# Concept

* Super important: In functional component, keep all executing code in useEffect method.or in control events. Don’t keep any executing code in raw body of function. Sometimes functional component is reloaded multiple times, in that case your executing code would run multiple times which will give wrong results.
* Because of useState when there is change there will be component reload. There is no bail out. You cannot stop it. For example:

Const [myObj, setMyObj] = useState({});

setMyObj({})

will always do reload of current component.

* Custom hooks: There are situations when you want some business logic to be returned by some function without any UI component. You can do it by custom hook. Create a functional component and export all elements which you want to be used by other elements. Don’t return any JSX. Now use the exported elements in any component. Name this by prefixing use…. say useMyHook. Keep the logic inside useEffect.
* A child component will always reload if parent component reloads even if there are no changes in props of child component. If you want to prevent child component from reloading even if parent component reloads then wrap the child component in React.memo(). This will prevent child component from reload and child component will reload only if there is change in props of child component.
* You can use lazy loading for react import of components along with Suspense.

const ReactForm = **lazy**(() => import('./react-form/ReactForm'));

* Note that there is difference between browser refresh and component refresh. During browser refresh always useEffect() method is called after render. In functional components the render is return statement of the function. At component reload / refresh the useEffect method is called based on the array argument of useEffect method.
  + If there is no array element, then useEffect method is always called at component reload
  + If there is empty array element, then useEffect is called only first time when component loads
  + If there are elements in array like [a,b] etc, then useEffect is called when there is change in value of any array element.
* When you do F5 or Browser page referesh or hard refresh then methods traverse sequence is:
  + Render (return)
  + Any return from the method inside useEffect
  + Method inside useEffect
* When [x,setx] = useState(0);

Suppose in click event of a button you do setx(x+1). That means value of x is changed. Remember that setx with useState does replacement and not merge. If there is no change in x then there will not be any component reload. Here x + 1 is change in value of x. So there will be component reload. So life cycle methods sequence will be:

* + Render method of function
  + return of method inside useEffect. This is component cleanup.
  + Method inside useEffect. This is initialization of component.

So component cleanup method is called before component initialization. If you do setx(x), then value of x does not change, and no component reload will take place.

* You used [] in useEffect as useEffect(()=>{ },[ ]). That means that you want useEffect to be called when items in the array [] change. If you do [x] or put x in the array then the useEffect will be called when value of x changes by calling setx(x+1). The cleanup method will be called before initialization as usual during component refresh.
* There can be multiple useEffect methods. The behavior of each useEffect will be identical. You can keep some useEffect methods with array arguments and some with not, some with return function (cleanup) and others with not.
* If you do [obj,setObj] = useState({}), then obj is an object. Now if you change some property of object or add new property in object and do setObj(obj) then component reload will not occur. Because objects are not compared but object references are compared, and they both are same object references. So you need to use object clone for component refresh as setObj({…obj}). For object clone {…obj} the object references change because they are pointing to two different objects so it works. Also setObj({}) will also do component reload.
* If you just want an empty refresh do like this:

const [, refresh] = useState({});

refresh({ ...{} });

* setObj({}) reloads the component.
* Suppose you are using multiple useState statements like const [value, setValue] = useState({}).

Changing the value directly without calling setValue will be lost at next reload. But setValue will perform reload. If there are many such instances then there will be unnecessary many reload. To avoid so many unnecessary reloads you can make use of useReducer in place of many useState. You can then control when and how to reload while persisting the values.

Signature of using useReducer is

[state, dispatch] = useReducer( reducer, initialState);

Steps:

Step 1: Create reducer method

const initialState: any = {

count: 0

}

const [state, dispatch] = **useReducer**(reducer, initialState);

function **reducer**(state: any, action: any) {

if (action === 'incr') {

state.count = state.count + 1;

return state;

*// const st = { count: state.count + 1 };*

*// return st;*

} else if (action === 'decr') {

const st = { count: state.count - 1 };

return st;

} else {

return state

}

}

Step 2: Call dispatch passing it the action

dispatch(‘incr’);

Note that in above when action is ‘incr’ then there is no component reload because only a property of state is changed, and same state object is returned. For action ‘decr’ a new object is returned hence the component will reload. The action can be an object having several properties. So effectively you can simulate many useState methods with great control over component reload.

Remember that object signature does not change when you change, add, delete a property or add, remove an item from array. These action do not produce reloads.

Let us examine the following code:

function runPromise() {

return new Promise((resolve: any) => {

setTimeout(() => {

resolve(1000);

}, 5000);

});

}

async function asyncPromise() {

setTimeout(() => {

return 100;

}, 5000);

}

<button

onClick={(e: any) => {

(async () => {

const res = await asyncPromise();

console.log(res);

})();

}}

>

Test promise123

</button>

As per our knowledge when we write code like const res = await runPromise() we expect that there will be wait till this line of code is not finished. So the next line will be reached when await line is completed. There is a catch here. The runPromise() function must be implemented as promise. If You have implemented runPromise() as async function then it does not work the expected way. That means that await does will not wait with asyncPromise() in above code in onClick() event but it will work with runPromise() in place of asyncPromise().

# Revisit 29-01-2019 (react hooks and effect)

Found React Hooks very good. Realized that Ibuki is better than EventEmitter for massive use. Hooks are used primarily to remove class structure in react. State is maintained at functional component when you use hook which was otherwise not possible in class components.

const [count, setCount] = useState(0);

This is basically destructuring of ES6 array. The useState creates an array and first two elements of the array are assigned to two variables count and setCount. The setCount is a function. The state of count is maintained when the component is re-rendered. Whenever you call setCount(someValue), then if count and someValue differs then the component is rendered. The useState does not merge values contrary to setState in class components. The useState() only replace values and does not merge. I found it convenient for say:

Const [myObj, setMyObj] = useState({});

Now you can do

myObj.name=’abc’; setMyObj(myObj); This maintains the value of myObj at render.

While using ibuki with hooks always use the effect hook as follows:

useEffect(() => {

const subs = ibuki.filterOn('do:validate').subscribe(d => {

});

return () => {

subs.unsubscribe();

}

});

If you don’t want to do render when you call the setMyObj above, then pass the empty array [] with the effect hook as follows:

function Product(props: any) {

let [contact, setContact] = useState({});

function loadData() {

fetch('http://chisel.cloudjiffy.net/contacts/short')

.then((res: any) =>

res.json()

).then((json) => {

contact = json[0];

setContact(contact);

console.log(contact);

}).catch(e => {

console.log('Error:', e);

}

);

}

let subs: any;

useEffect(() => {

const subs = ibuki.filterOn('load:data').subscribe(d => {

loadData();

});

loadData();

return () => {

subs.unsubscribe();

}

},[]);

return <div className="product">

Product

<div>{JSON.stringify(contact)}</div>

</div>

}

If I don’t use [] in above code it will go in infinite loop because loadData() is called in the useEffect(); in LoadData, the setContact(contact) is called which does re-render; This in turn again calls loadData();

I created custom hooks which were very effective for code reuse. The HOC were very complex but custom hooks nicely replaced the HOC. The custom hooks are functions which are named as **useMyHook.** Custom hook can return an object with various properties. You can destructure these properties into variables in the caller methods. The custom hook can in turn call other built in hooks. I implemented a pattern wherein I put all logic in custom hook and returned an object with several properties. In the caller components I destructured the object in variables. One custom hook catered many components. In the component I kept the code to minimum. Actually you can put anything inside custom hook. An example follows:

const useGeneric = (props: any) => {

const { item, dataHolder, callbacks } = props;

const [x, setx] = useState(0);// only render / refresh required no local persistence of data in text box

let [errors, setErrors]: any = useState({});

const [uniqueId] = useState(Date.now()); // to preserve unique id

const putErrors = (key: string, value: string) => {

errors[key] = value || undefined; //undefined value will not be displayed by ErrorDisplay

setErrors(errors);

const isAnyError = Object.values(errors).some(v => v != undefined);

callbacks.putErrors(uniqueId, isAnyError || undefined);

}

useEffect(() => {

const subs = ibuki.filterOn('do:validate').subscribe(d => {

errors = {};

validateItem({

data: dataHolder[item.name],

type: 'all',

item: item,

putErrors: putErrors

});

});

return () => {

subs.unsubscribe();

}

});

callbacks.initField(dataHolder, item.name, undefined);

const XLabel = () => item.label ? <label>{item.label}</label> : null;

const XErrorDisplay = () => <ErrorDisplay errors={Object.values(errors)}></ErrorDisplay>;

const onChangeEvent = (e: any, value: any) => {

callbacks.setField(dataHolder, e, value);

setx(1);// does only render

validateItem({

data: value, type: 'onChange', item: item, putErrors: putErrors

});

};

const onBlurEvent = (e: any, value: any) => {

validateItem({ data: value, type: 'onBlur', item: item, putErrors: putErrors });

}

return {

item: item, xName: item.name, xValue: dataHolder[item.name], xPlaceholder: item.placeholder, XLabel: XLabel, XErrorDisplay: XErrorDisplay, onChangeEvent: onChangeEvent, onBlurEvent: onBlurEvent

};

}

The useGeneric is custom hook which caters to many components. Following code consumes custom hook.

const ComponentStore: any = {

'radio': (props: any) => {

const { item, xName, XLabel, XErrorDisplay, onChangeEvent } = useGeneric(props);

const comp = <>

<XLabel></XLabel>

{

item.options.map((option: any, index: number) => {

return <div key={index}>

<label>

<input type='radio'

value={option.value}

name={xName}

onChange={e => onChangeEvent(e, e.target.value)}>

</input>{option.label}</label>

</div>

})

}

<XErrorDisplay></XErrorDisplay>

</>

return comp;

},

'text': (props: any) => {

const { xName, xValue, xPlaceholder, XLabel, XErrorDisplay, onChangeEvent, onBlurEvent } = useGeneric(props);

return <>

<XLabel></XLabel>

<input type='text'

value={xValue || ''}

placeholder={xPlaceholder}

name={xName}

onChange={(e) => onChangeEvent(e, e.target.value)}

onBlur={(e) => onBlurEvent(e, e.target.value)}

></input>

<XErrorDisplay></XErrorDisplay>

</>

}

}

# Revisited React.js on 15th Nov 2018

Original react site

<https://reactjs.org>

## Installation

Npm install -g create-react-app

Npx create-react-app my-app

Npm start

With typescript support

Npx create-react-app new-app --typescript

React app start in port 3000

For creating React application there are two steps

1. Create class which extends Component class. This should have a render method which returns JSX. The JSX can only be written within scope of React imported. So if you have not imported React then you cannot write JSX. The JSX can be put in a variable, can be passed as argument and many more. You can write JavaScript expression in curly braces {} inside JSX.

class App extends Component {

render() {

const ret = <div name1={test2()}>{myName}{test()}Test</div>;

return ret;

}

}

1. Using ReactDOM render the above class which is a component.

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

So, the react component is rendered inside an HTML element. In one HTML page you can have many such html elements wherein you can render react components.

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

ReactDOM.render(element, document.getElementById('root'));

React elements are immutable. They cannot be changed. Generally ReactDOM.render() is called once. By using the change state the things are managed.

## Hello world

import React from 'react'

import ReactDOM from 'react-dom'

class HelloWorld extends React.Component {

render() {

return (

<div>Hello World!</div>

)

}

}

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

## Important

When ReactDOM.render() is called multiple times, don’t think that whole DOM is re-painted. React compares the earlier rendered DOM with new DOM and only changes are updated to UI. Hence it is very fast.

## More on Functional components

 In React, any function that returns [JSX](https://facebook.github.io/react/docs/jsx-in-depth.html) is known as a Stateless Functional Component, or Functional Component for short

Any function returning JSX can be F component. There are two ways you can call a F component. 1) Like a function inside curly braces: {MyFComponent(myObj)} and

2) As JSX tag <MyFComponent item={some code}></MyFComponent>.

In second way item will be available in called function as props.item. The same F component you can call in both the manners. Be careful while passing the parameters. If you want to stay compatible with both manners, then in first manner remember to wrap your data in an object. This object will be passed as parameter to the F component.

When you create a component in class format then you must call it as jsx tag only. You cannot call it as function.

Note that {MyFunctionalComponent(props)} = <MyFunctionalComponent></ MyFunctionalComponent > except for passing parameters. When calling as function you pass exact parameters. When calling as tag you wrap the attributes in an object which is passed to function as argument.

React tags essentially executes the function while passing it the attributes wrapped in props as parameter. If you use React tag then strictly remember that this tag should be a class or a function returning JSX. If you use some variable as tag then wrap the variable’s value in ()=> function.

If you put some object in between the start and close tag then that function is obtainable in function as **props.children**.

<MyComponent>{item}</MyComponent>. If MyComponent(props){} is a functional component then inside MyComponent(props) function you can get **item** as **props.children**.

## More on Higher Order Components (HOC)

HOC’s are functions. HOC receives a react component X and returns a react component Y. Purpose of HOC is to provide additional functionality to the input component X. Most important to note that HOC does not mutate X. Remember that HOC does not execute a function or draws a component. It merely transforms a component X to component Y without mutation to X. The new component Y can be drawn using tags or using functional way.

HOC can perform following functions:

1. The X can be composed, means X can be put inside another component Y which is a wrapper of X. Hence another component Y can be created out of X and returned.
2. Some new properties can be added to X and then X itself can be returned.

## Tips

1. React tags need to always be in Capital case. They can be in lower case if they are objects like <props.comp></props.comp>. Associative arrays are not allowed as React tags. This is not allowed as tag: myObj[‘abc’].
2. When providing event say onChange event in JSX you call a function. If you provide non-arrow function syntax then you need to write a line in constructor for bind(). If arrow function is used the bind() is automatically taken care of.

**nameChanged** = (e) => {

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

}

**render**() {

const ret =

<div *className*="left"><input *type*='text' *value*={this.state.myName} *onChange*={this.nameChanged}></input>

</div>

return ret;

}

In above if you use nameChanged(){this.**setState**({ myName: e.target.value });} syntax then you must write in constructor as this.nameChanged = this.nameChanged.bind(this).

1. You can create a React component as 1) Function returning JSX and 2) A class with class extending from React.component. There are differences between the two.

Function component is less code. Function component is stateless, and you cannot use Component lifecycle methods. So, use function components where no need for lifecycle and state.

1. Always name react component starting with Capital letter. Jsx tags are always capital letter. React treats elements starting with lower case as HTML DOM and elements starting with upper case as react element.
2. State in React is local to a component. The setState method has two dialects. The first dialect has an object which is shallow merged with current state. Other dialect receives a function with prevState and props as arguments. Do not set the state directly but use the setState() method to change the state of a component. setState() will force the render() method. The merging is shallow, so this.setState({comments}) leaves this.state.posts intact, but completely replaces this.state.comments. Remember that setState() renders a component. You can pass new state object to setState(). Also you can pass a function to setState((prevState, props)=>{})
3. I tested with new context api. It worked fine. The concept is there is one provider and many consumers. Provider does provide data which can be consumed by consumers. The provider and consumer both are React components. When you create a new context say myContext, then the Provider and Consumer are automatically created inside the context myContext. Just wrap the parent component in the provider component and set its value. Now in child components where you want the value just wrap the component in a function which in turn wraps the actual component.

Process to implement Context api:

* 1. Create Context

Const myContext = React.createContext({});

* 1. In parent component render method

render() {

const myValue = { name: 'Sushant' };

return (

<myContext.Provider value={myValue}>

<div className="root">

<Header></Header>

<Body></Body>

<Footer></Footer>

</div>

</myContext.Provider>

);

}

* 1. Now in a deeply nested component’s render method

render() {

console.log('Right render');

return (

<myContext.Consumer>

{(data) => < div className="right">{data.name}</div>}

</myContext.Consumer>

);

}

See how data is provided as function argument and name inside data is fetched in the function.

1. There is a classnames npm library (npm install classnames). You can use this classnames library to conditionally add / remove class names in react components programmatically.

export:

export {component1} will export individual component1. There can be several such exports in a file. You need to individually import these as import {component1} from ‘./someFile’ in the source. There may be export default component1. Such export can only occur once in a file. You can import in the source file as import component1 from ‘./somefile’. Instead of component1 you can write import myComponent from ‘./someFile’.

1. HOC: Higher order component: HOC is in fact a function. It takes input of component and returns a component. The returned component has added functionality. For example the returned component can have a few new properties added up. If you want to make use of events such as the onBlur or onChange of original component then the wrapper component must call the event of original component through props. HOC are used to inject extra functionality in a component. You can add say validations to a component through HOC. For validation pass in the original component to a HOC, which in turn puts in the validation for onBlur and onChange events.
2. The setState method has two flavours. a) With one parameter object. This object is merged with original state. b) a function which returns the new state. This function received two values, 1) the previous state, 2) the props. Whenever your new state is dependent one previous state use this second flavour. The setState function is asynchronous. So in the next line you won’t get the updated state. So setState(something); console.log(this.state) won’t give the correct state. But if a control uses {this.state.someValue} then its value will be correctly shown.
3. Sometimes you need to return multiple items in the JSX. You generally put all tags in one <div> tag. In place of <div> tag you can place an empty tag <> and </>.
4. More on State: We know there are two flavours of changing state as explained above. Basically setState function merges the new value with the state and executes the render method. I have found from experimentation that you can use a custom object in place of state. For example you can set the UI with values from a custom object. On some event change the dependent property of custom object and execute the setState({}) or setState(()=>{return {};}). Your UI will be updated with new property of custom object. I checked with highly deeply nested custom object and this technique worked.

## Using semantic-ui-react

This is excellent library.

npm install semantic-ui-react --save

npm install semantic-ui-css –save

in source file

import { Input, Button, Icon } from 'semantic-ui-react';

import 'semantic-ui-css/semantic.min.css';

const Comp = <div>

<Input loading={false} icon='user' placeholder='Search...' onBlur={props.onBlur} ></Input>

<Button size='small' color='green'>

<Icon name='download' />

Download

</Button>

</div>

return Comp;

I checked the onBlur event with this library. it works fine.

## React component lifecycle

* **componentWillMount** is executed before rendering, on both the server and the client side.
* **componentDidMount** is executed after the first render only on the client side. This is where AJAX requests and DOM or state updates should occur. This method is also used for integration with other JavaScript frameworks and any functions with delayed execution such as **setTimeout** or **setInterval**. We are using it to update the state so we can trigger the other lifecycle methods.
* **componentWillReceiveProps** is invoked as soon as the props are updated before another render is called. We triggered it from **setNewNumber** when we updated the state.
* **shouldComponentUpdate** should return **true** or **false** value. This will determine if the component will be updated or not. This is set to **true** by default. If you are sure that the component doesn't need to render after **state** or **props** are updated, you can return **false** value.
* **componentWillUpdate** is called just before rendering.
* **componentDidUpdate** is called just after rendering.
* **componentWillUnmount** is called after the component is unmounted from the dom.

Notes: I checked that following is the chronological sequence:

* Constructor of parent is called
* Constructor of all immediate children is called and so on.
* ComponentDidMount of children is called
* ComponentDidMount of parent is called.

So it is a complete circle which starts at parent and closes at parent.

## Component Nesting

One component can contain another component. When react comes across any react component it collects its all attributes and push in props object and passes it to this component.

function Welcome(props){

return <div>Hello {props.name}</div>

}

class App extends Component {

render() {

const ret = <Welcome name = "Sushant" />

return ret;

}

}

Above will display Hello Sushant. Welcome is a react component. So when Welcome is encountered its attributes (currently only one attribute) name is pushed into props ands passed to Welcome, which later displays it.

## Events

constructor(props) {

super(props);

this.state = {isToggleOn: true};

// This binding is necessary to make `this` work in the callback

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

}

handleClick() {

this.setState(state => ({

isToggleOn: !state.isToggleOn

}));

}

render() {

return (

<button onClick={this.handleClick}>

{this.state.isToggleOn ? 'ON' : 'OFF'}

</button>

);

}

}

## Points

JSX — Allows us to write HTML like syntax which gets

transformed to lightweightJavaScript objects.

Virtual DOM — A JavaScript representation of the actual

DOM.

React.Component — The way in which you create a new component.

render (method) — Describes what the UI will look like for

the particular component.

ReactDOM.render — Renders a React component to a DOM node.

state — The internal data store (object) of a component.

constructor (this.state) - The way in which you establish

the initial state of a component.

setState — A helper method used for updating the state of a

component and re-rendering the UI

props — The data which is passed to the child component

from the parent component.

propTypes — Allows you to control the presence, or types of

certain props passed to the child component.

defaultProps — Allows you to set default props for your component.

Component LifeCycle

- componentDidMount — Fired after the component mounted

- componentWillUnmount — Fired before the component will unmount

- getDerivedStateFromProps - Fired when the component mounts and

whenever the props change. Used to update the state of a

component when its props change

Events

- onClick

- onSubmit

- onChange

## Good tutorials for basics

<https://tylermcginnis.com/reactjs-tutorial-a-comprehensive-guide-to-building-apps-with-react/>

<https://www.edureka.co/blog/reactjs-tutorial?utm_source=facebook&utm_medium=crosspost&utm_campaign=social-media-edureka-october-shivaprakash>

Code samples for Learning React Book from O’Really

<https://github.com/moonhighway/learning-react/>

React components and widgets. Too good

<https://github.com/brillout/awesome-react-components#ui-layout>

React JSON schema form build

<https://github.com/mozilla-services/react-jsonschema-form>

## FlexBox for react

Having solid knowledge of flexbox is necessary for react component layout. Flexbox completely matches with component structure of React.js. Flex is a super subset of float. Good tutorial for flexbox is here:

<https://internetingishard.com/html-and-css/flexbox/>

* Flexbox consists of container and items inside container. Each item in a container can in turn be a container. In css display: flex makes the item to behave as container.

# Tutorial

<https://facebook.github.io/react/docs/tutorial.html>

# Basics

React has component, props, state and events.

Component is like a view. Component maintains an immutable property bag called props. It also maintains a state which is user driven state of UI. Component creates an intermediate DOM and not the physical DOM. An additional step is taken to convert the intermediate DOM to real HTML DOM. Intermediate DOM is JavaScript object graph. A translation step converts this intermediate DOM to real HTML DOM. Dom manipulation is generally slow. The technique of intermediate DOM makes it fast.

You can compose your UI from a tree of components. The component implements a render method which creates the intermediate DOM. One component can have another component and so on. There can be deep nesting of components. When you call render on root component then all components are recursively converted to intermediate-DOM. This I-DOM is then converted to Real DOM.

For component creation React provides a convenient language called JSX which is extension of JavaScript. JSX is XML based and it directly maps to a component. The JSX has to be converted to JavaScript. There are two ways, one is provide JSX in browser and some library will create JavaScript at run time. Another is cli based which convert JSX to JavaScript, you can then provide this precompiled js. This is better for large JSX files or when there are many JSX files. You can even convert these all js files and other associated files to single bundle.js using webpack.

Remember that react.js is continuously massively changing. All react specific name should be uppercase otherwise it won’t work. Remember that React.createClass has been replaced by extend React.Component.

Following code is working:

var CommentBox = React.createClass({

render:function(){

return(<div className="commentBox">All comments</div>);

}

});

ReactDOM.render(<CommentBox />,document.getElementById('example'));

Render method does not return HTML markup. It generates the component. ReactDOM.render generates actual DOM and pushes to html container. Notice the uppercase of ‘CommentBox’, making it lowercase like ‘commentBox’ does not work.

Babel: A transpiler which converts ES6 to ES5.

webpack.js: packs js files in dist folder as a single js file.

Extracts from book Hacking Reactjs.

Chapter:1: Configure webpack, react, babel etc. through npm

Folder: minda. Npm init -y. Configure all as shown in book. Running webpack creates bundle.js in dist folder. Webpack-dev-server run in root folder starts the server. Automatically it takes content from dist folder. Localhost:8080 now runs the index.js in dist folder which in turn uses the bundle.js which is having all required js files in a single file. If you make changes in src folder it is auto reflected in output. It follows ES6.

Create folder pages in src and create details.js in pages as shown here <http://www.hackingwithreact.com/read/1/3/introduction-to-jsx>

Learnt so far:

* Setup dev environment using ‘hot-loading’ through webpack, **webpack-dev-server**
* Create index.html calling bundle.js created by webpack
* Using es6 create class extending react component and overriding its render method to render html
* Using react DOM to render the component created above.

Note:

1. Extend React.Component, override render method and create a component.
2. Using ReactDOM.render, render the component to browser, passing it the parameters to be consumed as props.

**Understanding this in JavaScript:** If an object **obj** has a function **f**, then **this** in **f** will refer to **obj**. In brief **this** in a function refers to antecedent or parent object. **this** has got the value of invoking object. **this** is a property of function which is automatically populated when an object calls it. When a function is defined in global space then **this** will refer to window object.

**This** is in raw terms the context. You can switch the context, i.e change the value of **this** and make **this** refer to another object by usage of **apply** or **call** method.

var person={

firstName:'Sushant',

lastName:'Agrawal',

showFullName:function(){

console.log(this.firstName + ' ' + this.lastName);

}

};

var anotherPerson={

firstName:'Niraj',

lastName:'Tenany'

};

person.showFullName.apply(anotherPerson);

In above it will show Niraj Tenany due to context switching.

**More on apply, call and bind:**

Every function by default has **this**, **apply**, **call** and **bind**.

These methods are exclusively used to set **this** in different ways. Call and apply are old whereas bind is from ECMA5.

Call and apply are executed immediately but bind executes at later stage. Call expects named and manual parameters. Apply expects array as parameters.

Bind does not execute immediately. It simply returns a function which can be executed later.

A good article on MDN (Mozilla Developer Network)

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply>

**Using react props**: These are read only passed at initiation of component. I did that. Props can be changed but recommended not to change.

Using chance.js a random data generator for first name, last name, SSN etc. Used small case for ‘c’ as chance.first() and not Chance.first().

The content inside the curly braces {} is javascript expression and not javascript statement. Expressions return a value but statements do not. You are not allowed to write statements inside {}. Only expressions are allowed.

For if-then-else to be used in {} you can use tenary operator: ?.

The render method always return one html component. If you try to return multiple then it throws error. There is a way out: wrap all multiple elements in a single div element.

**Event with JSX:**

We provide click events in component. Example working code:

import React from 'react';

import Chance from 'chance';

class Detail extends React.Component{

buttonClicked(){

console.log('Button clicked');

this.forceUpdate();

}

render(){

return (

<div>

<span>Hello {chance.first()}</span>

<span>{chance.last()}</span>

<button onClick={this.buttonClicked.bind(this)}>Click Me</button>

</div>);

}

};

export default Detail;

Notice in above code how you defined buttonClicked() without a ‘:’ and ‘;’ as you normally do in HTML. forceUpdate() does re-render. this.buttonClicked.bind(this) is important. The definition of this changes when inside an event. So you need to retain the **this** inside event. Hence bind(this) is done. You cannot do apply or call because apply and call immediately execute. The bind returns a function and does not immediately execute. So when the button is clicked a proper **this** is fed to the buttonClicked method.

**State and props**

Any change in component’s state or property forces it to re-render. this.state gets the state. this.setState sets the state to some object. There is a constructor(props) method which s initializer. You must give super(props) inside constructor to run the ancestor script.

In constructor if you want to initialize state then this.state = {some object}. Now you can refer to this.state in your render. If any other method of component wants to change the state then this.setState() to be called. Then there is re-render. To stop re-render you can provide a method **shouldComponentRender()** returning false.

Chapter 14

## React State

Check this

<https://facebook.github.io/react/docs/react-component.html>

Understanding state and ‘Change state’ is important in react. Every component in react has got a state which is denoted by ‘this.state’. State can be changed by setState() method. This setState() method does not mutate the original state but merges the new state to old state. Change of state necessarily does re-render of component.

To use redux you can maintain entire data / state of application in redux store. You can merge new values in redux store. Then you can subscribe to store and copy the new values from store to component’s state by using setState() method. Now when you apply dispatch() method on a redux store then all the subscriptions will be notified. In the subscriptions you can copy the application state to component state and the component will re-render. In this way data can be transferred from any UI component to any other UI component in the entire application irrespective of child and parent relationship. This is most important understanding for redux.

Redux presentational components are hand created and container components are generated by react-redux library’s **connect** method. The container components connect to the redux store and pass on the new state through props to presentational components.

**03-02-2017**

**Container and presentational components**:

https://scotch.io/tutorials/build-a-music-player-with-react-electron-i-setup-basic-concepts#enter-react-presentation-vs-container-components

It is important to understand them to understand Redux in depth. Pr. Components are only responsible to do display, they have only HTML UI code. They receive props from parent and all the data display and events firings are done through these props. Even the button click events are passed to Pr. from the parent. Say for example on button click an ajax call is to be initiated. Now parent composes the ajax call and in form of props it passes a function to the Pr. Component. Now Pr. Component executes this props.funcName in button click event. It is React way of working. Pr. Components do not connect outside world directly, they only interact the parent. Parent’s responsibility is to provide everything to the Pr. Component in form of props. Parent component is container component.

**React Router**

Library react-router (npm install) is used for routing.

Good blog for routing: <https://css-tricks.com/learning-react-router/>

**Simple routing**

In **Router** tag provide several **Route** tags. Evary Route tag has path and component. In url if path appears after # that component is placed.

<Router history={hashHistory}>

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

<Route path="/address" component={Address} />

<Route path="/street" component={Street} />

</Route>

</Router>

**Nested Routing**

One Route is defined in another Route. Intention is to show nested components. Normally you can put one component in another component inside jsx which works fine. But this is fixed and you cannot control it through url. Purpose of nested routing is to control the nested components through url. Say for example if url is ‘/home’ you want to show only home, when url is ‘/address’ you want to show address component inside home component, when url is ‘/street’ you want to show street component like home 🡪 address 🡪 street nested. It wll show all three components. Steps to follow:

1. Provide nested routing inside Router.
2. Make use of this.props.children inside the parent component to show the child components. Here you have to put this.props.children inside the **home** and **address** component. In that case **home** will display **address** as its child and **address** will display **street** as its child and all the three components will be displayed.

import React from 'react';

import Search from './search.jsx';

import {

Router,

Route,

Link,

IndexRoute,

hashHistory,

browserHistory

} from 'react-router'

class App extends React.Component {

render() {

return (

<Router history={hashHistory}>

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

<Route path="/address" component={Address}>

<Route path="/street" component={StreetNo}>

</Route>

</Route>

</Route>

</Router>

);

}

}

class Home extends React.Component {

render() {

return (

<div>

<div>This is home</div>

<div>{this.props.children}</div>

</div>

);

}

}

class Address extends React.Component {

render() {

return (

<div>

<div>This is address</div>

<div>{this.props.children}</div>

</div>

);

}

}

class StreetNo extends React.Component{

render(){

return(

<div>P / 161 VIP road, SCH VIIM</div>

);

}

}

export default App

## React Native

npm install -g react-native-cli

react-native init adam

cd adam

react-native start

in another command window

react-native run-android

I successfully created basic tutorial application following the react native blog. Text, Image, View are reactive-native specific controls. In emulator (the mobile phone screen which comes up during AVD startup) I pressed R two times in quick succession and the changes in JavaScript file reloaded. I clicked the emulator first then pressed **R** two times to do that.

All of core components in RN accepts style prop. Style is JSON object and values are CSS with one difference: it is camel. Background-color is backgroundColor.

class LotsOfStyles extends Component {

render() {

return (

<View>

<Text style={styles.red}>just red</Text>

<Text style={styles.bigblue}>just bigblue</Text>

<Text style={[styles.bigblue, styles.red]}>bigblue, then red</Text>

<Text style={[styles.red, styles.bigblue]}>red, then bigblue</Text>

</View>

);

}

}

const styles = StyleSheet.create({

bigblue: {

color: 'blue',

fontWeight: 'bold',

fontSize: 30,

},

red: {

color: 'red',

},

});

In RN two types of dimensions are there. Fixed width and Flex. Fixed width is absoluter number like height:10, width:10. They show the same sizes irrespective of device size.

Flex dimension enables a component to adjust dynamically based on available space. Flex is meaningful if a component’s parent also has a flex. If parent has no flex it is considered as flex = 0 and no children will be visible. The children will be sized based on Flex ratio. A child with flex:2 will have double size than with flex:1 in the same parent.

Layout with Flexbox: If you put flexDirection:’row’ in parent style then its children are drawn horizontally. If flexDirection:’column’ then vertically drawn children. justifyContent has flex-start, center, flex-end, space-around, space-between as possible values. Similarly alignItems in parent has flex-start, center, flex-end, and stretch as possible values.

In order to run your apk on real device you need to sign it and follow the procedure for release build. The release apk can only run on real Android device. I successfully ran the release app in real device

## Fetch and Promise

Promise is a special object having another object. Then callback is used to get this another object.

Basic signature of fetch is as follows:

fetch('https://facebook.github.io/react-native/movies.json')

.then((response)=>response.json())

.then((responseObject)=>{console.log(responseObject)});

The first **then** returns a promise so other **then** is required to get the object data. This object data can now be pushed into redux store or observable to be subscribed by the target.

Fetch can be universally used in react for all ajax calls.