**Table of Content**

[What is React?](#_sygih58pnmvi) 7

[Why React?](#_8f1z8tnee4lx) 7

[**Pros and cons of React JS**](#_5l6wxknm9t2x) **7**

[Pros](#_xdo8lir0ear7) 7

[Cons](#_o5trzu6dj47) 7

[**Why did Facebook build React?**](#_xa6wdwskli83) **8**

[**Comparison of JavaScript Frameworks**](#_ybmdop2j90z) **9**

[**Setting up Your Environment**](#_ndw3ouze4n20) **9**

[**Hello React**](#_cxmnywf2tczx) **10**

[Setting up a Web Server](#_kt5k7k7nicn4) 10

[Hello React](#_p4c15qz1ifhc) 10

[Setting up Babel](#_cbz7471cx88j) 12

[Exploring JSX](#_azs3rsqrtlr6) 12

[JSX Expressions](#_97s45nxvlmfk) 12

[Conditional Rendering in JSX](#_gzp2gb7oxb50) 12

[ES6 Aside const and let](#_qapazkkoghuk) 13

[ES6 Aside Arrow Functions](#_s944y8jjuxf1) 13

[Events and Attributes](#_r24tlolqtx5) 15

[Manual Data Binding](#_mrbizcz9gzji) 15

[Virtual DOM:](#_podwiexaxrks) 16

[Forms and Inputs](#_p3hjas2l87co) 18

[Arrays in JSX](#_zg8v6am7nvz8) 19

[**React Components**](#_o8lt2miwy942) **20**

[ES6 Classes](#_5ttisjuqhenk) 20

[Creating a React Component](#_r5jau0nvfkr5) 21

[Nesting Components](#_ead2fr1bza6i) 21

[Component Props](#_r99980jiio5p) 22

[Events Methods](#_l6bw1eh1uzka) 23

[Method Binding](#_hun3bt5ycj9j) 24

[Two way binding using props](#_mmcwee5fv1dv) 26

[What Is Component State](#_uus3capidx17) 27

[Alternative setState Syntax](#_qppox7xw522h) 28

[How to manipulate parent state object from child component?](#_zhqzkyoxex9c) 29

[How to pass data upstream?](#_wjlpk8v3b46y) 30

[Stateless Functional Components](#_nql0q7mbp3h7) 32

[Convert Class based component to Stateless Functional Component](#_7idxvutjqlxf) 33

[Default Prop Values](#_ya1nhx73ybnt) 33

[Component Types](#_dqktyq667ybq) 34

[Presentational Component](#_lb0s8qod31ri) 34

[Container Component](#_74mz1lb9sk6s) 34

[Class Components](#_59leqgh22ts8) 34

[Functional Components](#_o8wbpxcgasy6) 35

[**React Dev Tools**](#_y80d8mygfhcz) **35**

[**Lifecycle Methods**](#_8if80kgkl5eo) **35**

[Initialization](#_9i1rxt98spdn) 36

[Mounting](#_skqfxpqyytvw) 36

[Updation](#_w08uv386ns06) 36

[Unmounting](#_wtjllfjcrn5u) 37

[**Webpack**](#_3hnjuqmnyvlm) **37**

[Avoid Global Modules](#_z2rsjgl6oj5z) 38

[Installing Configuring Webpack](#_9s4878om3n1s) 38

[ES6 import export](#_hyoxd1alo6oj) 38

[Default Exports](#_kikpci15yd3) 39

[Importing npm Modules](#_9qzo1b17hmjn) 39

[Setting up Babel with Webpack](#_vkm6y0n2ojoz) 40

[One Component per File](#_dycdz2pvgqka) 40

[Source Maps with Webpack](#_ab5zcgolczbk) 41

[Webpack Dev Server](#_gv40781wf2b5) 41

[ES6 class properties](#_3z1lta94vmmb) 41

[**Using a Third-Party Component**](#_s4nw9imru6ly) **42**

[Passing Children to Component](#_rmlmm1c7lrku) 42

[Setting up React-Modal](#_w3lmub7qo1pt) 43

[**Styling React**](#_yq20xalxad1l) **45**

[Setting up Webpack with SCSS](#_jw4ji3qgonwv) 45

[Architecture and Header Styles](#_5hj9u735n2bs) 45

[Reset That](#_ifamuflzdfec) 46

[**React-Router**](#_47rk3qvzr8rm) **46**

[Server vs. Client Routing](#_d2415e48b4x4) 46

[Static Routing](#_rz2100kz5se4) 47

[Dynamic Routing](#_o4fuvej5a2fl) 47

[Responsive Routes](#_ergsnx1rgo4u) 47

[Installation](#_utp372rvxhkz) 47

[BrowserRouter](#_9hpium8pqi30) 49

[HashRouter](#_4gkv7hbksck6) 50

[<Link>](#_fpembq57dmf) 50

[<NavLink>](#_swa13m9ucjvm) 50

[Prompt](#_1d7v3nr54kcy) 51

[MemoryRouter](#_y1kzw2zceyzo) 52

[Redirect](#_rhfa5c6so14y) 52

[Route](#_glq14xq102w8) 52

[Router](#_4rrlvtqonrgo) 54

[StaticRouter](#_n8srrb2660h) 55

[Switch](#_x2bh25d15tlj) 55

[HashRouter vs BrowserRouter](#_9fp47cvh5858) 55

[History](#_ccxtpixyw84w) 56

[Setting up a 404](#_54aa82meq227) 56

[Linking Between Routes](#_u60uuwlrqps7) 57

[Query Strings and URL Parameters](#_jv3u7tc828g3) 58

[Nested Routes](#_bwbj2l8d2fm5) 59

[Index Routes](#_hv8pgqxi3m23) 61

[Authenticated routes in React Router 4](#_89m9th884y12) 62

[Lazy Loading Routes in React](#_m88l709bgdux) 62

[**Accessing DOM Elements in React**](#_vuk1n9nv3b1v) **62**

[Refs](#_5dmt3hd9whvk) 62

[When to Use Refs](#_twhvnip0xjzn) 63

[Creating Refs](#_50y24kezrkuq) 63

[Accessing Refs](#_6wpn3u873u5u) 63

[Adding a Ref to a DOM Element](#_g5g9k2sljz3p) 63

[Callback Refs](#_ro1m7j4kswxg) 64

[React Portal](#_l561z4dvsco1) 66

[**Immutability in JavaScript**](#_twwjutz767l3) **67**

[Advantages of using Immutability](#_e2gp84tut5n1) 67

[How we can achieve immutability in JavaScript?](#_r4otorb6lu2d) 68

[How Immutable Data Structures (E.g. Immutable.js) are Optimized](#_ix3otkinpx3w) 69

[**Function as Child Components**](#_qvcerem6kvua) **70**

[**React's Children API**](#_j8aspcnyrex5) **70**

[React.Children](#_830n7qwdfgcz) 71

[React.Children.map](#_wrh4awvyb646) 71

[React.Children.forEach](#_r2hzrm8q4xle) 71

[React.Children.count](#_s5h2ghxeqrcy) 71

[React.Children.only](#_ma18jvwbu9ah) 71

[React.Children.toArray](#_6su5oo217oeu) 71

[**Hot Module Replacement in Create-React-App**](#_k8gtr6gf14ud) **72**

[**Differentiate between Real DOM and Virtual DOM.**](#_4t0fsdyn1yg2) **72**

[**How is React different from Angular?**](#_4m132k4895yv) **72**

[**What’s the difference between client-side rendering and server-side rendering?**](#_ey58pstilxyk) **73**

[**Server Side Rendering**](#_89gh891l1tei) **73**

[The Benefits of Server-Side Rendering](#_y8a7cj9zim2u) 73

[Server Side Rendering: Data Fetching & Routing](#_ixczdesa2g5n) 74

[**Synthetic events in React**](#_7nwawjnaucqh) **74**

[Event Pooling](#_olqoai3wrrvh) 74

[**How ReactJS works**](#_lssfm1r94xxh) **74**

[Model-View-Controller](#_sfokq1aw7ses) 74

[Reconciliation](#_jc1y8x6gupjo) 74

[Diffing Algorithms](#_j0wkq91f0nrd) 75

[Browser DOM Update](#_giaiq0b35sga) 75

[shouldComponentUpdate](#_4ds6w6d7gh8q) 75

[**applyMiddleware**](#_4luqxsclovmb) **76**

[**Create-react-app**](#_5cfxfdmpmtxt) **76**

[**Controlled Forms**](#_ut08te4scj8r) **76**

[**Uncontrolled Forms**](#_nap8cz21gu57) **76**

[**React-Redux-Form**](#_rml9c3ript7) **77**

[**Cross-Fetch**](#_v4mm12f8ntt0) **77**

[**How to make AJAX calls in React**](#_r0b1xrjaei9e) **77**

[How can I make an AJAX call?](#_slgn4fbuct8) 77

[Where in the component lifecycle should I make an AJAX call?](#_rdnfj9z8tz2f) 77

[**React Fiber**](#_qk98rtiz4038) **79**

[**React Polyfill**](#_qswol9hfhnr9) **79**

[**Class Properties**](#_wlzqugsnysn2) **80**

[**Advanced React**](#_a1yjpr7pqmnq) **81**

[Contexts](#_xfzbk4qbs74d) 81

[Error Boundaries](#_cx1l5mfogus9) 87

[Forwarding Refs](#_7sf4d0xmpsoh) 88

[Fragments](#_2lxig5d5c0xv) 89

[Render Props](#_h5o0a27itdbm) 90

[Using Props Other Than render](#_7o78ny2urg9) 90

[Static Type Checking](#_3hx01d37x4pd) 90

[Flow](#_f42d0fprex9l) 90

[Strict Mode](#_cbnilig10ksi) 90

[Typechecking With PropTypes](#_e1t79jkpit30) 91

[Default Prop Values](#_glq406ugmk8a) 92

[**The Model View Controller Framework**](#_gggcf52w3puc) **93**

[**Flux**](#_lko7pspp9c80) **93**

[Flux](#_av59nvsjnu2r) 93

[Flux Store](#_vvq7xuf7o302) 94

[Flux Store Events](#_p0l6ge84n5mh) 95

[The Flux Dispatcher](#_sot1wb8vzweb) 97

[Flux Actions](#_tb6s4jah0uev) 99

[React & Flux Memory Leaks](#_5gihycbxti2e) 100

[**Redux**](#_t3vbpyq01649) **100**

[Why Do We Need Something Like Redux](#_775hu725l37o) 101

[What is Redux?](#_wus802nzu66v) 101

[Redux Concepts](#_dz11xle96x0p) 102

[Redux Store](#_jbfg89rckhuq) 102

[Understanding Redux as a CQRS system](#_ps14epnidyaq) 103

[Setting up Redux](#_39zqfwmfiwfk) 103

[Subscribing and Dynamic Actions](#_fdr012ef4ezb) 105

[ES6 Object Destructuring](#_hcoqu53e8ws) 106

[ES6 Array Destructuring](#_6xutqynnhmr5) 107

[Refactoring and Organizing](#_2hk1x9k46t4r) 107

[Redux Actions](#_ee6le0d85nu2) 107

[Action Creators](#_kffejjqo28ov) 107

[Reducers](#_76vzl8gybrl6) 108

[Working with Multiple Reducers](#_ci2v6ggljkh7) 109

[ES6 Spread Operator in Reducers](#_bxluubj4jbxz) 110

[Spreading Objects](#_30s8bev8aq0h) 112

[Redux Overview](#_vpq868h5lmzi) 114

[Difference between redux and flux](#_608hyxcuye6q) 114

[**Redux Middleware and Redux Thunk**](#_ub819ztxakgp) **114**

[Redux Middleware](#_js7fhexngp6y) 114

[Redux Thunk](#_h19i7r6ugzez) 115

[**React with Redux**](#_h9ky825qmsmu) **116**

[The Higher Order Component](#_isgz6pxv36g8) 116

[Connecting Store and Component with React-Redux](#_uun8tp5lz68r) 117

[Controlled Inputs for Filters](#_o9q8sfn8fwth) 118

[Dropdown for Picking SortBy](#_3jl5i1akki0g) 119

[Creating Expense AddEdit Form](#_p9gl005wuum7) 119

[Setting up a Date Picker](#_egfj0ls8qnys) 121

[Redux Dev Tools](#_8j8kllxfn2am) 122

[**FluxThis**](#_ob1ackjrwlsf) **122**

[High Level Objectives](#_ntve35akbha1) 123

[Creating an ImmutableStore](#_o0qhqa1hmuj2) 123

[The init Method](#_a0bd44wz48cc) 123

[The public Methods](#_tzwmayrbwmvb) 124

[The private Methods](#_s8t7fwbl7gce) 124

[Using bindActions](#_hbhpenbbpiak) 125

[Creating a Controller View](#_d1mv5c5nqfs9) 125

[Creating an ActionCreator](#_kdxxbs2raatt) 127

[**React.js Questions and Answers**](#_1brh2v4710cf) **129**

### What is React?

* React JS is a JavaScript library for building UI specially single page application.
* Component based
* Very fast (because of virtual DOM)
* Created by facebook

### Why React?

React was built to solve one problem: building large applications with data that changes over time.

**Conceived at Facebook**

Heavily used on products made by Facebook and Instagram.

Built to simplify the process of building complex UIs.

## Pros and cons of React JS

### Pros

1. easy to know how a component is rendered, you just look at the render function.
2. JSX makes it easy to read the code of your components. It is also really easy to see the layout, or how components are plugged/combined with each other.
3. you can render React on the server-side.
4. it is easy to test, and you can also integrate some tools like jest.
5. it ensures readability and makes maintainability easier.
6. you can use React with any framework (Backbone.js, Angular.js) as it is only a view layer.

### Cons

1. It is a UI library only. It is only a view layer, you have to pull modules for Ajax requests, events and so on. Some people get surprised by that.
2. the library itself is pretty large.
3. You DON'T GET any of the following:
   1. An event system (other than vanilla DOM events)
   2. Any AJAX capabilities whatsoever
   3. Any form of a data layer
   4. Promises
   5. Any application framework
4. Flux and data flow are complex patterns for beginners

## Why did Facebook build React?

**React isn’t an MVC framework.**

React is a library for building composable user interfaces. It encourages the creation of reusable UI components which present data that changes over time.

**React doesn’t use templates.**

Traditionally, web application UIs are built using templates or HTML directives. These templates dictate the full set of abstractions that you are allowed to use to build your UI.

React approaches building user interfaces differently by breaking them into components. This means React uses a real, full featured programming language to render views, which we see as an advantage over templates for a few reasons:

* JavaScript is a flexible, powerful programming language with the ability to build abstractions. This is incredibly important in large applications.
* By unifying your markup with its corresponding view logic, React can actually make views easier to extend and maintain.

**Reactive updates are dead simple**

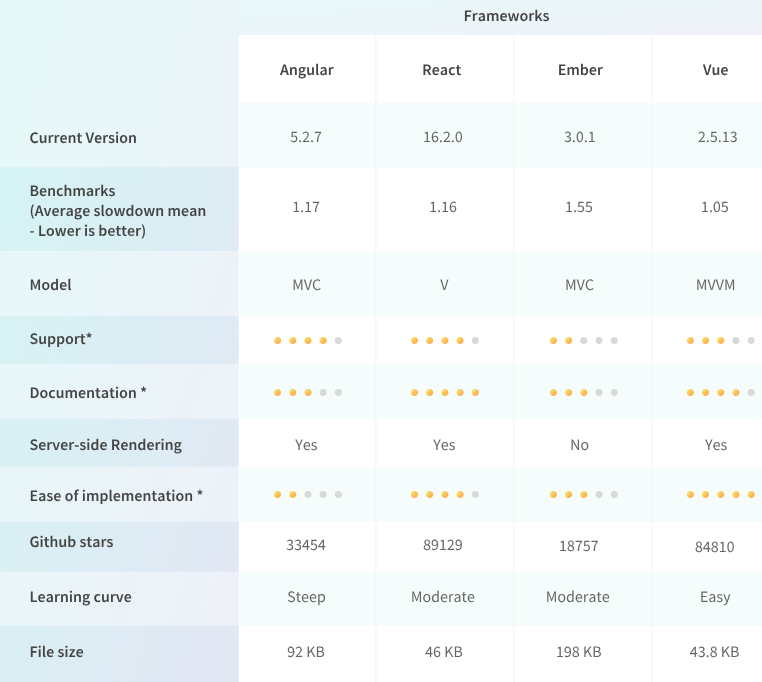
React really shines when your data changes over time.

In a traditional JavaScript application, you need to look at what data changed and imperatively make changes to the DOM to keep it up-to-date. Even AngularJS, which provides a declarative interface via directives and data binding requires a linking function to manually update DOM nodes.

React takes a different approach.

When your component is first initialized, the render method is called, generating a lightweight representation of your view. From that representation, a string of markup is produced, and injected into the document. When your data changes, the render method is called again. In order to perform updates as efficiently as possible, we diff the return value from the previous call to render with the new one, and generate a minimal set of changes to be applied to the DOM.

## Comparison of JavaScript Frameworks



Now that we have seen the good and the bad of the top JavaScript frameworks, the question arises - which is the best JavaScript framework? To put it in perspective, we firmly maintain that the choice of the framework depends on your preference, requirement, and the use-case.

Choosing the right JavaScript framework is never easy. However, it’s not about the number of features that the particular framework can provide. It’s about the actual functionality of the framework and how that functionality can be utilized within your own development project. Hence, choose JavaScript frameworks as per your project needs.

## Setting up Your Environment

1. Any text editor e.g Visual Studio Code
2. Install Node.js
3. Install yarn using below command

**npm install -g yarn**

## Hello React

### Setting up a Web Server

1. Create a new folder for your project
2. Open project folder in VS Code and dive into that folder in terminal
3. Create a folder named *public* in it then create a index.html file in *public* folder
4. Install live-server using below command

**yarn global add live-server** *or* **npm install -g live-server**

1. Now, run below command in project root folder

live-server public

Above command will serve index.html file in localhost server

### Hello React

Include ReactJS library in your html file

2 ways to set up React & Babel

1. Quick way
   1. Add React and Babel scripts to head of HTML
   2. <script crossorigin src="react.js"></script>  
      <script crossorigin src="react-dom.js"></script>

<script crossorigin src="browser.min.js"></script>

<script type=”text/babel” src="app.js"></script>

* 1. Use <script type=”text/babel” src=”index.js”>

1. Not so quick but better way
   1. Use npm to install React and Babel
   2. Use webpack to configure Babel
   3. Can also use the webpack-dev-server

**JSX:**

We write JSX code to create UI template.

const element = <h1>Hello, world!</h1>;

This funny tag syntax is neither a string nor HTML.

It is called JSX, and it is a syntax extension to JavaScript. We recommend using it with React to describe what the UI should look like. JSX may remind you of a template language, but it comes with the full power of JavaScript.

JSX produces React “elements”.

React **doesn’t require** using JSX, but most people find it helpful as a visual aid when working with UI inside the JavaScript code. It also allows React to show more useful error and warning messages.

But browser does not understand the JSX code so we need compiler.

We will be using **Babel** JavaScript compiler which will convert JSX code into Vanilla JS



[BABEL](https://babeljs.io/) is a javascript compiler, and importing its "browser.min.js" file, you are enabling it to "compile" the code inside "text/babel" script tags and execute it as vanilla javascript.

You write the code in ES6, JSX and any new concepts in JS, at the end it will convert into ES5 code that runs in the browser.

**Hello World example:**

**Index.html**

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

<script crossorigin src="https://unpkg.com/react@16/umd/react.development.js"></script>

<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>

<script src="./scripts/app.js"></script>

**App.js**

var template = React.createElement(

"h1",

{"id": "someid"},

"Hello World!!"

);

Using babel we can write code in JSX and babel convert it as above

var appRoot = document.getElementById('app');

ReactDOM.render(template, appRoot);

ReactDOM.render() takes two parameter, template and the element in which you want to render that template

### Setting up Babel

Babel does not compile automatically. We need to add plugins as per our requirements.

Use below command to install plugins

**npm install -g babel-cli@6.24.1**

Run **npm init,** that will create package.json file where you will install dependencies locally

Run below command to install dependencies

**npm install babel-core babel-loader babel-preset-es2015 babel-preset-react babel-preset-env --save-dev**

To compile JSX code use below command

**babel src/app.js --out-file=public/scripts/app.js --presets=env,react --watch**

### Exploring JSX

When working with JSX you can only have single root element

### JSX Expressions

var userName = "Ankit Chaurasia";

var userAge = 26;

var templateTwo = (

<div>

<h1>{userName + "!!"}</h1>

<p>Age: {userAge}</p>

<p>Location: Hyderabad</p>

</div>

);

var appRoot = document.getElementById('app');

ReactDOM.render(templateTwo, appRoot);

### Conditional Rendering in JSX

var user = {

userName: "Ankit Chaurasia",

userAge: 26,

location: "Hyderabad"

};

var templateTwo = (

<div>

<h1>{user.userName ? user.userName : 'Anonymous'}</h1>

{(user.userAge && user.userAge>= 18) && <p>Age: {user.userAge}</p>}

{getLocation(user.location)}

</div>

);

var appRoot = document.getElementById('app');

ReactDOM.render(templateTwo, appRoot);

### ES6 Aside const and let

Redefining of variable using ***let*** or ***const*** is not valid in ES6

var name = "Ankit";

var name = "Chaurasia";

console.log(name);

let name2 = "Ankit2";

let name2 = "Chaurasia2"; // This line will give error

console.log(name2);

const nameConst = "Ankit Const";

// const nameConst = "New Const"; // This line will give error

console.log('Const name: ' + nameConst);

### ES6 Aside Arrow Functions

// ES5 functions

const square = function(x){

return x\*x;

};

console.log(square(8));

// ES6 Arrow functions

const squareArrow = (x) => x\*x;

console.log("Arrow Function:" + squareArrow(8));

Argument object is no longer bound with arrow function

*this* keyword is no longer bound

const add = function (a, b) {

console.log(arguments);

return a + b;

};

console.log(add(3, 4));

// arguments in arrow function

const addArrow = (a, b) => {

// console.log(arguments); // ReferenceError: arguments is not defined

return a + b;

};

console.log(addArrow(3, 4));

const user = {

name: "Ankit",

cities: ['Jhansi', 'Hyderabad'],

printPlacesLived: function () {

// here this is bound to user object

console.log(this.name); // Ankit

console.log(this.cities); // ["Jhansi", "Hyderabad"]

var that = this;

this.cities.forEach(function (city) {

// here this is no longer bound to user because it is an anonymous function

// workaround: assigning this value in a variable and child function has access to its parent function

console.log(that.name + " has lived in " + city);

});

}

};

user.printPlacesLived();

// In Arrow function this is bound to the enclosing context

const user2 = {

name: "Ankit",

cities: ['Jhansi', 'Hyderabad'],

printPlacesLived: function () {

// here this is bound to user object

return this.cities.map((city)=> this.name + " has lived in " + city);

// this.cities.forEach( (city) => {

// // here this is no longer bound to user because it is an anonymous function

// // workaround: assigning this value in a variable and child function has access to its parent function

// console.log(this.name + " has lived in " + city);

// });

}

};

user2.printPlacesLived();

console.log(user2.printPlacesLived());

### Events and Attributes

Some html attributes work as like **id** but others got renamed

**Class** attribute renamed to **className** because class is a reserved keyword in JavaScript

let count = 0;

const someId = "myIdHere";

const addOne = () => {

console.log('addOne');

};

const templateTwo = (

<div>

<h1>Count: {count}</h1>

<button id={someId} className="button" onClick={addOne}>+1</button>

</div>

);

var appRoot = document.getElementById('app');

ReactDOM.render(templateTwo, appRoot);

### Manual Data Binding

JSX does not have built in data binding i.e if a variable is updating in JavaScript then variable’s updated value will not be reflected in view.

let count = 0;

const someId = "myIdHere";

const addOne = () => {

count++;

console.log('addOne', count);

};

const templateTwo = (

<div>

<h1>Count: {count}</h1>

<button id={someId} className="button" onClick={addOne}>+1</button>

</div>

);

var appRoot = document.getElementById('app');

ReactDOM.render(templateTwo, appRoot);

**Issue:** on clicking button count value is not updating in the view but updating in JavaScript

When we create JSX all the data that we use inside of it that happens at the time the code runs. In that case *count* is always be 0 because *count* was 0 when this code first ran

**Solution:**

let count = 0;

const addOne = () => {

count++;

renderCounterApp();

console.log('addOne', count);

};

var appRoot = document.getElementById('app');

**const renderCounterApp = () => {**

**const templateTwo = (<div>**

**<h1>Count: {count}</h1>**

**<button className="button" onClick={addOne}>+1</button>**

**</div>**

**);**

**ReactDOM.render(templateTwo, appRoot);**

**};**

renderCounterApp();

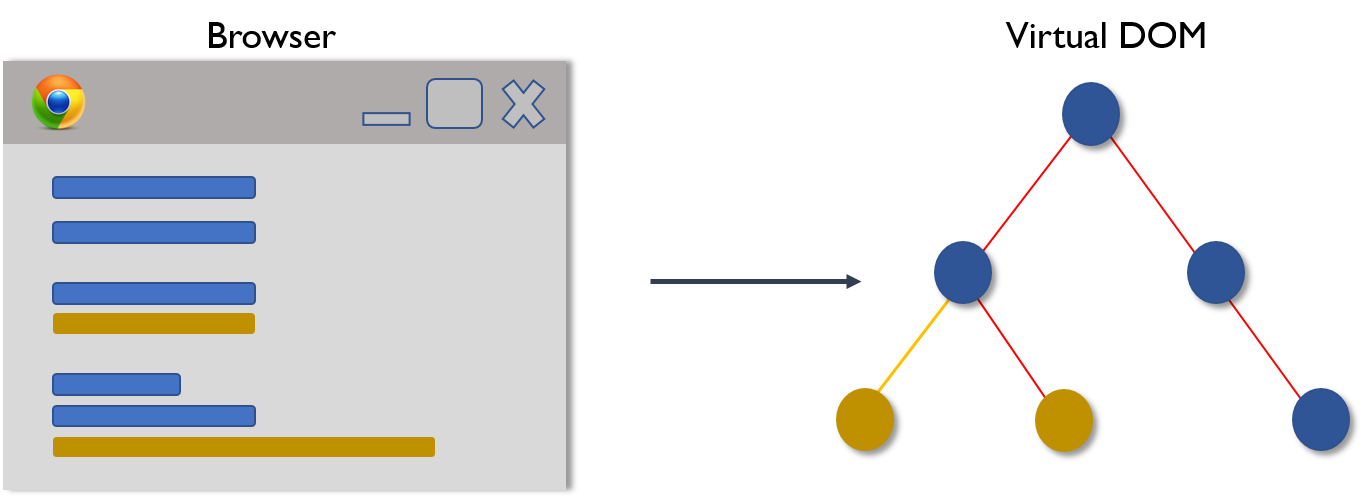
here , you might be thinking that React re rendering the entire DOM whenever we are updating count variable but it isn't so here comes React powerful feature Virtual DOM

#### Virtual DOM:

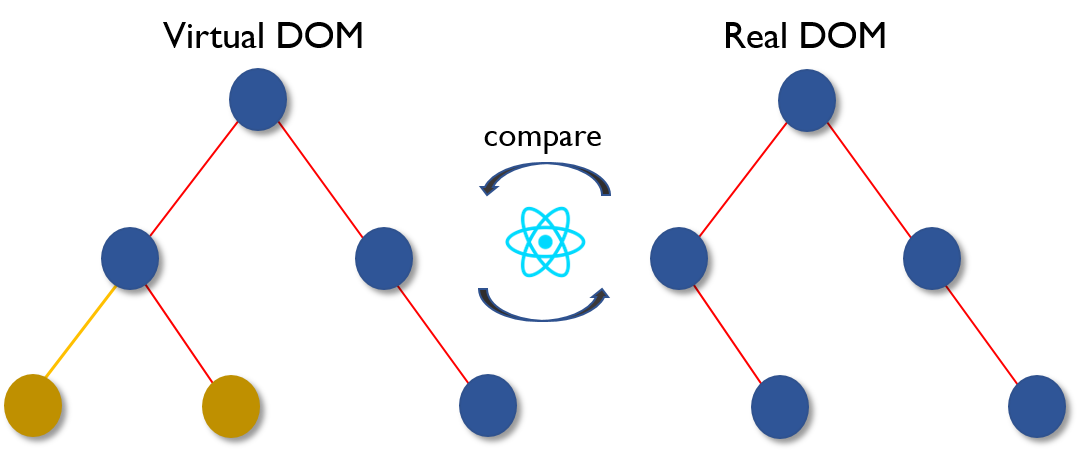
React internally uses Virtual DOM algorithm in JavaScript that to determine the minimal number changes that needs to be made in order to correctly render the new application.

So in the above code while calling *renderCounterApp()* function in *addOne()* function React updates only count variable in the DOM and improves performance of the page.

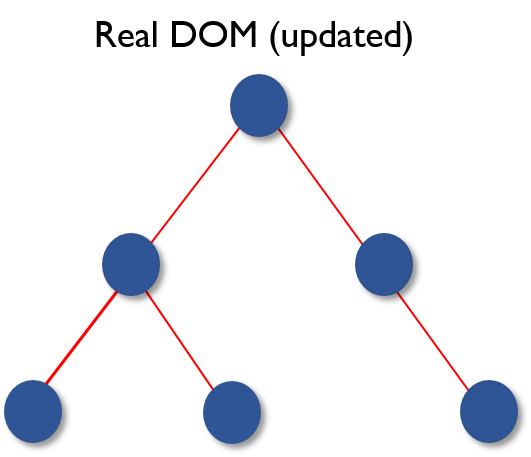
1. Whenever any underlying data changes, the entire UI is re-rendered in Virtual DOM representation.

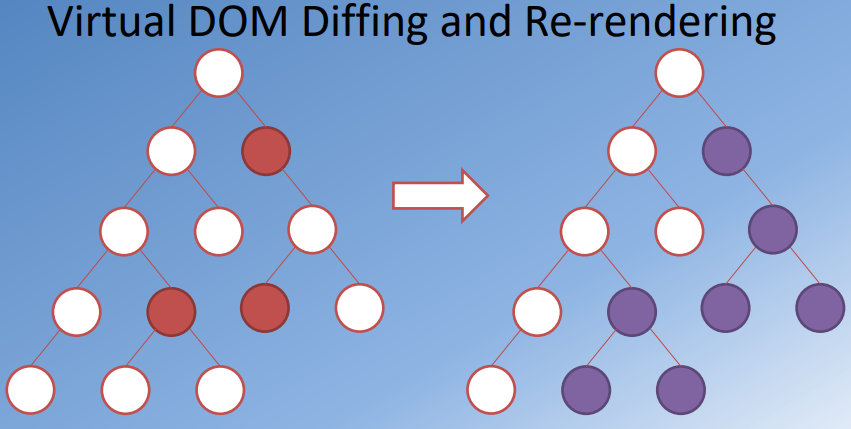


1. Then the difference between the previous DOM representation and the new one is calculated.



1. Once the calculations are done, the real DOM will be updated with only the things that have actually changed.





### Forms and Inputs

If we normally submit the form, then we gonna see full page refresh and forms data will be dumped in the url as the query string

To solve this issue, we can set up event handler in the form

var userName = "Ankit Chaurasia";

var userAge = 26;

const app = {

title: "Indecision App",

subtitle: "Put yourself in the hands of a computer",

options: ["One", 'Two']

};

const onFormSubmit = (e) => {

**e.preventDefault(); // Stop full page refresh on submitting form**

const option = e.target.elements.option.value;

if(option){

app.options.push(option);

e.target.elements.option.value = '';

rerenderForm();

}

};

const onRemoveAll = () => {

app.options = [];

rerenderForm();

};

var appRoot = document.getElementById('app');

const rerenderForm = () => {

var templateTwo = (

<div>

<h1>{app.title}</h1>

{app.subtitle && <p>{app.subtitle}</p>}

<p>{app.options.length > 0 ? 'Here are your options' : 'No options'}</p>

<p>Options Length: {app.options.length}</p>

<button onClick={onRemoveAll}>Remove All</button>

<ol>

<li>Item One</li>

<li>Item Two</li>

</ol>

<form onSubmit={onFormSubmit}>

<input type="text" name="option" />

<button>Add Option</button>

</form>

</div>

);

ReactDOM.render(templateTwo, appRoot);

};

rerenderForm();

### Arrays in JSX

JSX supports all types of values in arrays except null, undefined and boolean.

When JSX sees an array it renders each element side by side.

Null, undefined and boolean still be used in JSX array but they don’t show up in the page

{

[99,98,97, 'Ankit', null, undefined, true]

}

**Output**: 999897Ankit

**Note**: null, undefined and boolean don’t show up in the view

Above expression is equivalent to below line

{99}{98}{97}{'Ankit'}

If we use JSX inside JSX array, we get below error



To fix above issue, we need to add **key** attribute to each JSX value and key must be unique

{

[<p key='a'>a</p>, <p key='b'>b</p>, <p key='c'>c</p>]

}

To render items from array

var templateTwo = (

<div>

<ol>

{app.options.map((option)=> <li key={option}>{option}</li>) }

</ol>

</div>

);

ReactDOM.render(templateTwo, appRoot);

To disable a button based on some condition

<button onClick={onMakeDecision} disabled = {!app.options.length}>What should I do?</button>

## React Components

React components allows us to break our application into small reusable chunks.

Each little components has its own set of JSX that renders to the screen and handle events for those JSX elements and it allows us to create little self contained units

### ES6 Classes

class Person {

constructor(name = 'Anonymous', age = 0) {

this.name = name;

this.age = age;

}

getDescripton() {

return `${this.name} is ${this.age} year(s) old`;

}

}

class Student extends Person {

constructor(name, age, major = 'Information Technology') {

super(name, age); // Before *this* parent constructor should be called

this.major = major;

}

hasMajor() {

return !!this.major;

}

getDescripton() {

let descripton = super.getDescripton();

if (this.hasMajor()) {

descripton += ` Their major is ${this.major}`;

}

return descripton;

}

}

const me = new Person('Ankit Chaurasia', 25);

console.log(me.getDescripton());

### Creating a React Component

* React component is ES6 class.
* React Components require **render** method to be defined
* First character of React class name must be capital. If you use lowercase then it will not give any error but it will not display that component. This is how react differentiate HTML tag and component tags.
* To use component we just provide them in JSX. Looks like HTML tags

class Header extends React.Component {

render(){

return (

<div>

<h1>Indecision</h1>

<h2>Put your life in the hands of a computer.</h2>

</div>

);

}

}

const jsx = (

<div>

{ /\* Use Header Component as below \*/ }

<Header/>

</div>

);

ReactDOM.render(jsx, document.getElementById('app'));

### Nesting Components

class IndecisionApp extends React.Component {

render() {

return (

<div>

<Header />

<Options />

</div>

);

}

}

class Header extends React.Component {

render(){

return (

<div>

<h1>Indecision</h1>

<h2>Put your life in the hands of a computer.</h2>

</div>

);

}

}

class Options extends React.Component {

render() {

return (

<div>

Options component here

**<Option />**

</div>

);

}

}

class Option extends React.Component {

render() {

return (

<div>

Option Component

</div>

);

}

}

ReactDOM.render(**<IndecisionApp />**, document.getElementById('app'));

### Component Props

* Setting up Component props is same as HTML attribute.
* This is useful to send data from component instance to the component
* **this.props**  gives access to the property passed into the Component instance

class IndecisionApp extends React.Component {

render() {

const title = "Indecision App";

const subTitle = 'Put your life in the hands of a computer.';

return (

<div>

<Header **title={title} subTitle={subTitle}**/>

</div>

);

}

}

class Header extends React.Component {

render(){

console.log(this.props);

return (

<div>

<h1>{**this.props.title**}</h1>

<h2>{**this.props.subTitle**}</h2>

</div>

);

}

}

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

### Events Methods

class IndecisionApp extends React.Component {

render() {

const title = "Indecision App";

const subTitle = 'Put your life in the hands of a computer.';

const options = ['One', 'Two'];

return (

<div>

<Header title={title} subTitle={subTitle}/>

**<AddOption />**

</div>

);

}

}

...

class AddOption extends React.Component {

**handleAddOption**(e) {

e.preventDefault();

const option = e.target.elements.option.value.trim();

if(option){

console.log(option);

}

}

render() {

return (

<div>

<form **onSubmit={this.handleAddOption}**>

<input type='text' name='option' />

<button>Add Option</button>

</form>

</div>

);

}

}

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

### Method Binding

class IndecisionApp extends React.Component {

render() {

const options = ['One', 'Two'];

return (

<div>

**<Options options={options}/>**

</div>

);

}

}

class Options extends React.Component {

handleAllRemove() {

console.log('handleAllRemove' + **this.props.options**);

**// 'this' is no longer refers to our class instance**

**// To fix above issue, we can use bind method**

}

render() {

return (

<div>

<button onClick={**this.handleAllRemove.bind(this)**}>Remove All</button>

{'Options Length: ' + **this.props.options.length**} {

this.props.options.length && this.props.options.map((optionText) => <Option key={optionText} optionText={optionText} />)

}

</div>

);

}

}

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

To fix the issue we used **this.handleAllRemove.bind(this)**but this technique is little inefficient. It requires to rerun bind every time the component re renders

**Efficient way to bind method**

We can add binding in constructor so that binding runs once when the component gets initialized it is not need to rebound every single time the component is rendered

…

class Options extends React.Component {

constructor(props) {

super(props);

**this.handleRemoveAll = this.handleRemoveAll.bind(this);**

}

handleRemoveAll() {

console.log('handleAllRemove: ' + this.props.options);

}

render() {return ( <div>

{/\* <button onClick={this.handleRemoveAll.bind(this)}>Remove All</button> \*/}

<button onClick={this.handleRemoveAll}>Remove All</button>

{'Options Length: ' + this.props.options.length} {

this.props.options.length && this.props.options.map((optionText) => <Option key={optionText} optionText={optionText} />)}

</div>

);}}…

**Why do I have to .bind(this) for methods defined in React component class, but not in regular ES6 class?**

In the React docs it says the following

[React Component Class] Methods follow the same semantics as regular ES6 classes, meaning that they don't automatically bind this to the instance.

Render method is defined inside of the lifecycle methods and therefore do not need call .bind(this) on the render method.

#### Two way binding using props

class App extends React.Component {

state = {value : ''};

handleOnChange = (e) => {

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

}

render() {

return (

<div>

<TextInput handleOnChange={this.handleOnChange}/>

<div>{this.state.value}</div>

</div>

);

}

}

class TextInput extends React.Component {

render(){

return (

<input type='text' onChange={this.props.handleOnChange} />

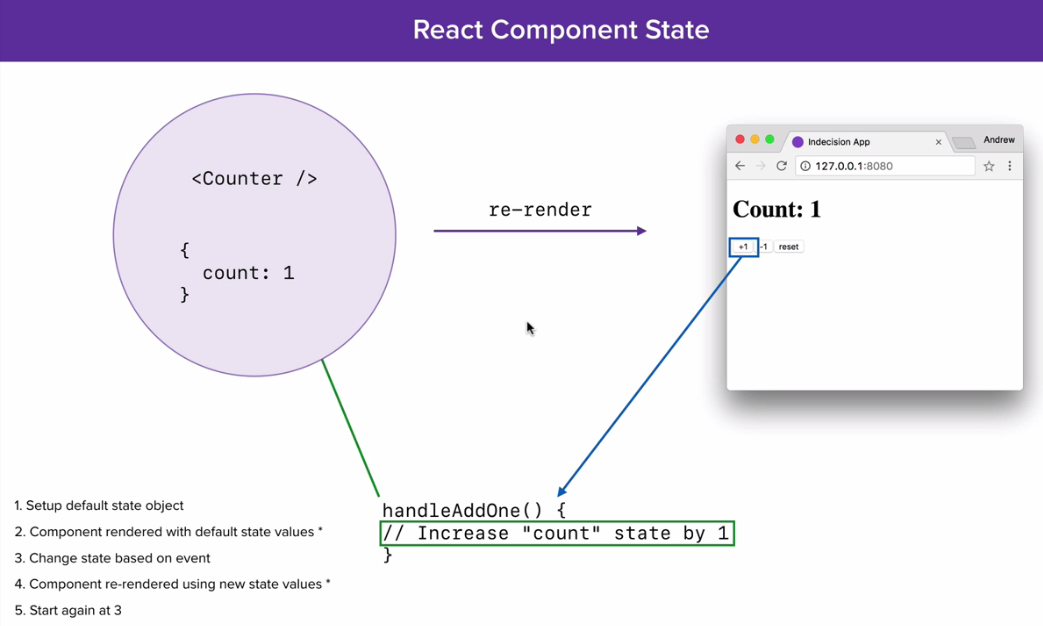
);

}

}

### What Is Component State

* Component state allows us to manage some data
* Component state just an object. It is a set of key value pair
* We define our initial data and that allow us to render component with default values to the screen
* State object can be changed based on event e.g click, change
* When we change state object, component re renders itself automatically



class Counter extends React.Component {

constructor(props) {

super(props);

this.handleAddOne = this.handleAddOne.bind(this);

**// Step 1: Set default state values**

**this.state = {**

**count: 0**

**};**

}

handleAddOne() {

**// Step 3: Change state value based on event**

// this.state.count++;

// If you directly update state value then component do not re render state value

// Use this.setState to manipulate state values

**this.setState((prevState) => {**

**return {**

**count: prevState.count + 1**

**};**

**});**

**// Step 4: Components re rendered automatically using new state value**

}

render() {

return (

<div>

**{/\* Step 2: Render component with default state value\*/}**

**<h1>Count: {this.state.count}</h1>**

<button onClick={this.handleAddOne}>+1</button>

</div>

);

}

}

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

### Alternative setState Syntax

*this.setState()* function allow us to directly pass the object instead of passing it in a callback function but this is the older approach.

this.setState({

count: 0

});

But above syntax has some drawback because this.setState is an asynchronous function.

In the below example we will not see the value that we are expecting because of asynchronous behaviour of this.setState function

this.setState({

count: 0

});

this.setState({

count: this.state.count + 1

});

Always use this.setState with a callback function to update state values

To solve above issue, we can do as below:

this.setState(() => ({count: 0}));

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

### How to manipulate parent state object from child component?

**props** are one way thing in React. You can pass **props** from parent to child component but not vice-versa

In props we can actually pass the entire function reference to the child component

class IndecisionApp extends React.Component {

constructor(props) {

super(props);

this.handlePick = this.handlePick.bind(this);

this.state = {

options: ['One', 'Two']

};

}

**handleDeleteOptions() {**

**this.setState(()=>{**

**return {**

**options: []**

**};**

**});**

**}**

render() {

const title = "Indecision App";

const subTitle = 'Put your life in the hands of a computer.';

return (

<div>

<Options options={**this.state.options**} **handleDeleteOptions={this.handleDeleteOptions}** />

</div>

);

}

}

class Options extends React.Component {

render() {

return (

<div>

<button onClick={**this.props.handleDeleteOptions**}>Remove All</button>

{'Options Length: ' + this.props.options.length} {

this.props.options.length && this.props.options.map((optionText) => <Option key={optionText} optionText={optionText} />)

}

</div>

);

}

}

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

In this example, we created **handleDeleteOptions** function in parent component and passing that function as a reference to child component using props. In child component we are accessing passed method reference using props and calling that function to update parent state object.

### How to pass data upstream?

Send method as a props in child component and use that props method to send data back to parent component

class IndecisionApp extends React.Component {

constructor(props) {

super(props);

this.handleAddOption = this.handleAddOption.bind(this);

this.state = {

options: ['One', 'Two']

};

**}**

**handleAddOption(option) {**

**if(!option) {**

**return 'Enter valid value to add item';**

**} else if(this.state.options.indexOf(option) > -1) {**

**return 'This option already exist.'**

**}**

**this.setState((prevState) => {**

**return {**

**options: prevState.options.concat([option])**

**};**

**});**

**}**

render() {

return (

<div>

**<AddOption handleAddOption={this.handleAddOption}/>**

</div>

);

}

}

class AddOption extends React.Component {

constructor(props) {

super(props);

this.handleAddOption = this.handleAddOption.bind(this);

this.state = {

error: undefined

};

}

**handleAddOption(e) {**

**e.preventDefault();**

**const option = e.target.elements.option.value.trim();**

**// Calling parent handleAddOption function and passing option value**

**const error = this.props.handleAddOption(option);**

**this.setState(() => {**

**// if key and value has same name**

**return { error };**

**});**

**}**

render() {

return (

<div>

{this.state.error && <p>{this.state.error}</p>}

<form onSubmit={this.handleAddOption}>

<input type='text' name='option' />

<button>Add Option</button>

</form>

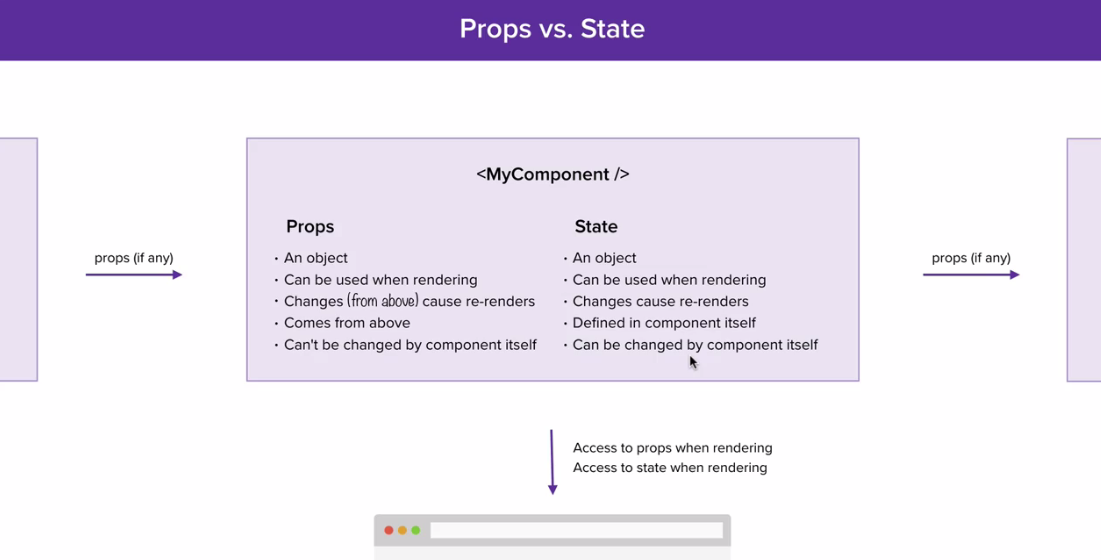
</div>

);

}

}

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



### Stateless Functional Components

* It is an alternative to Class based component
* It does not allow to add state but does allow to add props
* We can render stateless component in the same way we rendered Class based component
* Does not have life cycle hooks

**Advantages**

1. They are faster than Class based component
2. If we want to create presentational component, we should use function based component
3. Easy to understand

**Why functional components are faster than stateful components?**

1. Functional components don’t support state, refs, or lifecycle methods. They can’t extend PureComponent either

#### Convert Class based component to Stateless Functional Component

class Action extends React.Component {

render() {

return (

<div>

<button onClick={this.props.handlePick} disabled={!this.props.hasOption}>

What should I do?

</button>

</div>

);

}

}

1. Create a variable with the same name as the Class based component
2. Accept **props** as a parameter that will give access to all the props passed into the component
3. Change **this.props** to **props,** because arrow function does not have its own this
4. Arrow function same as **render()**  function in Class based component, so copy entire render function content into arrow function

const Action = (**props**) => {

return (

<div>

<button onClick={props.handlePick} disabled={!props.hasOption}>

What should I do?

</button>

</div>

);

};

#### Default Prop Values

**defaultProps** allows to set default value for Class based/functional Component

const Header = (props) => {

console.log(props);

return (

<div>

<h1>{props.title}</h1>

{props.subTitle && <h2>{props.subTitle }</h2>}

</div>

);

};

**Header.defaultProps = {**

**title: 'Indecision App'**

**};**

If we pass title props to Header component then default value will be overridden by the passed value

### Component Types

Can be classified based on how they used:

1. Presentation vs container
2. Skinny vs fat
3. Dumb vs smart
4. Stateless vs stateful

#### Presentational Component

1. Mainly concerned with rendering the view
2. Render the view based on data passed to them in props
3. Do not maintain their own state

#### Container Component

1. Responsible for making things work (data fetching, state updates)
2. Make use of presentational component for rendering
3. Provide data to the presentational components
4. Maintain state and communicate with data sources

#### Class Components

1. Extend React.Component to get class components
2. Need to implement the render() method that returns the view
3. Can have local state
4. Lifecycle hooks

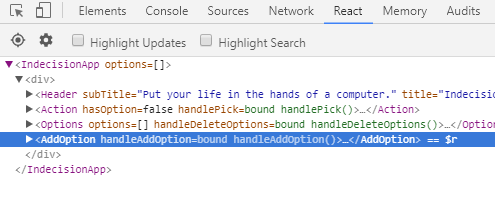
#### Functional Components

1. Simplest way to define React Components
2. JavaScript function that returns a React element or collection of React element that define the view
3. Receives a props object as a parameter
4. Cannot have local state or access life cycle methods

## React Dev Tools

React developer tools available for Chrome and firefox [Chrome React Dev Tool Link](https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en)

Once you install React Dev Tool extension for chrome, you can see **React** tab in chrome developer tool window



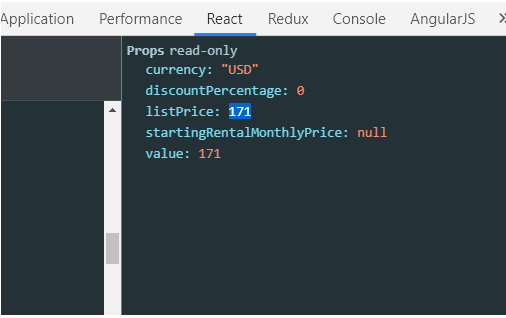
Select any of the component. You can see **$r** variable on right side of the component.

**$r** is a global variable provided by React so that you can check properties and methods of the selected component

You can update/change props/state in React Developer tool

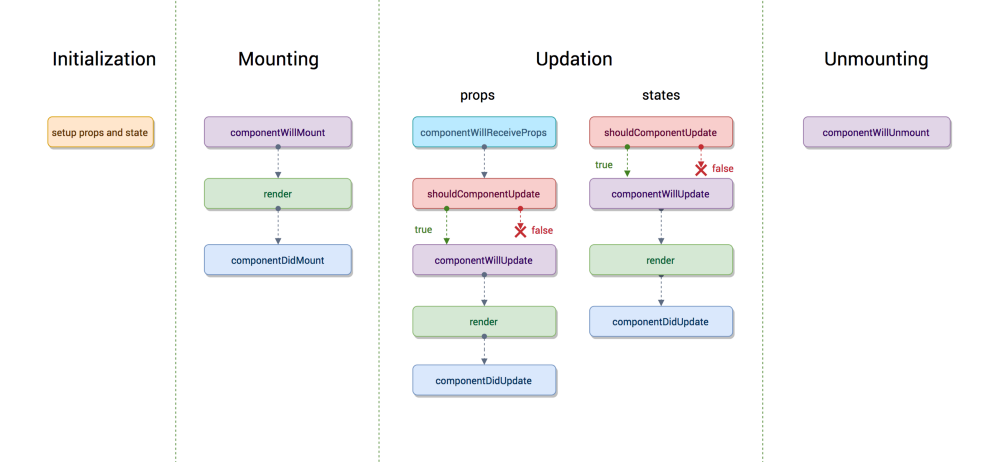
**Note**: You cannot update props of functional component

It looks like props for functional components are being marked at "read-only". From digging around, it seems like this is because functional component instances don't have a forceUpdate()



## Lifecycle Methods

* Lifecycle methods fires at various time in a given component life
* Lifecycle methods not available for Stateless Functional Component
* We don’t call these functions explicitly. React call these functions internally



### Initialization

In this phase the React component prepares for the upcoming tough journey, by setting up the initial states and default props, if any.

### Mounting

**componentWillMount**

1. is executed just before the React Component is about to mount on the DOM
2. This method is executed once in a lifecycle of a component and before first render.

**render**

1. mounts the component onto the browser. This is a pure method, which means it gives the same output every time the same input is provided.

**componentDidMount**

1. is executed after the component did mount on the dom.
2. This method is executed once in a lifecycle of a component and after the first render.
3. As, in this method, we can access the DOM, we can send network request etc.
4. The API calls should be made in componentDidMount method always.

### Updation

The component can be updated by two ways, sending new props or updating the state.

**shouldComponentUpdate**

1. Based on new and previous props you should return true or false, and accordingly the component would be re-rendered or skipped. By default, this method return true.

**componentWillUpdate**

1. is executed only after the shouldComponentUpdate returns true.
2. This method is only used to do the preparation for the upcoming render

**render** And then the component gets rendered.

**componentDidUpdate**

1. is executed when the new updated component has been updated in the DOM.

The list of methods that will get called when the parent sends new props are as follows:

**componentWillReceiveProps**

1. gets executed when the props have changed and is not first render.
2. Sometimes state depends on the props, hence whenever props changes the state should also be synced.
3. The rest of the methods behave exactly same defined above, in terms of state as well.
   * 1. shouldComponentUpdate
     2. componentWillUpdate
     3. Render
     4. componentDidUpdate

### Unmounting

**componentWillUnmount**

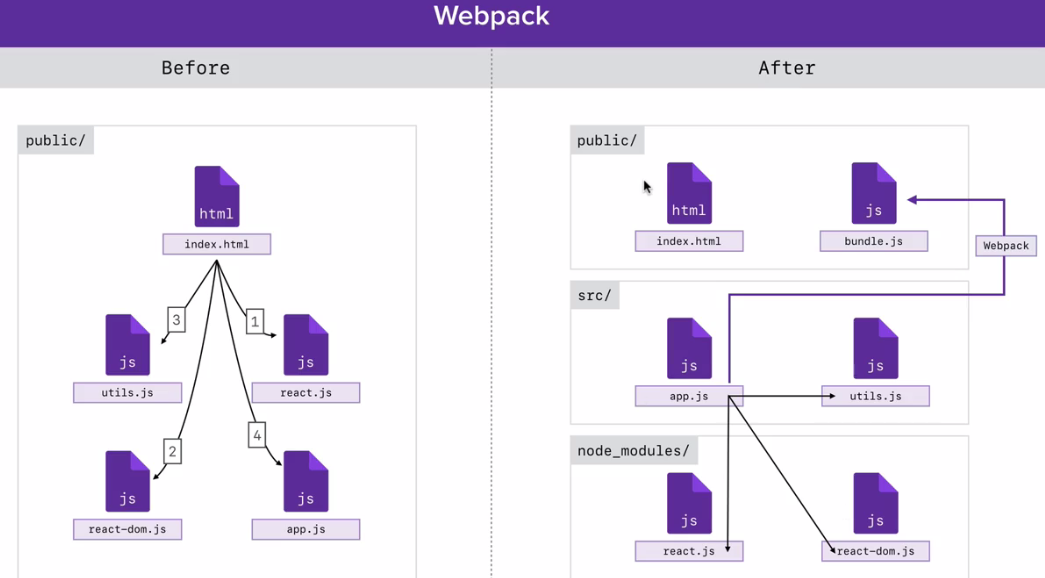
1. This method is the last method in the life cycle. This is executed just before the component gets removed from the DOM.
2. For example, on logout, the user details and all the auth tokens can be cleared before unmounting the main component.
3. Use this function to “clean up” after the component if it takes advantage of timers (setTimeout, setInterval), opens sockets or performs any operations we need to close / remove when no longer needed.

## Webpack

**Webpack** is a popular module bundling system built on top of Node.js. It can handle not only combination and minification of JavaScript and CSS files, but also other assets such as image files (spriting) through the use of plugins.

It allows us to manage JS files. When we run through Webpack we get single javascript file

That file called bundle that contains everything



### Avoid Global Modules

If you are collaborating project, the someone might not know what packages need to install because if you install globally they will not added in the package.json file

If you have many projects in your machine then all your projects will use same version of the globally installed package

We can put big commands in package.json file by giving it a name so that later on we can execute that command as below

npm run <script name>

### Installing Configuring Webpack

1. Install webpack in your project locally using below command

npm install webpack --save-dev

1. Add below key value pair in script section in package.json file so you can run webpack using following command: **npm run build**

"build": "webpack",

1. Create **webpack.config.js** file in project root folder (this is actually a node script)
2. All the properties outlined in webpack documentation
3. **Path** property in **output** property, require absolute path but that may differ from system to system so we use node path module to locate to local directory

### ES6 import export

Utils.js

console.log('utils.js is running!!');

const square = (x) => x \* x;

const add = (a, b) => a + b;

const subtract = (a, b) => a - b;

export { square, add}

App.js

// import './utils.js';

import { square, add } from './utils.js'

console.log('app.js is running');

console.log(square(4));

console.log(add(4,5))

### Default Exports

You can have only zero or one default export in your file

utils.js

const add = (a, b) => a + b;

const subtract = (a, b) => a - b;

**export { add, subtract as default }**

App.js

import **subtract**, { square, add } from './utils.js'

console.log(add(4,5))

console.log(subtract(4,5))

**Note**: default exports should be imported outside the curly braces. If you use inside curly brace you will get reference error

Naming is not important for default export you can import by other name but this is not possible for named exports

import **anythingYouWant**, { square, add } from './utils.js'

Two ways to export from a file

1. Declare the things you want to export in **export** object (this is not a key value pair object)

export { square, add, subtract as default }

1. You can also declare inline as below

export const square = (x) => x \* x;

export const add = (a, b) => a + b;

1. You cannot declare default export inline.Default export has to be declared after variable declaration

const subtract = (a, b) => a - b;

export default subtract;

Or you can directly export function

export default (a, b) => a - b;

### Importing npm Modules

Use below command to import react and react-dom

**npm install react react-dom --save**

Now, you can import these module in your file

import React from 'react';

import ReactDOM from 'react-dom';

// const template = <p>Testing</p>;

// Here we don't have support for JSX

const template = React.createElement("p", {}, 'testing');

ReactDOM.render(template, document.getElementById('app'));

### Setting up Babel with Webpack

Babel-core module allows to run babel through webpack

Babel-loader is a webpack plugin that teach webpack how to run babel when webpack sees certain files

npm install babel-core babel-loader --save-dev

To setup babel in webpack, we can use **module** property in webpack.config.js file

module: {

rules: [{

loader: 'babel-loader',

test: /\.js$/,

exclude: /node\_modules/

}]

}

Here we are saying to webpack that whenever it sees files ends with .js and not in node\_modules folder then run it through babel-loader module

To to set babel environment, create a file named **.babelrc** in the project root folder and add below code;

{

"presets": [

"env",

"react"

]

}

### One Component per File

We should make all the component related thing in single file so it will be easy to understand

### Source Maps with Webpack

Currently, if you get any error in any of the .js file, on developer tool you will get error in **bundle.min.js**  file, which is difficult to read.

To get the error in the file, webpack has configuration for that also

Add below property in **webpack.config.js** file

devtool: 'cheap-module-eval-source-map'

### Webpack Dev Server

It is a little dev server like **live-server** but webpack dev server comes with nice to have feature specific to webpack. It speed up the process between changing the file and viewing on the browser

To install webpack-dev-server use below command:

npm install webpack-dev-server --save-dev

Add **devServer** property in **webpack.config.js** file and set the path of the public folder as below;

devServer: {

contentBase: path.join(\_\_dirname, 'public')

}

Update scripts in script section in package.json file as below:

"build": "webpack",

"dev-server": "webpack-dev-server"

Now, you don’t need to run multiple command to run the project

Use **npm run dev-server** to run the project and it will automatically run webpack internally whenever needed

**webpack-dev-server serves your bundle.js from memory. It won't generate the file when you run it. So bundle.js is not present as a file in this scenario.**

**If you want to use bundle.js, for example to optimize it's size or test your production deployment, generate it with webpack using the webpack command and serve it in production mode.**

### ES6 class properties

Install below babel plugin

npm install --save-dev babel-plugin-transform-class-properties

This plugin will allow you to write class in a better way

class OldSyntax {

constructor() {

**this.name = "Mike";**

**this.getGreeting = this.getGreeting.bind(this);**

}

getGreeting() {

return `Hi, my name is ${this.name}`;

}

}

const oldSyntax = new OldSyntax();

console.log('oldSyntax.name' + oldSyntax.name);

console.log('oldSyntax.getGreeting()' + oldSyntax.getGreeting());

// but getGreeting binding can break if we do as below

const getGreeting = oldSyntax.getGreeting;

console.log("getGreeting() " + getGreeting()); // Will give error: name is not defined because ‘this’ is no longer referring to OldSysntax class

// To fix 'this' reference error we need to bind it in constructor

Issue in above syntax can be resolved using babel plugin syntax as below

class NewSyntax {

name = 'Jan';

// Because arrow function don't bind its this, so 'this' is always be referred to class

getGreeting = () => (`Hi, my name is ${this.name}`)

}

const newSyntax = new NewSyntax();

const newGetGreeting = newSyntax.getGreeting;

console.log('newSyntax.name: ' + newSyntax.name);

console.log('newGetGreeting(): ' + newGetGreeting());

## Using a Third-Party Component

### Passing Children to Component

**Method 1**

import React from 'react';

import ReactDOM from 'react-dom';

const Layout = (props) => {

return (

<div>

<p>Header</p>

{props.content}

<p>Footer</p>

</div>

);

};

const template = (

<div>

<h1>Page Title</h1>

<p>This is my page</p>

</div>

);

ReactDOM.render(<Layout content={template}/>, document.getElementById('app'));

**Method 2**

You can directly pass the template code inside custom HTML tags

ReactDOM.render((

<Layout>

**<div>**

**<h1>Page Title</h1>**

**<p>This is my page</p>**

**</div>**

</Layout>

), document.getElementById('app'));

### Setting up React-Modal

React-Modal is a react’s inbuilt modal dialogue

Use below command to install react modal

**npm install react-modal --save-dev**

indecisionApp.js

import React from 'react';

import AddOption from './AddOption';

import Options from './Options';

import Action from './Action';

import Header from './Header';

import OptionModal from './OptionModal';

export default class IndecisionApp extends React.Component {

state = {

options: [],

selectedOption: undefined

};

**handlePick** = () => {

const randonNum = Math.floor(Math.random() \* this.state.options.length);

const option = this.state.options[randonNum];

this.setState((prevState) => ({selectedOption: option }));

}

**handleClearSelectedOption** = () => {

this.setState(() => ({selectedOption: undefined}));

};

render() {

const subTitle = 'Put your life in the hands of a computer.';

return (

<div>

...

**<OptionModal selectedOption={this.state.selectedOption} handleClearSelectedOption={this.handleClearSelectedOption} />**

</div>

);

}

}

OptionModal.js

import React from 'react';

import Modal from 'react-modal';

const OptionModal = (props) => (

<Modal

isOpen={!!props.selectedOption}

onRequestClose = {props.handleClearSelectedOption} /\*\* To Close modal when user clicks outside or press esc button \*/

contentLabel="Selected Option"

>

<h3>Selected Option</h3>

{props.selectedOption && <p>{props.selectedOption}</p>}

<button onClick={props.handleClearSelectedOption}>Okay</button>

</Modal>

);

export default OptionModal;

## Styling React

### Setting up Webpack with SCSS

Install css loader packages using below command:

**npm install style-loader css-loader --save-dev**

**npm install sass-loader node-sass --save-dev**

To include css loaders in webpack update **module** object as below:

…

module: {

rules: [{

loader: 'babel-loader',

test: /\.js$/,

exclude: /node\_modules/

}, **{**

**test: /\.scss$/,**

**use: ['style-loader', 'css-loader', 'sass-loader']**

**}**

]

},

…

Whenever webpack sees **.scss** code in **.js** file it will load it using **sass-loader** and compile them using **node-sass** module and then uses **css-loader** to load compiled css codeand inject it into the DOM(in Style html tag) using **style-loader**

### Architecture and Header Styles

We should create a seperate folder to have styles files. Each component should have its own style file

### Reset That

CSS reset just makes sure that all browser starts at the same place.Each browser has its own set of default styles

There are libraries available for this. We will be using **normalize.css** Use below command to install it

**npm install normalize.css --save-dev**

Import the normalize.css file

import '../node\_modules/normalize.css/normalize.css';

Because it is a css file, so we need to update webpack.config.js file to look for .css files as well

{

**test: /\.s?css$/,**

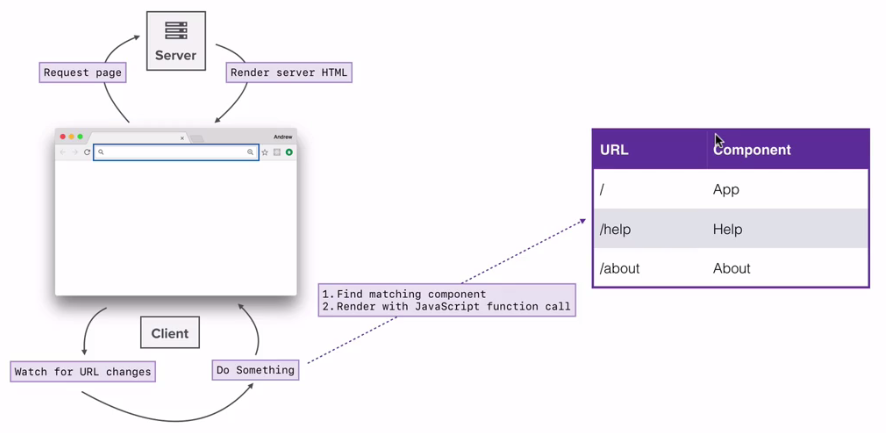
use: ['style-loader', 'css-loader', 'sass-loader']

}

## React-Router

### Server vs. Client Routing

|  |  |
| --- | --- |
| **Server Side Routing** | **Client Side Routing** |
| Traditional approach where we define routing on server side | Modern approach where we define routing on client side |
| Whenever url change, browser makes http request, after that server sends the html and browser render it | First time require http request and further request handles on client side |
| This process is expensive, sending http request and waiting for response takes time | Uses HTML history API available by browsers that allows us to watch URL changes and run some JavaScript code |



We call it “Dynamic Routing”, which is quite different from the “Static Routing” you’re probably more familiar with.

### Static Routing

If you’ve used Rails, Express, Ember, Angular etc. you’ve used static routing. In these frameworks, you declare your routes as part of your app’s initialization before any rendering takes place. React Router pre-v4 was also static (mostly).

### Dynamic Routing

When we say dynamic routing, we mean routing that takes place as your app is rendering, not in a configuration or convention outside of a running app. That means almost everything is a component in React Router

First, grab yourself a Router component for the environment you’re targeting and render it at the top of your app.

Next, grab the link component to link to a new location:

Finally, render a Route to show some UI when the user visits /dashboard.

### Responsive Routes

As React Router V4 is dynamic, you can also render routes based on viewport. E.g. some routes will be valid for small screens and some will be valid for large screens

### Installation

1. Install router using below command
   1. **npm install react-router-dom --save-dev**
2. Router component: <BrowserRouter>
   1. Creates specialized history object
   2. Enclose your app in BrowserRouter

Import in your file

import { BrowserRouter, Route } from 'react-router-dom';

**App.js**

import React from 'react';

import ReactDOM from 'react-dom';

**import { BrowserRouter, Route } from 'react-router-dom';**

const ExpenseDashboardPage = () => (

<div>

This is from my dashboard component;

</div>

);

const AddExpensePage = () => (

<div>

This is from my AddExpensePage page.

</div>

);

**const routes = (**

**<BrowserRouter>**

**<div>**

**<Route path="/" component={ExpenseDashboardPage} />**

**<Route path="/create" component={AddExpensePage} />**

**</div>**

**</BrowserRouter>**

**);**

ReactDOM.render(**routes**, document.getElementById('app'));

At this point if you try to load URL: <http://127.0.0.1:8080>, you will get only **ExpenseDashboardPage**  component

To load **AddExpensePage** use this URL: http://127.0.0.1:8080/create

But this will give 404 error, because here browser trying to fetch page from server side.

So it’s using server side routing for first page load

**Note:** For the first time we visit our application the browser needs to grab the initial html and load the javaScript before it can do anything

We can do a tweak here, we can configure dev server to send **index.html**  for all routes and let react-router figure it out what should get shown on the screen so in order to do that make changes in **webpack.config.js** file

Update devserver property as below:

devServer: {

contentBase: path.join(\_\_dirname, 'public'),

**historyApiFallback: true**

}

This tells the browser that we are handling routes on client side and it will return index.html

Now, if you load <http://127.0.0.1:8080/create> it will show you output from both the component

Because react-router matches path which start at least with the mentioned path

We can fix above issue by adding **exact** property to **Route** component and its default value is false

<Route path="/" component={ExpenseDashboardPage} **exact={true}**/>

### BrowserRouter

A <Router> that uses the HTML5 history API (pushState, replaceState and the popstate event) to keep your UI in sync with the URL.

<BrowserRouter

basename={optionalString}

getUserConfirmation={optionalFunc}

forceRefresh={optionalBool}

keyLength={optionalNumber}

>

<App/>

</BrowserRouter>

**basename: string**

The base URL for all locations. If your app is served from a sub-directory on your server, you’ll want to set this to the sub-directory. A properly formatted basename should have a leading slash, but no trailing slash.

**getUserConfirmation: func**

A function to use to confirm navigation. Defaults to using window.confirm.

**forceRefresh: bool**

If true the router will use full page refreshes on page navigation. You probably only want this in browsers that don’t support the HTML5 history API.

**keyLength: number**

The length of location.key. Defaults to 6.

### HashRouter

A <Router> that uses the hash portion of the URL (i.e. window.location.hash) to keep your UI in sync with the URL.

<HashRouter basename="/calendar"/>

<Link to="/today"/> // renders <a href="#/calendar/today">

**basename: string**

The base URL for all locations. A properly formatted basename should have a leading slash, but no trailing slash.

### <Link>

Provides declarative, accessible navigation around your application.

**to: string**

A string representation of the location to link to, created by concatenating the location’s pathname, search, and hash properties.

<Link to='/courses?sort=name'/>

**to: object**

An object that can have any of the following properties:

pathname: A string representing the path to link to.

search: A string representation of query parameters.

hash: A hash to put in the URL, e.g. #a-hash.

state: State to persist to the location.

<Link to={{

pathname: '/courses',

search: '?sort=name',

hash: '#the-hash',

state: { fromDashboard: true }

}}/>

All above properties will be available as a props in the rendered component of the mentioned path

**This.props.location** will have below properties

hash, key, pathname, search, state

### <NavLink>

A special version of the <Link> that will add styling attributes to the rendered element when it matches the current URL

**activeClassName: string**

The class to give the element when it is active. The default given class is active. This will be joined with the className prop.

<NavLink

to="/faq"

activeClassName="selected"

>FAQs</NavLink>

**activeStyle: object**

The styles to apply to the element when it is active.

<NavLink

to="/faq"

activeStyle={{

fontWeight: 'bold',

color: 'red'

}}

>FAQs</NavLink>

**exact: bool**

When true, the active class/style will only be applied if the location is matched exactly.

**strict: bool**

When true, the trailing slash on a location’s pathname will be taken into consideration when determining if the location matches the current URL.

Match from exactly; equivalent to Route.exact.

strict: bool

Match from strictly; equivalent to Route.strict.

### Prompt

Used to prompt the user before navigating away from a page. When your application enters a state that should prevent the user from navigating away (like a form is half-filled out), render a <Prompt>.

import { Prompt } from 'react-router'

<Prompt

when={formIsHalfFilledOut}

message="Are you sure you want to leave?"

/>

**message: string**

The message to prompt the user with when they try to navigate away.

**message: func**

Will be called with the next location and action the user is attempting to navigate to. Return a string to show a prompt to the user or true to allow the transition.

**when: bool**

Instead of conditionally rendering a <Prompt> behind a guard, you can always render it but pass when={true} or when={false} to prevent or allow navigation accordingly.

### MemoryRouter

A <Router> that keeps the history of your “URL” in memory (does not read or write to the address bar). Useful in tests and non-browser environments like React Native.

import { MemoryRouter } from 'react-router'

<MemoryRouter

initialEntries={[ '/one', '/two', { pathname: '/three' } ]}

initialIndex={1}

>

<App/>

</MemoryRouter>

**initialEntries: array**

An array of locations in the history stack. These may be full-blown location objects with { pathname, search, hash, state } or simple string URLs.

**initialIndex: number**

The initial location’s index in the array of initialEntries.

### Redirect

Rendering a <Redirect> will navigate to a new location. The new location will override the current location in the history stack, like server-side redirects (HTTP 3xx) do.

**to: string**

The URL to redirect to

<Redirect to="/somewhere/else" />

**to: object**

A location to redirect to. pathname can be any valid URL path that path-to-regexp@^1.7.0 understands.

<Redirect

to={{

pathname: "/login",

search: "?utm=your+face",

state: { referrer: currentLocation }

}}

/>

**push: bool**

When true, redirecting will push a new entry onto the history instead of replacing the current one.

<Redirect push to="/somewhere/else" />

### Route

The Route component is perhaps the most important component in React Router to understand and learn to use well. Its most basic responsibility is to render some UI when a location matches the route’s path.

import { BrowserRouter as Router, Route } from 'react-router-dom'

<Router>

<div>

<Route exact path="/" component={Home}/>

<Route path="/news" component={NewsFeed}/>

</div>

</Router>

<div>

<Home/>

<!-- react-empty: 2 -->

</div>

<div>

<!-- react-empty: 1 -->

<NewsFeed/>

</div>

**Route render methods**

There are 3 ways to render something with a <Route>:

* <Route component>
* <Route render>
* <Route children>

Each is useful in different circumstances. You should use only one of these props on a given <Route>. Most of the time you’ll use component.

**Route props**

All three render methods will be passed the same three route props

* match
* location
* history

**component**

A React component to render only when the location matches. It will be rendered with route props.

**render: func**

you can pass in a function to be called when the location matches. The render prop receives all the same route props as the component render prop.

// wrapping/composing

const FadingRoute = ({ component: Component, ...rest }) => (

<Route {...rest} **render**={props => (

<FadeIn>

<Component {...props}/>

</FadeIn>

)}/>

)

<FadingRoute path="/cool" component={Something}/>

**children: func**

Sometimes you need to render whether the path matches the location or not. In these cases, you can use the function children prop. It works exactly like render except that it gets called whether there is a match or not.

const ListItemLink = ({ to, ...rest }) => (

<Route path={to} children={({ match }) => (

<li className={match ? 'active' : ''}>

<Link to={to} {...rest}/>

</li>

)}/>

)

**path: string**

Any valid URL

### Router

The common low-level interface for all router components. Typically apps will use one of the high-level routers instead:

1. <BrowserRouter>
2. <HashRouter>
3. <MemoryRouter>
4. <NativeRouter>
5. <StaticRouter>

The most common use-case for using the low-level <Router> is to synchronize a custom history with a state management lib like Redux or Mobx. Note that this is not required to use state management libs alongside React Router, it’s only for deep integration.

import { Router } from 'react-router'

import createBrowserHistory from 'history/createBrowserHistory'

const history = createBrowserHistory()

<Router history={history}>

<App/>

</Router>

### StaticRouter

A <Router> that never changes location.

An advantage of static routing is that it allows for inspection and matching of routes before rendering. Hence it proves useful especially on server side.

### Switch

Renders the first child <Route> or <Redirect> that matches the location.

import { Switch, Route } from 'react-router'

<Switch>

<Route exact path="/" component={Home}/>

<Route path="/about" component={About}/>

<Route path="/:user" component={User}/>

<Route component={NoMatch}/>

</Switch>

Now, if we’re at /about, <Switch> will start looking for a matching <Route>. <Route path="/about"/> will match and <Switch> will stop looking for matches and render <About>

### HashRouter vs BrowserRouter

|  |  |
| --- | --- |
| **BrowserRouter** | **HashRouter** |
| It uses HTML5 history API | It uses the hash in the URL |
| The <BrowserRouter> should be used when you have a server that will handle dynamic requests (knows how to respond to any possible URI) | the <HashRouter> should be used for static websites (where the server can only respond to requests for files that it knows about). |
| Client-side React application is able to maintain clean routes like example.com/react/route but needs to be backed by web server. Usually this means that web server should be configured for single-page application, i.e. index.html is served for /react/route or any other route. | HashRouter uses a hash symbol in the URL, which has the effect of all subsequent URL path content being ignored in the server request (ie you send "www.mywebsite.com/#/person/john" the server gets "www.mywebsite.com". As a result the server will return the pre # URL response, and then the post # path will be handled by parsed by your client side react application. |

### History

The following terms are also used:

* “browser history” - A DOM-specific implementation, useful in web browsers that support the HTML5 history API
* “hash history” - A DOM-specific implementation for legacy web browsers
* “memory history” - An in-memory history implementation, useful in testing and non-DOM environments like React Native

history objects typically have the following properties and methods:

* length - (number) The number of entries in the history stack
* action - (string) The current action (PUSH, REPLACE, or POP)
* location - (object) The current location. May have the following properties:
* pathname - (string) The path of the URL
* search - (string) The URL query string
* hash - (string) The URL hash fragment
* state - (object) location-specific state that was provided to e.g. push(path, state) when this location was pushed onto the stack. Only available in browser and memory history.
* push(path, [state]) - (function) Pushes a new entry onto the history stack
* replace(path, [state]) - (function) Replaces the current entry on the history stack
* go(n) - (function) Moves the pointer in the history stack by n entries
* goBack() - (function) Equivalent to go(-1)
* goForward() - (function) Equivalent to go(1)
* block(prompt) - (function) Prevents navigation

### Setting up a 404

What if someone visits route that we do not have route for.

If I go <http://127.0.0.1:8080/testing> , we will not get any error for this route

React router check all the routes and if none of the match it shows none

If you add route as below, then it will be visible on all the page

<Route component={NotFoundPage} />

To solve above problem, React router provides a **Switch** component in **React Router**

Switch does not take any parameter but it contains routes in the order

**App.js**

import React from 'react';

import ReactDOM from 'react-dom';

**import { BrowserRouter, Route, Switch } from 'react-router-dom';**

const ExpenseDashboardPage = () => (

<div>

This is from my dashboard component;

</div>

);

const NotFoundPage = () => (

<div>

404!!

</div>

);

const routes = (

<BrowserRouter>

**<Switch>**

<Route path="/" component={ExpenseDashboardPage} exact={true}/>

<Route path="/create" component={AddExpensePage} />

<Route path="/edit" component={EditExpensePage} />

<Route path="/help" component={HelpPage} />

**<Route component={NotFoundPage} />**

**</Switch>**

</BrowserRouter>

);

ReactDOM.render(routes, document.getElementById('app'));

Switch component check routes and return the first matched component, if none of the routes matches then it will return last or default route component

**Note: BrowerRouter takes only one route element**

### Linking Between Routes

When we change the URL manually we get full page refresh which we don’t want

We want to switch pages without going to full page refresh

How to add links to navigate other pages?

Generally we use anchor tag <a href="/">Go Home</a> but if we click on this link then it will do full page refresh

To solve above issue, React Router gives a component **Link** that we can use to link other pages

import { BrowserRouter, Route, Switch, Link } from 'react-router-dom';

. . .

const NotFoundPage = () => (

<div>

404!! - <Link to="/">Go Home</Link>

</div>

);

Some place we need <a></a> tag and some places we need <Link /> tag provided by the react-router-dom

Login button - because we are navigating to different domain

<a href="/auth/google">Login with Google</a>

Logout button - because we are navigating to different domain

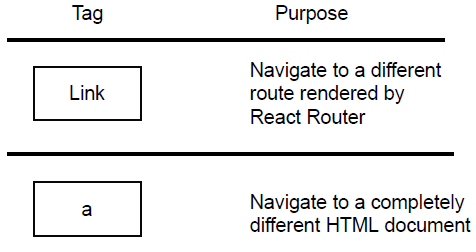
<a href="/api/logout">Logout</a>

Logo button - because we are navigating to same domain(client route)

<Link to={this.props.auth ? '/surveys' : '/'} className="left brand-logo">

Emaily

</Link>



For navigation link React Router provides separate component **NavLink** in which we can add customize link styles and color of selected NavLink

const Header = () => (

<header>

<h1>Expensify</h1>

<NavLink to="/" activeClassName="is-active" **exact={true}**>Dashboard</NavLink>

<NavLink to="/create" activeClassName="is-active">Create Expense</NavLink>

<NavLink to="/edit" activeClassName="is-active">Edit Expense</NavLink>

<NavLink to="/help" activeClassName="is-active">Help</NavLink>

</header>

);

Here **exact={true}** behaving same as Routes

### Query Strings and URL Parameters

**Route props**

All three below methods will be passed in the props of the component defined in the routes

* match
* location
* history

**AppRouter.js**

import React from "react";

import { BrowserRouter, Route, Switch, Link, NavLink } from "react-router-dom";

import EditExpensePage from "../components/EditExpensePage";

const AppRouter = () => (

<BrowserRouter>

<div>

<Header />

<Switch>

**<Route path="/edit/:id" component={EditExpensePage} />**

<Route component={NotFoundPage} />

</Switch>

</div>

</BrowserRouter>

);

export default AppRouter;

**EditExpensePage.js**

import React from "react";

const EditExpensePage = **props** => {

console.log(props);

return <div>Editing the expense with id of {**props.match.params.id**}.</div>;

};

export default EditExpensePage;

### Nested Routes

<https://github.com/reactjs/react-router-tutorial/tree/master/lessons>

Let's nest our **About** and **Repos** components inside of **App** so that we can share the navigation with all screens in the app. We do it in two steps:

First, let the App Route have children, and move the other routes underneath it.

// index.js

// ...

render((

<Router history={hashHistory}>

**<Route path="/" component={App}>**

**{/\* make them children of `App` \*/}**

**<Route path="/repos" component={Repos}/>**

**<Route path="/about" component={About}/>**

**</Route>**

</Router>

), document.getElementById('app'))

Next, render children inside of App.

// modules/App.js

// ...

render() {

return (

<div>

<h1>React Router Tutorial</h1>

<ul role="nav">

<li><Link to="/about">About</Link></li>

<li><Link to="/repos">Repos</Link></li>

</ul>

{/\* add this \*/}

**{this.props.children}**

</div>

)

}

// ...

Alright, now go click the links and notice that the App component continues to render while the child route component gets swapped around as **this.props.children**

React Router is constructing your UI like this:

// at /about

<App>

<About/>

</App>

// at /repos

<App>

<Repos/>

</App>

### Index Routes

consider the **home** component:

// modules/Home.js

import React from 'react'

export default React.createClass({

render() {

return <div>Home</div>

}

})

but with this routing:

// index.js

// new imports:

// add `IndexRoute` to 'react-router' imports

import { Router, Route, hashHistory, IndexRoute } from 'react-router'

// and the Home component

import Home from './modules/Home'

// ...

render((

<Router history={hashHistory}>

**<Route path="/" component={App}>**

**{/\* add it here, as a child of `/` \*/}**

**<IndexRoute component={Home}/>**

**<Route path="/repos" component={Repos}>**

**<Route path="/repos/:userName/:repoName" component={Repo}/>**

**</Route>**

**<Route path="/about" component={About}/>**

**</Route>**

</Router>

), document.getElementById('app'))

Now open http://localhost:8080 and you'll see the new component is rendered.

Notice how the IndexRoute has no path. It becomes this.props.children of the parent when no other child of the parent matches, or in other words, when the parent's route matches exactly.

Index routes can twist people's brains up sometimes. Hopefully it will sink in with a bit more time. Just think about a web server that looks for index.html when you're at /. Same idea, React Router looks for an index route if a route's path matches exactly.

### Authenticated routes in React Router 4

Use the Redirect component

function PrivateRoute ({component: Component, authed, ...rest}) {

return (

<Route

{...rest}

render={(props) => authed === true

? <Component {...props} />

: <Redirect to={{pathname: '/login', state: {from: props.location}}} />}

/>

)

}

Now your Routes can look something like this

<Route path='/' exact component={Home} />

<Route path='/login' component={Login} />

<Route path='/register' component={Register} />

<PrivateRoute authed={this.state.authed} path='/dashboard' component={Dashboard} />

### Lazy Loading Routes in React

## Accessing DOM Elements in React

Two ways to access the DOM

1. **Refs:** allow you to directly access the DOM element
2. **Portals:** allow you to render content to any arbitrary DOM. It will work outside your component. No binding to the current component.

### Refs

Refs provide a way to access DOM nodes or React elements created in the render method.

We can add *ref*  to class based component but you may not use the *ref* attribute on functional components because they don’t have instances

You can, however, use the ref attribute inside a functional component as long as you refer to a DOM element or a class component:

#### When to Use Refs

There are a few good use cases for refs:

1. Managing focus, text selection, or media playback.
2. Triggering imperative animations.
3. Integrating with third-party DOM libraries.

#### Creating Refs

Refs are created using **React.createRef()** and attached to React elements via the ref attribute. Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.

#### Accessing Refs

The value of the ref differs depending on the type of the node:

1. When the ref attribute is used on an HTML element, the ref created in the constructor with React.createRef() receives the underlying DOM element as its **current** property.
2. When the ref attribute is used on a custom class component, the ref object receives the mounted instance of the component as its current.
3. You may not use the ref attribute on functional components because they don’t have instances.

#### Adding a Ref to a DOM Element

class CustomTextInput extends React.Component {

constructor(props) {

super(props);

// create a ref to store the textInput DOM element

**this.textInput = React.createRef();**

this.focusTextInput = this.focusTextInput.bind(this);

}

focusTextInput() {

// Explicitly focus the text input using the raw DOM API

// Note: we're accessing "current" to get the DOM node

**this.textInput.current.focus();**

}

render() {

// tell React that we want to associate the <input> ref

// with the `textInput` that we created in the constructor

return (

<div>

<input

type="text"

**ref={this.textInput} />**

<input

type="button"

value="Focus the text input"

onClick={this.focusTextInput}

/>

</div>

);

}

}

#### Callback Refs

React also supports another way to set refs called “callback refs”, which gives more fine-grain control over when refs are set and unset.

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

class RefExample extends React.Component {

onButtonClick = () => {

if (this.input) {

this.input.focus();

}

}

render() {

return (

<div>

<button onClick={this.onButtonClick}>Click here!!</button>

**<input ref={(el) => this.input = el} />**

</div>

);

}

}

ReactDOM.render(<RefExample />, document.querySelector('#app'));

More general way

**const TextInput = (props) => (<input ref={props.reference} />);**

class RefExample extends React.Component {

onButtonClick = () => {

if (this.input) {

this.input.focus();

}

}

**setInputRef = (ref) => (this.input = ref);**

render() {

return (

<div>

<button onClick={this.onButtonClick}>Click here!!</button>

<TextInput **reference={this.setInputRef**} />

</div>

);

}

}

ReactDOM.render(<RefExample />, document.querySelector('#app'));

We can add ref as below as well. Here we will get ref value in **this.refs**

class MyComponent extends React.Component {

state = {

clicks: 0

};

componentDidMount() {

**this.refs.myComponentDiv.addEventListener('click', this.clickHandler);**

}

clickHandler() {

this.setState({

clicks: this.clicks + 1

});

}

render() {

return (

<div className="my-component" **ref="myComponentDiv"**>

<h2>My Component ({this.state.clicks} clicks})</h2>

</div>);

}}

ReactDOM.render(<MyComponent />, document.getElementById('app'))

### React Portal

Portals provide a first-class way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.

render() {

// React does \*not\* create a new div. It renders the children into `domNode`.

// `domNode` is any valid DOM node, regardless of its location in the DOM.

**return ReactDOM.createPortal(**

**this.props.children,**

**domNode**

);

}

A typical use case for portals is when a parent component has an overflow: hidden or z-index style, but you need the child to visually “break out” of its container. For example, dialogs, hovercards, and tooltips.

Code Pen Example: <https://codepen.io/gaearon/pen/yzMaBd?editors=1010>

<script type="text/babel">

**class ColorLabel extends React.Component {**

**render() {**

**return ReactDOM.createPortal(**

**':' + this.props.color,**

**document.querySelector("#colorHeading")**

**);**

**}**

**}**

class Colorizer extends React.Component {

...

render() {

var squareStyle = {

backgroundColor: this.state.bgColor

};

return (

<div className="colorArea">

<div style={squareStyle} className="colorSquare"></div>

<form onSubmit={this.setNewColor}>

<input onChange={this.colorValue}

placeholder="Enter a color value"></input>

<button type="submit">go</button>

</form>

**<ColorLabel color={this.state.bgColor} />**

</div>

);

}

}

...

## Immutability in JavaScript

Immutability is required for some state management library like redux and FluxThis

Requirement for

1. Redux
2. React

JavaScript does not treat data as immutable by default but we can make immutable operations in plain JavaScript

Immutability is a concept that has its root in functional programming. Whenever we wanted to make changes to some data e.g an object or an array, we should get a new object back with updated data instead of directly modifying the original one

### Advantages of using Immutability

1. Provide stricter control over data
2. Makes it easier to implement:
   1. Undo/redo
   2. Time-travel debugging
   3. Optimistic updates
3. Better performance in react
4. Components can take advantages of it and re renders whenever needed
5. **Predictability**: Mutation hides change, which create (unexpected) side effects, which can cause nasty bugs. When you enforce immutability you can keep your application architecture and mental model simple, which makes it easier to reason about your application.
6. **Performance**: Even though adding values to an immutable Object means that a new instance needs to be created where existing values need to be copied and new values need to be added to the new Object which cost memory, **immutable Objects can make use of structural sharing to reduce memory overhead.**
7. **Mutation Tracking:** Besides reduced memory usage, immutability allows you to optimize your application by making use of reference- and value equality. This makes it really easy to see if anything has changed. For example a state change in a react component. You can use *shouldComponentUpdate* to check if the state is identical by comparing state Objects and prevent unnecessary rendering.

### How we can achieve immutability in JavaScript?

JavaScript is not a good language if data is in immutable fashion because it does not provide immutability by default. Array and Object are always stored in reference

const a = [2, 4, 1, 5];

const b = a.sort();

console.log(a); // [1, 2, 4, 5]

console.log(b); // [1, 2, 4, 5]

Here we wanted to sort array in *b*  variable but *a* is also sorted

But there are Array methods which works in immutable way or non destructive array methods:

1. Slice: gives a copy of the array
2. ES6 spread operator for Array: returns new array
3. ES6 spread operator for Objects
4. Filter, map and reduce method

Spread operator makes a shallow copy of the object i.e they go only one level down while copying the object so you need to update the child objects also in immutable manner

const state = {

name: 'John',

age: 20,

skills: []

};

const newState = { ...state, age: 25 };

console.log(newState); // {name: "John", age: 25, skills: Array(0)}

// If we do like this

//newState.skills.push('new skill'); // It will update the original array

console.log(newState); //{name: "John", age: 25, skills: Array(0)}

console.log(state); //{name: "John", age: 20, skills: Array(0)}

**// Update array also in immutable way**

**newState.skills = [...newState.skills, 'immutable skills'];**

console.log(newState); // {name: "John", age: 25, skills: Array(1)}

Or we can make copy as below:

const state = {

name: 'John',

age: 20,

skills: []

};

const newState={...state, age:25, skills: [...state.skills, 'newStateSkill']};

console.log(newState); // {name: "John", age: 25, skills: Array(1)}

console.log(state); //{name: "John", age: 20, skills: Array(0)}

### How Immutable Data Structures (E.g. Immutable.js) are Optimized

For an example, say that we have an array of integers. We need to change one of the integers in the array. Now if we want to stay immutable, instead of changing the array in place, we need to keep the original array intact and return a new array with the changed integer. For this we need to create a new array and copy over the old elements. This is much more expensive than changing the array in place.

a method that is used to optimize immutable data structures, called “Structural Sharing”. To begin let’s learn what a persistent data structure is.

**Persistent data structures**

In computing, a persistent data structure is a data structure that always preserves the previous version of itself when it is modified. Such data structures are effectively immutable, as their operations do not (visibly) update the structure in-place, but instead always yield a new updated structure.

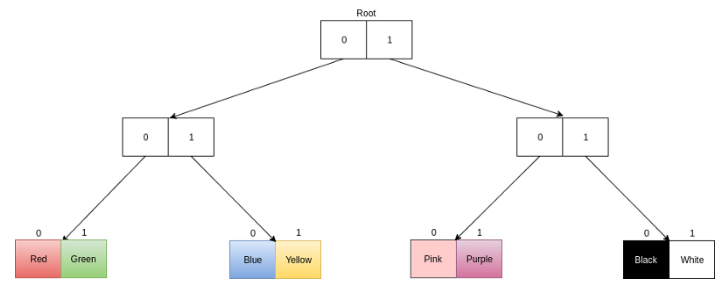
Persistent data structures are commonly used in Functional Programming as this enforces immutability. Almost all functional programming languages has implementations of persistent data structures. **Immutable.js** is a JavaScript library that implements persistent data structures.

**Using a Tree to represent an array**

Since now we know what a Trie is, let’s take a look at how to represent an array using a Trie. Take the following array as an example,

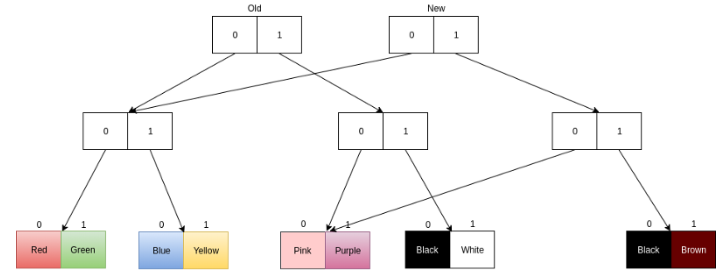
[“red”, “green”, “blue”, “yellow”, “pink”, “purple”, “black”, “white”]

Let’s see how we can try and represent this using a Trie.



If you follow the path “001” (0 means the left node, 1 means the right node. look at the diagram) from the root node you can find element with index 1 (Green).

Say we need to change the last element of this array from “White” to “Brown”. We want this to be done in a persistent manner. Therefore changing the original data structure is not a solution. Let’s take a look at the following diagram.



You can see that the old root is still there and you can access the old array using that. And the array after the new element is added, is the structure with the new root. We are creating a new array with the new element by reusing the old structure. If we had a traditional array we had to copy all the elements.

## Function as Child Components

“Function as Child Component” is a pattern where the parent component accepts a function as child, instead of a nested React Node.

## React's Children API

A component can receive no children at all, a string, single element, multiple element.

Inside the parent component these child component are available to a special prop named children **this.props.children**

App.js

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

**import Parent from './Parent;**

import './App.css';

class App extends Component {

render() {

return (

<div className="App">

**<Parent>**

**<img src="http://placekitten.com/300/200" alt="Cat Pic" />**

**</Parent>**

</div>

);

}

}

export default App;

Parent.js

class Parent extends Component {

render() {

return (

<div>

**{ this.props.children }**

</div>

)

}

}

export default Parent;

### React.Children

React.Children provides utilities for dealing with the this.props.children opaque data structure.

#### React.Children.map

**React.Children.map(children, function[(thisArg)])**

Invokes a function on every immediate child contained within children with this set to thisArg. If children is a keyed fragment or array it will be traversed: the function will never be passed the container objects. If children is null or undefined, returns null or undefined rather than an array.

#### React.Children.forEach

**React.Children.forEach(children, function[(thisArg)])**

Like React.Children.map() but does not return an array.

#### React.Children.count

**React.Children.count(children)**

Returns the total number of components in children, equal to the number of times that a callback passed to map or forEach would be invoked.

#### React.Children.only

**React.Children.only(children)**

Verifies that children has only one child (a React element) and returns it. Otherwise this method throws an error.

**Note**: React.Children.only() does not accept the return value of React.Children.map() because it is an array rather than a React element.

#### React.Children.toArray

**React.Children.toArray(children)**

Returns the children opaque data structure as a flat array with keys assigned to each child. Useful if you want to manipulate collections of children in your render methods, especially if you want to reorder or slice this.props.children before passing it down.

## Hot Module Replacement in Create-React-App

Hot Module Replacement (or HMR) is a Webpack feature that allows "on the fly" module updates for your app. Create-React-App uses webpack internally, but HMR working out-of-the-box only for stylesheets (and not for React components)

It lets you see your code changes instantaneously without reloading the page

It does not run automatically on any code, developers need to explicitly mark the module that can be updated on the fly and how these update should handle

There are library available that automate this

**Create-React-App**  is official command line for building new React application

Create React App have hot module replacement automatically for stylesheets but not for React Components

Solution:

1. Create a new react app
2. Make changes to index file
3. Page is updating without page refresh
4. But one issue, introduce a state in the component and update it by some click event
5. After that make changes in the component, you will observe that state value reset
6. To fix this issue, we can use **react-hot-loader** npm module

## Differentiate between Real DOM and Virtual DOM.

|  |  |
| --- | --- |
| **Real DOM** | **Virtual DOM** |
| It updates slowly. | It updates faster. |
| Can directly update HTML | Can’t directly update HTML. |
| Creates a new DOM if element updates. | Updates the JSX if element updates. |
| DOM manipulation is very expensive. | DOM manipulation is very easy. |
| Too much of memory wastage. | No memory wastage. |

## How is React different from Angular?

|  |  |  |
| --- | --- | --- |
| **TOPIC** | **REACT** | **ANGULAR** |
| **ARCHITECTURE** | Only the View of MVC | Complete MVC |
| **RENDERING** | Server side rendering | Client side rendering |
| **DOM** | Uses virtual DOM | Uses real DOM |
| **DATA BINDING** | One-way data binding | Two-way data binding |
| **DEBUGGING** | Compile time debugging | Run time debugging |
| **AUTHOR** | Facebook | Google |

## What’s the difference between client-side rendering and server-side rendering?

**In Client-side rendering**, your browser downloads a minimal HTML page. It renders the JavaScript and fills the content into it.

**Server-side rendering**, on the other hand, renders the React components on the server. The output is HTML content.

## Server Side Rendering

If pages aren't showing up on Google and don't look good when posted to Facebook. Seems solvable, right?

You figure out that to solve this you'll need to render your React pages from the server on initial load so that crawlers from search engines and social media sites can read your markup.

1. React library is itself isomorphic also known as universal, i.e it does not contain browser specific code it is only javascript
2. **renderToString()** function that we can use to generate markup on the server
3. Runs react component on server
4. We use JSX in react so need compiler to convert it into plain JS and front-end module system is not same as node module system. Babel allows us to do both the things

### The Benefits of Server-Side Rendering

1. server-side rendering is always recommended if you want to ensure **good SEO** and **compatibility with other services like Facebook, Twitter.**
2. **server-side rendering displays pages faster**. With server-side rendering, your server's response to the browser is the HTML of your page that is ready to be rendered so the browser can start rendering without having to wait for all the JavaScript to be downloaded and executed. There's no "white page" while the browser downloads and executes the JavaScript and other assets needed to render the page, which is what might happen in an entirely client-rendered React site.
3. Page is sent pre-populated from the server

**Implementation**

Create-react-app does not support server side rendering. You should use alternative or configured it for SSR

<https://github.com/ac13793/server-side-rendering-react-poc>

#### Server Side Rendering: Data Fetching & Routing

We generally fetch data in **componentDidMount** method but when we do SSR no DOM will be available because we use **renderToString()**  which does not mount to DOM so **componentDidMount** does not call.

To solve this, we get initial data by calling API on server side

## Synthetic events in React

Synthetic events are the objects which act as a cross-browser wrapper around the browser’s native event. They combine the behavior of different browsers into one API. This is done to make sure that the events show consistent properties across different browsers.

**Note:** As of v0.14, returning false from an event handler will no longer stop event propagation. Instead, e.stopPropagation() or e.preventDefault() should be triggered manually, as appropriate.

### Event Pooling

The SyntheticEvent is pooled. This means that the SyntheticEvent object will be reused and all properties will be nullified after the event callback has been invoked. This is for performance reasons. As such, **you cannot access the event in an asynchronous way.**

**Note:** If you want to access the event properties in an asynchronous way, you should call **event.persist()** on the event, which will remove the synthetic event from the pool and allow references to the event to be retained by user code.

## How ReactJS works

### Model-View-Controller

**Model -** Manages the data and rules of the application (React Component)

**View -** The output, the React application, the browser DOM (render())

**Controller -** Takes user input and converts them into commands for the Model or View layer (click events or API requests)

### Reconciliation

It is important to remember that the reconciliation algorithm is an implementation detail. React could re render the whole app on every action; the end result would be the same. Just to be clear, rerender in this context means calling *render* for all components, it doesn’t mean React will unmount and remount them. It will only apply the differences following the rules

#### Diffing Algorithms

React implements a heuristic algorithm based on two assumptions:

1. Two elements of different types will produce different trees
2. Using key you hint child element as stable. No need to re-render where keys do not change
3. Created completely from scratch on every setState

Any component below the root will also get unmounted and have their state destroyed. E.g when diffing

<div>

<Counter />

</div>

With

<span>

<Counter />

</span>

This will destroy the old Counter and remount a new one

* When comparing two React DOM elements of the same type, React looks at the attributes of both, keeps the same underlying DOM node, and only updates the changed attributes. Works the same way for the styles tag
* React traverse the tree breadth first
* This ensures that a node isn’t added to the update list if one of its parents also need to be updated

### Browser DOM Update

* Once react creates a new virtual DOM and diff it vs.the old one, it creates a list of the minimum possible changes to the browser DOM.
* Once it completes its list, it will fire off all of the changes one after the other as fast as possible
* Accomplish this in one continuous write cycle without any reflow until the end
* Reflow is the process that the browser performs to re-calculate the positions, geometries, and colors of elements on the page

### shouldComponentUpdate

* shouldComponentUpdate() is life cycle hook that React gives us so we can speed up our app
* Return false to block the update of this component

## applyMiddleware

Middleware lets you wrap the store's **dispatch** method for fun and profit. The key feature of middleware is that it is composable. Multiple middleware can be combined together, where each middleware requires no knowledge of what comes before or after it in the chain.

The most common use case for middleware is to support asynchronous actions without much boilerplate code or a dependency on a library like Rx. It does so by letting you **dispatch async actions** in addition to normal actions.

For example, **redux-thunk** lets the action creators invert control by dispatching functions. They would **receive dispatch as an argument and may call it asynchronously**. Such functions are called thunks. Another example of middleware is redux-promise. It lets you dispatch a Promise async action, and dispatches a normal action when the Promise resolves.

## Create-react-app

Create-react-app is a command line tool for scaffolding out a react application. It has babel and webpack pre configured and has best practice as per the react community. Just run project using **npm start** command

## Controlled Forms

1. Make the React component to controlled the form that it renders
   1. Single source of truth
   2. Tying the form state to component state
   3. Controlled component
2. Every state mutation will have an associated handler function

<https://github.com/ac13793/React-JS-Learning/commit/022365ee1dcc590adec99eed00cb5f15d1f55e33>

<https://github.com/ac13793/React-JS-Learning/commit/51fe6fce7614900b5078c746e6a108748d1dbbc2>

## Uncontrolled Forms

1. Ideally you should implement forms within controlled components
2. Sometimes this approach may be too tedious
3. Uncontrolled component approach allows you to handle the form data by the DOM itself
4. Instead of writing an event handler for every state update, **use a ref to get form values from the DOM**
5. More easier to integrate React with non-React code

<https://github.com/ac13793/React-JS-Learning/commit/99d8e0562df06d488cb95518c78c04ae3d39b2b4>

## React-Redux-Form

* A versatile, fast, and intuitive library for creating complex and performant forms in React and Redux
* Collection of reducer creators and action creators
* Form data stored in Redux store in a model
* Validation support for forms

When you use this library you don’t need to do below things:

1. No need to manage state for form fields
2. No need to create function for handle input change to update state for form fields
3. No need to create function for handle blur to show validation errors
4. No need to create function to validate form fields

<https://github.com/ac13793/React-JS-Learning/commit/f0c64690d595b15a263850f59a7f47e9fba8b0f5>

<https://github.com/ac13793/React-JS-Learning/commit/9c753e82f42ccae0827af2976d17453e392c6157>

## Cross-Fetch

1. Cross-Fetch library provides support for Fetch both in Node applications and Browsers
   1. Uses whatwg-fetch polyfill in browsers that do not have fetch support
   2. Uses node-fetch for Node applications on the server
2. To use in the application:

**import fetch from ‘cross-fetch’;**

1. Fetch alternatives: Axios, Superagent

## How to make AJAX calls in React

### How can I make an AJAX call?

You can use any AJAX library you like with React. Some popular ones are Axios, jQuery AJAX, and the browser built-in window.fetch.

### Where in the component lifecycle should I make an AJAX call?

You should populate data with AJAX calls in the **componentDidMount** lifecycle method. This is so you can use setState to update your component when the data is retrieved.

**Example: Using AJAX results to set local state**

The component below demonstrates how to make an AJAX call in **componentDidMount** to populate local component state.

The example API returns a JSON object like this:

class MyComponent extends React.Component {

constructor(props) {

super(props);

**this.state = {**

**error: null,**

**isLoaded: false,**

**items: []**

**};**

}

**componentDidMount() {**

**fetch("https://api.example.com/items")**

**.then(res => res.json())**

**.then((result) => {**

**this.setState({**

**isLoaded: true,**

**items: result.items**

**});**

**},**

**// Note: it's important to handle errors here**

**// instead of a catch() block so that we don't swallow**

**// exceptions from actual bugs in components.**

**(error) => {**

**this.setState({**

**isLoaded: true,**

**error**

**});**

**}**

**)**

**}**

render() {

const { error, isLoaded, items } = this.state;

if (error) {

return <div>Error: {error.message}</div>;

} else if (!isLoaded) {

return <div>Loading...</div>;

} else {

return (

<ul>

{items.map(item => (

<li key={item.name}>

{item.name} {item.price}

</li>

))}

</ul>

);

}

}

}

## React Fiber

React Fiber is an ongoing reimplementation of React's core algorithm.

The goal of React Fiber is to increase its suitability for areas like animation, layout, and gestures.

**Features in React Fiber**

1. Its headline feature is **incremental rendering**: the ability to split rendering work into chunks and spread it out over multiple frames.
2. Other key features include the ability to pause, abort, or reuse work as new updates come in; the ability to assign priority to different types of updates; and new concurrency primitives.
3. Removed adding extra div/span just to make a root element. You don’t need a root element to create UI template.
4. New reconciler focused on Task Scheduling

## React Polyfill

Polyfills are scripts which are providing standard APIs which are not supported yet by all the browsers and patching it conditional. So you will check if functionality is not there then you will provide a workaround to run it successfully across all the browsers

## Class Properties

Use class properties to clean up your classes and React components

Fed up of binding event handlers in a constructor? Forget to declare propTypes or defaultProps in your React component? Start using class properties now!

you can use the **class-properties-transform** plugin for Babel.

Example:

When you want to access *this* inside a class methods of your React component, you need to bind it to your constructor:

class Button extends Component {

constructor(props) {

super(props);

this.state = { clicked: false };

**this.handleClick = this.handleClick.bind(this);**

}

handleClick() {

this.setState({ clicked: true });

}

render() {

return <button onClick={this.handleClick}>Click Me!</button>;

}

}

Binding *this* to *handleClick* in the constructor allows us to use *this.setState* from Component inside handleClick. Without *this* binding, *this* is re-scoped for handleClick and loses context of the component’s setState method.

But this is completely unnecessary, extra code!

You can clean up this ugliness by using some new ES6+ features. Here is the same component, rewritten using class properties and arrow functions to **avoid binding this** to handleClick:

class Button extends Component {

state = { clicked: false };

**handleClick = () => this.setState({ clicked: true });**

render() {

return <button onClick={this.handleClick}>Click Me!</button>;

}

}

## Advanced React

### Contexts

Context makes it possible to pass data through the component hierarchy, without needing intermediate components to know about it. It can be useful for data that never (or rarely) change, such as theming and localization, global variables

To implement context that will be accessible by any child component

1. Declare **childContextTypes**
2. Set up **getChildContext** method which will set the value for these properties

App.js

import React, { Component, **PropTypes** } from 'react';

import Panel from './components/Panel';

import './App.css';

import en from './locales/en.json';

import pt from './locales/pt.json';

const locales = {en, pt};

class App extends Component {

**static childContextTypes = {**

**locale: PropTypes.object**

**}**

state = {

currentLocale: 'en'

}

**getChildContext() {**

**return {locale: locales[this.state.currentLocale]}**

**}**

changeLocale(locale){

this.setState({currentLocale: locale})

}

render() {

return (

<div>

<nav>

<a onClick={() => this.changeLocale('en')}>🇺🇸</a>

<a onClick={() => this.changeLocale('pt')}>🇧🇷</a>

</nav>

<Panel />

</div>

);

}

}

export default App;

To consume context in the child component

1. White list the properties that you want to access in **contextTypes**
2. Use **this.context**  to access the context properties

import React, { Component, **PropTypes** } from 'react';

class ContentPanel extends Component {

**static contextTypes = {**

**locale: PropTypes.object**

**}**

render() {

**const { locale } = this.context;**

return (

<div className="contentPanel">

<h1>{locale.header}</h1>

<p> {locale.text} </p>

<button>{locale.buttonLabel}</button>

<footer>ContentPanel.js</footer>

</div>

);

}

}

export default ContentPanel;

Context has 2 problems -

1. The fact that the API is experimental (it will end up changing)
2. The fact that updates might not propagate if any component in the middle of the hierarchy implements *ShouldComponentUpdate*.

To use context safely, we can use Higher Order Component to solve first problems

Instead of directly using context in the individual components we should wraps context in Higher Order Component and reuse behaviour

WithLocalHOC.js

import React, { Component, **PropTypes** } from 'react';

const WithLocaleHOC = (WrappedComponent) => {

return class WithLocaleHOC extends Component {

**static contextTypes = {**

**locale: PropTypes.object**

**}**

componentDidMount() {

this.context.locale.subscribe(() => this.forceUpdate());

}

render() {

**const { locale } = this.context;**

return <WrappedComponent {...this.props} **locale={locale}** />

}

}

}

export default WithLocaleHOC;

ContentPanel.js

import React, { Component, **PropTypes** } from 'react';

**import WithLocaleHOC from './WithLocaleHOC';**

class ContentPanel extends Component {

**static propTypes = { // As we are receiving these as props from HOC**

**locale: PropTypes.object**

**}**

render() {

**const { locale } = this.props; // As we are receiving these as props from HOC**

return (

<div className="contentPanel">

<h1>{locale.header}</h1>

<p> {locale.text} </p>

<button>{locale.buttonLabel}</button>

<footer>ContentPanel.js</footer>

</div>

);

}

}

export default **WithLocaleHOC(ContentPanel);**

To solve second problem, updates that don’t propagate

React provide **ComponentShouldUpdate**  life cycle method to short circuit rendering process to improve performance

If any component in the middle implemented **ComponentShouldUpdate** and returnings false, then all child component will never render more than once

The reason, **ComponentShouldUpdate**  is returning false because these component are not notified that **context** has updated

So we should somehow notify components that context has been updated

Previously, for every state change we are creating new object . We need to make context shallowing mutable

Locale.js

import en from './en.json';

import pt from './pt.json';

const locales = {en, pt};

class Locale {

constructor(language) {

**this.strings = locales[language];**

**this.subscriptions = [];**

}

setLanguage(language) {

**this.strings = locales[language];**

**this.subscriptions.forEach(cb => cb());**

}

**subscribe(callback) {**

**this.subscriptions.push(callback);**

**}**

}

export default Locale;

App.js

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

import Panel from './components/Panel';

**import Locale from './locales/Locale';**

import './App.css';

class App extends Component {

constructor(props, context) {

super(props, context);

**this.locale = new Locale('en');**

}

**static childContextTypes = {**

**locale: PropTypes.object**

**}**

state = {

currentLocale: 'en'

}

**componentWillUpdate(nextProps, nextState) {**

**this.locale.setLanguage(nextState.currentLocale);**

**}**

**getChildContext() {**

**return {locale: this.locale}**

**}**

changeLocale(locale){

**this.setState({currentLocale: locale})**

}

render() {

return (

<div>

<nav>

<a onClick={() => this.changeLocale('en')}>🇺🇸</a>

<a onClick={() => this.changeLocale('pt')}>🇧🇷</a>

</nav>

<Panel />

</div>

);

}

}

export default App;

WithLocalHOC.js

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

const WithLocaleHOC = (WrappedComponent) => {

return class WithLocaleHOC extends Component {

**static contextTypes = {**

**locale: PropTypes.object**

**}**

**componentDidMount() {**

**this.context.locale.subscribe(() => this.forceUpdate());**

**}**

render() {

**const { locale } = this.context;**

return <WrappedComponent {...this.props} locale={locale} />

}

}

}

export default WithLocaleHOC;

ContentPanel.js

import React, { Component, **PropTypes** } from 'react';

**import WithLocaleHOC from './WithLocaleHOC';**

class ContentPanel extends Component {

**static propTypes = { // As we are receiving these as props from HOC**

**locale: PropTypes.object**

**}**

render() {

**const { locale } = this.props; // As we are receiving these as props from HOC**

return (

<div className="contentPanel">

**<h1>{locale.strings.header}</h1>**

**<p> {locale.strings.text} </p>**

**<button>{locale.strings.buttonLabel}</button>**

</div>

);

}

}

export default **WithLocaleHOC(ContentPanel);**

### Error Boundaries

**Why?** In the past, JavaScript errors inside components used to corrupt React’s internal state and cause it to emit cryptic errors on next renders. These errors were always caused by an earlier error in the application code, but React did not provide a way to handle them gracefully in components, and could not recover from them.

A JavaScript error in a part of the UI shouldn’t break the whole app. To solve this problem for React users, React 16 introduces a new concept of an “error boundary”.

**What?** Error boundaries are React components that **catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI** instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

**Note:** Error boundaries do not catch errors for:

1. Event handlers
2. Asynchronous code (e.g. setTimeout or requestAnimationFrame callbacks)
3. Server side rendering
4. Errors thrown in the error boundary itself (rather than its children)

A class component becomes an error boundary if it defines a new lifecycle method called **componentDidCatch(error, info)**

class ErrorBoundary extends React.Component {

constructor(props) {

super(props);

this.state = { hasError: false };

}

**componentDidCatch(error, info)** {

// Display fallback UI

this.setState({ hasError: true });

// You can also log the error to an error reporting service

logErrorToMyService(error, info);

}

render() {

if (this.state.hasError) {

// You can render any custom fallback UI

return <h1>Something went wrong.</h1>;

}

return this.props.children;

}

}

Then you can use it as a regular component:

<ErrorBoundary>

<MyWidget />

</ErrorBoundary>

The componentDidCatch() method works like a JavaScript catch {} block, but for components. Only class components can be error boundaries. In practice, most of the time you’ll want to declare an error boundary component once and use it throughout your application.

### Forwarding Refs

Ref forwarding is a technique for automatically passing a **ref** through a component to one of its children. This is typically not necessary for most components in the application. However, it can be useful for some kinds of components, especially in reusable component libraries.

const TextInput = **React.forwardRef((props, ref) => (<input ref={ref} />));**

class RefExample extends React.Component {

onButtonClick = () => {

if (this.input) {

this.input.focus();

}

}

setInputRef = (ref) => (this.input = ref);

render() {

return (

<div>

<button onClick={this.onButtonClick}>Click here!!</button>

<TextInput **ref={this.setInputRef}** />

</div>

);

}

}

ReactDOM.render(<RefExample />, document.querySelector('#app'));

**Note:** The second **ref** argument only exists when you define a component with *React.forwardRef* call. Regular functional or class components don’t receive the ref argument, and ref is not available in props either.

Ref forwarding is not limited to DOM components. You can forward refs to class component instances, too.

### Fragments

A common pattern in React is for a component to return multiple elements. Fragments let you group a list of children without adding extra nodes to the DOM.

class Table extends React.Component {

render() {

return (

<table>

<tr>

**<Columns />**

</tr>

</table>

);

}

}

class Columns extends React.Component {

render() {

return (

**<React.Fragment>**

**<td>Hello</td>**

**<td>World</td>**

**</React.Fragment>**

);

}

which results in a correct <Table /> output of:

}

<table>

<tr>

<td>Hello</td>

<td>World</td>

</tr>

</table>

### Render Props

The term “render prop” refers to a simple technique for sharing code between React components using a prop whose value is a function.

A component with a render prop takes a function that returns a React element and calls it instead of implementing its own render logic.

<DataProvider render={data => (  
 <h1>Hello {data.target}</h1>  
)}/>

#### Using Props Other Than render

It’s important to remember that just because the pattern is called “render props” you don’t have to use a prop named render to use this pattern. In fact, **any prop that is a function that a component uses to know what to render is technically a “render prop”**.

### Static Type Checking

Static type checkers like **Flow** and **TypeScript** identify certain types of problems before you even run your code. They can also improve developer workflow by adding features like auto-completion.

#### Flow

Flow is a static type checker for your JavaScript code. It is developed at Facebook and is often used with React. It lets you annotate the variables, functions, and React components with a special type syntax, and catch mistakes early.

To use Flow, you need to:

1. Add Flow to your project as a dependency.
2. Ensure that Flow syntax is stripped from the compiled code.
3. Add type annotations and run Flow to check them.

If your project was set up using Create React App, congratulations! The Flow annotations are already being stripped by default so you don’t need to do anything

### Strict Mode

**StrictMode** is a tool for highlighting potential problems in an application. Like Fragment, StrictMode does not render any visible UI. It activates additional checks and warnings for its descendants.

**Note:** Strict mode checks are run in development mode only; they do not impact the production build.

import React from 'react';

function ExampleApplication() {

return (

<div>

<Header />

**<React.StrictMode>**

<div>

<ComponentOne />

<ComponentTwo />

</div>

**</React.StrictMode>**

<Footer />

</div>

);

}

### Typechecking With PropTypes

As your app grows, you can catch a lot of bugs with typechecking. For some applications, you can use JavaScript extensions like Flow or TypeScript to typecheck your whole application. But even if you don’t use those, React has some built-in typechecking abilities. To run typechecking on the props for a component, you can assign the special **propTypes** property:

**import PropTypes from 'prop-types';**

class Greeting extends React.Component {

render() {

return (

<h1>Hello, {this.props.name}</h1>

);

}

}

**Greeting.propTypes = {**

**name: PropTypes.string**

**};**

In the case of static propTypes, propTypes need to be declared on the class itself, not on the instance of the class.

<https://reactjs.org/docs/typechecking-with-proptypes.html>

PropTypes exports a range of validators that can be used to make sure the data you receive is valid. In this example, we’re using PropTypes.string. When an invalid value is provided for a prop, a warning will be shown in the JavaScript console. For performance reasons, propTypes is only checked in development mode.

#### Default Prop Values

You can define default values for your **props** by assigning to the special **defaultProps** property:

class Greeting extends React.Component {

render() {

return (

<h1>Hello, {this.props.name}</h1>

);

}

}

// Specifies the default values for props:

**Greeting.defaultProps = {**

**name: 'Stranger'**

**};**

// Renders "Hello, Stranger":

ReactDOM.render(

<Greeting />,

document.getElementById('example')

);

If you are using a Babel transform-class-properties , you can also declare props and defaultProps as static property within a React component class.

class Greeting extends React.Component {

**static props = {**

**name: PropTypes.string**

**}**

**static defaultProps = {**

**name: 'stranger'**

**}**

render() {

return (

<div>Hello, {this.props.name}</div>

)

}

}

## The Model View Controller Framework

**Model**

1. manages the behavior and data of the application domain
2. Responds to requests for information about its state(usually from the view)
3. Responds to instructions to change state (usually from the controller)
4. In event-driven systems, the model notifies observers (usually views) when the information changes so that they can react

**View**

1. Renders the model into a form suitable for interaction, typically a user interface element
2. Multiple views can exist for a single model for different purposes
3. A viewport typically has a one to one correspondence with a display surface and knows how to render to it

**Controller**

1. Receives user input and initiates a response by making calls on model objects
2. A controller accepts input from the user and instructs the model and viewport to perform actions based on that input

**Disadvantages of MVC Architecture**

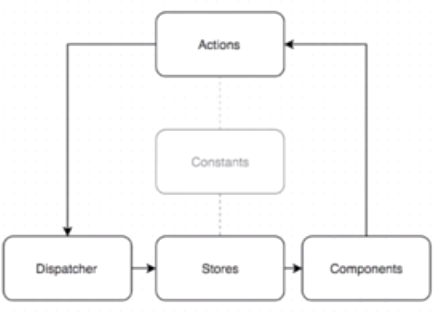
1. View depends both on Controller and Model
2. Model is doing too much work
3. In this design pattern the more amount of work is done by controller which makes the mvc design pattern complex.

## Flux

### Flux

Flux is the application architecture that Facebook, AddThis, and many other companies use for building client-side web applications. It complements React's composable view components by utilizing a unidirectional data flow. It's more of a pattern rather than a formal framework

React by default does not give any specific way to manage all the data. Flux is not a framework, it is a pattern that React has laid out for building react application.



In flux pattern, components fire off actions and listen for Stores. Whenever store updates component re renders itself. Actions inform about something happened by dispatcher and dispatcher job is to aware every store that hearing about the actions. Store may be ignore the actions and update/change the state

### Flux Store

**ToDoStore.js**

import { EventEmitter } from "events";

class TodoStore extends EventEmitter {

constructor() {

super()

this.todos = [

{

id: 113464613,

text: "Go Shopping",

complete: false

},

{

id: 235684679,

text: "Pay Water Bill",

complete: false

},

];

}

getAll() {

return this.todos;

}

}

const todoStore = new TodoStore;

export default todoStore;

**ToDos.js**

import React from "react";

import Todo from "../components/Todo";

**import TodoStore from "../stores/TodoStore";**

export default class Todos extends React.Component {

constructor() {

super();

this.state = {

**todos: TodoStore.getAll()**

};

}

render() {

**const { todos } = this.state;**

const TodoComponents = todos.map((todo) => {

return <Todo key={todo.id} {...todo}/>;

});

return (

<div>

<button onClick={this.reloadTodos.bind(this)}>Reload!</button>

<h1>Todos</h1>

<ul>{TodoComponents}</ul>

</div>

);

}

}

### Flux Store Events

**ToDoStore.js**

import { EventEmitter } from "events";

class TodoStore extends EventEmitter {

constructor() {

super()

this.todos = [

{

id: 113464613,

text: "Go Shopping",

complete: false

},

{

id: 235684679,

text: "Pay Water Bill",

complete: false

},

];

}

**createTodo(text) {**

**const id = Date.now();**

**this.todos.push({**

**id,**

**text,**

**complete: false,**

**});**

**this.emit("change");**

**}**

getAll() {

return this.todos;

}

}

const todoStore = new TodoStore;

export default todoStore;

**ToDos.js**

import React from "react";

import Todo from "../components/Todo";

import TodoStore from "../stores/TodoStore";

export default class Todos extends React.Component {

constructor() {

super();

this.getTodos = this.getTodos.bind(this);

this.state = {

todos: TodoStore.getAll(),

};

}

**componentWillMount() {**

**TodoStore.on("change", this.getTodos);**

**}**

**getTodos() {**

**this.setState({**

**todos: TodoStore.getAll(),**

**});**

**}**

render() {

const { todos } = this.state;

const TodoComponents = todos.map((todo) => {

return <Todo key={todo.id} {...todo}/>;

});

return (

<div>

<button onClick={this.reloadTodos.bind(this)}>Reload!</button>

<h1>Todos</h1>

<ul>{TodoComponents}</ul>

</div>

);

}

}

### The Flux Dispatcher

So we are getting initial data from Store and listening to Store whenever we are changing the Store. Now, lets add dispatcher. For that we will register dispatcher with the Store so any Action that comes through the pipeline, our store will get notified

Install **flux** module using below command:

Npm install flux --save-dev

**Dispatcher.js**

import { Dispatcher } from "flux";

export default new Dispatcher;

**ToDoStore.js**

import { EventEmitter } from "events";

**import dispatcher from "../dispatcher";**

class TodoStore extends EventEmitter {

constructor() {

super()

this.todos = [

{id: 113464613,

text: "Go Shopping",

complete: false},

{id: 235684679,

text: "Pay Water Bill",

complete: false},

];

}

createTodo(text) {

const id = Date.now();

this.todos.push({ id, text, complete: false });

this.emit("change");

}

getAll() {

return this.todos;

}

**handleActions(action) {**

**switch(action.type) {**

**case "CREATE\_TODO": {**

**this.createTodo(action.text);**

**break;**

**}**

**case "RECEIVE\_TODOS": {**

**this.todos = action.todos;**

**this.emit("change");**

**break;**

**}**

**}**

**}**

}

const todoStore = new TodoStore;

**dispatcher.register(todoStore.handleActions.bind(todoStore));**

export default todoStore;

**App.js**

**dispatcher.dispatch({type: "CREATE\_TODO", text:"new tod"})**

We registered dispatcher with **todoStore,** so whenever dispatcher dispatch any action handle Action will be notified

### Flux Actions

**ToDoActions.js**

import dispatcher from "../dispatcher";

export function createTodo(text) {

dispatcher.dispatch({

type: "CREATE\_TODO",

text,

});

}

export function deleteTodo(id) {

dispatcher.dispatch({

type: "DELETE\_TODO",

id,

});

}

**ToDos.js**

import React from "react";

import Todo from "../components/Todo";

**import \* as TodoActions from "../actions/TodoActions";**

import TodoStore from "../stores/TodoStore";

export default class Todos extends React.Component {

constructor() {

super();

this.getTodos = this.getTodos.bind(this);

this.state = {

todos: TodoStore.getAll()

};

}

componentWillMount() {

TodoStore.on("change", this.getTodos);

}

getTodos() {

this.setState({

todos: TodoStore.getAll()

});

}

**createTodos = () => {**

**TodoActions.createTodo("My First Action");**

**};**

render() {

const { todos } = this.state;

const TodoComponents = todos.map(todo => {

return <Todo key={todo.id} {...todo} />;

});

return (

<div>

**<button onClick={this.createTodos}>Create</button>**

<h1>Todos</h1>

<ul>{TodoComponents}</ul>

</div>

);

}

}

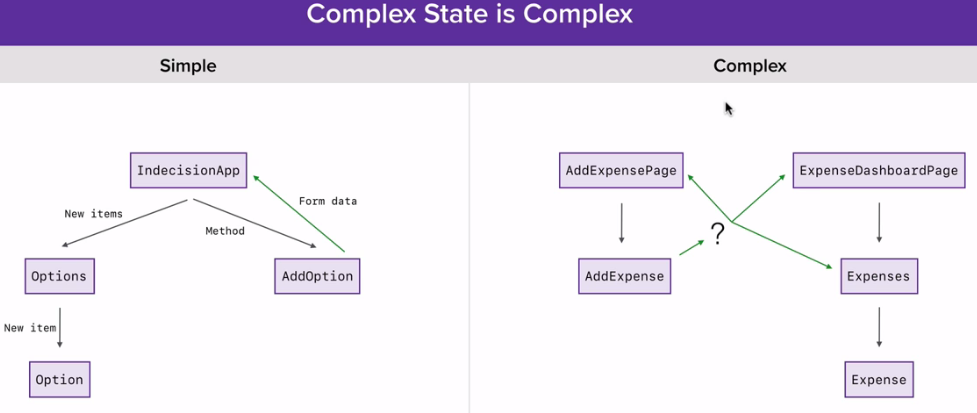
### React & Flux Memory Leaks

Whenever we bind any event in the component, it will keep on binding multiple time whenever that component re renders

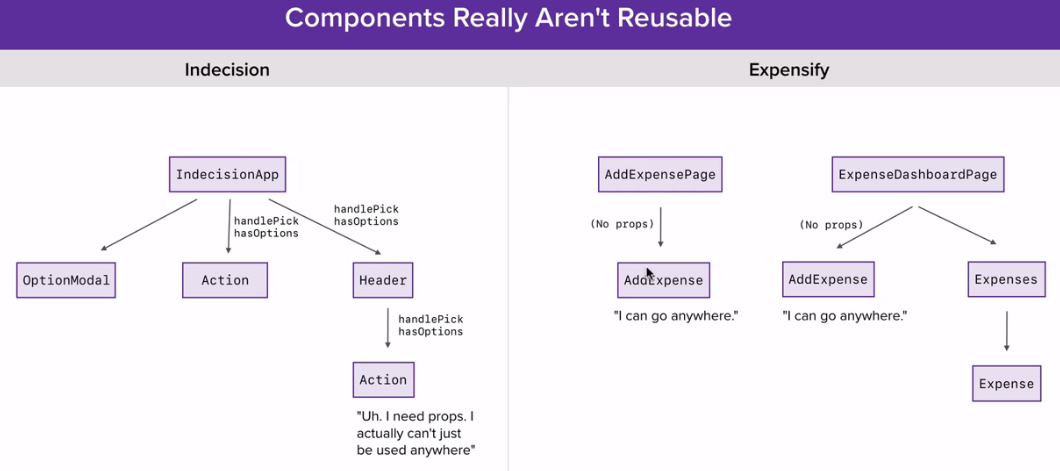
**Solution:** Unbind those event in **componentWillUnmount()** life cycle methods

## Redux

### Why Do We Need Something Like Redux



For complex application there will not be a parent component that maintains the data and sharing data is not that easy in complex applications

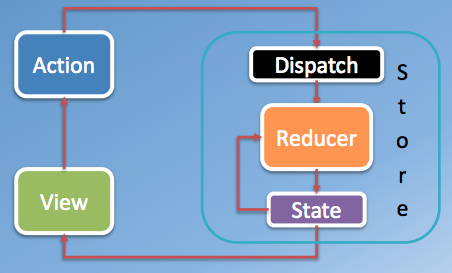


Where do I store my app state in an complex React app?

How do I create components that are actually reusable?

### What is Redux?

1. Redux is a state container which is exactly what our class based components are. There is an object, we can change the object, we can read from the object
2. Components does not communicate among each other but they communicate with redux container
3. Predictable state container for JavaScript apps
4. Inspired by Flux, Elm, Immutable
5. Makes state mutations predictable
6. Single source of truth
   1. Single state object tree within a single store
   2. State is read-only (only getters, no setters)
   3. Changes should only be done through actions
   4. Changes are made with pure functions
7. Take previous state and action and return next state
8. No mutation of the previous state
9. Single store and single state tree enables powerful techniques:
   1. Logging
   2. API handling
   3. Undo/redo
   4. State persistence
   5. “time-travel debugging”
10. Unidirectional data flow

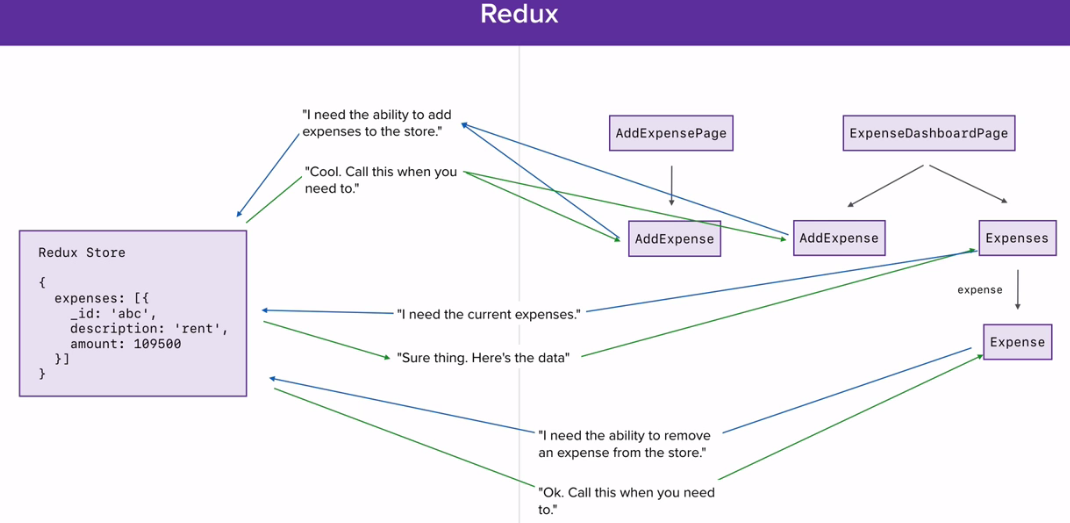


### Redux Concepts

* **State**: stored in plain JS object
* **Action**: plain JS object with a type field that specifies how to change something in the state
* **Reducer**: pure functions that take the current state and action and return a new state
  + Update data immutably (do not modify inputs)

### Redux Store

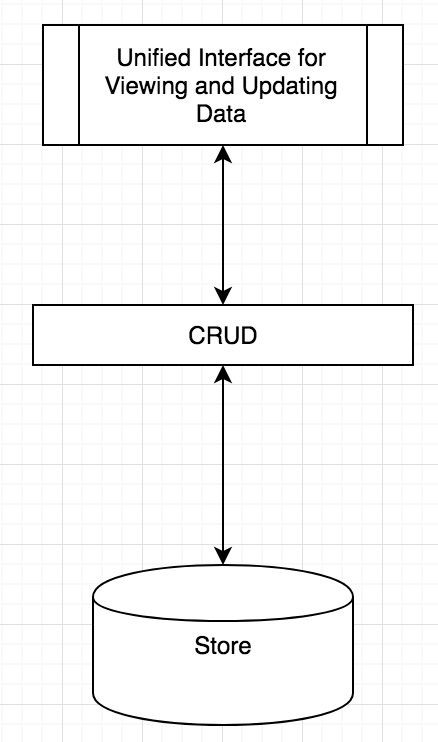
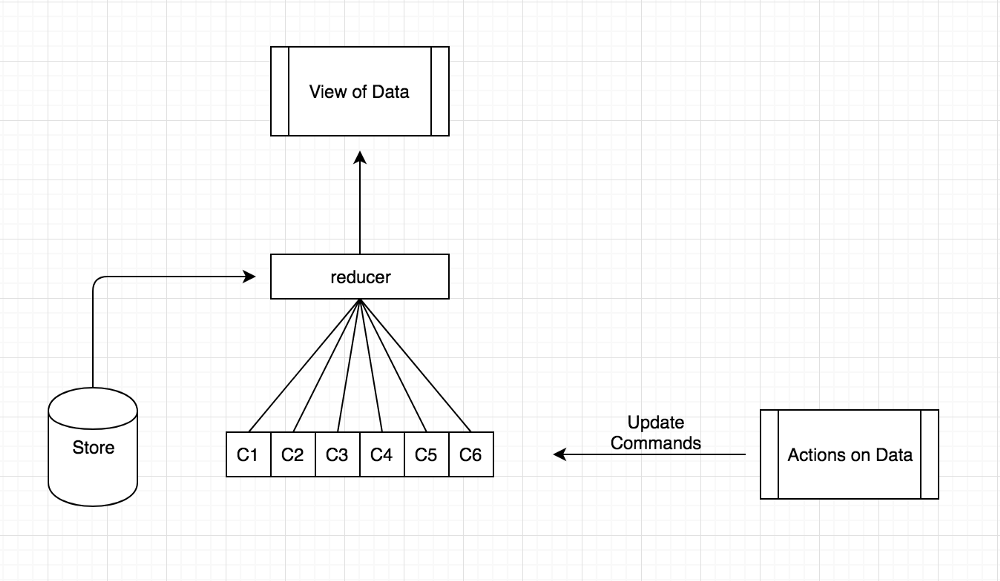
* Holds the current state value
* Created using createStore()
* Supplies three methods:
  + dispatch(): states state update with the provided action object
  + getState(): returns the current stored state value
  + subscribe(): accepts a callback function that will be run every time an action is dispatched



### Understanding Redux as a CQRS system

**CQRS — Command Query Responsibility Segregation** is a pattern where we use different interfaces to read and update data. This is as opposed to CRUD — Create Read Update Delete, where we use a single interface to perform all the reads and updates.

While implementing a CQRS system, all the commands are modelled as events. So whenever a user performs any action, a command is dispatched as an event and the event is stored in a store. Whenever we want to read data, we take the data at its initial state and apply all the commands to the data to get the most up to date data. Hopefully, the below diagram does a decent job of capturing the differences between CQRS and CRUD.



### Setting up Redux

Install redux using below command:

**npm install redux --save-dev**

import { createStore } from "redux";

const store = createStore((state = { count: 0 }) => {

return state;

});

// getState() returns current state object

console.log(store.getState());

To access redux we need to access **createStore** which tracks our data over time

When redux call this function first time, no state object present so it takes the default value

Dispatching Actions

import { createStore } from "redux";

const store = createStore((state = { count: 0 }, **action**) => {

switch **(action.type)** {

case "INCREMENT":

return {

count: state.count + 1

};

case "DECREMENT":

return {

count: state.count - 1

};

case "RESET":

return {

count: 0

};

default:

return state;

}

});

// getState() returns current state object

console.log(store.getState());

store.dispatch({

type: "INCREMENT"

});

store.dispatch({

type: "DECREMENT"

});

store.dispatch({

type: "RESET"

});

console.log(store.getState());

Action an object that gets sent to the store. **dispatch()** method allows us to send off Action object. Whenever you dispatch an action using **dispatch** method **createStore()** function gets called that takes **reducer** function as a parameter and reducer takes two parameter **state and action.** Reducer specify and change state as per the action

### Subscribing and Dynamic Actions

How can we actually watch changes to the redux store state?

Redux gives a method called **subscribe(),**  which gets called everytime store state changes

import { createStore } from "redux";

const store = createStore((state = { count: 0 }, action) => {

switch (action.type) {

case "INCREMENT":

return {

count: state.count + 1

};

case "DECREMENT":

return {

count: state.count - 1

};

default:

return state;

}

});

store.subscribe(() => {

console.log("Subscribe: " + store.getState());

});

store.dispatch({

type: "INCREMENT"

});

store.dispatch({

type: "DECREMENT"

});

Redux also gives method to unsubscribe the store object. **subscribe()** method returns a function that we can use it later to unsubscribe the store object

const unsubscribe = store.subscribe(() => {

console.log("Subscribe: " + store.getState());

});

store.dispatch({

type: "INCREMENT"

});

unsubscribe();

**Note: type** property is required to dispatch an action

import { createStore } from "redux";

const store = createStore((state = { count: 0 }, **action**) => {

switch (action.type) {

case "INCREMENT":

**const incrementBy =**

**typeof action.incrementBy === "number" ? action.incrementBy : 1;**

return {

count: state.count + incrementBy

};

default:

return state;

}

});

store.subscribe(() => {

console.log(store.getState());

});

store.dispatch({

type: "INCREMENT",

incrementBy: 5

});

### ES6 Object Destructuring

const person = {

name: "Ankit",

age: 26,

location: {

city: "Hyderabad",

temp: 92

}

};

// ES6 destructuring

// Renaming name to firstName and setting default value

const { name: firstName = "defalt name", age } = person;

console.log(`${firstName} is ${age}`);

const { city, temp: temprature } = person.location;

console.log(`It's ${temprature} in ${city}`);

### ES6 Array Destructuring

const address = [

"1299 S Juniper Street",

"Philadelphia",

"Pennsylvania",

"19147"

];

// Matching by position

const [, city, state = "New York"] = address;

console.log(` You are in ${city} ${state}`);

### Refactoring and Organizing

### Redux Actions

1. **Actions** are JavaScript objects that use type property to inform about the data that should be sent to the store.
2. payloads of information that send data from your application to the store
   1. Done through store.dispatch()
3. Plain JS object that must have
   1. A type property that indicates the type of action to be performed
      1. Best supported by defining action types as String constants
   2. Rest of the object contains the data necessary for the action (payload)

### Action Creators

1. Functions that create actions
   1. Encapsulate the process of creating the action objects
   2. Return the action object
   3. Resulting action object can be passed to the store through dispatch()

**Action generators**: functions that returns action objects

Action generators are preferable over inline action objects because if you accidentally types incorrect value in type property you will not get any error but if you type incorrect function name you will get error.

const incrementCount = ({ incrementBy = 1 } = {}) => ({

type: "INCREMENT",

incrementBy

});

const store = createStore((state = { count: 0 }, action) => {

switch (action.type) {

case "INCREMENT":

return {

count: state.count + action.incrementBy

};

default:

return state;

}

});

// store.dispatch({

// type: "INCREMENT",

// incrementBy: 5

// });

store.dispatch(incrementCount({ incrementBy: 5 }));

store.dispatch(incrementCount());

### Reducers

Actions describe the fact that something happened but don’t specify how the application’s state change in response. This is the job of reducers.

1. Reducers are pure functions i.e output is purely determined by the input
2. Takes two parameter: state and action
3. Calculate and returns updated state
4. Reducers should be able to take the previous state and action and return next state:
   1. Do not mutate state
      1. Make a copy and modify the copy before returning it
   2. Actions typically handled through a switch statement switching on the action type
   3. Return the previous state in the default case

**Note:** Redux initially trigger a action { type: '@@INIT' } to set up default state

const countReducers = (state = { count: 0 }, action) => {

switch (action.type) {

case "INCREMENT":

return {

count: state.count + action.incrementBy

};

case "DECREMENT":

const decrementBy =

return {

count: state.count - action.decrementBy

};

case "SET":

return {

count: action.count

};

case "RESET":

return {

count: 0

};

default:

return state;

}

};

const store = **createStore(countReducers());**

### Working with Multiple Reducers

import { createStore, **combineReducers** } from "redux";

// Expenses Reduer

const expenseReducerDefaultState = [];

const expenseReducer = (state = expenseReducerDefaultState, action) => {

switch (action.type) {

default:

return state;

}

};

// Filters Reducer

const filterReducerDefaultState = {

text: "",

sortBy: "date",

startDate: undefined,

endDate: undefined

};

const filterReducer = (state = filterReducerDefaultState, action) => {

switch (action.type) {

default:

return state;

}

};

// Store creation

**const store = createStore(**

**combineReducers({**

**expenses: expenseReducer,**

**filters: filterReducer**

**})**

**);**

console.log(store.getState());

### ES6 Spread Operator in Reducers

import { createStore, combineReducers } from "redux";

import uuid from "uuid"; // To generate random ids

// ADD\_Expense Action generator

const addExpense = ({

description = "",

note = "",

amount = 0,

createdAt = 0

} = {}) => ({

type: "ADD\_EXPENSE",

expense: {

id: uuid(),

description,

amount,

createdAt

}

});

// Remove expense Action generator

const removeExpense = ({ id } = {}) => ({

type: "REMOVE\_EXPENSE",

id

});

// Expenses Reduer

const expenseReducerDefaultState = [];

const expenseReducer = (state = expenseReducerDefaultState, action) => {

switch (action.type) {

case "ADD\_EXPENSE":

// state.push(action.expense); push changes the original array

// return state.concat(action.expense);

// ES6 spread operator: does not change the original array

**return [...state, action.expense];**

case "REMOVE\_EXPENSE":

return state.filter(({ id }) => id !== action.id);

default:

return state;

}

};

// Filters Reducer

const filterReducerDefaultState = {

text: "",

sortBy: "date",

startDate: undefined,

endDate: undefined

};

const filterReducer = (state = filterReducerDefaultState, action) => {

switch (action.type) {

default:

return state;

}

};

// Store creation

const store = createStore(

combineReducers({

expenses: expenseReducer,

filters: filterReducer

})

);

store.subscribe(() => {

console.log(store.getState());

});

const expenseOne = store.dispatch(

addExpense({ description: "rent", amount: 100 })

);

const expenseTwo = store.dispatch(

addExpense({ description: "Coffee", amount: 200 })

);

store.dispatch(removeExpense({ id: expenseOne.expense.id }));

### Spreading Objects

Install babel plugin to support rest-spread

npm install --save-dev babel-plugin-transform-object-rest-spread

Example:

const user = {

name: "jan",

age: 24

};

console.log({

...user,

location: "Philadelphia",

age: 27 // here we defined age property after user object so end result will be overridden by the age value

});

**Redux-expensify.js**

import { createStore, combineReducers } from "redux";

import uuid from "uuid";

...

// EDIT\_EXPENSE Action

const editExpense = (id, updates) => ({

type: "EDIT\_EXPENSE",

id,

updates

});

// Expenses Reduer

const expenseReducerDefaultState = [];

const expenseReducer = (state = expenseReducerDefaultState, action) => {

switch (action.type) {

...

case "EDIT\_EXPENSE":

return state.map(expense => {

if (expense.id === action.id) {

**return {**

**...expense,** // Passing entire current expense object having all the properties and will be overridden by action.updates object

**...action.updates**

**};**

} else {

return expense;

}

});

default:

return state;

}

};

// Store creation

const store = createStore(

combineReducers({

expenses: expenseReducer,

filters: filterReducer

})

);

store.subscribe(() => {

console.log(store.getState());

});

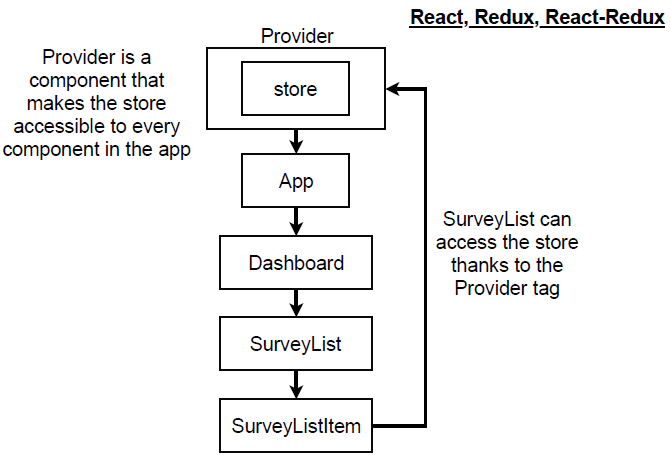
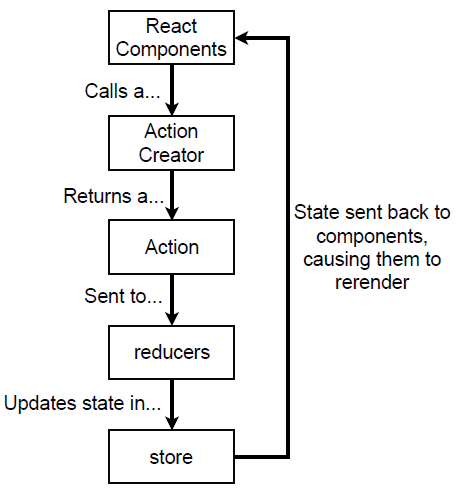
const expenseTwo = store.dispatch(

addExpense({ description: "Coffee", amount: 200 })

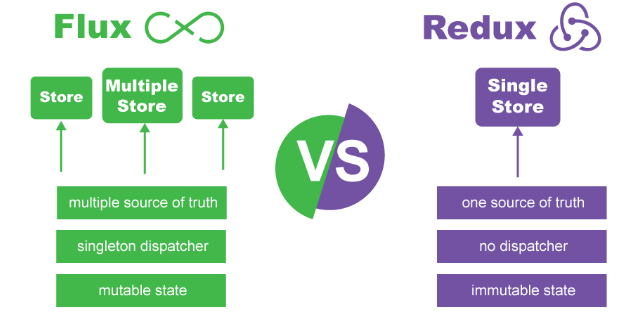
);

store.dispatch(editExpense(expenseTwo.expense.id, { amount: 500 }));

### Redux Overview



### Difference between redux and flux



## Redux Middleware and Redux Thunk

Redux Thunk [middleware](https://github.com/reduxjs/redux/blob/master/docs/advanced/Middleware.md) allows you to write action creators that return a function instead of an action

### Redux Middleware

1. Provides the capability to run code after an action is dispatched, but before it reaches the reducer
   1. Third-party extension point
   2. e.g., logging, async API calls
2. Middleware:
   1. Forms pipeline that wraps around the dispatch()
   2. Pass actions onward
   3. Restart the dispatch pipeline
   4. Access the store state
3. Middleware typically used for:
   1. Inspecting the actions and the state,
   2. Modify actions,
   3. Dispatch other actions,
   4. Stop actions from reaching the reducers, etc.
4. The **applyMiddleware()** function:
   1. Sets up the middleware pipeline
   2. Returns a “store enhancer” that is passed to createStore()

### Redux Thunk

Generally , Action creators returns Actions **Redux Thunk** library breaks that rule and allows you to write Action Creators that returns a function instead of an Action

If redux thunk sees that we return a function from Action Creators it will automatically call that function and pass in **dispatch** as an argument

1. In programming, a thunk is a subroutine used to inject an additional calculation in another subroutine
   1. Delay a calculation until its result is needed,
   2. Insert operations at the beginning or end of the other subroutine
2. **Middleware that allows you to write action creators that return a function instead of an action**
   1. Can be used to delay the dispatch of an action, or
   2. Dispatch only if a certain condition is met
3. **Inner function receives the dispatch() and getState() store methods**
4. Useful for complex synchronous logic
   1. Multiple dispatches
   2. Conditional dispatches
   3. Simple Async logic
5. Redux Saga: Uses ES6 generators to control pausable functions
   1. Complex async logic
   2. Ongoing “background thread” like processing behavior

<https://github.com/ac13793/React-JS-Learning/commit/97efaf6f972dfab99a961b6102922d62c2ae6b25>

## React with Redux

### The Higher Order Component

React-Redux is a library that allows us to connect redux stores to our react components and it makes heavy use of patterns known as higher order components

Higher order component is a component that renders another component(normal component)

HOC are custom components which wraps another component within it. They can accept any dynamically provided child component but they won’t modify or copy any behavior from their input components. You can say that HOC are ‘pure’ components.

**Advantages:**

1. Reuse code
2. **Render hijacking:** The concept of render hijacking is the ability to control what a component will output from another component". It actually means that you decorate your component by wrapping it into a Higher-Order component. By wrapping you can inject additional props or make other changes, which can cause changing logic of rendering. It does not actually "ENABLES" hjacking, but by using HOC you make your component behave in different way
3. Prop manipulation
4. Abstract state

import React from "react";

import ReactDOM from "react-dom";

const Info = props => (

<div>

<h1>Info</h1>

<p>The info is: {props.info}</p>

</div>

);

// Higher Order Component

const withAdminWarning = WrappedComponent => {

return props => (

<div>

{props.isAdmin && <p>This is private info.Please don't share it.</p>}

<WrappedComponent {...props} />

</div>

);

};

const AdminInfo = withAdminWarning(Info);

ReactDOM.render(

<AdminInfo info="These are the details" isAdmin={false} />,

document.getElementById("app")

);

### Connecting Store and Component with React-Redux

Install package using below command

**npm install react-redux --save-dev**

Step 1:

**Provider** component in react-redux library allow us to provide the store to all of the components that make our application so that individual component can have access to stores

**import { Provider } from "react-redux";**

...

const store = configureStore();

store.dispatch(addExpense({ description: "Water bill" }));

...

const jsx = (

**<Provider store={store}>**

**<AppRouter />**

**</Provider>**

);

ReactDOM.render(jsx, document.getElementById("app"));

Step2:

Create a higher order component using **connect** function provided by react-redux**.** **Connect** component connects your component to Redux store

**ExpenseList.js**

import React from "react";

**import { connect } from "react-redux";**

const ExpenseList = props => (

<div>

<h1>Expense list</h1>

{props.filters.text}

{props.expenses.length}

</div>

);

const mapStateToProps = state => {

return {

expenses: state.expenses,// here expenses is props that you will use in component as above and state.expenses is the state defined in your store

filters: state.filters

};

};

**export default connect(mapStateToProps)(ExpenseList);**

In connect function, we are passing the things that we want from the redux store to the component.

When ever store changes it will automatic re render the connected component

### Controlled Inputs for Filters

Here we are updating value in redux store by dispatching from a component

**ExpenseListFilters.js**

import React from "react";

import { connect } from "react-redux";

import { setTextFilter } from "../actions/filters";

const ExpenseListFilters = props => (

<div>

<input

type="text"

value={props.filters.text}

onChange={e => {

**props.dispatch(setTextFilter(e.target.value));**

}}

/>

</div>

);

const mapStateToProps = state => {

return {

filters: state.filters

};

};

export default connect(mapStateToProps)(ExpenseListFilters);

When you add a function in connect function to get the redux store, you will get **dispatch by default** which is same dispatch that we get from **createStore**

### Dropdown for Picking SortBy

import React from "react";

import { connect } from "react-redux";

import { setTextFilter, sortByAmount, sortByDate } from "../actions/filters";

const ExpenseListFilters = props => (

<div>

<input type="text" value={props.filters.text} onChange={e => {

props.dispatch(setTextFilter(e.target.value));

}}

/>

**<select value={props.filters.sortBy} onChange={e => {**

**if (e.target.value === "date") {**

**props.dispatch(sortByDate());**

**} else if (e.target.value === "amount") {**

**props.dispatch(sortByAmount());**

**}**

**}}**

**>**

**<option value="date">Date</option>**

**<option value="amount">Amount</option>**

**</select>**

</div>

);

const mapStateToProps = state => {

return {

filters: state.filters

};

};

export default connect(mapStateToProps)(ExpenseListFilters);

### Creating Expense AddEdit Form

import React from "react";

export default class ExpenseForm extends React.Component {

state = {

description: "",

note: "",

amount: ""

};

onDescriptionChange = e => {

const description = e.target.value;

this.setState(() => ({ description }));

};

onNoteChange = e => {

const note = e.target.value;

this.setState(() => ({ note }));

};

onAmountChange = e => {

const amount = e.target.value;

if (amount.match(/^\d\*(\.\d{0,2})?$/)) {

this.setState(() => ({ amount }));

}

};

render() {

return (

<div>

<form>

<input

type="text"

placeholder="Description"

autoFocus

value={this.state.description}

onChange={this.onDescriptionChange}

/>

<input

type="text"

placeholder="Amount"

value={this.state.amount}

onChange={this.onAmountChange}

/>

<textarea

placeholder="Add a note for your expense (optional)"

value={this.state.note}

onChange={this.onNoteChange}

/>

<button>Add Expense</button>

</form>

</div>

);

}

}

### Setting up a Date Picker

**Moment JS:** For this we will use Moment JS library. This is used for Parse, validate, manipulate, and display dates and times in JavaScript.

Npm install moment --save-dev

We will also use one open source project for datepicker in React JS

**React-dates:** An easily internationalizable, mobile-friendly datepicker library for the web <http://airbnb.io/react-dates>

Npm install react-dates --save-dev

Npm install react-addons-shallow-compare --save-dev

import React from "react";

**import moment from "moment";**

**import "react-dates/initialize";**

**import { SingleDatePicker } from "react-dates";**

**import "react-dates/lib/css/\_datepicker.css";**

export default class ExpenseForm extends React.Component {

state = {

description: "",

note: "",

amount: "",

**createdAt: moment(),**

**calendarFocused: false**

};

...

onDateChange = createdAt => {

this.setState(() => ({ createdAt }));

};

onFocusChange = ({ focused }) => {

this.setState(() => ({ calendarFocused: focused }));

};

render() {

return (

<div>

<form>

...

**<SingleDatePicker**

**date={this.state.createdAt}**

**onDateChange={this.onDateChange}**

**focused={this.state.calendarFocused}**

**onFocusChange={this.onFocusChange}**

**numberOfMonths={1}**

**isOutsideRange={() => false}**

**/>**

...

</form>

</div>

);

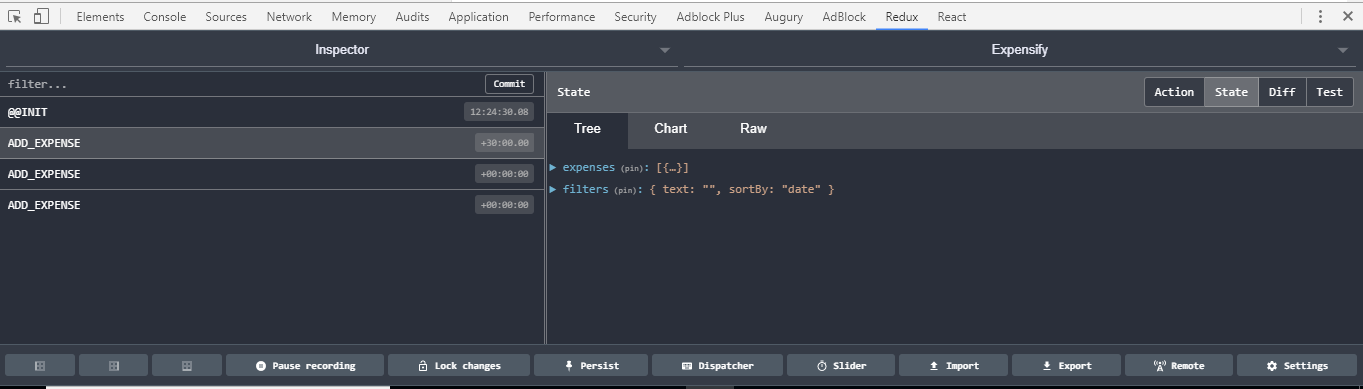
}

}

### Redux Dev Tools

Follow the step mentioned in the below link: <https://github.com/zalmoxisus/redux-devtools-extension>

After installing Chrome Redux Developer Tool, a Redux tab will start appearing in developer tools for Redux application as below:



## FluxThis

The super-opinionated, yell-at-you-for-everything, immutable Flux framework by AddThis.

Flux is a great design pattern, but following patterns is a matter of convention, and conventions are easy to break. FluxThis takes the pattern and turns it into a super-strict framework, to help large teams of devs avoid bad practices, and write sustainable, bug-free code.

### High Level Objectives

1. Enforce Flux design patterns
2. Make debugging extremely easy
3. Reduce boilerplate and create simple modular apis
4. Allow individual modules to be used standalone

### Creating an ImmutableStore

MyFirstStore.es6.js

'use strict';

const ImmutableStore = require('fluxthis/src/ImmutableStore');

const ACTION\_TYPES = require('constants/ActionTypes');

const Immutable = ImmutableStore.Immutable;

export default new ImmutableStore({

displayName: 'MyFirstStore',

init () {

/\* Your code here! \*/

},

public: {

/\* Your code here! \*/

},

private: {

/\* Your code here! \*/

}

});

#### The init Method

Inside of a stores **init** method, you should declare every property of your store with a default value. Here we add a list of todos, as well as an integer to keep track of todo IDs. Every property of an **ImmutableStore** is required to be an Immutable object or primitive javascript type. Don't worry, FluxThis will quickly catch your mistakes, if you make any.

init () {  
 **this**.todos = Immutable.List();  
 **this**.lastID = 0;  
},

### The public Methods

Inside of the **public** object, add accessor functions which can be used to get the data inside your store. In our case, we just need to return our list of todos.

**Note** : Immutable Stores can only return immutable objects or primitive types.

public {  
 getTodos () {  
 **return** **this**.todos;  
 }  
},

### The private Methods

Private methods are called internally by the store to respond to updates from the dispatcher. These methods should update your store's internal state. Referring to our design that we laid out earlier, you can see we will need to deal with three possible actions: add, remove, and toggle.

private {  
 addTodo (description) {  
 **let** todo = Immutable.fromJS({  
 description: description,  
 id: **this**.lastID++,  
 done: false  
 });   
 // update this.todos with the new immutable list, containing the new todo  
 **this**.todos = **this**.todos.push(todo);  
 },  
 removeTodo (id) {  
 **let** index = **this**.todos.findIndex(todo => todo.get('id') === id);  
 // remove the todo with the ID of id, but only if we have it to begin with  
 **this**.todos = index > -1 ?  
 **this**.todos.remove(index) :  
 **this**.todos;  
 },  
 toggleTodo (id) {  
 **let** index = **this**.todos.findIndex(todo => todo.get('id') === id);  
 **this**.todos = **this**.todos.update(index, (todo) => {  
 **return** todo.set('done', !todo.get('done'));  
 });  
 }  
}

Since everything in an ImmutableStore is immutable, any collections of items need to be completely replaced in order to update them. That is why every single private method in this store reassigns a new value to this.todos. Be sure to check out the docs for ImmutableJS if any of the immutability bits are confusing.

**Note**: Private methods being called is what triggers your view to update!

### Using bindActions

If you're familiar with flux, you're probably wondering where the giant switch case of action types belongs. Things look a little different in FluxThis. Inside the init method, we add a call to **this.bindActions**, passing to it action types and private method handlers.

init () {  
 **this**.todos = Immutable.List();  
 **this**.lastID = 0;  
 **this**.bindActions(  
 'ADD\_TODO', **this**.addTodo,  
 'REMOVE\_TODO', **this**.removeTodo,  
 'TOGGLE\_TODO', **this**.toggleTodo  
 );  
},  
The call to **bindActions** takes pairs of arguments. The first argument in a pair is an action type. The second argument in the pair is a reference to a private method which will handle the action.In general, it's better to use constants for action types instead of strings. See the documentation for [ConstantCollection](https://fluxthis.io/#/docs/constant-collections) for more details.At this point, your store has been created, and we can move on to creating a view that depends on it.

### Creating a Controller View

In flux, certain high-level components which get their state directly from stores are referred to as Controller Views. FluxThis uses a mixin approach to creating controller views, and stores all information from relevant stores on **this.state** through the method **getStateFromStores**.

Open the file src/components/MyFirstComponent.jsx to get started.  
export **default** React.createClass({  
 mixins: [MyFirstStore.mixin],  
 getStateFromStores() {  
 /\* Your code here! \*/  
 },  
 render() {  
 /\* Your code here! \*/  
 }  
});

The first thing to notice here is the mixins attribute. For every store that this view depends on, add the store's mixin to the list. For the most part, only top level components will need to use FluxThis mixins as it's good practice to resist have too many control views in your react hierarchy.

The second thing to see is **getStateFromStores**. This method is used to translate store public methods into internal component state. It should return an object, similar to getInitialState. Note: getInitialState, if defined, will run before getStateFromStores. In addition, getInitialState will always call getStateFromStores.

getInitialState should be used to setup any component state that does not live in stores.

getStateFromStores() {  
 **return** {  
 todos: MyFirstStore.getTodos()  
 };  
},

Next, lets implement the rest of the views methods.

getInitialState () {  
 **return** {newTodoText: ''};  
},  
handleAddClick () {  
 /\* We'll come back to this later \*/   
},  
handleTodoToggleClick () {  
 /\* We'll come back to this later \*/   
},  
handleTodoRemoveClick () {  
 /\* We'll come back to this later \*/   
},  
renderTodoItems () {  
 **return** **this**.state.todos.map(todo => {  
 **let** id = todo.get('id');  
 **let** toggleFn = **this**.handleTodoToggleClick.bind(**this**, id);  
 **let** removeFn = **this**.handleTodoRemoveClick.bind(**this**, id);  
 **let** style = {textDecoration: todo.get('done') ? 'line-through' : ''};  
  
 **return** <**div**>  
 <**span** style={style}>  
 {todo.get('description')}  
 </**span**>  
 <**input** type='button' value='remove' onClick={removeFn}/>  
 <**input** type='button' value='toggle' onClick={toggleFn}/>  
 </**div**>;  
 });  
},  
updateTodoText (event) {  
 **this**.setState({ newTodoText: event.target.value });  
},  
render () {  
 **return** <**div**>  
 {this.renderTodoItems()}  
 <**input** type='text' value={this.state.newTodoText} onChange={this.updateTodoText}/>  
 <**input** type='button' value='add' onClick={this.handleAddClick}/>  
 </**div**>  
}

That's it for the view for now. We will move ahead to creating an ActionCreator, and then revisit the view to add in functionality for adding/removing/toggling todos.

### Creating an ActionCreator

The FluxThis action creator is probably a little different than others you might have seen. The most unique thing about it is that it does not provide any access to the dispatcher. Instead, a dispatched action is mostly defined by configuration.

Open up the file src/actions/MyFirstActionCreator.es6.js to get started. The first thing we will add to the action creator is a displayName, which provides you with an easy way to identify the source of actions when you're debugging.

export **default** **new** ActionCreator({  
 displayName: 'Todo'  
});

Next, we define our actions. At the minimum, actions need only an type to function. We will provide payloads as well, which will spit out warnings when payloads don't look like we expect them to, similar to react's propTypes .

export **default** **new** ActionCreator({  
 displayName: 'Todo',  
 createTodo: {  
 type: 'ADD\_TODO',  
 payload: ActionCreator.PayloadTypes.string.isRequired  
 },  
 removeTodo: {  
 type: 'REMOVE\_TODO',  
 payload: ActionCreator.PayloadTypes.number.isRequired  
 },  
 toggleTodo: {  
 type: 'TOGGLE\_TODO',  
 payload: ActionCreator.PayloadTypes.number.isRequired  
 }  
});

Now, the exported ActionCreator will expose three methods: createTodo, removeTodo, and toggleTodo. Let's hop back into the view we created earlier to wire things together.

/\* src/components/MyFirstComponent.es6.js \*/  
handleAddClick () {  
 MyFirstActionCreator.createTodo(**this**.state.newTodoText);  
 **this**.setState({  
 newTodoText: ''  
 });  
},  
handleTodoToggleClick (id) {  
 MyFirstActionCreator.toggleTodo(id);  
},  
handleTodoRemoveClick (id) {  
 MyFirstActionCreator.removeTodo(id);  
},

## 

## React.js Questions and Answers

**Question: How to use same component multiple times on a page in React?**

**Answer:** To use same component multiple times we need something to differentiate them. Here I am using *id.* Using id I am creating computed property in setState method to update state specific to component

const Visitor = (props) => {

**showName = (e) => props.showName(e.target.value, props.id);**

return (

<input onChange={showName} />

);

}

class App extends React.Component {

constructor(props) {

super(props);

this.state = {name: ''}

}

showName = (value, id) => {

**this.setState(() => ({[id]: value}));**

}

render() {

return (

<div>

**<Visitor showName={this.showName} id='visitor1'/>**

**<Visitor showName={this.showName} id='visitor2'/>**

**<Visitor showName={this.showName} id='visitor3'/>**

**<Visitor showName={this.showName} id='visitor4'/>**

<p>**{`${this.state.visitor1} || ${this.state.visitor2} || ${this.state.visitor3} || ${this.state.visitor4}`}**</p>

</div>

);

}

}

ReactDOM.render(<App />, document.querySelector('#app'));

**Question: Can we create function in render method?**

**Answer:** A function in the render method will be created each render which is a slight performance hit. It's also messy if you put them in the render, which is a much bigger reason, you shouldn't have to scroll through code in render to see the html output. Always put them on the class instead.

**Question: What is the significance of keys in React?**

**Answer:** Keys are used for identifying unique Virtual DOM Elements with their corresponding data driving the UI. They help React to optimize the rendering by recycling all the existing elements in the DOM. These keys must be a unique number or string, using which React just reorders the elements instead of re-rendering them. This leads to increase in application’s performance.

**Question: What state object will print if we access, state object just after the setState()**

**Answer:** It will print values from previous state object.To solve this

There are two ways as mentioned in the official React documentation.

1. Using a callback passed to setState.
2. Using componentDidUpdate life cycle method

class App extends React.Component {

state = {

name: 'test',

age: 20

};

handleClick = () => {

// this.setState({age: 40});

**this.setState({ age: 40 }, () => {**

**console.log(`Inside ${this.state.age}`); // Inside: 40**

**});**

**console.log(`Outside; ${this.state.age}`); // Outside: 20**

};

render() {

return (

<div>

hello

<button onClick={this.handleClick}>Click</button>

</div>

);

}

}

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

**Question: Create a component that will print width of the window while resizing**

**Answer:**

class WindowWidth extends React.Component {

state = {

height: window.innerHeight,

width: window.innerWidth

};

componentDidMount() {

**window.addEventListener('resize', this.updateDimensions);**

}

updateDimensions() {

this.setState({

height: window.innerHeight,

width: window.innerWidth

});

}

componentWillUnmount() {

**window.removeEventListener('resize', this.updateDimensions);**

}

render() {

return (

<h3>

Window width: {this.state.width} and height: {this.state.height}

</h3>

);

}

}