removal

`componentWillUnmount()`

This method is called **before** a component is unmounted from the DOM.

It is a good place to perform cleaning operations like:

- Removing event listeners.
- Clearing timers.
- Stopping sockets.
- Cleaning up redux states.

```
componentWillUnmount(){
  ...
}
```

An example of removing attached event listener in `componentWillUnMount`

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

export default class SideMenu extends Component {
```

```
  constructor(props) {
    super(props);
    this.state = {
        ...
      };
    this.openMenu = this.openMenu.bind(this);
    this.closeMenu = this.closeMenu.bind(this);
  }

  componentDidMount() {
    document.addEventListener("click", this.closeMenu);
  }

  componentWillUnmount() {
    document.removeEventListener("click", this.closeMenu);
  }

  openMenu() {
    ...
  }

  closeMenu() {
    ...
  }

  render() {
    return (
      <div>
        <a
          href      = "javascript:void(0)"
          className = "closebtn"
          onClick   = {this.closeMenu}
        >
          ×
        </a>
        <div>
          Some other structure
        </div>
      </div>
    );
  }
}
```

Section 6.3: Component Update

`componentWillReceiveProps(nextProps)`

This is the **first function called on properties changes**.

When **component's properties change**, React will call this function with the **new properties**. You can access to the old props with *this.props* and to the new props with *nextProps*.

With these variables, you can do some comparison operations between old and new props, or call function because a property change, etc.

```
componentWillReceiveProps(nextProps){
  if (nextProps.initialCount && nextProps.initialCount > this.state.count){
    this.setState({
      count : nextProps.initialCount
    });
```

```
  }
}
```

shouldComponentUpdate(nextProps, nextState)

This is the **second function called on properties changes and the first on state changes**.

By default, if another component / your component change a property / a state of your component, **React** will render a new version of your component. In this case, this function always return true.

You can override this function and **choose more precisely if your component must update or not**.

This function is mostly used for **optimization**.

In case of the function returns **false**, the **update pipeline stops immediately**.

```
componentShouldUpdate(nextProps, nextState){
  return this.props.name !== nextProps.name ||
    this.state.count !== nextState.count;
}
```

componentWillUpdate(nextProps, nextState)

This function works like `componentWillMount()`. **Changes aren't in DOM**, so you can do some changes just before the update will perform.

/!\ : you cannot use **this.setState()**.

```
componentWillUpdate(nextProps, nextState){}
```

render()

There's some changes, so re-render the component.

componentDidUpdate(prevProps, prevState)

Same stuff as `componentDidMount()` : **DOM is refreshed**, so you can do some work on the DOM here.

```
componentDidUpdate(prevProps, prevState){}
```

Section 6.4: Lifecycle method call in different states

This example serves as a complement to other examples which talk about how to use the lifecycle methods and when the method will be called.

This example summarize Which methods (componentWillMount, componentWillReceiveProps, etc) will be called and in which sequence will be different for a component **in different states**:

When a component is initialized:

1. getDefaultProps
2. getInitialState
3. componentWillMount
4. render
5. componentDidMount

When a component has state changed:

1. shouldComponentUpdate

2. componentWillUpdate
3. render
4. componentDidUpdate

When a component has props changed:

1. componentWillReceiveProps
2. shouldComponentUpdate
3. componentWillUpdate
4. render
5. componentDidUpdate

When a component is unmounting:

1. componentWillUnmount

Section 6.5: React Component Container

When building a React application, it is often desirable to divide components based on their primary responsibility, into Presentational and Container components.
Presentational components are concerned only with displaying data - they can be regarded as, and are often implemented as, functions that convert a model to a view. Typically they do not maintain any internal state.
Container components are concerned with managing data. This may be done internally through their own state, or by acting as intermediaries with a state-management library such as Redux. The container component will not directly display data, rather it will pass the data to a presentational component.

```
// Container component
import React, { Component } from 'react';
import Api from 'path/to/api';

class CommentsListContainer extends Component {
    constructor() {
        super();
        // Set initial state
        this.state = { comments: [] }
    }

    componentDidMount() {
        // Make API call and update state with returned comments
        Api.getComments().then(comments => this.setState({ comments }));
    }

    render() {
        // Pass our state comments to the presentational component
        return (
            <CommentsList comments={this.state.comments} />;
        );
    }
}

// Presentational Component
const CommentsList = ({ comments }) => (
    <div>
        {comments.map(comment => (
            <div>{comment}</div>
        )}
    </div>
);
```

```
CommentsList.propTypes = {
    comments: React.PropTypes.arrayOf(React.PropTypes.string)
}
```

Chapter 7: Forms and User Input

Section 7.1: Controlled Components

Controlled form components are defined with a `value` property. The value of controlled inputs is managed by React, user inputs will not have any direct influence on the rendered input. Instead, a change to the `value` property needs to reflect this change.

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

    this.onChange = this.onChange.bind(this);

    this.state = {
      name: ''
    };
  }

  onChange(e) {
    this.setState({
      name: e.target.value
    });
  }

  render() {
    return (
      <div>
        <label for='name-input'>Name: </label>
        <input
          id='name-input'
          onChange={this.onChange}
          value={this.state.name} />
      </div>
    )
  }
}
```

The above example demonstrates how the `value` property defines the current value of the input and the `onChange` event handler updates the component's state with the user's input.

Form inputs should be defined as controlled components where possible. This ensures that the component state and the input value is in sync at all times, even if the value is changed by a trigger other than a user input.

Section 7.2: Uncontrolled Components

Uncontrolled components are inputs that do not have a `value` property. In opposite to controlled components, it is the application's responsibility to keep the component state and the input value in sync.

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

    this.onChange = this.onChange.bind(this);

    this.state = {
      name: 'John'
```

```
    };
  }

  onChange(e) {
    this.setState({
      name: e.target.value
    });
  }

  render() {
    return (
      <div>
        <label for='name-input'>Name: </label>
        <input
          id='name-input'
          onChange={this.onChange}
          defaultValue={this.state.name} />
      </div>
    )
  }
}
```

Here, the component's state is updated via the `onChange` event handler, just as for controlled components. However, instead of a `value` property, a `defaultValue` property is supplied. This determines the initial value of the input during the first render. Any subsequent changes to the component's state are not automatically reflected by the input value; If this is required, a controlled component should be used instead.

Chapter 8: React Boilerplate [React + Babel + Webpack]

Section 8.1: react-starter project

About this Project

This is simple boilerplate project. This post will guide you to set up the environment for ReactJs + Webpack + Bable.

Lets get Started

we will need node package manager for fire up express server and manage dependencies throughout the project. if you are new to node package manager, you can check here. Note : Installing node package manager is require here.

Create a folder with suitable name and navigate into it from terminal or by GUI.Then go to terminal and type `npm init` this will create a package.json file, Nothing scary , it will ask you few question like name of your project ,version, description, entry point, git repository, author, license etc. Here entry point is important because node will initially look for it when you run the project. At the end it will ask you to verify the information you provide. You can type *yes* or modify it. Well that's it , our *package.json* file is ready.

Express server setup run *npm install express@4 --save*. This is all the dependencies we needed for this project.Here save flag is important, without it *package.js* file will not be updated. Main task of *package.json* is to store list of dependencies. It will add express version 4. Your *package.json* will look like `"dependencies": { "express": "^4.13.4", ..... },`

After complete download you can see there is *node_modules* folder and sub folder of our dependencies. Now on the root of project create new file *server.js* file. Now we are setting express server. I am going to past all the code and explain it later.

```
var express = require('express');
// Create our app
var app = express();

app.use(express.static('public'));

app.listen(3000, function () {
  console.log('Express server is using port:3000');
});
```

var express = require('express'); this will gave you the access of entire express api.

var app = express(); will call express library as function. *app.use();* let the add the functionality to your express application. *app.use(express.static('public'));* will specify the folder name that will be expose in our web server. *app.listen(port, function(){})* will here our port will be *3000* and function we are calling will verify that out web server is running properly. That's it express server is set up.

Now go to our project and create a new folder public and create *index.html* file. *index.html* is the default file for you application and Express server will look for this file. The *index.html* is simple html file which looks like

```
<!DOCTYPE html>
<html>

<head>
```

```
  <meta charset="UTF-8"/>
</head>

<body>
  <h1>hello World</h1>
</body>

</html>
```

And go to the project path through the terminal and type *node server.js*. Then you will see * console.log('Express server is using port:3000');*.

Go to the browser and type *http://localhost:3000* in nav bar you will see *hello World*.

Now go inside the public folder and create a new file *app.jsx*. JSX is a preprocessor step that adds XML syntax to your JavaScript.You can definitely use React without JSX but JSX makes React a lot more elegant. Here is the sample code for *app.jsx*

```
ReactDOM.render(
  <h1>Hello World!!!</h1>,
  document.getElementById('app')
);
```

Now go to *index.html* and modify the code , it should looks like this

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8"/>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23
                /browser.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react.js">
    </script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react-dom.js">    </script>
</head>

<body>
  <div id="app"></div>

  <script type="text/babel" src="app.jsx"></script>
</body>

</html>
```

With this in place you are all done, I hope you find it simple.

Section 8.2: Setting up the project

You need Node Package Manager to install the project dependencies. Download node for your operating system from Nodejs.org. Node Package Manager comes with node.

You can also use Node Version Manager to better manage your node and npm versions. It is great for testing your project on different node versions. However, it is not recommended for production environment.

Once you have installed node on your system, go ahead and install some essential packages to blast off your first React project using Babel and Webpack.

Before we actually start hitting commands in the terminal. Take a look at what Babel and Webpack are used for.

You can start your project by running `npm init` in your terminal. Follow the initial setup. After that, run following commands in your terminal-

Dependencies:

```
npm install react react-dom --save
```

Dev Dependecies:

```
npm install babel-core babel-loader babel-preset-es2015 babel-preset-react babel-preset-stage-0
webpack webpack-dev-server react-hot-loader --save-dev
```

Optional Dev Dependencies:

```
npm install eslint eslint-plugin-react babel-eslint --save-dev
```

You may refer to this sample package.json

Create `.babelrc` in your project root with following contents:

```
{
  "presets": ["es2015", "stage-0", "react"]
}
```

Optionally create `.eslintrc` in your project root with following contents:

```
{
  "ecmaFeatures": {
    "jsx": true,
    "modules": true
  },
  "env": {
    "browser": true,
    "node": true
  },
  "parser": "babel-eslint",
  "rules": {
    "quotes": [2, "single"],
    "strict": [2, "never"],
  },
  "plugins": [
    "react"
  ]
}
```

Create a `.gitignore` file to prevent uploading generated files to your git repo.

```
node_modules
npm-debug.log
.DS_Store
dist
```

Create `webpack.config.js` file with following minimum contents.

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

```
module.exports = {
  devtool: 'eval',
  entry: [
    'webpack-dev-server/client?http://localhost:3000',
    'webpack/hot/only-dev-server',
    './src/index'
  ],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'bundle.js',
    publicPath: '/static/'
  },
  plugins: [
    new webpack.HotModuleReplacementPlugin()
  ],
  module: {
    loaders: [{
      test: /\.js$/,
      loaders: ['react-hot', 'babel'],
      include: path.join(__dirname, 'src')
    }]
  }
};
```

And finally, create a `sever.js` file to be able to run `npm start`, with following contents:

```
var webpack = require('webpack');
var WebpackDevServer = require('webpack-dev-server');
var config = require('./webpack.config');

new WebpackDevServer(webpack(config), {
  publicPath: config.output.publicPath,
  hot: true,
  historyApiFallback: true
}).listen(3000, 'localhost', function (err, result) {
  if (err) {
    return console.log(err);
  }

  console.log('Serving your awesome project at http://localhost:3000/');
});
```

Create `src/app.js` file to see your React project do something.

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

export default class App extends Component {
  render() {
    return (
      <h1>Hello, world.</h1>
    );
  }
}
```

Run `node server.js` or `npm start` in the terminal, if you have defined what `start` stands for in your `package.json`

Chapter 9: Using ReactJS with jQuery

Section 9.1: ReactJS with jQuery

Firstly, you have to import jquery library . We also need to import findDOmNode as we're going to manipulate the dom. And obviously we are importing React as well.

```
import React from 'react';
import { findDOMNode } from 'react-dom';
import $ from 'jquery';
```

We are setting an arrow function 'handleToggle' that will fire when an icon will be clicked. We're just showing and hiding a div with a reference naming 'toggle' onClick over an icon.

```
handleToggle = () => {
    const el = findDOMNode(this.refs.toggle);
    $(el).slideToggle();
};
```

Let's now set the reference naming 'toggle'

```
<ul className="profile-info additional-profile-info-list" ref="toggle">
  <li>
    <span className="info-email">Office Email</span>   me@shuvohabib.com
  </li>
</ul>
```

The div element where we will fire the 'handleToggle' on onClick.

```
<div className="ellipsis-click" onClick={this.handleToggle}>
  <i className="fa-ellipsis-h"/>
</div>
```

Let review the full code below, how it looks like .

```
import React from 'react';
import { findDOMNode } from 'react-dom';
import $ from 'jquery';

export default class FullDesc extends React.Component {
    constructor() {
        super();
    }

    handleToggle = () => {
        const el = findDOMNode(this.refs.toggle);
        $(el).slideToggle();
    };

    render() {
        return (
            <div className="long-desc">
                <ul className="profile-info">
                    <li>
                        <span className="info-title">User Name : </span> Shuvo Habib
```

```
                    </li>
                </ul>

                <ul className="profile-info additional-profile-info-list" ref="toggle">
                    <li>
                        <span className="info-email">Office Email</span> me@shuvohabib.com
                    </li>
                </ul>

                <div className="ellipsis-click" onClick={this.handleToggle}>
                    <i className="fa-ellipsis-h"/>
                </div>
            </div>
        );
    }
}
```

We are done! This is the way, how we can use **jQuery in React** component.

Chapter 10: React Routing

Section 10.1: Example Routes.js file, followed by use of Router Link in component

Place a file like the following in your top level directory. It defines which components to render for which paths

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';
import New from './containers/new-post';
import Show from './containers/show';

import Index from './containers/home';
import App from './components/app';

export default(
  <Route path="/" component={App}>
    <IndexRoute component={Index} />
    <Route path="posts/new" component={New} />
    <Route path="posts/:id" component={Show} />

  </Route>
);
```

Now in your top level index.js that is your entry point to the app, you need only render this Router component like so:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, browserHistory } from 'react-router';
// import the routes component we created in routes.js
import routes from './routes';


// entry point
ReactDOM.render(
    <Router history={browserHistory} routes={routes} />
  , document.getElementById('main'));
```

Now it is simply a matter of using `Link` instead of `<a>` tags throughout your application. Using Link will communicate with React Router to change the React Router route to the specified link, which will in turn render the correct component as defined in routes.js

```
import React from 'react';
import { Link } from 'react-router';

export default function PostButton(props) {
  return (
    <Link to={`posts/${props.postId}`}>
      <div className="post-button" >
        {props.title}
        <span>{props.tags}</span>
      </div>
    </Link>
  );
}
```

Section 10.2: React Routing Async

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';

import Index from './containers/home';
import App from './components/app';

//for single Component lazy load use this
const ContactComponent = () => {
  return {
      getComponent: (location, callback)=> {
        require.ensure([], require => {
          callback(null, require('./components/Contact')["default"]);
        }, 'Contact');
      }
    }
};

//for multiple componnets
 const groupedComponents = (pageName) => {
  return {
      getComponent: (location, callback)=> {
        require.ensure([], require => {
          switch(pageName){
            case 'about' :
                callback(null, require( "./components/about" )["default"]);
                    break ;
            case 'tos' :
                callback(null, require( "./components/tos" )["default"]);
                    break ;
          }
        }, "groupedComponents");
      }
    }
};
export default(
  <Route path="/" component={App}>
    <IndexRoute component={Index} />
    <Route path="/contact" {...ContactComponent()} />
    <Route path="/about" {...groupedComponents('about')} />
    <Route path="/tos" {...groupedComponents('tos')} />
  </Route>
);
```

Chapter 11: Communicate Between Components

Section 11.1: Communication between Stateless Functional Components

In this example we will make use of `Redux` and `React Redux` modules to handle our application state and for auto re-render of our functional components., And ofcourse `React` and `React Dom`

You can checkout the completed demo here

In the example below we have three different components and one connected component

- **UserInputForm**: This component display an input field And when the field value changes, it calls `inputChange` method on `props` (which is provided by the parent component) and if the data is provided as well, it displays that in the input field.
- **UserDashboard**: This component displays a simple message and also nests `UserInputForm` component, It also passes `inputChange` method to `UserInputForm` component, `UserInputForm` component inturn makes use of this method to communicate with the parent component.
    - **UserDashboardConnected**: This component just wraps the `UserDashboard` component using `ReactRedux connect` method., This makes it easier for us to manage the component state and update the component when the state changes.
- **App**: This component just renders the `UserDashboardConnected` component.

```
const UserInputForm = (props) => {

  let handleSubmit = (e) => {
    e.preventDefault();
  }

  return(
    <form action="" onSubmit={handleSubmit}>
      <label htmlFor="name">Please enter your name</label>
      <br />
      <input type="text" id="name" defaultValue={props.data.name || ''} onChange={
props.inputChange } />
    </form>
  )

}


const UserDashboard = (props) => {

  let inputChangeHandler = (event) => {
    props.updateName(event.target.value);
  }

  return(
    <div>
      <h1>Hi { props.user.name || 'User' }</h1>
      <UserInputForm data={props.user} inputChange={inputChangeHandler} />
    </div>
```

```
  )

}

const mapStateToProps = (state) => {
  return {
    user: state
  };
}
const mapDispatchToProps = (dispatch) => {
  return {
    updateName: (data) => dispatch( Action.updateName(data) ),
  };
};

const { connect, Provider } = ReactRedux;
const UserDashboardConnected = connect(
  mapStateToProps,
  mapDispatchToProps
)(UserDashboard);



const App = (props) => {
  return(
    <div>
      <h1>Communication between Stateless Functional Components</h1>
      <UserDashboardConnected />
    </div>
  )
}



const user = (state={name: 'John'}, action) => {
  switch (action.type) {
    case 'UPDATE_NAME':
      return Object.assign( {}, state, {name: action.payload}  );

    default:
      return state;
  }
};

const { createStore } = Redux;
const store = createStore(user);
const Action = {
  updateName: (data) => {
    return { type : 'UPDATE_NAME', payload: data }
  },
}


ReactDOM.render(
  <Provider store={ store }>
    <App />
  </Provider>,
  document.getElementById('application')
);
```

JS Bin URL

Chapter 12: How to setup a basic webpack, react and babel environment

Section 12.1: How to build a pipeline for a customized "Hello world" with images

Step 1: Install Node.js

The build pipeline you will be building is based in Node.js so you must ensure in the first instance that you have this installed. For instructions on how to install Node.js you can checkout the SO docs for that here

Step 2: Initialise your project as an node module

Open your project folder on the command line and use the following command:

```
npm init
```

For the purposes of this example you can feel free to take the defaults or if you'd like more info on what all this means you can check out this SO doc on setting up package configuration.

Step 3: Install necessary npm packages

Run the following command on the command line to install the packages necessary for this example:

```
npm install --save react react-dom
```

Then for the dev dependencies run this command:

```
npm install --save-dev babel-core babel-preset-react babel-preset-es2015 webpack babel-loader css-loader style-loader file-loader image-webpack-loader
```

Finally webpack and webpack-dev-server are things that are worth installing globally rather than as a dependency of your project, if you'd prefer to add it as a dependency then that will work to, I don't. Here is the command to run:

```
npm install --global webpack webpack-dev-server
```

Step 3: Add a .babelrc file to the root of your project

This will setup babel to use the presets you've just installed. Your .babelrc file should look like this:

```
{
  "presets": ["react", "es2015"]
}
```

Step 4: Setup project directory structure

Set yourself up a directory stucture that looks like the below in the root of your directory:

```
|- node_modules
|- src/
  |- components/
  |- images/
  |- styles/
  |- index.html
```

```
  |- index.jsx
|- .babelrc
|- package.json
```

NOTE: The `node_modules`, `.babelrc` and `package.json` should all have already been there from previous steps I just included them so you can see where they fit.

Step 5: Populate the project with the Hello World project files

This isn't really important to the process of building a pipeline so I'll just give you the code for these and you can copy paste them in:

src/components/HelloWorldComponent.jsx

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

class HelloWorldComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {name: 'Student'};
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.setState({name: e.target.value});
  }

  render() {
    return (
      <div>
        <div className="image-container">
          <img src="./images/myImage.gif" />
        </div>
        <div className="form">
          <input type="text" onChange={this.handleChange} />
          <div>
            My name is {this.state.name} and I'm a clever cloggs because I built a React build
pipeline
          </div>
        </div>
      </div>
    );
  }
}

export default HelloWorldComponent;
```

src/images/myImage.gif

Feel free to substitute this with any image you'd like it's simply there to prove the point that we can bundle up images as well. If you provide your own image and you name it something different then you'll have to update the `HelloWorldComponent.jsx` to reflect your changes. Equally if you choose an image with a different file extension then you need to modify the `test` property of the image loader in the webpack.config.js with appropriate regex to match your new file extension..

src/styles/styles.css

```
.form {
  margin: 25px;
```

```
  padding: 25px;
  border: 1px solid #ddd;
  background-color: #eaeaea;
  border-radius: 10px;
}

.form div {
  padding-top: 25px;
}

.image-container {
  display: flex;
  justify-content: center;
}
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Learning to build a react pipeline</title>
</head>
<body>
  <div id="content"></div>
  <script src="app.js"></script>
</body>
</html>
```

index.jsx

```
import React from 'react';
import { render } from 'react-dom';
import HelloWorldComponent from './components/HelloWorldComponent.jsx';

require('./images/myImage.gif');
require('./styles/styles.css');
require('./index.html');

render(<HelloWorldComponent />, document.getElementById('content'));
```

Step 6: Create webpack configuration

Create a file called webpack.config.js in the root of your project and copy this code into it:

webpack.config.js

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

var config = {
  context: path.resolve(__dirname + '/src'),
  entry: './index.jsx',
  output: {
    filename: 'app.js',
    path: path.resolve(__dirname + '/dist'),
  },
  devServer: {
    contentBase: path.join(__dirname + '/dist'),
    port: 3000,
    open: true,
```

```
  },
  module: {
    loaders: [
      {
        test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        loader: 'babel-loader'
      },
      {
        test: /\.css$/,
        loader: "style!css"
      },
      {
        test: /\.gif$/,
        loaders: [
          'file?name=[path][name].[ext]',
          'image-webpack',
        ]
      },
      { test: /\.(html)$/,
        loader: "file?name=[path][name].[ext]"
      }
    ],
  },
};

module.exports = config;
```

Step 7: Create npm tasks for your pipeline

To do this you will need to add two properties to the scripts key of the JSON defined in the package.json file in the root of your project. Make your scripts key look like this:

```
  "scripts": {
    "start": "webpack-dev-server",
    "build": "webpack",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
```

The `test` script will have already been there and you can choose whether to keep it or not, it's not important to this example.

Step 8: Use the pipeline

From the command line, if you are in the project root directory you should now be able to run the command:

```
npm run build
```

This will bundle up the little application you've built and place it in the `dist/` directory that it will create in the root of your project folder.

If you run the command:

```
npm start
```

Then the application you've built will be served up in your default web browser inside of a webpack dev server instance.

Chapter 13: React.createClass vs extends React.Component

Section 13.1: Create React Component

Let's explore the syntax differences by comparing two code examples.

React.createClass (deprecated)

Here we have a `const` with a React class assigned, with the `render` function following on to complete a typical base component definition.

```
import React from 'react';

const MyComponent = React.createClass({
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

Let's take the above React.createClass definition and convert it to use an ES6 class.

```
import React from 'react';

class MyComponent extends React.Component {
  render() {
    return (
      <div></div>
    );
  }
}

export default MyComponent;
```

In this example we're now using ES6 classes. For the React changes, we now create a class called **MyComponent** and extend from React.Component instead of accessing React.createClass directly. This way, we use less React boilerplate and more JavaScript.

PS: Typically this would be used with something like Babel to compile the ES6 to ES5 to work in other browsers.

Section 13.2: "this" Context

Using React.createClass will automatically bind `this` context (values) correctly, but that is not the case when using ES6 classes.

React.createClass

Note the `onClick` declaration with the `this.handleClick` method bound. When this method gets called React will apply the right execution context to the `handleClick`.

```
import React from 'react';

const MyComponent = React.createClass({
  handleClick() {
    console.log(this); // the React Component instance
  },
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
});

export default MyComponent;
```

React.Component

With ES6 classes `this` is `null` by default, properties of the class do not automatically bind to the React class (component) instance.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick() {
    console.log(this); // null
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

There are a few ways we could bind the right `this` context.

Case 1: Bind inline:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick() {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick.bind(this)}></div>
    );
  }
}

export default MyComponent;
```

Case 2: Bind in the class constructor

Another approach is changing the context of `this.handleClick` inside the `constructor`. This way we avoid inline repetition. Considered by many as a better approach that avoids touching JSX at all:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

Case 3: Use ES6 anonymous function

You can also use ES6 anonymous function without having to bind explicitly:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick = () => {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

Section 13.3: Declare Default Props and PropTypes

There are important changes in how we use and declare default props and their types.

React.createClass

In this version, the `propTypes` property is an Object in which we can declare the type for each prop. The `getDefaultProps` property is a function that returns an Object to create the initial props.

```
import React from 'react';

const MyComponent = React.createClass({
```

```
  propTypes: {
    name: React.PropTypes.string,
    position: React.PropTypes.number
  },
  getDefaultProps() {
    return {
      name: 'Home',
      position: 1
    };
  },
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

This version uses `propTypes` as a property on the actual **MyComponent** class instead of a property as part of the `createClass` definition Object.

The `getDefaultProps` has now changed to just an Object property on the class called defaultProps, as it's no longer a "get" function, it's just an Object. It avoids more React boilerplate, this is just plain JavaScript.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div></div>
    );
  }
}
MyComponent.propTypes = {
  name: React.PropTypes.string,
  position: React.PropTypes.number
};
MyComponent.defaultProps = {
  name: 'Home',
  position: 1
};

export default MyComponent;
```

Additionally, there is another syntax for `propTypes` and `defaultProps`. This is a shortcut if your build has ES7 property initializers turned on:

```
import React from 'react';

class MyComponent extends React.Component {
  static propTypes = {
    name: React.PropTypes.string,
    position: React.PropTypes.number
  };
```

```
  static defaultProps = {
    name: 'Home',
    position: 1
  };
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div></div>
    );
  }
}

export default MyComponent;
```

Section 13.4: Mixins

We can use `mixins` only with the React.createClass way.

React.createClass

In this version we can add `mixins` to components using the mixins property which takes an Array of available mixins. These then extend the component class.

```
import React from 'react';

var MyMixin = {
  doSomething() {

  }
};
const MyComponent = React.createClass({
  mixins: [MyMixin],
  handleClick() {
    this.doSomething(); // invoke mixin's method
  },
  render() {
    return (
      <button onClick={this.handleClick}>Do Something</button>
    );
  }
});

export default MyComponent;
```

React.Component

React mixins are not supported when using React components written in ES6. Moreover, they will not have support for ES6 classes in React. The reason is that they are considered harmful.

Section 13.5: Set Initial State

There are changes in how we are setting the initial states.

React.createClass

We have a `getInitialState` function, which simply returns an Object of initial states.

```
import React from 'react';

const MyComponent = React.createClass({
  getInitialState () {
    return {
      activePage: 1
    };
  },
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

In this version we declare all state as a simple **initialisation property in the constructor**, instead of using the `getInitialState` function. It feels less "React API" driven since this is just plain JavaScript.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      activePage: 1
    };
  }
  render() {
    return (
      <div></div>
    );
  }
}

export default MyComponent;
```

Section 13.6: ES6/React “this” keyword with ajax to get data from server

```
import React from 'react';

class SearchEs6 extends React.Component{
    constructor(props) {
        super(props);
        this.state = {
            searchResults: []
        };
    }

    showResults(response){
        this.setState({
            searchResults: response.results
        })
    }
```

```
    search(url){
        $.ajax({
            type: "GET",
            dataType: 'jsonp',
            url: url,
            success: (data) => {
                this.showResults(data);
            },
            error: (xhr, status, err) => {
                console.error(url, status, err.toString());
            }
        });
    }

    render() {
        return (
            <div>
                <SearchBox search={this.search.bind(this)} />
                <Results searchResults={this.state.searchResults} />
            </div>
        );
    }
}
```

Chapter 14: React AJAX call

Section 14.1: HTTP GET request

Sometimes a component needs to render some data from a remote endpoint (e.g. a REST API). A standard practice is to make such calls in `componentDidMount` method.

Here is an example, using superagent as AJAX helper:

```
import React from 'react'
import request from 'superagent'

class App extends React.Component {
  constructor () {
    super()
    this.state = {}
  }
  componentDidMount () {
    request
      .get('/search')
      .query({ query: 'Manny' })
      .query({ range: '1..5' })
      .query({ order: 'desc' })
      .set('API-Key', 'foobar')
      .set('Accept', 'application/json')
      .end((err, resp) => {
        if (!err) {
          this.setState({someData: resp.text})
        }
      })
  },
  render() {
    return (
      <div>{this.state.someData || 'waiting for response...'}</div>
    )
  }
}

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

A request can be initiated by invoking the appropriate method on the `request` object, then calling `.end()` to send the request. Setting header fields is simple, invoke `.set()` with a field name and value.

The `.query()` method accepts objects, which when used with the GET method will form a query-string. The following will produce the path `/search?query=Manny&range=1..5&order=desc`.

POST requests

```
request.post('/user')
  .set('Content-Type', 'application/json')
  .send('{"name":"tj","pet":"tobi"}')
  .end(callback)
```

See Superagent docs for more details.

Section 14.2: HTTP GET request and looping through data

The following example shows how a set of data obtained from a remote source can be rendered into a component.

We make an AJAX request using fetch, which is build into most browsers. Use a fetch polyfill in production to support older browsers. You can also use any other library for making requests (e.g. axios, SuperAgent, or even plain Javascript).

We set the data we receive as component state, so we can access it inside the render method. There, we loop through the data using map. Don't forget to always add a unique key attribute (or prop) to the looped element, which is important for React's rendering performance.

```
import React from 'react';

class Users extends React.Component {
  constructor() {
    super();
    this.state = { users: [] };
  }

  componentDidMount() {
    fetch('/api/users')
      .then(response => response.json())
      .then(json => this.setState({ users: json.data }));
  }

  render() {
    return (
      <div>
        <h1>Users</h1>
        {
          this.state.users.length == 0
            ? 'Loading users...'
            : this.state.users.map(user => (
              <figure key={user.id}>
                <img src={user.avatar} />
                <figcaption>
                  {user.name}
                </figcaption>
              </figure>
            ))
        }
      </div>
    );
  }
}

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

Working example on JSBin.

Section 14.3: Ajax in React without a third party library - a.k.a with VanillaJS

The following would work in IE9+

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

```
class App extends React.Component {
  constructor () {
    super()
    this.state = {someData: null}
  }
  componentDidMount () {
    var request = new XMLHttpRequest();
    request.open('GET', '/my/url', true);

    request.onload = () => {
      if (request.status >= 200 && request.status < 400) {
        // Success!
        this.setState({someData: request.responseText})
      } else {
        // We reached our target server, but it returned an error
        // Possibly handle the error by changing your state.
      }
    };

    request.onerror = () => {
      // There was a connection error of some sort.
      // Possibly handle the error by changing your state.
    };

    request.send();
  },
  render() {
    return (
      <div>{this.state.someData || 'waiting for response...'}</div>
    )
  }
}

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

Chapter 15: Communication Between Components

Section 15.1: Child to Parent Components

Sending data back to the parent, to do this we simply **pass a function as a prop from the parent component to the child component**, and **the child component calls that function**.

In this example, we will change the Parent state by passing a function to the Child component and invoking that function inside the Child component.

```
import React from 'react';

class Parent extends React.Component {
    constructor(props) {
        super(props);
        this.state = { count: 0 };

        this.outputEvent = this.outputEvent.bind(this);
    }
    outputEvent(event) {
        // the event context comes from the Child
        this.setState({ count: this.state.count++ });
    }

    render() {
        const variable = 5;
        return (
            <div>
                Count: { this.state.count }
                <Child clickHandler={this.outputEvent} />
            </div>
        );
    }
}

class Child extends React.Component {
    render() {
        return (
            <button onClick={this.props.clickHandler}>
                Add One More
            </button>
        );
    }
}

export default Parent;
```

Note that the Parent's `outputEvent` method (that changes the Parent state) is invoked by the Child's button `onClick` event.

Section 15.2: Not-related Components

The only way if your components does not have a parent-child relationship (or are related but too further such as a grand grand grand son) is to have some kind of a signal that one component subscribes to, and the other writes into.

Those are the 2 basic operations of any event system: **subscribe/listen** to an event to be notify, and **send/trigger/publish/dispatch** a event to notify the ones who wants.

There are at least 3 patterns to do that. You can find a comparison here.

Here is a brief summary:

- Pattern 1: **Event Emitter/Target/Dispatcher**: the listeners need to reference the source to subscribe.
    - to subscribe: `otherObject.addEventListener('click', () => { alert('click!'); });`
    - to dispatch: `this.dispatchEvent('click');`
- Pattern 2: **Publish/Subscribe**: you don't need a specific reference to the source that triggers the event, there is a global object accessible everywhere that handles all the events.
    - to subscribe: `globalBroadcaster.subscribe('click', () => { alert('click!'); });`
    - to dispatch: `globalBroadcaster.publish('click');`
- Pattern 3: **Signals**: similar to Event Emitter/Target/Dispatcher but you don't use any random strings here. Each object that could emit events needs to have a specific property with that name. This way, you know exactly what events can an object emit.
    - to subscribe: `otherObject.clicked.add( () => { alert('click'); });`
    - to dispatch: `this.clicked.dispatch();`

Section 15.3: Parent to Child Components

That the easiest case actually, very natural in the React world and the chances are - you are already using it.

You can **pass props down to child components**. In this example `message` is the prop that we pass down to the child component, the name message is chosen arbitrarily, you can name it anything you want.

```
import React from 'react';

class Parent extends React.Component {
    render() {
        const variable = 5;
        return (
            <div>
                <Child message="message for child" />
                <Child message={variable} />
            </div>
        );
    }
}

class Child extends React.Component {
    render() {
        return <h1>{this.props.message}</h1>
    }
}

export default Parent;
```

Here, the **`<Parent />`** component renders two **`<Child />`** components, passing `message for child` inside the first component and `5` inside the second one.

Chapter 16: Stateless Functional Components

Section 16.1: Stateless Functional Component

Components let you split the UI into *independent*, *reusable* pieces. This is the beauty of React; we can separate a page into many small reusable **components**.

Prior to React v14 we could create a stateful React component using `React.Component` (in ES6), or `React.createClass` (in ES5), irrespective of whether it requires any state to manage data or not.

React v14 introduced a simpler way to define components, usually referred to as **stateless functional components**. These components use plain JavaScript functions.

For example:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

This function is a valid React component because it accepts a single `props` object argument with data and returns a React element. We call such components **functional** because they are literally JavaScript *functions*.

Stateless functional components typically focus on UI; state should be managed by higher-level "container" components, or via Flux/Redux etc. Stateless functional components don't support state or lifecycle methods.

Benefits:

1. No class overhead
2. Don't have to worry about `this` keyword
3. Easy to write and easy to understand
4. Don't have to worry about managing state values
5. Performance improvement

Summary: If you are writing a React component that doesn't require state and would like to create a reusable UI, instead of creating a standard React Component you can write it as a **stateless functional component**.

Let's take a simple example :

Let's say we have a page that can register a user, search for registered users, or display a list of all the registered users.

This is entry point of the application, `index.js`:

```
import React from 'react';
import ReactDOM from 'react-dom';

import HomePage from './homepage'

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

The `HomePage` component provides the UI to register and search for users. Note that it is a typical React component including state, UI, and behavioral code. The data for the list of registered users is stored in the `state` variable, but our reusable `List` (shown below) encapsulates the UI code for the list.

`homepage.js`:

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

import List from './list';

export default class Temp extends Component{

    constructor(props) {
        super();
        this.state={users:[], showSearchResult: false, searchResult: []};
    }

    registerClick(){
        let users = this.state.users.slice();
        if(users.indexOf(this.refs.mail_id.value) == -1){
            users.push(this.refs.mail_id.value);
            this.refs.mail_id.value = '';
            this.setState({users});
        }else{
            alert('user already registered');
        }
    }

    searchClick(){
        let users = this.state.users;
        let index = users.indexOf(this.refs.search.value);
        if(index >= 0){
            this.setState({searchResult: users[index], showSearchResult: true});
        }else{
            alert('no user found with this mail id');
        }
    }

    hideSearchResult(){
        this.setState({showSearchResult: false});
    }

    render() {
        return (
            <div>
                <input placeholder='email-id' ref='mail_id'/>
                <input type='submit' value='Click here to register'
onClick={this.registerClick.bind(this)}/>
                <input style={{marginLeft: '100px'}} placeholder='search' ref='search'/>
                <input type='submit' value='Click here to register'
onClick={this.searchClick.bind(this)}/>
                {this.state.showSearchResult ?
                    <div>
                        Search Result:
                        <List users={[this.state.searchResult]}/>
                        <p onClick={this.hideSearchResult.bind(this)}>Close this</p>
                    </div>
                    :
                    <div>
                        Registered users:
```

```
                        <br/>
                        {this.state.users.length ?
                            <List users={this.state.users}/>
                            :
                            "no user is registered"
                        }
                    </div>
                }
            </div>
        );
    }
}
```

Finally, our **stateless functional component** `List`, which is used display both the list of registered users *and* the search results, but without maintaining any state itself.

`list.js`:

```
import React from 'react';
var colors = ['#6A1B9A', '#76FF03', '#4527A0'];

var List = (props) => {
    return(
        <div>
            {
                props.users.map((user, i)=>{
                    return(
                        <div key={i} style={{color: colors[i%3]}}>
                            {user}
                        </div>
                    );
                })
            }
        </div>
    );
}

export default List;
```

Reference: https://facebook.github.io/react/docs/components-and-props.html

Chapter 17: Performance

Section 17.1: Performance measurement with ReactJS

You can't improve something you can't measure. To improve the performance of React components, you should be able to measure it. ReactJS provides with *addon* tools to measure performance. Import the `react-addons-perf` module to measure the performance

```
import Perf from 'react-addons-perf' // ES6
var Perf = require('react-addons-perf') // ES5 with npm
var Perf = React.addons.Perf; // ES5 with react-with-addons.js
```

You can use below methods from the imported `Perf` module:

- Perf.printInclusive()
- Perf.printExclusive()
- Perf.printWasted()
- Perf.printOperations()
- Perf.printDOM()

The most important one which you will need most of the time is `Perf.printWasted()` which gives you the tabular representation of your individual component's wasted time

(index)	Owner > component	Wasted time (ms)	Instances
0	"Todos > TodoItem"	102.76999999977124	1000

Total time: 132.71 ms react-with-addons.js:9900

You can note the **Wasted time** column in the table and improve Component's performance using **Tips & Tricks** section above

Refer the React Official Guide and excellent article by Benchling Engg. on React Performance

Section 17.2: React's diff algorithm

Generating the minimum number of operations to transform one tree into another have a complexity in the order of O(n^3) where n is the number of nodes in the tree. React relies on two assumptions to solve this problem in a linear time - O(n)

1. Two components of the same class will generate similar trees and tw components of different classes will generate different trees.

2. It is possible to provide a unique key for elements that is stable across different renders.

In order to decide if two nodes are different, React differentiates 3 cases

1. Two nodes are different, if they have different types.

- for example, ** ... ** is different from **...**

2. Whenever two nodes have different keys

- for example, `<div key="1">...</div>` is different from `<div key="2">...</div>`

Moreover, **what follows is crucial and extremely important to understand** if you want to optimise performance

> If they [two nodes] are not of the same type, React is not going to even try at matching what they render. It is just going to remove the first one from the DOM and insert the second one.

Here's why

> It is very unlikely that a element is going to generate a DOM that is going to look like what a would generate. Instead of spending time trying to match those two structures, React just re-builds the tree from scratch.

Section 17.3: The Basics - HTML DOM vs Virtual DOM

HTML DOM is Expensive

Each web page is represented internally as a tree of objects. This representation is called *Document Object Model*. Moreover, it is a language-neutral interface that allows programming languages (such as JavaScript) to access the HTML elements.

In other words

> The HTML DOM is a standard for how to get, change, add, or delete HTML elements.

However, those **DOM operations** are extremely **expensive**.

Virtual DOM is a Solution

So React's team came up with the idea to abstract the *HTML DOM* and create its own *Virtual DOM* in order to compute the minimum number of operations we need to apply on the *HTML DOM* to replicate current state of our application.

The Virtual DOM saves time from unnecessary DOM modifications.

How Exactly?

At each point of time, React has the application state represented as a `Virtual DOM`. Whenever application state changes, these are the steps that React performs in order to optimise performance

1. Generate a new *Virtual DOM* that represents the new state of our application
2. Compare the old Virtual DOM (which represents the current HTML DOM) vs the new Virtual DOM
3. Based on 2. find the minimum number of operations to transform the old Virtual DOM (which represents the current HTML DOM) into the new Virtual DOM

- to learn more about that - read React's Diff Algorithm

4. After those operations are found, they are mapped into their equivalent *HTML DOM* operations

- remember, the *Virtual DOM* is only an abstraction of the *HTML DOM* and there is a isomorphic relation between them

5. Now the minimum number of operations that have been found and transferred to their equivalent *HTML DOM* operations are now applied directly onto the application's *HTML DOM*, which saves time from modifying the *HTML DOM* unnecessarily.

Note: Operations applied on the Virtual DOM are cheap, because the Virtual DOM is a JavaScript Object.

Section 17.4: Tips & Tricks

When two nodes are not of the same type, React doesn't try to match them - it just removes the first node from the DOM and inserts the second one. This is why the first tip says

1. If you see yourself alternating between two components classes with very similar output, you may want to make it the same class.

2. Use shouldComponentUpdate to prevent component from rerender, if you know it is not going to change, for example

```
shouldComponentUpdate: function(nextProps, nextState) {
  return nextProps.id !== this.props.id;
}
```

Chapter 18: Introduction to Server-Side Rendering

Section 18.1: Rendering components

There are two options to render components on server: `renderToString` and `renderToStaticMarkup`.

renderToString

This will render React components to HTML on server. This function will also add `data-react-` properties to HTML elements so React on client won't have to render elements again.

```
import { renderToString } from "react-dom/server";
renderToString(<App />);
```

renderToStaticMarkup

This will render React components to HTML, but without `data-react-` properties, it is not recommended to use components that will be rendered on client, because components will rerender.

```
import { renderToStaticMarkup } from "react-dom/server";
renderToStaticMarkup(<App />);
```

Chapter 19: Setting Up React Environment

Section 19.1: Simple React Component

We want to be able to compile below component and render it in our webpage

Filename: src/index.jsx

```
import React from 'react';
import ReactDOM from 'react-dom';

class ToDo extends React.Component {
    render() {
        return (<div>I am working</div>);
    }
}

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

Section 19.2: Install all dependencies

```
# install react and react-dom
$ npm i react react-dom --save

# install webpack for bundling
$ npm i webpack -g

# install babel for module loading, bundling and transpiling
$ npm i babel-core babel-loader --save

# install babel presets for react and es6
$ npm i babel-preset-react babel-preset-es2015 --save
```

Section 19.3: Configure webpack

Create a file `webpack.config.js` in the root of your working directory

Filename: webpack.config.js

```
module.exports = {
    entry: __dirname + "/src/index.jsx",
    devtool: "source-map",
    output: {
        path: __dirname + "/build",
        filename: "bundle.js"
    },
    module: {
        loaders: [
            {test: /\.jsx?$/, exclude: /node_modules/, loader: "babel-loader"}
        ]
    }
}
```

Section 19.4: Configure babel

Create a file `.babelrc` in the root of our working directory

Filename: .babelrc

```
{
    "presets": ["es2015","react"]
}
```

Section 19.5: HTML file to use react component

Setup a simple html file in the root of the project directory

Filename: index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="App"></div>
    <script src="build/bundle.js" charset="utf-8"></script>
  </body>
</html>
```

Section 19.6: Transpile and bundle your component

Using webpack, you can bundle your component:

$ webpack

This will create our output file in `build` directory.

Open the HTML page in a browser to see component in action

Chapter 20: Using React with Flow

How to use the Flow type checker to check types in React components.

Section 20.1: Using Flow to check prop types of stateless functional components

```
type Props = {
  posts: Array<Article>,
  dispatch: Function,
  children: ReactElement
}

const AppContainer =
  ({ posts, dispatch, children }: Props) => (
    <div className="main-app">
      <Header {...{ posts, dispatch }} />
      {children}
    </div>
  )
```

Section 20.2: Using Flow to check prop types

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

type Props = {
  posts: Array<Article>,
  dispatch: Function,
  children: ReactElement
}

class Posts extends Component {
  props: Props;

  render () {
    // rest of the code goes here
  }
}
```

Chapter 21: JSX

Section 21.1: Props in JSX

There are several different ways to specify props in JSX.

JavaScript Expressions

You can pass **any JavaScript expression** as a prop, by surrounding it with `{}`. For example, in this JSX:

```
<MyComponent count={1 + 2 + 3 + 4} />
```

Inside the `MyComponent`, the value of `props.count` will be `10`, because the expression `1 + 2 + 3 + 4` gets evaluated.

If statements and for loops are not expressions in JavaScript, so they can't be used in JSX directly.

String Literals

Of course, you can just pass any `string literal` as a `prop` too. These two JSX expressions are equivalent:

```
<MyComponent message="hello world" />

<MyComponent message={'hello world'} />
```

When you pass a string literal, its value is HTML-unescaped. So these two JSX expressions are equivalent:

```
<MyComponent message="&lt;3" />

<MyComponent message={'<3'} />
```

This behavior is usually not relevant. It's only mentioned here for completeness.

Props Default Value

If you pass no value for a prop, **it defaults to `true`**. These two JSX expressions are equivalent:

```
<MyTextBox autocomplete />

<MyTextBox autocomplete={true} />
```

However, the React team says in their docs **using this approach is not recommended**, because it can be confused with the ES6 object shorthand `{foo}` which is short for `{foo: foo}` rather than `{foo: true}`. They say this behavior is just there so that it matches the behavior of HTML.

Spread Attributes

If you already have props as an object, and you want to pass it in JSX, you can use `...` as a spread operator to pass the whole props object. These two components are equivalent:

```
function Case1() {
  return <Greeting firstName="Kaloyab" lastName="Kosev" />;
}
```

```
function Case2() {
  const person = {firstName: 'Kaloyan', lastName: 'Kosev'};
  return <Greeting {...person} />;
}
```

Section 21.2: Children in JSX

In JSX expressions that contain both an opening tag and a closing tag, the content between those tags is passed as a special prop: `props.children`. There are several different ways to pass children:

String Literals

You can put a string between the opening and closing tags and `props.children` will just be that string. This is useful for many of the built-in HTML elements. For example:

```
<MyComponent>
    <h1>Hello world!</h1>
</MyComponent>
```

This is valid JSX, and `props.children` in MyComponent will simply be `<h1>Hello world!</h1>`.

Note that **the HTML is unescaped**, so you can generally write JSX just like you would write HTML.

Bare in mind, that in this case JSX:

- removes whitespace at the beginning and ending of a line;
- removes blank lines;
- new lines adjacent to tags are removed;
- new lines that occur in the middle of string literals are condensed into a single space.

JSX Children

You can provide more JSX elements as the children. This is useful for displaying nested components:

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

You can **mix together different types of children, so you can use string literals together with JSX children**. This is another way in which JSX is like HTML, so that this is both valid JSX and valid HTML:

```
<div>
  <h2>Here is a list</h2>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

Note that a React component **can't return multiple React elements, but a single JSX expression can have multiple children**. So if you want a component to render multiple things you can wrap them in a `div` like the example above.

JavaScript Expressions

You can pass any JavaScript expression as children, by enclosing it within {}. For example, these expressions are equivalent:

```
<MyComponent>foo</MyComponent>

<MyComponent>{'foo'}</MyComponent>
```

This is often useful for rendering a list of JSX expressions of arbitrary length. For example, this renders an HTML list:

```
const Item = ({ message }) => (
  <li>{ message }</li>
);

const TodoList = () => {
  const todos = ['finish doc', 'submit review', 'wait stackoverflow review'];
  return (
    <ul>
      { todos.map(message => (<Item key={message} message={message} />)) }
    </ul>
  );
};
```

Note that JavaScript expressions can be mixed with other types of children.

Functions as Children

Normally, JavaScript expressions inserted in JSX will evaluate to a string, a React element, or a list of those things. However, `props.children` works just like any other prop in that it can pass any sort of data, not just the sorts that React knows how to render. For example, if you have a custom component, you could have it take a callback as `props.children`:

```
const ListOfTenThings = () => (
  <Repeat numTimes={10}>
    {(index) => <div key={index}>This is item {index} in the list</div>}
  </Repeat>
);

// Calls the children callback numTimes to produce a repeated component
const Repeat = ({ numTimes, children }) => {
  let items = [];
  for (let i = 0; i < numTimes; i++) {
    items.push(children(i));
  }
  return <div>{items}</div>;
};
```

Children passed to a custom component can be anything, as long as that component transforms them into something React can understand before rendering. This usage is not common, but it works if you want to stretch what JSX is capable of.

Ignored Values

Note that **`false`**, **`null`**, **`undefined`**, and **`true`** are valid children. But they simply don't render. These JSX expressions

will all render to the same thing:

```
<MyComponent />

<MyComponent></MyComponent>

<MyComponent>{false}</MyComponent>

<MyComponent>{null}</MyComponent>

<MyComponent>{true}</MyComponent>
```

This is extremely useful to conditionally render React elements. This JSX only renders a if `showHeader` is true:

```
<div>
  {showHeader && <Header />}
 <Content />
</div>
```

One important caveat is that some "falsy" values, such as the `0` number, are still rendered by React. For example, this code will not behave as you might expect because `0` will be printed when `props.messages` is an empty array:

```
<div>
  {props.messages.length &&
   <MessageList messages={props.messages} />
 }
</div>
```

One approach to fix this is to make sure that the expression before the `&&` is always boolean:

```
<div>
  {props.messages.length > 0 &&
   <MessageList messages={props.messages} />
 }
</div>
```

Lastly, bare in mind that if you want a value like **`false`**, **`true`**, **`null`**, or **`undefined`** to appear in the output, you have to convert it to a string first:

```
<div>
  My JavaScript variable is {String(myVariable)}.
</div>
```

Chapter 22: React Forms

Section 22.1: Controlled Components

A controlled component is bound to a value and its changes get handled in code using event based callbacks.

```
class CustomForm extends React.Component {
constructor() {
    super();
    this.state = {
        person: {
            firstName: '',
            lastName: ''
        }
    }
}

handleChange(event) {
    let person = this.state.person;
    person[event.target.name] = event.target.value;
    this.setState({person});
}

render() {
    return (
        <form>
            <input
            type="text"
            name="firstName"
            value={this.state.firstName}
            onChange={this.handleChange.bind(this)} />

            <input
            type="text"
            name="lastName"
            value={this.state.lastName}
            onChange={this.handleChange.bind(this)} />
        </form>
    )
}
```

}

In this example we initialize state with an empty person object. We then bind the values of the 2 inputs to the individual keys of the person object. Then as the user types, we capture each value in the `handleChange` function. Since the values of the components are bound to state we can rerender as the user types by calling `setState()`.

NOTE: Not calling `setState()` when dealing with controlled components, will cause the user to type, but not see the input because React only renders changes when it is told to do so.

It's also important to note that the names of the inputs are same as the names of the keys in the person object. This allows us to capture the value in dictionary form as seen here.

```
handleChange(event) {
    let person = this.state.person;
    person[event.target.name] = event.target.value;
    this.setState({person});
```

```
}
```

`person[event.target.name]` is the same is a `person.firstName || person.lastName`. Of course this would depend on which input is currently being typed in. Since we don't know where the user will be typing, using a dictionary and matching the input names to the names of the keys, allows us to capture the user input no matter where the onChange is being called from.

Chapter 23: User interface solutions

Let's say we get inspired of some ideas from modern user interfaces used in programs and convert them to React components. That's what "**User interface solutions**" topic consists of. Attribution is appretiated.

Section 23.1: Basic Pane

```
import React from 'react';

class Pane extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        return React.createElement(
            'section', this.props
        );
    }
}
```

Section 23.2: Panel

```
import React from 'react';

class Panel extends React.Component {
    constructor(props) {
        super(props);
    }

    render(...elements) {
        var props = Object.assign({
            className: this.props.active ? 'active' : '',
            tabIndex: -1
        }, this.props);

        var css = this.css();
        if (css != '') {
            elements.unshift(React.createElement(
                'style', null,
                css
            ));
        }

        return React.createElement(
            'div', props,
            ...elements
        );
    }

    static title() {
        return '';
    }
    static css() {
        return '';
    }
}
```

Major differences from simple pane are:

- panel has focus in instance when it is called by script or clicked by mouse;
- panel has `title` static method per component, so it may be extended by other panel component with overridden `title` (reason here is that function can be then called again on rendering for localization purposes, but in bounds of this example `title` doesn't make sense);
- it can contain individual stylesheet declared in `css` static method (you can pre-load file contents from `PANEL.css`).

Section 23.3: Tab

```
import React from 'react';

class Tab extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        var props = Object.assign({
            className: this.props.active ? 'active' : ''
        }, this.props);
        return React.createElement(
            'li', props,
            React.createElement(
                'span', props,
                props.panelClass.title()
            )
        );
    }
}
```

`panelClass` property of `Tab` instance must contain class of *panel* used for description.

Section 23.4: PanelGroup

```
import React from 'react';
import Tab from './Tab.js';

class PanelGroup extends React.Component {
    constructor(props) {
        super(props);
        this.setState({
            panels: props.panels
        });
    }

    render() {
        this.tabSet = [];
        this.panelSet = [];
        for (let panelData of this.state.panels) {
            var tabIsActive = this.state.activeTab == panelData.name;
            this.tabSet.push(React.createElement(
                Tab, {
                    name: panelData.name,
                    active: tabIsActive,
                    panelClass: panelData.class,
                    onMouseDown: () => this.openTab(panelData.name)
                }
```

```
            ));
            this.panelSet.push(React.createElement(
                panelData.class, {
                    id: panelData.name,
                    active: tabIsActive,
                    ref: tabIsActive ? 'activePanel' : null
                }
            ));
        }
        return React.createElement(
            'div', { className: 'PanelGroup' },
            React.createElement(
                'nav', null,
                React.createElement(
                    'ul', null,
                    ...this.tabSet
                )
            ),
            ...this.panelSet
        );
    }

    openTab(name) {
        this.setState({ activeTab: name });
        this.findDOMNode(this.refs.activePanel).focus();
    }
}
```

`panels` property of `PanelGroup` instance must contain array with objects. Every object there declares important data about panels:

- **`name`** - identifier of panel used by controller script;
- **`class`** - panel's class.

Don't forget to set property `activeTab` to name of needed tab.

Clarification

When tab is down, needed panel is getting class name `active` on DOM element (means that it gonna be visible) and it's focused now.

Section 23.5: Example view with `PanelGroup`s

```
import React from 'react';
import Pane from './components/Pane.js';
import Panel from './components/Panel.js';
import PanelGroup from './components/PanelGroup.js';

class MainView extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        return React.createElement(
            'main', null,
            React.createElement(
                Pane, { id: 'common' },
                React.createElement(
                    PanelGroup, {
```

```
                        panels: [
                            {
                                name: 'console',
                                panelClass: ConsolePanel
                            },
                            {
                                name: 'figures',
                                panelClass: FiguresPanel
                            }
                        ],
                        activeTab: 'console'
                    }
                )
            ),
            React.createElement(
                Pane, { id: 'side' },
                React.createElement(
                    PanelGroup, {
                        panels: [
                            {
                                name: 'properties',
                                panelClass: PropertiesPanel
                            }
                        ],
                        activeTab: 'properties'
                    }
                )
            )
        );
    }
}

class ConsolePanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Console';
    }
}

class FiguresPanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Figures';
    }
}

class PropertiesPanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Properties';
    }
}
```

Chapter 24: Using ReactJS in Flux way

It comes very handy to use Flux approach, when your application with ReactJS on frontend is planned to grow, because of limited structures and a little bit of new code to make state changes in runtime more easing.

Section 24.1: Data Flow

> This is outline of comprehensive Overview.

Flux pattern assumes the use of unidirectional data flow.

1. **Action** — simple object describing action `type` and other input data.
2. **Dispatcher** — single action receiver and callbacks controller. Imagine it is central hub of your application.
3. **Store** — contains the application state and logic. It registers callback in dispatcher and emits event to view when change to the data layer has occurred.
4. **View** — React component that receives change event and data from store. It causes re-rendering when something is changed.

> As of Flux data flow, views may also **create actions** and pass them to dispatcher for user interactions.

Reverted

To make it more clearer, we can start from the end.

- Different React components (*views*) get data from different stores about made changes.

> Few components may be called **controller-views**, cause they provide the glue code to get the data from the stores and to pass data down the chain of their descendants. Controller-views represent any significant section of the page.

- *Stores* can be remarked as callbacks that compare action type and other input data for business logic of your application.
- *Dispatcher* is common actions receiver and callbacks container.
- *Actions* are nothing than simple objects with required `type` property.

> Formerly, you'll want to use constants for action types and helper methods (called **action creators**).

Chapter 25: React, Webpack & TypeScript installation

Section 25.1: webpack.config.js

```
module.exports = {
    entry: './src/index',
    output: {
        path: __dirname + '/build',
        filename: 'bundle.js'
    },
    module: {
        rules: [{
            test: /\.tsx?$/,
            loader: 'ts-loader',
            exclude: /node_modules/
        }]
    },
    resolve: {
        extensions: ['.ts', '.tsx']
    }
};
```

The main components are (in addition to the standard `entry`, `output` and other webpack properties):

The loader

For this you need to create a rule that tests for the `.ts` and `.tsx` file extensions, specify `ts-loader` as the loader.

Resolve TS extensions

You also need to add the `.ts` and `.tsx` extensions in the `resolve` array, or webpack won't see them.

Section 25.2: tsconfig.json

This is a minimal tsconfig to get you up and running.

```
{
    "include": [
        "src/*"
    ],
    "compilerOptions": {
        "target": "es5",
        "jsx": "react",
        "allowSyntheticDefaultImports": true
    }
}
```

Let's go through the properties one by one:

`include`

This is an array of source code. Here we have only one entry, `src/*`, which specifies that everything in the `src` directory is to be included in compilation.

`compilerOptions.target`

Specifies that we want to compile to ES5 target

compilerOptions.jsx

Setting this to **true** will make TypeScript automatically compile your tsx syntax from ** ** to `React.createElement("div")`.

compilerOptions.allowSyntheticDefaultImports

Handy property which will allow you to import node modules as if they are ES6 modules, so instead of doing

```
import * as React from 'react'
const { Component } = React
```

you can just do

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

without any errors telling you that React has no default export.

Section 25.3: My First Component

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';


interface AppProps {
    name: string;
}
interface AppState {
    words: string[];
}

class App extends Component<AppProps, AppState> {
    constructor() {
        super();
        this.state = {
            words: ['foo', 'bar']
        };
    }

    render() {
        const { name } = this.props;
        return (<h1>Hello {name}!</h1>);
    }
}

const root = document.getElementById('root');
ReactDOM.render(<App name="Foo Bar" />, root);
```

When using TypeScript with React, once you've downloaded the React DefinitelyTyped type definitions (`npm install --save @types/react`), every component will require you to add type annotations.

You do this like so:

```
class App extends Component<AppProps, AppState> { }
```

where `AppProps` and `AppState` are interfaces (or type aliases) for your components' props and state respectively.

Chapter 26: How and why to use keys in React

Whenever you are rendering a list of React components, each component needs to have a key attribute. The key can be any value, but it does need to be unique to that list.

When React has to render changes on a list of items, React just iterates over both lists of children at the same time and generates a mutation whenever there's a difference. If there are no keys set for the children, React scans each child. Otherwise, React compares the keys to know which were added or removed from the list

Section 26.1: Basic Example

For a class-less React component:

```
function SomeComponent(props){

    const ITEMS = ['cat', 'dog', 'rat']
    function getItemsList(){
        return ITEMS.map(item => <li key={item}>{item}</i>);
    }

    return (
        <ul>
            {getItemsList()}
        </ul>
    );
}
```

For this example, the above component resolves to:

```
<ul>
    <li key='cat'>cat</li>
    <li key='dog'>dog</li>
    <li key='rat'>rat</li>
<ul>
```

Chapter 27: Keys in react

Keys in react are used to identify a list of DOM elements from the same hierarchy internally.

So if you are iterating over an array to show a list of li elements, each of the li elements needs a unique identifier specified by the key property. This usually can be the id of your database item or the index of the array.

Section 27.1: Using the id of an element

Here we are having a list of todo items that is passed to the props of our component.

Each todo item has a text and id property. Imagine that the id property comes from a backend datastore and is a unique numeric value:

```
todos = [
  {
    id: 1,
    text: 'value 1'
  },
  {
    id: 2,
    text: 'value 2'
  },
  {
    id: 3,
    text: 'value 3'
  },
  {
    id: 4,
    text: 'value 4'
  },
];
```

We set the key attribute of each iterated list element to `todo-${todo.id}` so that react can identify it internally:

```
render() {
  const { todos } = this.props;
  return (
    <ul>
      { todos.map((todo) =>
        <li key={ `todo-${todo.id}` }>
          { todo.text }
        </li>
      }
    </ul>
  );
}
```

Section 27.2: Using the array index

If you don't have unique database ids at hand, you could also use the numeric index of your array like this:

```
render() {
  const { todos } = this.props;
  return (
    <ul>
      { todos.map((todo, index) =>
```

```
        <li key={ `todo-${index}` }>
          { todo.text }
        </li>
      }
    </ul>
  );
}
```

Chapter 28: Higher Order Components

Higher Order Components ("HOC" in short) is a react application design pattern that is used to enhance components with reusable code. They enable to add functionality and behaviors to existing component classes.

A HOC is a pure javascript function that accepts a component as it's argument and returns a new component with the extended functionality.

Section 28.1: Higher Order Component that checks for authentication

Let's say we have a component that should only be displayed if the user is logged in.

So we create a HOC that checks for the authentication on each render():

AuthenticatedComponent.js

```
import React from "react";

export function requireAuthentication(Component) {
    return class AuthenticatedComponent extends React.Component {

        /**
         * Check if the user is authenticated, this.props.isAuthenticated
         * has to be set from your application logic (or use react-redux to retrieve it from global
state).
         */
        isAuthenticated() {
            return this.props.isAuthenticated;
        }

        /**
         * Render
         */
        render() {
            const loginErrorMessage = (
                <div>
                    Please <a href="/login">login</a> in order to view this part of the
application.
                </div>
            );

            return (
                <div>
                    { this.isAuthenticated === true ? <Component {...this.props} /> :
loginErrorMessage }
                </div>
            );
        }
    };
}

export default requireAuthentication;
```

We then just use this Higher Order Component in our components that should be hidden from anonymous users:

MyPrivateComponent.js

```
import React from "react";
import {requireAuthentication} from "./AuthenticatedComponent";

export class MyPrivateComponent extends React.Component {
    /**
     * Render
     */
    render() {
        return (
            <div>
                My secret search, that is only viewable by authenticated users.
            </div>
        );
    }
}

// Now wrap MyPrivateComponent with the requireAuthentication function
export default requireAuthentication(MyPrivateComponent);
```

This example is described in more detail here.

Section 28.2: Simple Higher Order Component

Let's say we want to console.log each time the component mounts:

hocLogger.js

```
export default function hocLogger(Component) {
  return class extends React.Component {
    componentDidMount() {
      console.log('Hey, we are mounted!');
    }
    render() {
      return <Component {...this.props} />;
    }
  }
}
```

Use this HOC in your code:

MyLoggedComponent.js

```
import React from "react";
import {hocLogger} from "./hocLogger";

export class MyLoggedComponent extends React.Component {
    render() {
        return (
            <div>
                This component gets logged to console on each mount.
            </div>
        );
    }
}

// Now wrap MyLoggedComponent with the hocLogger function
export default hocLogger(MyLoggedComponent);
```

Chapter 29: React with Redux

Redux has come to be the status quo for managing application-level state on the front-end these days, and those who work on "large-scale applications" often swear by it. This topic covers why and how you should use the state management library, Redux, in your React applications.

Section 29.1: Using Connect

Create a Redux store with *createStore*.

```
import { createStore } from 'redux'
import todoApp from './reducers'
let store = createStore(todoApp, { inistialStateVariable: "derp"})
```

Use *connect* to connect component to Redux store and pull props from store to component.

```
import { connect } from 'react-redux'

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

Define actions that allow your components to send messages to the Redux store.

```
/*
 * action types
 */

export const ADD_TODO = 'ADD_TODO'

export function addTodo(text) {
  return { type: ADD_TODO, text }
}
```

Handle these messages and create a new state for the store in reducer functions.

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

Appendix A: Installation

Section A.1: Simple setup

Setting up the folders

This example assumes code to be in `src/` and the output to be put into `out/`. As such the folder structure should look something like

```
example/
|-- src/
|   |-- index.js
|   `-- ...
|-- out/
`-- package.json
```

Setting up the packages

Assuming a setup npm environment, we first need to setup babel in order to transpile the React code into es5 compliant code.

```
$npm install --save-dev babel-core babel-loader babel-preset-es2015 babel-preset-react
```

The above command will instruct npm to install the core babel libraries as well as the loader module for use with webpack. We also install the es6 and react presets for babel to understand JSX and es6 module code. (More information about the presets can be found here Babel presets)

```
$npm i -D webpack
```

This command will install webpack as a development dependency. (**i** is the shorthand for install and **-D** the shorthand for --save-dev)

You might also want to install any additional webpack packages (such as additional loaders or the webpack-dev-server extension)

Lastly we will need the actual react code

```
$npm i -D react react-dom
```

Setting up webpack

With the dependencies setup we will need a webpack.config.js file to tell webpack what to do

simple webpack.config.js:

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

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'out'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
```

```
        exclude: /(node_modules)/,
        loader: 'babel-loader',
        query: {
          presets: ['es2015', 'react']
        }
      }
    ]
  }
};
```

This file tells webpack to start with the index.js file (assumed to be in src/) and convert it into a single bundle.js file in the `out` directory.

The `module` block tells webpack to test all files encountered against the regular expression and if they match, will invoke the specified loader. (`babel-loader` in this case) Furthermore, the `exclude` regex tells webpack to ignore this special loader for all modules in the `node_modules` folder, this helps speed up the transpilation process. Lastly, the `query` option tells webpack what parameters to pass to babel and is used to pass along the presets we installed earlier.

Testing the setup

All that is left now is to create the `src/index.js` file and try packing the application

src/index.js:

```
'use strict'

import React from 'react'
import { render } from 'react-dom'

const App = () => {
  return <h1>Hello world!</h1>
}

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

This file would normally render a simple **`<h1>Hello world!</h1>`** Header into the html tag with the id 'app', but for now it should be enough to transpile the code once.

`$./node_modules/.bin/webpack` . Will execute the locally installed version of webpack (use `$webpack` if you installed webpack globally with -g)

This should create the file `out/bundle.js` with the transpiled code inside and concludes the example.

Section A.2: Using webpack-dev-server

Setup

After setting up a simple project to use webpack, babel and react issuing `$npm i -g webpack-dev-server` will install the development http server for quicker development.

Modifying webpack.config.js

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

```
module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'out'),
    publicPath: '/public/',
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: 'babel',
        query: {
          presets: ['es2015', 'react']
        }
      }
    ]
  },
  devServer: {
    contentBase: path.resolve(__dirname, 'public'),
    hot: true
  }
};
```

The modifications are in

- `output.publicPath` which sets up a path to have our bundle be served from (see Webpack configuration files for more info)
- `devServer`
    - `contentBase` the base path to serve static files from (for example `index.html`)
    - `hot` sets the webpack-dev-server to hot reload when changes get made to files on disk

And finally we just need a simple index.html to test our app in.

index.html:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>React Sandbox</title>
  </head>
  <body>

    <div id="app" />

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

With this setup running `$webpack-dev-server` should start a local http server on port 8080 and upon connecting should render a page containing a `<h1>Hello world!</h1>`.