Introduction to React.JS

Learner Manual

Table of Contents

Welcome to the Academic Institute of Excellence

We pride ourselves on our innovative approach to quality education, excellent service delivery, and a modern outlook to technology. AIE creates, develops, supports, and presents innovative progams to create employable, productive, emotionally intelligent, and skilled learners.

Our learning paths are designed by analysing global skills demands and by providing relevant programmes to meet these demands. Our facilitators are qualified subject matter experts with both practical industry experience and tertiary education experience.

The AIE is a revolutionised family of brands, people, and students. We present our programs in a smarter, more efficient, and cost-effective way by utilising innovation and technology, which results in an expanded reach of our learning interventions. We strive towards excellence and on empowering future generations to become problem solvers, critical thinkers and innovators to create the leaders of tomorrow.

We cultivate excellence.

Our VISION

To deliver demand-driven education, built upon the principal of quality education through innovation and technology.

AIE Academic Institute of Excellence

the cad corporation
CK No: 2002/013213/23

learnfast

GREENSIDE DESIGN CENTER

Getting Started with Reactjs

Module 1

Module Overview

In this module we will look at the open-source ReactJS

Topics covered in this module:

- What is ReactJS
- Installation or Setup
- Hello World with Stateless Functions
- Absolute Basics of Creating Reusable Components
- Create React App
- Hello World
- Hello World Component

What is ReactJS?

ReactJS is an open-source, component based front end library responsible only for the view layer of the application. It is maintained by Facebook.

ReactJS uses virtual DOM based mechanism to fill in data (views) in HTML DOM. The virtual DOM works fast owning to the fact that it only changes individual DOM elements instead of reloading complete DOM every time

A React application is made up of multiple components, each responsible for outputting a small, reusable piece of HTML. Components can be nested within other components to allow complex applications to be built out of simple building blocks. A component may also maintain internal state – for example, a TabList component may store a variable corresponding to the currently open tab.

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.

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.

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.

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.

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 build
```

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
```

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>
```

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
```

Take NOTE!

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

Codepen Link to Example.

Components

Module 2

Module Overview

In this module we will look at various components as well as using ReactJS with Typescript

Topics covered in this module:

- Creating Components
- Basic Component
- Nesting Component
- Props
- Components states – Dynamic user-interface
- Variations of Stateless Functional Components
- SetState pitfalls
- ReactJS component written in Typescript
- Installation and Setup
- Stateless React Components in Typescript
- Stateless and property-less Components

Components

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.

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>
```

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.

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.

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.

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.

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.

Using ReactJS with TypeScript

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.

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"
},
```

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>;
```

Take NOTE!

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

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.

State in React and Props

Module 3

Module Overview

In this module we will look at State in React components that are important to manage data in your application. We will also look at props that are used to pass data and methods from a parent component to a child component.

Topics covered in this module:

- Basic State
- Common Antipattern
- setState ()
- State, Events And Managed Controls
- Props in React – Introduction
- Default props
- PropTypes
- Passing down props using spread operator
- Props.children and component composition
- Detecting the type of Children components

State in React

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 s79oome 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>
    );
  }
}
```

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.

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'));
```

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

Props in React

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.

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.

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,
...
```

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'
```

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>
    );
}
```

Take NOTE!

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.

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.

React Component Lifecycle

Module 4

Module Overview

In this module we will look at Lifecycle methods that are used to run code and interact with your component at different points in the components life.

Topics covered in this module:

- Component Creation
- Component Removal
- Component Update
- Lifecycle method call in different states
- React Component Container

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.

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
  };
}
```

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>
    );
  }
}
```

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){}
```

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

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)
}
```

Forms and User Input

Module 5

Module Overview

In this module we will look at Forms and User Input

Topics covered in this module:

- Controlled Components
- Uncontrolled Components

Forms and User Input

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.

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.

React Boilerplate

Module 6

Module Overview

In this module we will look at React Boilerplate. This module will show you how to set up the environment for Reactjs + Webpack + Bable.

Topics covered in this module:

- React-starter project
- Setting up the project

React Boilerplate [React + Babel + Webpack]

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.

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

Using ReactJS with jQuery

Module 7

Module Overview

In this module we will look at using Reactjs with jQuery

Topics covered in this module:

- Using ReactJS with jQuery

Using ReactJS with jQuery

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
```

```
: </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"></i>
                </div>
            </div>
        );
    }
}
```

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

React Routing/Communication with Components

Module 8

Module Overview

In this module we will look at React routing as well as React Routing Async

Topics covered in this module:

- Example Routes.js file, followed by use of Router Link in component
- React Routing Async

React Routing

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>
  );
}
```

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>
);
```

Communicate Between Components

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

Credits and References

Module 9

About this Book

AIE has reviewed and modified the content of this book for our training purposes from the Javascript® Notes for Professionals book.

This book is compiled from Stack Overflow Documentation and the content is written by the beautiful people at Stack Overflow.

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content.

Ahmad Chapter 5
akashrajkn Chapter 2
Alex Young Chapters 1 and 4
Alexander Chapter 4
Alexg2195 Chapter 6
Anuj Chapters 1, 5 and 6
Bart Riordan Chapters 1, 2
Bond Chapter 2
Brad Colthurst Chapter 4
Brandon Roberts Chapter 2
Daksh Gupta Chapter 1
Danillo Corvalan Chapter 5
F. Kauder Chapter 6
Faktor 10 Chapter 5
Gianluca Esposito Chapter 1
goldbullet Chapter 2
GordyD Chapter 2
hmnzr Chapter 2
Inanc Gumus Chapter 1
ivarni Chapter 2
Jack7 Chapter 5
Jagadish Upadhyay Chapter 5
John Ruddell Chapters 3 and 6
jolyonruss Chapters 1 and 2
Jon Chan Chapters 1 and 2
jonathangoodman Chapter 2
JordanHendrix Chapters 1 and 2
juandemarco Chapter 1
justabuzz Chapter 2
Kousha Chapters 2 and 4
Leone Chapter 3
Maayan Glikser Chapter 2
MaxPRafferty Chapters 1 and 5
Md Sifatul Islam Chapter 1
Md. Nahiduzzaman Rose Chapter 1
Michael Peyper Chapters 2
Mihir Chapter 8
MMachinegun Chapter 1
m_callens Chapter 2
Nick Bartlett Chapter 1
orvi Chapter 1
parlad neupane Chapter 8
QoP Chapters 4, 5 and 6
Rajab Shakirov Chapter 3
rossipedia Chapter 1
Salman Saleem Chapter 6
Sergii Bishyr Chapter 5
Shabin Hashim Chapter 1
Simplans Chapter 1
sjmarshy Chapter 2
skav Chapters 4 and 6
Sunny R Gupta Chapters 1
Timo Chapters 1, 4, 6 and 7
user2314737 Chapter 1
Vivian Chapter 6
Vlad Bezden Chapter 2
WitVault Chapters 5 and 6
Zakaria Ridouh Chapter 2