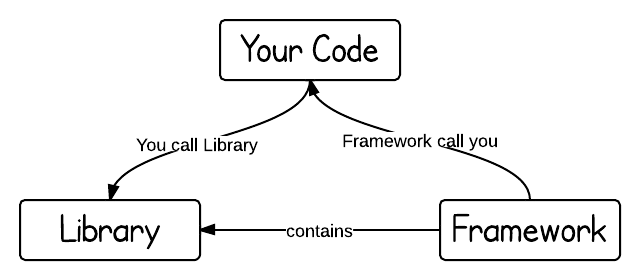
**React**

**Library vs Framework**

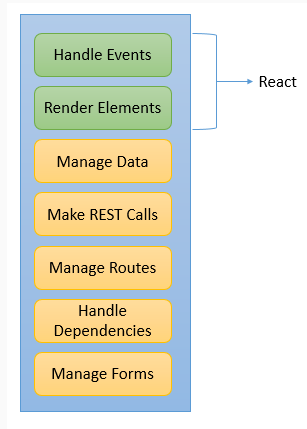
<https://www.programcreek.com/2011/09/what-is-the-difference-between-a-java-library-and-a-framework/>



<https://www.geeksforgeeks.org/software-framework-vs-library/>

A framework is full fledged solution for the different aspects of application development. A library on the other hand covers only a small part of the big picture and satisfies a specific requirement alone. Therefore we can use different libraries for different parts of the application.

React specifically caters to only two things: Rendering elements on the page efficiently and handling events. For everything else, we would have to write our own code or use other libraries. The below diagram shows the different needs of a typical complex front end application and which parts of it are covered by React.



**Advantages** are: **Flexibility**(Choice), **Lightweight**and **High Performance**

React being a library also has its own *disadvantages*. Choice is not always a good thing.

For example:

* React cannot be used for the entire front-end development
* One may have to depend on other libraries
* Choosing between multiple libraries can be difficult

Features of React

One of the key features of React is its use of **Virtual DOM**. This makes React very fast.

DOM is the tree representation of the various elements in an HTML page. When the browser receives an HTML page, it does the below steps:

1. Parses the HTML content
2. Creates the DOM structure
3. The position and layout is calculated
4. The DOM structure is painted on the browser

Whenever a DOM element changes, the browser re-calculates the position of that element and its children elements and repaints them again. This is a costly process.

Imagine, if we have thousand DOM elements, then the re-positioning and re-painting operation done thousand times is a slow process. React simplifies it to a great extent using a Virtual DOM.

# How does Virtual DOM work?

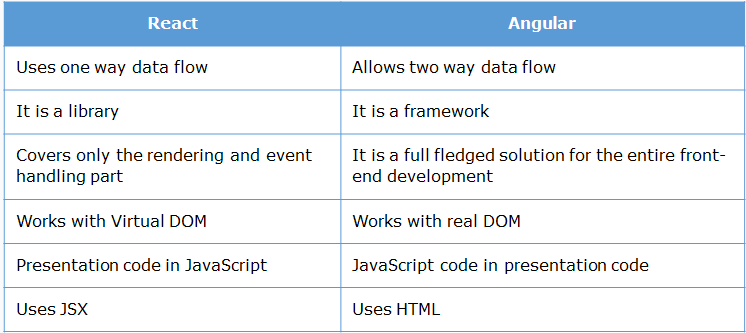
A virtual DOM is a in-memory representation of the actual DOM. The way virtual DOM works is:

* The initial Virtual DOM is exact representation of the actual DOM
* When anything needs to be changed, it does not compare with the actual DOM. Instead it compares which parts of the Virtual DOM needs changes. This makes it faster.
* Instead of re-painting and re-positioning for every change, it creates a collection or batch of changes in the virtual DOM.
* It repaints only those areas in the DOM which have changed.

Consider an example where the current time needs to be updated every second on the screen. If we code it without React, the DOM updates the whole h2 tag, whereas React updates only the time portion of the h2 tag and does not repaint the other parts of the h2 tag, as shown below:

|  |  |  |
| --- | --- | --- |
| **Actual DOM** |  | **Virtual DOM** |
|  |  |  |

Some developers find React as a competitive alternative to Angular. Let us compare some of the salient differences between them:

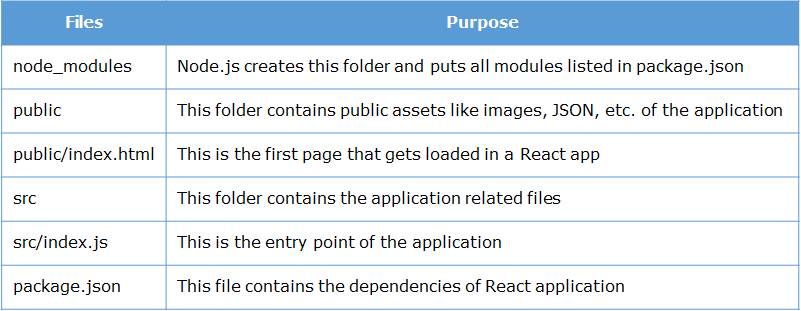


Creating React app

Follow the below steps to install and use 'create-react-app':

1. npm install -g create-react-app

2. create-react-app <projectName>



**Step 3**: To run the application

npm start

**Note**: The above can also be done using the *npx* command as shown below:

1. npx create-react-app my-new-app
2. cd my-new-app
3. npm start

Here **my-new-app** is the name of the React app that gets created. You can obviously have your own app name!

Next, let's take a look at how we can modify this app.

Creating React app – Demo

**Step 1:** In the app which you have created, navigate to in **src/index.js**and replace the whole code with the below code:

1. import React from "react";
2. import ReactDOM from "react-dom";
3. const element = <h1>Hello World</h1>;
4. ReactDOM.render(element,document.getElementById("root"));

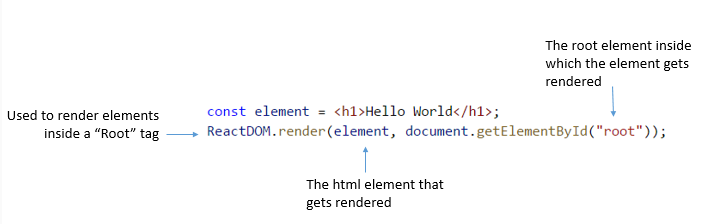
 The page which is is being displayed is the **index.html** page under public folder. The <h1> tag you created has been dynamically inserted into the <div> tag in the index.html page.

Let us see a sample JSX code:

var ele = <h1>This is sample JSX</h1>;

The above code snippet somewhat looks like HTML and it also uses a JavaScript-like variable but is neither of HTML or JavaScript, it is JSX. JSX is basically a syntax extension of regular JavaScript and is used to create React elements.

Rendering JSX Elements



# Element

An element is plain object which represents DOM node. However this is not the actual DOM. This becomes part of what is called as a Virtual DOM, which is a in-memory representation of the actual DOM.

Virtual DOM does not exist in the browser. React will compare the Virtual DOM with the actual DOM. Wherever there is a change, it will update only that part instead of updating the entire DOM making it very performance efficient.

We can create an element by just typing HTML tags and storing it in a variable as shown below:

1. const element = <h1> Displaying an element </h1>

# Render

To render anything on the UI, we will use ReactDOM.***render()*** method as follows:

1. ReactDOM.render(elementToBeRendered, htmlNode);

Here, the first param represents the element to be rendered and second param represents the HTML node in **index.html** page where it has to be rendered.

# [What is Virtual DOM?](https://stackoverflow.com/questions/21965738/what-is-virtual-dom)

React creates a tree of custom objects representing a part of the DOM. For example, instead of creating an actual DIV element containing a UL element, it creates a React.div object that contains a React.ul object. It can manipulate these objects very quickly without actually touching the real DOM or going through the DOM API. Then, when it renders a component, it uses this virtual DOM to figure out what it needs to do with the real DOM to get the two trees to match.

You can think of the virtual DOM like a blueprint. It contains all the details needed to construct the DOM, but because it doesn't require all the heavyweight parts that go into a real DOM, it can be created and changed much more easily.

Rendering JSX Elements

JSX is a special syntax introduced in ReactJS to write presentation code. It is an extension of the JavaScript syntax. It is syntactically similar to HTML and hence it can be easily read and written to help visualize the elements on a page.

Consider the below code:

1. const element = <h1> Hello World!!! </h1>;

In the above line, the value assigned to the element in not enclosed within double quotes. So if it is not a string, what is the datatype of this variable? React extends the capability of JavaScript in this way by using what is known as JSX ( JavaScript XML ).

 As the browser does not understand JSX code, this has to be converted to JavaScript and this conversion is done by React automatically using a JavaScript compiler called **Babel**.

Conversion of JSX to JavaScript happens as shown below:

**<h1>Hello World</h1>**to  **React.createElement("h1",null,"Hello World");**

Conversion shown above is done using babel, a JavaScript compiler and you can see conversion of your JSX code to JavaScript on [Babel Playground](https://babeljs.io/repl/). You can go to this link, paste your JSX code there and you will see JavaScript output of that code.

**Note**: All tags in JSX MUST BE closed. Else it will give an error.

We saw that JSX is converted into React.createElement() JavaScript code.

# Syntax

**React.createElement(arg1,arg2,arg3);**

This method creates a HTML element and renders it.

arg1 - this will be the name of the tag which gets rendered

arg2 - this will be data which will be passed along with element rendering

arg3 - this will be the children of the rendered element

# Example 1

**<h1>Hello World</h1>**will be converted into

**React.createElement("h1",null,"Hello World");**

Here arg1 - "h1", arg2 - null, arg3 - "Hello World"

# Example 2

**<div text-align="center">Hello World</div>**will be converted into:

**React.createElement("div",{ "text-align": "center" },"Hello World");**

Here arg1 - "div", arg2 - {"text-align":"center"}, arg3 - "Hello World"

JSX ultimately is an object. Thus we can write any valid JavaScript expression in it. To embed any JavaScript expressions within JSX, it has to be written using curly braces i.e {expression}.

# Accessing an object

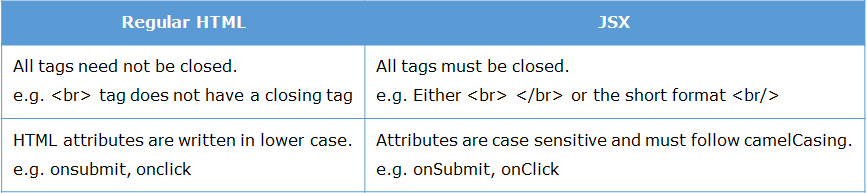
1. const product={
2. productId: 128,
3. price: 1200
4. }
5. const element = <p>Product ID:{product.productId}</p>

We have seen that we can include JavaScript expression in JSX. Now that JavaScript expression may in turn return another JSX.

1. function generateJSX() {
2. return <h1>Hello World</h1>
3. }
4. function display() {
5. return (
6. <div>
7. This is inside a div
8. {generateJSX()}
9. This is after h1 tag
10. </div>
11. )
12. }
13. ReactDOM.render(display(),document.getElementById('root'));

Although JSX resembles HTML code, it is not actually HTML code. But in the end, JSX is just a JavaScript object.

Therefore, writing JSX differs from HTML in the below ways:



Also note that:

* Attributes can either have double quotes (" ") or flower bracket ( {} ) and not both.
* Not all attributes of the HTML tags are available in JSX. For example, the **class** attribute becomes **className** attribute in JSX.

The below code will throw an error:

1. var element = (
2. <div>Hello World</div>
3. <div>Hello World</div>
4. );

This is because, since JSX is finally converted into *React.createElement()*, we cannot have multiple elements in it.

To avoid this we usually wrap all the elements inside a container div as shown:

1. var element = (
2. <div>
3. <div>Hello World</div>
4. <div>Hello World</div>
5. </div>
6. );

# React Fragments

Having a <div> to wrap other elements introduces an unnecessary div in the DOM as shown:

This can be avoided by using React fragments. React fragments solve the same problem without introducing a new element in the DOM.

1. render() {
2. return (
3. <React.Fragment>
4. <div>Hello World</div>
5. <div>Hello World</div>
6. </React.Fragment>
7. );
8. }
9. <div>
10. <span>Name: {emp1.empName}</span><br/>
11. <span>Id: {emp1.empId}</span><br/>
12. <span>Age: {emp1.age}</span><br/>
13. <span>Salary: {emp1.salary}</span><br/>
14. <img src="emp1.PNG" width="100px"/><br/>
15. <a href="#">Edit Details</a>
16. </div>

Observe the following in the code:

* All tags have a closing tag, including img and br
* *The images in public folder are accessed directly instead of mentioning any relative path*
* All the tags are nested inside a single div tag

Conditional Rendering

We cannot write if-else directly inside a JSX expression. Hence, we can use if-else logic as shown below:

1. import React from 'react';
2. import ReactDOM from 'react-dom';
3. const today=new Date().getDay();
4. var element;
5. if(today===0){ *//Sunday has 0 value for Day*
6. element=<h1>Happy holiday</h1>
7. }
8. else{
9. element=<h1>Happy coding!</h1>
10. }
11. ReactDOM.render(element, document.getElementById('root'));

We can also use the ternary operator as shown below:

1. const today=new Date().getDay();
2. var element= today===0?<h1>HappyHoliday</h1>:<h1>Happy Coding!</h1>
3. ReactDOM.render(element, document.getElementById('root'));

We cannot directly write for-loop inside a JSX. However we can write JSX inside a for-loop.

So one approach is to create an array for storing the JSX code generated for each iteration of loop and include that JSX array inside another JSX, as shown below in index.js.

1. import React from 'react';
2. import ReactDOM from 'react-dom';
3. const empArray = [
4. { empId: 100, empName: "Kirk", age: 30 },
5. { empId: 101, empName: "James", age: 31 },
6. { empId: 102, empName: "Robert", age: 32 },
7. { empId: 103, empName: "Dave", age: 50 },
8. { empId: 104, empName: "Rob", age: 40 }
9. ]
10. function generateTable() {
11. var rowArr = [];
12. for (let i = 0; i < empArray.length; i++) {
13. var row = (
14. <tr>
15. <td>{empArray[i].empId}</td>
16. <td>{empArray[i].age}</td>
17. <td>{empArray[i].empName}</td>
18. </tr>
19. )
20. rowArr.push(row);
21. }
22. return (
23. <table className="table">
24. <thead>
25. <tr>
26. <th>EmpId</th><th>Name</th><th>Age</th>
27. </tr>
28. </thead>
29. <tbody>
30. {rowArr}
31. </tbody>
32. </table>
33. )
34. }
35. ReactDOM.render(generateTable(), document.getElementById('root'));

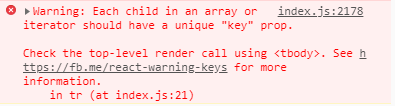
Note: While creating tables in React, <tr> cannot be direct child of the <table> tag. This is how it is defined in HTML specification itself. The valid child tags of <table> are <thead>, <tbody> and <tfoot>.

The above code can be further simplified as shown below:

1. function generateTable() {
2. return (
3. <table>
4. <tbody>
5. {empArray.map((emp)=> <tr><td>{emp.empId}</td><td>{emp.empName}</td><td>{emp.age}</td></tr>)}
6. </tbody>
7. </table>
8. )
9. }

The code which we wrote to display a table will give the output, but the developer console shows a warning as given below:

Warning: Each child in an array or iterator should have a unique "key" prop.



Let us see how we can render a list of employee objects in a table. We want to display the following data:

1. [{ empId: 100, empName: "Kirk", age: 30 },
2. { empId: 101, empName: "James", age: 31 },
3. { empId: 102, empName: "Robert", age: 32 },
4. { empId: 103, empName: "Dave", age: 50 },
5. { empId: 104, empName: "Rob", age: 40 }];

# 

# Adding Keys to array items

The React framework uses Keys to help render list items quickly. Keys helps React in identifying which items have changed i.e. added or removed. Keys should be a string that uniquely identifies a list item from the other items on the list, such as an ID attribute. Hence, it should be string and unique identifier.

For example, here we will be using empId attribute as a key:

Observe the code below:

1. import React from 'react';
2. import ReactDOM from 'react-dom';
3. import 'bootstrap/dist/css/bootstrap.min.css';
4. const empArray = [
5. { empId: 100, empName: "Kirk", age: 30 },
6. { empId: 101, empName: "James", age: 31 },
7. { empId: 102, empName: "Robert", age: 32 },
8. { empId: 103, empName: "Dave", age: 50 },
9. { empId: 104, empName: "Rob", age: 40 }
10. ]
11. function generateTable() {
12. return (
13. <table className="table col-4">
14. <tbody>
15. {empArray.map((emp)=> <tr key={emp.empId}><td>{emp.empId}</td><td>{emp.empName}</td><td>{emp.age}</td></tr>)}
16. </tbody>
17. </table>
18. )
19. }

Binding an Event

When a button is clicked,

1. const button = <button onClick= {display} >MyClick</button>
2. function display(){
3. console.log("Button clicked");
4. }

**Note:**

1. Unlike in HTML, we do not call the function while handling events. For example, in HTML the code will be <button onclick="display()">. In JSX, the code will be *<button onClick={display}>*
2. The event attribute must be in camelCase. In other words use *onClick*rather than onclick

**Use className attribute**

we can use the JSX attribute className=Css-Class-Name to add CSS classes to elements.

**Using class vs className**

You'll notice that React uses className instead of the traditional HTML attribute class.

Since JSX is JavaScript, identifiers such as class and for are discouraged as XML attribute names. Instead, React DOM components expect DOM property names like className and htmlFor, etc.

For adding Bootstrap style classes:

1. Install Bootstrap locally:

1. npm install bootstrap --save

2. Import the *bootstrap.min.css* into the required JS file as shown below:

1. import 'bootstrap/dist/css/bootstrap.min.css';

**Using inline styling**

In React, components will have inline styles and it should be specified as an **object**. CSS attribute should be written in *camelCased* version. For example, the CSS attribute *text-align* will become *textAlign.*

In HTML, a CSS inline style is defined as follows:

1. <h1 style = "font-family: Arial" > Styled element </h1>

In React, the inline styling is mentioned as a JavaScript literal object as shown below:

1. const h1Style={
2. fontFamily:'Arial'
3. }

The changes from a regular HTML inline styling are:

* The style is mentioned as an object
* The style is mentioned as key value pairs
* If the value is a string, it must be within quotes
* The key is mentioned in camelCase

We can then use the object in JSX as shown below:

1. <h1 style = { h1Style }> Styled element </h1>

Instead of creating an object and storing it in a separate variable, we can directly mention the object in the JSX as shown below:

<h1 style={ {fontFamily:'Arial'} }> Styled element </h1>

Creating Components

To create a component we need to create a class which extends the **React.Component** class and implement the render(). The render() must return a JSX.

For example, create a new file called HelloComponent.js in **src**folder and write the below code:

src/**HelloComponent.js**

1. import React from "react";
2. class HelloComp extends React.Component {
3. render() {
4. return (
5. <h1>Hello New Component</h1>
6. );
7. }
8. }
9. export default HelloComp;

**HelloComp** - component name should be in PascalCasing.

**render()** - method will render the component’s elements. Should have a mandatory return, which returns a JSX.

**export default HelloComp** - HelloComp component has to be exported so that it could be used in any other files.

# ****Rendering a component****

By creating a component we are actually creating our own custom tag. To display this component in the page, update the src/**index.js** file as shown below:

1. Add the below import statement:

1. import HelloComp from './HelloComponent.js'

2. Modify the ReactDOM.render() to use the new component created as shown below:

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

Component which has to be rendered will be written as selector i.e. <HelloComp> </HelloComp>

**Component conventions**

# ****Component Name****

Component name should be in **PascalCase**which means first letter should always be capital.

If your Component name is not in PascalCase then your React application will throw an error because React consider tags starting with lowercase letter as HTML/DOM tags and tags starting with Uppercase letter as Component.

# ****Render****

If you don't implement render() method in any class Component you will get **TypeError** while when you run your react application. You must always implement render() method while creating any class Component. Error will be: **TypeError: instance.render is not a function.**

# ****return ()****

One should always enclose return of Component with parentheses () because if after return our code start with next line, JavaScript will automatically put a semicolon after return and any code from next will not execute.

Observe the below code, Why does it not work?

1. class HelloComp extends React.Component {
2. render() {
3. return *// the starting '(' is missing, code will not work*
4. <h1>Hello World</h1>
5. }
6. }

The above example will not work as HelloComp will return nothing as <h1> is starting from next line and JavaScript will put a semicolon after return in line 3 during execution.

The correct code is as shown below:

1. class HelloComp extends React.Component {
2. render() {
3. return ( *// starting '(' is present*
4. <h1>Hello New Component</h1>
5. ); *// ending ')' is also present*
6. }
7. }

**Component - constructor, methods and variables**

A Component is essentially a JS class, and can have class methods and instance variables. A React component class should follow the rules:

* For creating **class methods**, *arrow function syntax* must be used.
* If a **constructor**is being used, a *super();*method call should be the first line of the constructor if **this** keyword is used in the constructor.

Observe the code below:

1. class HelloComp extends React.Component {
2. constructor() {
3. super();
4. this.name = "Jack";
5. }
6. *//arrow functions should be used*
7. *// code might behave unpredictably if arrow not used!!*
8. displayName = () => {
9. return (
10. <h1>Hello, {this.name}'s World</h1>
11. );
12. }
14. render() {
15. return (
16. <div>
17. {this.displayName()}
18. </div>
19. );
20. }
21. }

*Employee*component as shown below:

1. import React from "react"
2. class Employee extends React.Component {
3. constructor() {
4. super();
5. const emp1 = {
6. "empId": 100,
7. "empName": "Jack",
8. "age": 30,
9. "salary": 50000,
10. "image": "emp1.png"
11. }
12. const emp2 = {
13. "empId": 101,
14. "empName": "Jane",
15. "age": 24,
16. "salary": 40000,
17. "image": "emp2.png"
18. }
19. this.empArr=[emp1,emp2]
20. }
21. createCard(emp) {
22. var note = null;
23. if (emp.age < 25) {
24. note = <span className="text-info"> - Fresher</span>
25. }
27. return (
28. <div key={emp.empId} className="card" style={{ "width": 200 }}>
29. <img className="card-img-top" src={emp.image} height="200" alt="Card image cap" />
30. <div className="card-body">
31. <h5 className="card-title text-center">{emp.empName}</h5>
32. <p className="card-text">
33. <span>Id: {emp.empId}</span><br />
34. <span>Age: {emp.age}</span> {note}<br />
35. <span>Salary: {emp.salary}</span><br />
36. </p>
37. <button type="button" className="btn btn-primary">Edit Details</button>
38. </div>
39. </div>
40. )
41. }
43. render(){
44. return (
45. <div>
46. <h3 className="text-center text-primary">Employee Details</h3>
47. {this.empArr.map(emp => {return this.createCard(emp)})}
48. </div>
49. )
50. }
51. }
52. export default Employee;

Managing State

# What is state?

**State**is a special variable in React which represents the data associated with a component and the view. A React component must represent the state of view at any point in time, and not just during initialization. When we think of state, we should think of an internal data-set which affects the rendering of components.  
   
Data in the Virtual DOM is state and only when this Virtual DOM is updated, React will update the actual DOM.

In React, **state**is a built-in object associated with a component. The state represents a component's internal data-set.

Accessing the state will help us solve the problem of displaying the updated counter.

In the below example, count value is hardcoded in a component and we cannot update it dynamically.

1. class HelloComp extends React.Component {
2. constructor() {
3. super();
4. *//this.counter = 0; //added to state now!*
5. this.state = { *// state should be assigned an Object Literal with relevant attributes*
6. counter: 0
7. };
8. }
9. incrementCounter = () => {
10. this.state.counter += 1; *// trying to increment(or mutate) the state! //not possible!*
11. console.log("Button Clicked," , this.state.counter, "times" );
12. };
13. render() {
14. return (
15. <div>
16. <button onClick={this.incrementCounter}> Click </button>
17. <p>{this.state.counter}</p>
18. </div>
19. );
20. }
21. }

We need to update it dynamically based on the user interaction, we have added the**{counter:0}**object to the *this.****state*** of a component.

We can say that the *state* is reserved for interaction, whenever a user interacts, we have to track his interaction and come up with a new component state. Whenever we have any data changes in a component, that data has to be declared in the *state*object.

Hence any user interaction is actually a state change.

**Note**: State of a component should be an **object literal**. For example,

1. this.state = {
2. counter:0,
3. flag:true,
4. ... ,
5. ...
6. }

The state's *counter*attribute would be accessed as *this****.state.counter.***

 We can only access the state, but we can not mutate state directly.

**Updating state - setState()**

On running the code on previous page, you would have observed that the code does not work. You would be getting a warning saying - *"Do not mutate state directly. Use setState()  react/no-direct-mutation-state*".

State of a component can be updated using ***setState()*** method.

* We can pass a new *object literal* to ***setState(****{..., ..., ...}****)***
* This new object literal should represent the updated state.

Consider the below updated example:

1. class HelloComp extends React.Component {
2. constructor() {
3. super();
4. this.state = {
5. counter: 0
6. };
7. }
8. incrementCounter = () => {
9. *//this.state.counter += 1; //trying to increment(or mutate) the state!*
10. this.setState({counter : this.state.counter + 1 }) *//updating state using setState()*
11. console.log("Button Clicked," , this.state.counter, "times" );
12. };
13. render() {
14. return (
15. <div>
16. <button onClick={this.incrementCounter}> Click </button>
17. <p>{this.state.counter}</p>
18. </div>
19. );
20. }
21. }

Whenever any state updates happens, *setState()* method causes re-rendering of a component to reflect the changes.

**Important Note**:

* If the attributes in the state object are existing, it will update them, else add the new attributes in the updated object literal.
* We can not re-assign the state without using *setState().*

Modify the ***incrementCounter***method as shown below. Re-run the code and observe.

1. incrementCounter = () => {
2. this.setState({counter : this.state.counter + 1 });
3. this.setState({counter : this.state.counter + 1 }); *//trying to use the updated state*
4. this.setState({counter : this.state.counter + 1 }); *//trying to use the updated state again*
5. console.log("Button Clicked," , this.state.counter, "times" );
6. };

The code does not work as intended. Instead of displaying "Button Clicked 3 times" it displays "Button Clicked 1 times". This is because setState() updates **what is currently rendered.**Since 0 is what is currently rendered, calling setState() multiple times updates 0 only.

If we want to update the state **based on previous state**,

* the *setState()*method can take a *callback* with a parameter. This param holds the previous state of the component.
* The callback should return a new object literal having the updated state attributes.
* The final syntax will be: *setState(****(****previousState****) => {****return****{..., ..., ...} }****)*

Observe the updated incrementCounter's code below:

1. incrementCounter = () => {
2. this.setState({counter : this.state.counter + 1 });
3. this.setState((prevState) => {
4. return {counter: prevState.counter + 1}
5. });
6. this.setState((prevState) => {
7. return {counter: prevState.counter + 1}
8. });
9. console.log("Button Clicked," , this.state.counter);
10. };

# Final note on setState()

* Do not call setState() inside render(). It causes an infinite loop.
* Do not call setState() in constructor. Use this.state = {...}, for initializing state for the first time.
* setState() should be called from events like button clicks or from life-cycle methods.

Demosteps:

Let us add an additional button for each employee to view their achievements. When the button is clicked their respective achievement must be displayed as shown below:

**Step 1:** Modify the **Employee.js** component as shown below:

1. class Employee extends React.Component {
2. constructor() {
3. super();
4. const emp1 = {
5. "empId": 100,
6. "empName": "Jack",
7. "age": 30,
8. "salary": 50000,
9. "image": "emp1.png",
10. "achievements": "Has got 3 bravo awards and 1 MVP award. Has worked on cutting edge technologies as well"
11. }
12. const emp2 = {
13. "empId": 101,
14. "empName": "Jane",
15. "age": 24,
16. "salary": 40000,
17. "image": "emp2.png",
18. "achievements": "No major achievements so far"
19. }
20. this.empArr=[emp1,emp2]
21. this.state={
22. selectedEmp:null
23. }
24. }
25. createCard(emp) {
26. var note = null;
27. var achievements=null;
28. if (emp.age < 25) {
29. note = <span className="text-info"> - Fresher</span>
30. }
31. if(this.state.selectedEmp==emp.empId){
32. achievements=<p><i>{emp.achievements}</i></p>
33. }
34. return (
35. <div key={emp.empId} className="card" style={{ "width": 200 }}>
36. <img className="card-img-top" src={emp.image} height="200" alt="Card image cap" />
37. <div className="card-body">
38. <h5 className="card-title text-center">{emp.empName}</h5>
39. <p className="card-text">
40. <span>Id: {emp.empId}</span><br />
41. <span>Age: {emp.age}</span> {note}<br />
42. <span>Salary: {emp.salary}</span><br />
43. </p>
44. {achievements}
45. <button type="button" className="btn btn-primary">Edit</button> <button className="btn btn-success" onClick={()=>{
46. this.setState({selectedEmp:emp.empId})
47. }}>View</button>
48. </div>
49. </div>
50. )
51. }
53. render(){
54. return (
55. <div>
56. <h3 className="text-center text-primary">Employee Details</h3>
57. {this.empArr.map(emp => {return this.createCard(emp)})}
58. </div>
59. )
60. }
61. }
62. export default Employee;

Observe that:

* onClick of the button **we are invoking an arrow function instead of a regular function.**This is because, we need to pass data of which empId was selected. If it was a regular function, we would have to pass the data as function parameter and thus it would become a function call. In other words, if we had used a regular function, that function would be called automatically even without clicking the button.

Using Props

# Creating attributes for component

Lets say that we want to create attributes called greeting and name. So that

**<HelloComp greeting="Evening" name="Jack"/>** should display Hello Jack, Good Evening and

**<HelloComp greeting="Morning" name="Jane"/>** should display Hello Jane, Good Morning.

This can be done by props.

# Props

Props is a special keyword in React. It is an object which allows you to pass data from one component to the other component.

Props are immutable - a component cannot change its props however it is responsible for putting together.

Props can be accessed as **this.props**.

In order to pass data to an component as attributes, we need to do two things:

1) Pass values as attributes wherever the component is being rendered.

2) Update the component to accept the data through props.

The updated**index.js** will look like below:

1. ReactDOM.render(<HelloComp greeting="Morning" name="Jack"/>, document.getElementById('root'));

In the above code we are passing two parameters to the *HelloComp*: **greeting**and **name**.

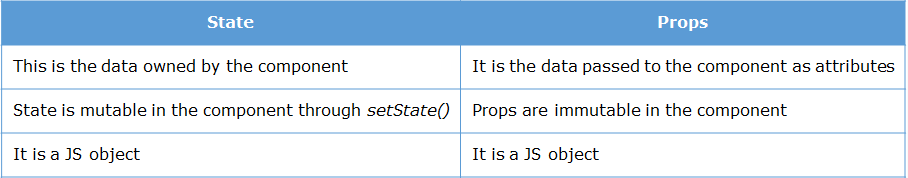
To access these values in the *HelloComp*, we need to use props as shown below:

1. class HelloComp extends React.Component {
2. render() {
3. return (
4. <h2>Hello {this.props.name}, Good {this.props.greeting}</h2>
5. );
6. }
7. }

The value of the props can be access by any method of the component using **this.props.**Here we are using**this.props** to access the *name*and *greeting*data passed as attributes.

**Props vs State**

Both state and props make up the data used by a component. A component converts this data into HTML presentation. Though both are data, they have certain similarities and differences as shown below:



**Note**: The component re-renders if props are updated from parent component.

Life-cycle Methods

The life-cycle methods are various methods provided by React, which are invoked at different phases of the life-cycle of a component.

Every component will have following phases in its life-cycle:

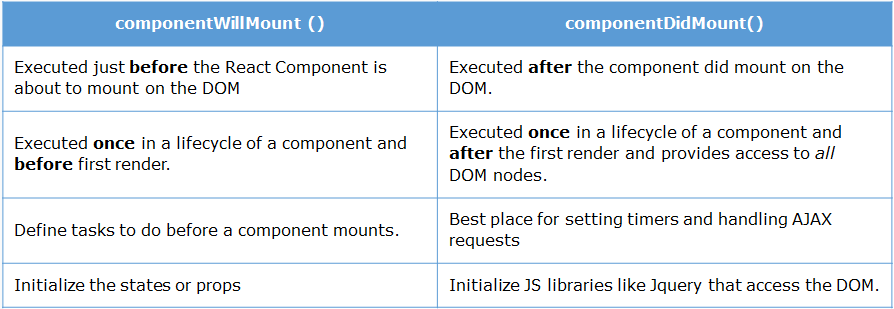
* **Initialization -**Initial component setup, setting up props and state
* **Mounting**- when the component is mounted to DOM tree
* **Updating** - when component is being updated with new state, new props are being received
* **Un-mounting** - destroying component from DOM tree

Every phase in the life-cycle of a component has few methods which will be invoked during that phase of a component's life-cycle. We can override these methods to provide the desired functionality.

**Mounting phase**

After preparing the React Component with basic needs, state and props, it is ready to mount in the browser DOM. This phase gives hook methods for *before*and *after mounting*of components.

The methods which gets called in this phase are *componentWillMount()* and *componentDidMount().*



**Unmounting phase**

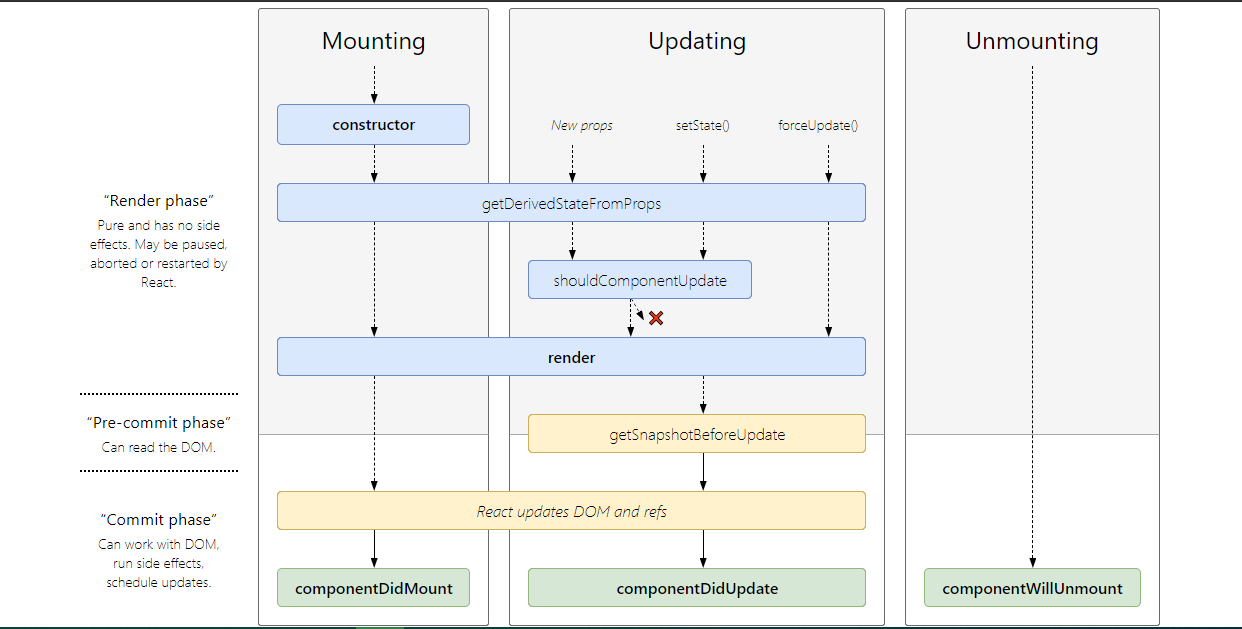
In this phase, the component is no longer needed and the component will get un-mounted from the DOM.

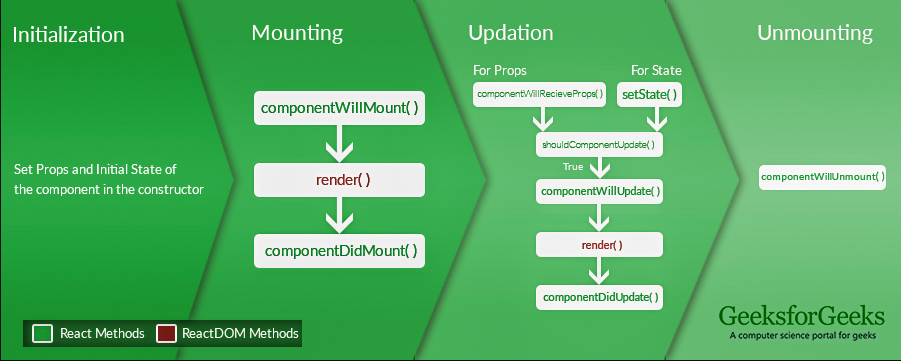
This is the **last**phase in a component's life-cycle. This is executed just **before**the component gets removed from the DOM.

# componentWillUnmount()

* This method is used at un-mounting process of a component
* By un-mounting the component, browser memory will be cleaned up
* A component can be also be unmounted from DOM using ReactDOM.unmountComponentAtNode()

**Usage:**This method is used to perform clean-up tasks associated with that component.   
For example, on logout, the user details and all the auth tokens can be cleared before un-mounting the main component.





<http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/>

**The Component Lifecycle**

Each component has several “lifecycle methods” that you can override to run code at particular times in the process. **You can use**[**this lifecycle diagram**](http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/)**as a cheat sheet.** In the list below, commonly used lifecycle methods are marked as **bold**. The rest of them exist for relatively rare use cases.

Mounting

These methods are called in the following order when an instance of a component is being created and inserted into the DOM:

* [**constructor()**](https://reactjs.org/docs/react-component.html#constructor)
* [static getDerivedStateFromProps()](https://reactjs.org/docs/react-component.html#static-getderivedstatefromprops)
* [**render()**](https://reactjs.org/docs/react-component.html#render)
* [**componentDidMount()**](https://reactjs.org/docs/react-component.html#componentdidmount)

**Note:**

These methods are considered legacy and you should [avoid them](https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html) in new code:

* [UNSAFE\_componentWillMount()](https://reactjs.org/docs/react-component.html#unsafe_componentwillmount)

Updating

An update can be caused by changes to props or state. These methods are called in the following order when a component is being re-rendered:

* [static getDerivedStateFromProps()](https://reactjs.org/docs/react-component.html#static-getderivedstatefromprops)
* [shouldComponentUpdate()](https://reactjs.org/docs/react-component.html#shouldcomponentupdate)
* [**render()**](https://reactjs.org/docs/react-component.html#render)
* [getSnapshotBeforeUpdate()](https://reactjs.org/docs/react-component.html#getsnapshotbeforeupdate)
* [**componentDidUpdate()**](https://reactjs.org/docs/react-component.html#componentdidupdate)

**Note:**

These methods are considered legacy and you should [avoid them](https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html) in new code:

* [UNSAFE\_componentWillUpdate()](https://reactjs.org/docs/react-component.html#unsafe_componentwillupdate)
* [UNSAFE\_componentWillReceiveProps()](https://reactjs.org/docs/react-component.html#unsafe_componentwillreceiveprops)

Unmounting

This method is called when a component is being removed from the DOM:

* [**componentWillUnmount()**](https://reactjs.org/docs/react-component.html#componentwillunmount)

Error Handling

These methods are called when there is an error during rendering, in a lifecycle method, or in the constructor of any child component.

* [static getDerivedStateFromError()](https://reactjs.org/docs/react-component.html#static-getderivedstatefromerror)
* [componentDidCatch()](https://reactjs.org/docs/react-component.html#componentdidcatch)

Further read: <https://reactjs.org/docs/react-component.html#the-component-lifecycle>

Functional Components

So far, we have created components as a class. But, React offers a shorter way to create a component.

Any function which returns a JSX is a component in React.

For example, the below function written in index.js is a valid component:

const Hello=()=><h1>Hello World</h1>

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

Such components created using functions are called as Functional components.

We can also pass data to it through props, though the keyword props is not used.

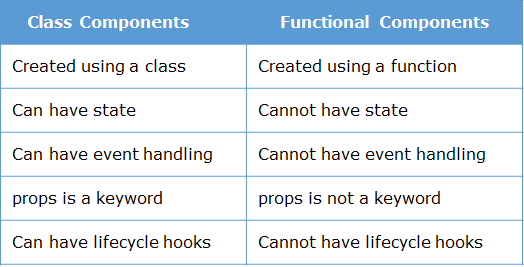
The props variable can be named anything. As shown below, we have named the props variable data:

const Hello=(data)=><h1>Hello {data.name}, Good {data.greeting}</h1>

ReactDOM.render(<Hello name="Jack" greeting="Afternoon"/>,document.getElementById('root'));

These functional components do not have state. Hence, they are also called Stateless components.

Difference between class and functional components are:



**Component life-cycle methods**

**Example:**

var course = [ {id: 1, name: "Angular", educator: "Rob"},

{id: 2, name: "ReactJS", educator: "John"},

{id: 3, name: "Ajax", educator: "Mack"}

];

class Educator extends React.Component {

constructor(props) {

super(props);

this.state = { courseData: course, selected: false };

console.log("Initialization Phase: 'constructor()' executed");

}

componentWillMount = () => {

console.log("Mounting Phase: 'componentWillMount()' executed");

}

render = () => {

var users = this.state.courseData;

console.log("Mounting Phase: 'render()' executed");

return (

<div>

<h4>UI Courses list</h4>

<div> {

users.map((userDetails)=> {

return (<div key={userDetails.id}>

<b>Course: </b>{userDetails.name} <br/>

<b>Educator: </b>{userDetails.educator} <br/>

<button onClick={this.clickHandler}>Details</button><br/><br/>

</div>);

})}

</div>

<button onClick={this.exitComponent}>Exit</button>

</div>

);

}

componentDidMount = () => {

console.log("Mounting Phase: 'componentDidMount()' executed");

}

clickHandler = () => {

console.log("Details button is clicked, state will change...");

this.setState({ selected: true });

}

exitComponent = () => {

console.log("Exit button is clicked, component will be destroyed");

ReactDOM.unmountComponentAtNode(document.getElementById('root'));

}

componentWillUnmount = () => {

console.log("Un-Mounting Phase: 'componentWillUnmount()' executed");

}

}

Basics of routing

To implement routing in our application, we first need to install a few modules that will help perform these functionalities.

npm install react-router-dom –save

React routing differs from Angular routing. In Angular, we declare routes before the app begins listening. All the routes are first defined and then imported to the top level module.

In React however, we will observe that routing take place as the app renders.

This implies that we can almost imagine routing to be a component by itself and we need to render it when we require some navigation in our application.

Our application will also require - BrowserRouter, Switch, Route and Link

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

2. **Switch**- Renders the first child <Route> or <Redirect> that matches the location.

3. **Link**- Provides declarative, accessible navigation around your application.

4. **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.

Using react-router

The first functionality we are going to implement is to render a new component on changing the URL.

To implement this, we first need to understand the use of the components we will add to our app.js file.

# ****<BrowserRouter />****

The first new component we use is called **<BrowserRouter>**

We use **BrowserRouter**because it can handle dynamic requests. This will be added to our App component present in **App.js** as shown below:

1. import { BrowserRouter } from 'react-router-dom';
2. const Home = () => <h2> Home </h2>;
3. const NewsFeed = () => <h2> News Feed </h2>;
4. class App extends Component {
5. render() {
6. return (
7. <BrowserRouter>
8. <React.Fragment>
9. ...content...
10. </React.Fragment>
11. </BrowserRouter>
12. );
13. }
14. }

As of now, the App component has the **BrowserRouter** directly present inside the render() method.

This means that the App component will be loaded in the root element of index.html and within this App component, the first component to be loaded will be the **BrowserRouter**which can handle incoming dynamic requests for various other child components like 'Home' or 'NewsFeed'.

Routing Navigable Links

**Link - to trigger the routing**

Our App component should have a navigation bar, which can have various links. On clicking on these links, the users are navigated other pages or more specifically, other components.

Observe the code below:

1. *//other imports...*
2. import { BrowserRouter as Router, Link } from 'react-router-dom';
3. class App extends React.Component {
4. render() {
5. return (
6. <Router>
7. <React.Fragment>
8. <nav>
9. <Link to={'/'}> Home </Link><br />
10. <Link to={'/news'}> News feed </Link><br />
11. <Link to={'/contact'}> Contact </Link><br />
12. </nav>
13. </React.Fragment>
14. </Router>
15. );
16. }
17. }
18. export default App;

# 

# <Link />

The <Link> element provides declarative, accessible navigation around our application. <Link/> is similar to an anchor <a> tag and helps us navigate to a different url.

The <Link> element has a prop called '***to***' , which help us specify the the new url that we want to navigate to.

In lines 8, 9 and 10, when user clicks on the 'Home' link, the url changes to http://localhost:3000/.

Similarly, when user clicks on the 'News Feed' link, the url changes to http://localhost:3000**/news**, and when user clicks on the 'Contact' link, the url changes to http://localhost:3000**/contact**.

But on clicking the link, only the URL changes. The corresponding component is not rendered yet.

**Route - to load the view**

As we saw earlier, the <Link> tag provides a clickable link for the user to navigate around the application. But only the URL gets updated and the component is not displayed yet.

Let's take a look at the App component present inside the **App.js** file.

1. *// other imports...*
2. import {BrowserRouter as Router, Route, Link } from 'react-router-dom';
3. const Home = () => <h2> Home </h2>;
4. const NewsFeed = () => <h2> News Feed </h2>;
5. class App extends React.Component {
6. render() {
7. return (
8. <Router>
9. <React.Fragment>
10. <nav>
11. <Link to={'/'}> Home </Link><br />
12. <Link to={'/news'}> News feed </Link><br />
13. <Link to={'/contact'}> Contact </Link><br />
14. </nav>
15. <div>
16. <Route exact path="/" component={Home}/>
17. <Route path="/news" component={NewsFeed}/>
18. <Route path="/contact" render={() => <h3>Contact Us</h3>}/>
19. </div>
20. </React.Fragment>
21. </Router>
22. );
23. }
24. }
25. export default App;

In the above code, 'Home' and 'NewsFeed' are components the need to be rendered based on URL change.

# <Route />

This tag is used to specify **which**component to render when the user clicks on a link and the URL is updated.

The Route component needs us to specify two props - path and component.

The path specifies the URL and component specifies **which component**should be loaded dynamically.

So we can say the <Route> tag is responsible for displaying the appropriate component after the URL is changed by clicking a link.

There are 2 ways to specify the component to be rendered:

1. Specify the component name as shown in lines 14 and 15

**Syntax**:

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

2. Use render() to specify inline rendering as in line 16

**Syntax**:

1. <Route path="/user" render={() => <div> User </div>}/>

The render prop allows for convenient inline rendering and wrapping without undesired remounting.

**Redirect**

In some cases, when the user clicks on a particular link, i.e the URL is pointing at a particular route, we want the user to be redirected to a different page or route.

As of now, the App component has the Route element to display or load the routes. Let us look at using Redirect to change the navigation from one URL to another.

Let look at the App component present inside the **App.js** file.

1. *// other imports...*
2. import { Link, Route, Redirect } from 'react-router-dom';
3. class App extends React.Component {
4. render() {
5. return (
6. <React.Fragment>
7. <nav>
8. <Link to={'/home'}> Home </Link><br />
9. <Link to={'/news-feed'}> News feed </Link><br />
10. <Link to={'/contact'}> Contact </Link><br />
11. </nav>
12. <div>
13. <Route exact path="/" render={() => (<Redirect to="/home">)} />
14. <Route path="/home" component={Home}/>
15. <Route path="/news-feed" component={NewsFeed}/>
16. <Route path="/contact" render={() => <h3>Contact Us</h3>}/>
17. </div>
18. </React.Fragment>
19. );
20. }
21. }
22. export default App;

In this case, if the user navigates to the root of the app, i.e the URL ends with just /, the URL will get updated to /home.

We want to redirect the user from /  to the /home page.

In the above code, we specify in the <Redirect> tag that if the URL is / , the user should be directed to the /home page instead.

We also configure the <Route> for /home so that when the URL is updated, the component is also rendered.

# <Redirect />

We will see the following props associated with Redirect:

1. **to**: string :- The URL to redirect to. Any valid URL path that path-to-regexp@^1.7.0 understands. All URL parameters that are used in to must be covered by from.

We use **to**for redirecting from a particular route path to a different path.

2. **push**: bool :- When true, redirecting will push a new entry onto the history instead of replacing the current one.

We use **push**so that the browser's back and forward buttons can navigate between the pages previously accessed by the user.

Handling parameterized routes

Sometimes, we need to pass certain data or parameters along with the URL, such that the corresponding component can be rendered. These parameters are called **route parameters**.

For example: The URL http://localhost:7777/profile/***James***should display the profile of James.

To render a component, based on the parameterized URL changes the route can be configured as shown:

1. <Route path="/profile/:username" component={UserProfile}/>

To an end user, this URL will not look any different from a non-parameterized URL.

But in the code, the"**:**" in front of username signifies the dynamic parameter.

# Using the parameter passed in the URL

In the above code snippet, **UserProfile**component gets rendered, only when a parameter is passed to the path.

Here, **username**is the route param. It is like we have declared a variable to store the parameter provided by the user. It makes sense to now use the passed parameter in the component we want to render, i.e, UserProfile component.

In UserProfile component, the route param value can be accessed as follows:

1. this.props.match.params.username

**match:** react-router-dom passes in a prop called match into every route that is rendered. Inside this match object there is an other object called params.

**params:** It is an object containing the passed URL parameters

**username:**In this example, the parameter is called username. The keyword has to match the keyword used in the route path.

**Switch - for rendering components**

In the previous slide, you saw how to pass parameters in the URL.

A problem that can come up in any app is that we have several named routes in addition to parameterized routes.

In React, when the URL matches multiple routes, all the corresponding components will be rendered.

Observe the code given below:

1. import { Route } from 'react-router-dom';
2. class App extends React.Component {
3. render() {
4. return (
5. <div>
6. <Route path={'/contact'} render={() => <h3>Contact Us</h3>} />
7. <Route path={'/:username'} render={({match}) => {return <h1> Hello {match.params.username}!</h1> }}/>
8. </div>
9. );
10. }
11. }

In this case, when the user hits the URL http://localhost:7777/contact, both the headings will be rendered! The URL will match the path '/contact' and also '/:username' since the second path is a parameterized route.

This is because in React, multiple matching routes will lead to multiple components being rendered.

So now let's take a look at how we can prevent this behavior.

# 

# <Switch />

Using the Switch tag around the Route tags will ensure that once the URL matches a route, no other routes will be matched. This means, only the first matching route and the corresponding component will be rendered.

Observe the code below:

1. import { Route, Switch } from 'react-router-dom';
2. class App extends React.Component {
3. render() {
4. return (
5. <div>
6. <Switch>
7. <Route path={'/contact'} render={() => <h3>Contact Us</h3>} />
8. <Route path={'/:username'} render={({match}) => {return <h1> Hello {match.params.username}!</h1> }}/>
9. </Switch>
10. </div>
11. );
12. }
13. }

Now, since we have encased the Route tags inside the Switch tags, once the URL matches the first route, no other routes are matched.

This means, the absolute paths should always be defined before the parameterized routes. If the URL does not match the absolute path, the parameterized paths will be matched.

If the parameterized paths are defined first, every URL will match the parameterized path instead of the absolute path, even if the route is defined!

# [Why do you need to import React multiple times in parent and child components?](https://stackoverflow.com/questions/44404730/why-do-you-need-to-import-react-multiple-times-in-parent-and-child-components)

In nodejs each file is a module, that has its own scope of variables. When you import variable into file (React for example) you add this variable to the module scope, but not to the global scope.

dirty hack to avoid importing React in every file with JSX. Just add React to window before ReactDOM.render.

import React from 'react';

window.React = React;

# [why to import React](https://stackoverflow.com/questions/36568791/why-to-import-react) even if we are not using it anywhere?

# ex-

import ReactDOM from 'react-dom'

import React, {Component} from 'react'

class App extends Component {

render () {

return (

<div>comp </div>

)

}

}

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

ReactDOM.render(<div>sv</div>, document.getElementById('root'))

Although you don't explicitly use the React instance you've imported, JSX is transpiled to [React.createElement()](https://facebook.github.io/react/docs/top-level-api.html#react.createelement) call, which uses it.

In your example, <div>comp </div> is transpiled by Babel to React.createElement('div', null, 'comp').

# Any time you use the shorthand <Component /> code what's actually happening is the JSX is transpiled by Babel into something that is regular Javascript.

# Forms

# Single Source of Truth

Let us learn how to create forms in React.

For this, let us take a simple example of a text field with a length validation as shown below:

We have already seen that React components have states.

When we create a form in React, the form has its own state as well. For example, the data entered by the user is part of the form and *not*of the component.

Thus, there are**two** different places where state is maintained: one in the form and one in the component.

React observes what is called a "**single source of truth**". That means that the state must be maintained in only one place.

We will see how we can maintain the state in the component and make the component control the form.

Let us consider a simple form with a text box and no button. The goal is to display some default text in the text box when the page loads. For this, we will create a state variable and assign a value to it. This state variable will be used for the value attribute of the text box.

The *HelloComp*component is modified as shown below:

1. class HelloComp extends React.Component {
2. constructor(props) {
3. super(props);
4. this.state = {
5. txtUsername: "Infosys"
6. };
7. }
8. render() {
9. return (
10. <form>
11. <input type="text" name="txtUsername" value={this.state.txtUsername} />
12. </form>
13. );
14. }
15. }

In the above code, the state exists only in one place, which is the component. The text box will now display **Infosys**.

Till now, we have seen how to maintain single source of truth and how to set predefined value in the text box. However, we will not be able to modify/change the text in the text box. Next, we will see how to update the text in text box as soon as the user changes/modifies the text.

We had created a simple form with a text box with a predefined value.

However, if we try to modify it it in the text box, we will find that we are unable to change the value in the text box. This is because the form now does not have it's own state.

*The state is in the component.*

To change the text in the text box, we need to change the state value in the component.

For this, we need to do three things:

1. Create a method for onChange event of the text box. This will allows to capture the data entered by the user when trying to change the text
2. Get the data of the text box by using **event.target.value**. This is a JavaScript DOM code which allows us to access the value of the DOM element on which the event had taken place
3. Update the state variable with the value entered by the user. This will in turn update the value in the text box.

The below code illustrates this concept:

1. class HelloComp extends React.Component {
2. constructor(props) {
3. super(props);
4. this.state = {
5. txtUsername: "Infosys"
6. };
7. this.getName = event => {
8. this.setState({ txtUsername: event.target.value });
9. };
10. }
11. render() {
12. return (
13. <form>
14. <input type="text" name="txtUsername" value={this.state.txtUsername}
15. onChange={this.getName}
16. />
17. </form>
18. );
19. }
20. }

**Note**: In normal HTML, onChange on a text box will fire when you have changed the text *and* also done a blur event. In JSX, onChange works like keyup.

Error Handling in Forms

We have seen how to maintain the state in the component and how to access the value entered by the user. Let us see how to perform validation of the message.

We can perform our validation logic in the event handler method. To display the error messages, we have to set the message as part of the state and display the message from the state inside a tag.

Usually, we maintain a state variable called *formErrors*which will be an object to store error messages for all the fields in the form.

In JavaScript, we can dynamically add keys and delete keys from an object. Therefore, before setting an error message in this object, we need to retrieve the object from the state variable and modify only the relevant error message. Else, we will end up modifying the whole object.

This is illustrated in the below code:

1. class HelloComp extends React.Component {
2. constructor(props) {
3. super(props);
4. this.state = {
5. txtUsername: "Infosys",
6. formErrors: {
7. txtUsernameErr:''
8. },
9. fieldValidity: {
10. txtUsername: false
11. },
12. formValid: false
13. };
14. ValidateUsername = (event) => {
15. const name = event.target.value;
16. var formErrors = this.state.formErrors;
17. var fieldValidity = this.state.fieldValidity;
18. this.setState({txtUsername:event.target.value});
19. if (name.length < 5) {
20. formErrors.txtUsernameErr = "Cannot be less than 5";
21. fieldValidity.txtUsername = false;
22. } else {
23. formErrors.txtUsernameErr = "";
24. fieldValidity.txtUsername = true;
25. }
26. this.setState({ formErrors: formErrors });
27. this.setState({ formValid: fieldValidity.txtUsername})
28. };
29. }
30. render() {
31. return (
32. <form>
33. <input type="text" name="txtUsername" value={this.state.txtUsername}
34. onChange={this.ValidateUsername}
35. />
36. <div id="errorMsg">{this.state.formErrors.txtUsernameErr}</div>
37. </form>
38. );
39. }
40. }

         The **onChange**event in the line 37, will invoke the function **ValidateUsername**in the line 16. The function *ValidateUsername*takes *event*as a parameter.

In line 20, the state's *txtUsername*value is set to user entered value. In lines 20-29, user input is validated. On failure, the state's *txtUsernameErr*variable is set to appropriate error message.

Introduction to Axios

**Making HTTP request in React application**

Most, if not all, moderns applications are data driven.

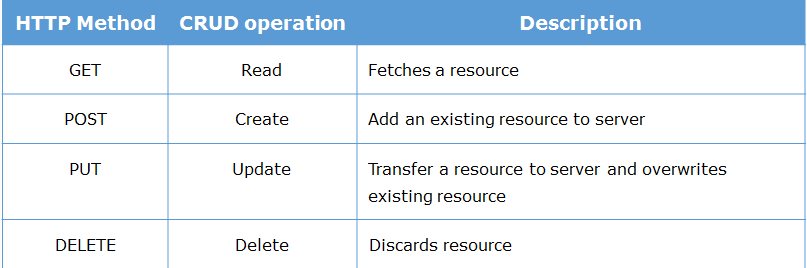
This means these applications communicate with one or more web services to consume data, update data, or display dynamic data. Data can change based on which user is logged in, or the location from where the user logged in, and correspondingly show different results on the webpage.

These web services have RESTful APIs that can easily be consumed in an application by using HTTP requests. React application use HTTP to communicate the user's interaction to the web services and to return the requested content for the user to browse.

When the user clicks on a link, a new HTTP request is generated to fetch the required content from the server. Once the requested content has been found, the server then generates a response. This response contains the content. The user will be able to see or interact with the content once it has been rendered.

As you have seen earlier, HTTP has 4 main methods that we are familiar with: GET, POST, PUT and DELETE.

To recap, given below is a short description of these methods and when to use them.



**HTTP libraries for React**

So, how do we use HTTP in our React application?

There are several libraries that can help handle HTTP functionalities, for example:

1. [Fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API): This API has been a developer favorite, thanks to the familiarity of XMLHttpRequest combined with a more powerful and flexible feature set.

2. [SuperAgent](https://www.npmjs.com/package/superagent): An API that was created to be flexible, lightweight and readable. This API was created with an easy learning curve and favors callbacks over promises.

3. [Axios](https://www.npmjs.com/package/axios): While it is similar to the Fetch API, Axios prefers using promises and is based on the $http service within Angular.js.

Axios has some advantages over Fetch:

* It can automatically convert a response to JSON
* A request can be cancelled
* It has wide range of browser support.

Thus, we will be using Axios as we progress in this course.

**Getting started with Axios**

To start using Axios in our project, first we need to install it using the command given below.

1. npm install axios --save

Then, we need to include Axios in our component using *import*.

1. import axios from 'axios';

**GET request using Axios**

To perform a GET request using axios, we can use*axios.get(URL)* function which takes one parameter which is the string URL and returns a Promise object.

The *.then()* block is used to handle promise response and *.catch()* block is used to handle errors, if any:

1. axios.get('URL')
2. .then(function (response) {
3. console.log(response);
4. })
5. .catch(function (error) {
6. console.log(error);
7. });

Now we have created a GET request using Axios. What about the response?

The response generated from the server will contain some data. This is the data requested by the user. In order to access this data, we need to use *response.data.*This data, from the server's response, will be in JSON format.

1. axios.get('URL')
2. .then(function (response) {
3. console.log(response.data); *// This will print response data received from web service.*
4. })

We can make a GET request to any back-end URL as shown below:

1. axios.get('./public/data.json')
2. .then(function (response) {
3. console.log(response.data); *// This will print response data received from web service.*
4. })

**Note:-** React and vanilla Bootstrap don't really mix well. Bootstrap is supposed to alter DOM elements on DOM element changes. React has this concept of a virtual DOM to render the page based on what changed since previous render.

For React with Bootstrap, you can probably check libraries like <https://react-bootstrap.github.io/>, which are rebuilt to work with React.

**Using Axios in component**

We want to display data received from Axios GET request in component. So it makes sense to write the Axios request inside the same component. We will be writing all Axios request method inside component.

Also, sometime we want Axios method to be invoked when component gets loaded. To achieve this, we make use of **componentDidMount()**life-cycle method.

Next, where should we store data received from web service? We generally keep all of component data in state, so we will store this data in state by using setState() method.

Let's create a component which makes GET request to URL "http://localhost:2000/retreiveEmployeelist" and display data received from web service.

1. import React, { Component } from "react";
2. import axios from "axios";
3. class AppComp extends Component {
4. constructor(props) {
5. super(props);
6. this.state = {
7. result: ""
8. };
9. this.fetchEmployees = this.fetchEmployees.bind(this);
10. }
11. componentDidMount() {
12. this.fetchEmployees();
13. }
14. fetchEmployees() {
15. axios
16. .get("http://localhost:2000/retreiveEmployeelist")
17. .then(response => {
18. this.setState({
19. result: response.data,
20. error: ""
21. });
22. });
23. }
24. render() {
25. const { result } = this.state;
26. return (
27. <div>
28. <div className="row">
29. <div className="col-8 offset-2">
30. <h3>Employee Data:</h3>
31. <br />
32. {result ? <Table list={result} /> : null}
33. </div>
34. </div>
35. </div>
36. );
37. }
38. }
39. const Table = ({ list }) => (
40. <table className="table table-striped">
41. <thead>
42. <tr>
43. <th>Employee Id</th>
44. <th>Employee Name</th>
45. <th>Age</th>
46. <th>Employee Achievement</th>
47. <th>Employee Salary</th>
48. </tr>
49. </thead>
50. <tbody>
51. {list.map(item => (
52. <tr key={item.empId}>
53. <td>
54. <span>{item.empId}</span>
55. </td>
56. <td>
57. <span>{item.empName}</span>
58. </td>
59. <td>
60. <span>{item.age}</span>
61. </td>
62. <td>
63. <span>{item.achievements}</span>
64. </td>
65. <td>
66. <span>{item.salary}</span>
67. </td>
68. </tr>
69. ))}
70. </tbody>
71. </table>
72. );
73. export default AppComp;

**Axios POST request**

To make POST request using axios, we have a method *axios.post(url, data)*where first parameter is URL string and second parameter is JSON data.

1. axios.post('URL', postData)
2. .then(function (response) {
3. console.log(response);
4. })
5. .catch(function (error) {
6. console.log(error);
7. });

Similar to GET request, *response.data*is used to get response data received from web service. Its will return response in JSON format.

1. axios.post('URL', data)
2. .then(function (response) {
3. console.log(response.data); *// This will print response data received from web service.*
4. })
5. addEmployee = () => {
6. var formJSON = {
7. empName: this.state.empName,
8. age: this.state.age,
9. salary: this.state.salary,
10. achievements: this.state.achievements
11. };
12. axios
13. .post("http://localhost:2000/InsertEmployee/", formJSON)
14. .then(response => {
15. this.setState({ successMessage: response.data.message, error: "" });
16. console.log(response.data);
17. })
18. .catch(error => {
19. if (error.response) {
20. this.setState({ error: error.response.data.message, success: "" });
21. } else {
22. this.setState({ error: error.message, success: "" });
23. }
24. });
25. };

Error Handling in Axios

Handling errors is a very important aspect of any HTTP request as it helps us know whether the request got fulfilled successfully or some error occurred in its process. Also, if some error has occurred, the developer would need to identify that error and resolve it.

We can handle errors by adding the .catch() method to the end of the promise chain.

1. axios.get('URL')
2. .then(function (response) {
3. console.log(response);
4. })
5. .catch(function (error) {
6. console.log(error.response);
7. console.log(error.response.data);
8. console.log(error.message)
9. });

.catch() method will handle an error with the following attributes:

**error.response -**It is the error response received from the web service which has details like data, status code and headers.

**error.response.data -**It will have response data returned from the web service in JSON format.

**error.message -**When there are no responses from the web service, or if something goes wrong while sending a request, we can use this attribute to display a different error message. It can have error string or error status code or both.

To handle errors properly, first we check for a response from web service, if a response is available, we display response error message otherwise we display the message from error.message.

1. axios.get('URL')
2. .then(function (response) {
3. console.log(response);
4. })
5. .catch(function (error) {
6. if (error.response) {
7. console.log(error.response.data.message);
8. } else {
9. console.log(error.message);
10. }
11. });

This approach has advantages since we are handling the errors regardless of whether an error response came from the web service or not. In this course, we will continue handling errors in the above manner.

Let's see how we can handle errors in the Employee component we created previously:

1. fetchEmployees() {
2. axios
3. .get("http://localhost:2000/retreiveEmployeelist")
4. .then(response => {
5. this.setState({
6. result: response.data,
7. error: ""
8. });
9. })
10. .catch(error => {
11. if (error.response) {
12. this.setState({ error: error.response.data.message, result: "" });
13. } else {
14. this.setState({ error: error.message, result: "" });
15. }
16. });
17. }

We add a new property called error in component state. In case of an error response, first we will check if there is some error response data received from back-end.

If yes, we will assign the same response data to the error state.

Otherwise, we will assign the error.message to error state.

In PUT request, send empID which already exist in URl parameter and editable details into body of request. Write a method updateEmployee() in component which make a PUT request to backend.

1. updateEmployee = () => {
2. var formJSON = {
3. empId: this.props.match.params.empId,
4. empName: this.state.empName,
5. age: this.state.age,
6. salary: this.state.salary,
7. achievements: this.state.achievements
8. };
9. axios
10. .put(
11. "http://localhost:2000/updateEmployeeDetails/" +
12. this.props.match.params.empId,
13. formJSON
14. )
15. .then(response => {
16. this.setState({ successMessage: response.data.message, error: "" });
17. console.log(response.data);
18. })
19. .catch(error => {
20. if (error.response) {
21. this.setState({ error: error.response.data.message, success: "" });
22. } else {
23. this.setState({ error: error.message, success: "" });
24. }
25. });
26. };

# Tools for testing React component

There are many tools for testing React components. Few of them are **chai, mocha, jasmine, karma, enzyme, jest** and others.

Jest

Jest is an open source, zero configuration testing tool created by Facebook, which is used to test Javascript code including React components. Facebook itself uses jest for testing React applications. Jest also provide **snapshot testing**which make it more suitable for React application. We will learn about snapshot testing later in this course.

**Installing jest:**

When you create a new React application using create-react-app, **jest**comes with it so you do not have to do anything to install **jest**.

For rendering snapshot, we need react-test-renderer which we have to install. To install it, open terminal and type below command in project root directory:

1. npm install --save-dev react-test-renderer

It will install react-test-renderer as well as add it as dev dependency in package.json file.

**Test file naming convention for Jest:**

Since jest is zero configuration testing tool, we do not need to specify any configuration to locate test files. Any files which is placed inside **src/\_\_tests\_\_**directory or file name having extension **.spec.js**or **.test.js** will be automatically captured as test file by jest.

In this course, we encourage you to have test file in same directory where component file resides. We will use **.test.js**extension to name test file in this course. You can choose any of above mentioned methods, but we will encourage you stick to **.test.js**while following this course for simplicity.

**Jest Matchers**

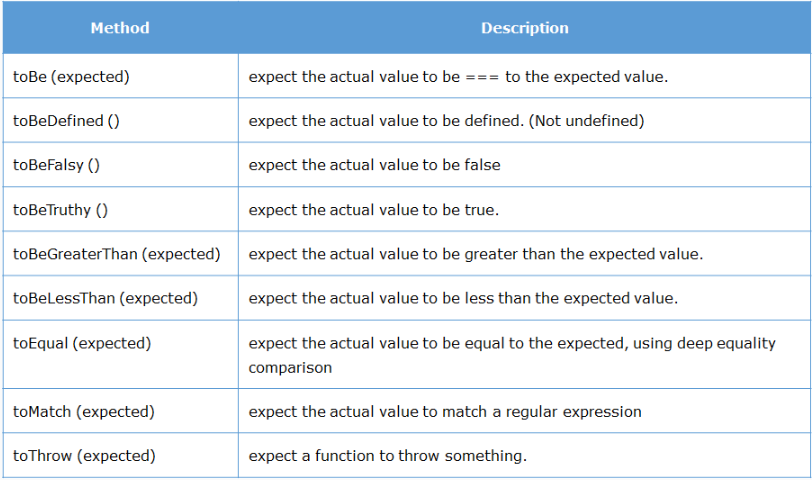
In Jest, matchers are used to test value in various ways. Matchers are used on expect() function.

Lets see a very simple matcher example:

1. expect(2 + 2).toEqual(4);

**toEqual**is a matcher here and it is used to check that expect value is same as value passed to toEqual.

Few commonly used Jest matchers are mentioned below table:



**Creating counter app for testing**

**Step 2:3**

In **src/App.js**we will create a counter state and two methods to increment and decrement counter which will increment and decrement counter by click of their respective button. Paste below code in **src/App.js** file.

1. import React, { Component } from "react";
2. export const doIncrement = prevState => ({
3. counter: prevState.counter + 1
4. });
5. export const doDecrement = prevState => ({
6. counter: prevState.counter - 1
7. });
8. class App extends Component {
9. constructor() {
10. super();
11. this.state = {
12. counter: 0
13. };
14. this.onIncrement = this.onIncrement.bind(this);
15. this.onDecrement = this.onDecrement.bind(this);
16. }
17. onIncrement() {
18. this.setState(doIncrement);
19. }
20. onDecrement() {
21. this.setState(doDecrement);
22. }
23. render() {
24. const { counter } = this.state;
25. return (
26. <div>
27. <h4>My Counter</h4>
28. <Counter counter={counter} />
29. <button type="button" className="increment" onClick={this.onIncrement}>
30. Increment
31. </button>
32. &nbsp;
33. <button type="button" className="decrement" onClick={this.onDecrement}>
34. Decrement
35. </button>
36. </div>
37. );
38. }
39. }
40. export const Counter = ({ counter }) => <p>{counter}</p>;
41. 3
42. export default App;

This will create a counter state which we can increment or decrement by clicking their respective buttons.

**Step 3:**

Open terminal and run this app:

1. npm start

**Testing with Jest**

**Step 1:**

Go to the counter app created in previous demo.

Open **src/App.test.js** file and import doIncrement and doDecrement.

1. import { doDecrement, doIncrement } from "./App";

You need to import to use in **App.test.js** file.

**Step 2:**

Create test case for doIncrement and doDecrement as shown below:

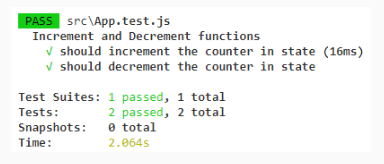
1. describe('Increment and Decrement functions', () => {
2. test('should increment the counter in state', () => {
3. const state = { counter: 0 };
4. const newState = doIncrement(state);
5. expect(newState.counter).toEqual(1);
6. });
7. test('should decrement the counter in state', () => {
8. const state = { counter: 0 };
9. const newState = doDecrement(state);
10. expect(newState.counter).toEqual(-1);
11. });
12. });

These test cases will check that increment and decrement functions are working properly.

**Step 3:**

Open terminal and type below command:

1. npm test



Enzyme

# Need for Enzyme

In the previous demo, we tested doIncrement and doDecrement methods which were outside of component. We also have functions, state and UI inside the component which we need to test. In order to test anything inside component, we need to render component. Enzyme provides functions using which we can very easily render component and test component functions, state and markup.

Enzyme helps us to easily assert, manipulate and traverse React component output. Enzyme provides many functions which make testing React component very easy. Without Enzyme testing React component will take more time and effort. It also helps us in reducing code complexity.

Therefore, we will be using Enzyme to test React component in this course.

# Enzyme

Enzyme is a JavaScript testing utility for React created by Airbnb, which provide methods to easily assert, manipulate and traverse React component output.

It provides three methods for rendering react component which are **shallow, mount**and **render**. We will learn about these methods in detail later.

**Installing enzyme:**

To use enzyme with React, we need to install enzyme along with an adapter with corresponding React version. We are using React 16 in this course, so we will be using  enzyme-adapter-react-16 as adapter.

Open terminal and type below command inside project root directory:

1. npm install --save-dev enzyme enzyme-adapter-react-16

You have successfully installed enzyme along with adapter.

**Configuring enzyme adapter:**

To use enzyme in test files, we need to configure adapter. By default, create-react-app takes testing configuration from **src/setupTests.js** file. If we create an adapter in this file, we will be able to access this adapter in every test file of our React application.

Create a file **src/setupTests.js** and create an enzyme adapter as shown below:

1. import { configure } from "enzyme";
2. import Adapter from "enzyme-adapter-react-16";
4. configure({ adapter: new Adapter() });

This will configure enzyme adapter and it will be accessible to all test files which means we can use enzyme in all test files without creating an adapter in any test file.

**Enzyme functions**

Enzyme provide three functions to render component. They are **shallow, mount** and **render.**Lets understand each of them.

# ****Shallow****

Shallow rendering will only render the component, it does not render its children. This helps in rendering component as a unit independent of its children. Any changes in child components do not affect rendering of parent component. This means if there is a bug in child component, it will not break parent component test cases.

We use shallow function when we want to run some tests on a parent component, while ensuring that we are not indirectly asserting the behavior of the child component.

To render component using shallow function, just pass component as a parameter to shallow function as shown below:

1. import { shallow } from "enzyme";
3. const wrapper = shallow(<App />);

It will render App Component without its children. wrapper will be of type ShallowWrapper and we can call any ShallowWrapper method on wrapper which we will learn in next page.

# Mount

Mount rendering will render the component as well as all of its children, which means it renders the complete component tree. This is also know **Full DOM rendering**.

Since mount rendering will mount the component along with its children to DOM, tests are more likely to break since they contain all children component and their logic.

To render component using mount, you need to pass component as a parameter to mount function:

1. import { mount} from "enzyme";
3. const wrapper = mount(<App />)

This will render complete component tree.

# Unmount

Once the component is mounted and elements are rendered, to complete the component life-cycle, the component needs to finally unmount.

To simulate this component unmounting, we can use the unmount method of enzyme.

The syntax of using this method is as shown below:

1. wrapper.unmount()

Of course, a component can be unmounted only if it has been mounted. Similarly, the unmount method of enzyme should also be used only after the mount method, hence the binding with wrapper.

# Render

Render function will render the component along with its children to static HTML and analyze the resulting HTML structure. It is useful when you want to access all child component without their life-cycle methods. It is also known as **Static rendering**.

To render a component using render function, you need to pass component as a parameter to render function:

1. import { render } from "enzyme";
3. const wrapper = render(<App />);

This will render App component as well as all its children.

**NOTE:**In this course we will be only using **Shallow**rendering as we want to test out component as a unit independent of its children component so that test cases do not break even when children component gets changed.

**ShallowWrapper methods**

ShallowWrapper is the type of wrapper instance which is returned after shallow rendering of component. ShallowWrapper provides many methods which makes very easy to access element of rendered component.

Some of popular ShallowWrapper methods are mentioned below which we will be using in this course:

**1. find(selector):** find() method takes selector as an argument and find every node which matches selector.

1. const wrapper = shallow(<App />);
2. wrapper.find("button");

The above example will find all *button*in rendered component.

**2. state():**state() method return state of the rendered component.

1. const wrapper = shallow(<App />);
2. expect(wrapper.state().counter);

The above example will return the *counter* state of the component.

**3. props():**props() return props of the given component.

1. const wrapper = shallow(<App />);
2. let counterWrapper = wrapper.find(Counter);
3. expect(counterWrapper.props());

**4. setState():**setState() is used to manually set state of rendered component.

1. const wrapper = shallow(<App />);
2. wrapper.setState({ counter: 1 });

The above example will set state of counter to 1.

**5. simulate(event):**simulate() is used to simulate an event on current node. It takes event as parameter.

1. const wrapper = shallow(<App />);
2. wrapper.find("button").simulate("click");

The above example will click the button.

You can find a complete list of ShallowWrapper methods [here](http://airbnb.io/enzyme/docs/api/shallow.html#shallowwrapper-api). In this course, we are going to use only above mentioned ShallowWrapper methods.

**Create test cases using shallow rendering**

Demo steps:

**Step 1:** Open '**counter-test-app**' created in previous demo in VSCode. Open terminal, go to '**counter-test-app'** directory and run below command in terminal:

1. npm install --save-dev enzyme enzyme-adapter-react-16 react-test-renderer

It will install enzyme, enzyme-adapter and react-test-renderer. We need to install enzyme and enzyme adapter to use enzyme.

**Step 2:** We need to setup enzyme adapter so that it can be accessible from anywhere. Create a file **setupTests.js**inside **src/**directory. Configure enzyme adapter in **src/setupTests.js** as shown below:

1. import { configure } from "enzyme";
2. import Adapter from "enzyme-adapter-react-16";
3. configure({ adapter: new Adapter() });

This will configure an enzyme adapter and it will be accessible to all test files.

**Step 3:**Go to **src/App.test.js** file and if there is any code, delete it.

Import React, App and Counter component and shallow from enzyme in **src/App.test.js**file.

1. import React from "react";
2. import App, { Counter } from "./App";
3. import { shallow } from "enzyme";

We have successfully imported App, Counter and shallow.

**Step 4: describe**function is used to create a test suite in jest. It is similar to jasmine testing which you have learnt in Modern JavaScript course. **describe**function takes two parameter, first is an string to describe the test suite and second function is a callback function in which we will write all test cases.

In **src/App.test.js**file, create a test suites for App component as shown below:

1. describe("App Component", () => {
2. *// test cases will be written here.*
3. });

This will create a test suite to write test cases for App component. In next step, we will add test cases inside this test suite.

**Step 5: test** function is used to create test cases in **jest**. **test** function takes two parameter, first parameter is a string describing the test case and second parameter is a callback function where test case will be written.

In  **src/App.test.js**file, in **describe** callback function, write a test case.

1. describe("App Component", () => {
2. test("renders the Counter wrapper", () => {
3. const wrapper = shallow(<App />);
4. expect(wrapper.find(Counter).length).toEqual(1);
5. });
6. });

Inside **describe**function we have created a test where we are rendering App component, finding Counter inside component and we are checking its length. Since we used Counter only once inside App component, its length should be 1.  If there will be only one Counter inside App component, this test case will pass.

**Step 6:**Similar to previous step, let's write **test**functions for increment function as well as decrement function.

In  **src/App.test.js**file, write test cases for increment and decrement functions.

1. describe("App Component", () => {
2. test("increments the counter", () => {
3. const wrapper = shallow(<App />);
4. wrapper.setState({ counter: 0 });
5. wrapper.find("button.increment").simulate("click");
6. expect(wrapper.state().counter).toEqual(1);
7. });
8. test("decrements the counter", () => {
9. const wrapper = shallow(<App />);
10. wrapper.setState({ counter: 0 });
11. wrapper.find("button.decrement").simulate("click");
12. expect(wrapper.state().counter).toEqual(-1);
13. });
14. });

In both test cases, we are shallow rendering App component, setting counter state to 0, finding respective buttons for increment and decrement using selector with the help of classname. Then we are simulating a click of each button and testing the counter value.

**Step 7:**Let's create a test case to check props also. In App component, we are passing counter a props to Counter component. Let's write a test case for it also.

In **src/App.test.js**file, write test case inside describe function callback:

1. test("passes all props to Counter wrapper", () => {
2. const wrapper = shallow(<App />);
3. let counterWrapper = wrapper.find(Counter);
4. expect(counterWrapper.props().counter).toEqual(0);
5. wrapper.setState({ counter: -1 });
6. counterWrapper = wrapper.find(Counter);
7. expect(counterWrapper.props().counter).toEqual(-1);
8. });

In this test case, we are testing props value of Counter initially and then checking it again after update counter state.

**Step 8: src/App.test.js**file after adding all above code is given below:

1. import React from "react";
2. import App, { Counter } from "./App";
3. import { shallow } from "enzyme";
4. describe("App Component", () => {
5. test("renders the Counter wrapper", () => {
6. const wrapper = shallow(<App />);
7. expect(wrapper.find(Counter).length).toEqual(1);
8. });
9. test("passes all props to Counter wrapper", () => {
10. const wrapper = shallow(<App />);
11. let counterWrapper = wrapper.find(Counter);
12. expect(counterWrapper.props().counter).toEqual(0);
13. wrapper.setState({ counter: -1 });
14. counterWrapper = wrapper.find(Counter);
15. expect(counterWrapper.props().counter).toEqual(-1);
16. });
17. test("increments the counter", () => {
18. const wrapper = shallow(<App />);
19. wrapper.setState({ counter: 0 });
20. wrapper.find("button.increment").simulate("click");
21. expect(wrapper.state().counter).toEqual(1);
22. });
23. test("decrements the counter", () => {
24. const wrapper = shallow(<App />);
25. wrapper.setState({ counter: 0 });
26. wrapper.find("button.decrement").simulate("click");
27. expect(wrapper.state().counter).toEqual(-1);
28. });
29. });

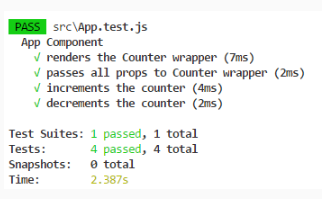
In this file, we have one test suite and four test cases which we have created in previous step.

**Step 9:**Its time to run test cases. To run tests cases, open terminal and type below command:

1. npm test

It will run all test cases.

Output of test cases will be similar to given image.



**Snapshot testing**

# Need for snapshot testing

We have tested the logic and the functionality of counter app till now, but we also need to test UI of React component to make sure that no unexpected changes occurs in UI. If any changes occur, we want test cases to inform us that some changes are made in UI so that we can check if these changes are made purposefully or unexpectedly.

In counter app, we display counter state and two buttons, increment button and decrement button. Increment button is displayed first and decrement button is displayed later. To make sure that button position does not change accidentally, we make use of snapshot testing. Snapshot testing is used to test UI and markup of component. Snapshot testing will fail test cases if we change button position or make any other UI changes. This helps us prevent unexpected UI changes.

# Snapshot testing

Snapshot testing is testing of UI and markup. It creates a snapshot of rendered component when it runs for the first time. When we run tests for the next time, it will compare difference between new rendered component snapshot and saved snapshot.

If both snapshot will match, test case will pass otherwise it will fail.

If test cases fail, it will inform you where test cases failed and ask you if you have made theses changes and want to keep it.

You can update it and snapshot will also get updated and test cases will pass further.

If those changes are needed, you can go to component and change it as per snapshot.

**Creating snapshot test cases**

To create a snapshot test case, we need to import renderer from react-test-renderer which will help us create snapshot of any component.

1. import renderer from "react-test-renderer";

Once we imported renderer, we can create snapshot of any component.

# renderer.create()

renderer.create() method creates the snapshot of any component which is passed to it as parameter. You can create snapshot as shown below:

1. const component = renderer.create(<App />);

By passing component to renderer.create() function, it will create a snapshot.

Jest store snapshots in a text file and for this we need to create a JSON file of snapshot. **toJSON()**function is used to create JSON tree for any snapshot which needs to be used later.

To convert any snapshot into JSON tree, invoke toJSON() on snapshot:

1. const component = renderer.create(<App />);
2. let tree = component.toJSON();

This will create a JSON tree of snapshot which we will use for testing.

# toMatchSnapshot()

toMatchSnapshot() is a jest matcher which is used to match snapshot by comparing JSON tree. We can invoke this function as shown below:

1. const component = renderer.create(<App />);
2. let tree = component.toJSON();
3. expect(tree).toMatchSnapshot();

This will be used to match snapshot.

Complete snapshot test code:

1. describe("App Snapshot", () => {
2. test("renders", () => {
3. const component = renderer.create(<App />);
4. let tree = component.toJSON();
5. expect(tree).toMatchSnapshot();
6. });
7. });

When we run this test case for first time, jest will create a snapshot inside **src/\_\_snapshots\_\_** folder. After that, whenever we run snapshot test in future, it will compare new snapshot with this saved snapshot. If they will match, test cases will pass otherwise it will fail.

Redux

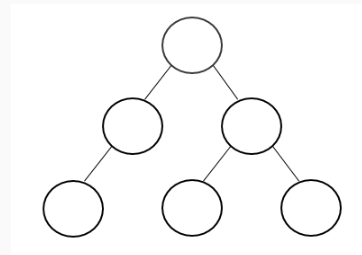
**Redux**is a component based library developed by **Dan Abramov** and **Andrew Clark**in 2015 with an inspiration from **Facebook's Flux**and **Elm**(functional programming language) for easing the development of react applications. It became popular in the developers' community very soon due to its small size, simplicity and detailed documentation.

Basically, Redux is a tool using which we can manage the data state and UI state of any react application in a much easier manner. Redux also makes the application more consistent.

Redux takes away some of the hassles faced with state management in large applications.

**Why Redux**

# How Redux would help in scenarios where the application has components nested at multiple levels?



As you know, React flows data in one direction i.e. from parent to child component, communicating between two child components is very difficult and it is not recommended to have direct component to component interaction as shown above as it is more error prone and code becomes very unstructured and difficult to maintain and follow.

Such are the scenarios where the need for Redux arises. Redux helps in easing out the state management by storing all the state's of the application (across all the components) at a single place called as "Store".

# For State management

In larger apps with a lot of moving pieces, state management becomes a huge concern. **Redux**takes away that effort quite well without performance concerns or trading off testability.

It can be ideally used for developing SPA's where state management becomes complex over time. While Redux was developed keeping React in mind, but it is framework independent and can be used even with angular or jquery applications.

One other reason a lot of developers love Redux is the developer experience that comes with it. Redux takes away some of the hassles faced with state management in large applications. It provides us a great developer experience, and makes sure that the ease of testing our app isn’t sacrificed for any of those.

Some of the nice things we get with using Redux include:

* logging
* hot reloading
* time travel
* universal apps
* record and replay — all without doing so much on your end as the developer.

# An Analogy: A visit to the Library

Let’s consider an event that we all are likely to be familiar with — visiting a library to borrow a few books.

Let's say, one morning you wake up and want to read some books.

Rather than buying them, you decide to visit your local library to issue them instead. While going to the library there’s just one intention or action you’ve got in mind i.e. **BORROW**(issue/borrow a book).

Here’s where things will get interesting, in the above scenario, there are three main tasks that are happening:

1. A **Library**which has all the books stored for not only you, but many others.
2. The **Customer**(you) wants to access the books in library without much hassle
3. You relied on someone else (a **Librarian**), to smoothly and efficiently let you borrow some books.

You are being the customer of the Library, looking to borrow some books. You won't go directly to the Library's book racks and take a book on your own. Rather you need to rely upon the library system that includes a librarian to handover your required books to you.

# What is Redux?

Redux is a tool to manage the data state and UI state of any React application in a much easier manner. Redux also makes the application more consistent. Redux gives us a way to define a global state for our application and maintain it with user's interactions.

**How is the analogy related to Redux?**

The previous scenario is very common, and is faced by almost all of us.

Your **issued books** in the library signifies your '**state**' with respect to the Library. Library as institution manages everyone else's books along with managing your books specifically. For which, the Library has created different accounts (or library cards) to preserve the 'state' of each of its customers.

The *library company* can be thought of as **Redux** as a whole. This scenario has three main actors involved:

1. **Book racks**: This is the only single source of books, which is being used for everyone's transactions (borrow/returning/extension etc).
2. **You**: You are the one who want to *borrow* (or *return*) the books. The library has your existing state with it, and you are relying on the library system to get your book out of the pool of books sitting inside the library.
3. **A Librarian**: They are the agents for the actual management of money. A librarianwill validate if you have the authority and the necessary availability of books for your transaction.

A library transaction results in the manipulation of books or the '**state**' in the Library. Hence, the librarianis the person who is responsible for changing from one 'state' to another.

Likewise, **Redux** is similar to the library company. Redux also has three main parts:

* **Store** - Store is the place where all the possible states of all the components of your application are stored. **Store** in Reduxcan be understood as the **book racks of a library**where the books for all the customers are kept.
* **Actions**- Action is the one which initiates a request to change the 'state'. In our example, **Action** is analogous to**you**. As Action initiates state change, similarly, you are the one who initiates any book borrow or book return from your account.
* **Reducers** - These are the agents that change the state of app, in a smooth and predictable manner. **Reducers**are analogous to the **librarian** who are responsible for any borrow or return of books happening in the library.3.

Hence, the main actors in **Redux** for any state management are **Store, Reducers**and **Actions**.

**Three principles of Redux**

Redux has three fundamental principles:

1. **Single source of truth**
2. **State is read-only**
3. **Changes are made with pure functions**

Lets understand these three principles.

**1. Single source of truth**

The state of complete application is stored in a **single state object** tree within a store. Having single store in a application makes it easy to create, debug and test. It also means components should read data from this single source and not keep their own version of the same state separately. The state of whole application is centralized and stores in a single state object. In Redux, store is the single source of truth.

**2. State is read-only**

State is **immutable**. The only way to change a state is by dispatching an action. Since whole application is stored in single state tree, having immutable state make sure that no views or network callbacks wrote directly to state. Instead, they dispatch an action, which have type property which indicates the  type of action being performed.

**3. Changes are made with pure functions**

**Pure functions** are those functions which do not change input value and always provide same output for same input. In Redux, reducers are pure functions which takes previous state and an action as parameter and return new state.Since reducers are pure functions, it will never changes previous state, it will always return new state object.

**Immutable objects**

Immutable means it cannot be changed. To change immutable objects, we need to create a copy of it and modify the copy.

In Redux state is an immutable object. State object cannot be changed directly in Redux. Triggering actions are the only way to modify the state. Every time the state is modified a new object should be returned which represents the modifications that are made to state object.

# Handling immutable objects

There are many ways to handle immutable objects. Object.assign and spread operator for arrays are the most popular approach.

**Object.assign()**

Consider state object a shown below

1. state={status: 'logged out', value: 'guest',email:'',tel:''}

When Login action is triggered we want to change only the status and value properties. We can use Object.assign to create a copy and modify only few properties as shown below.

1. Object.assign({}, state, {
2. status: 'logged in',
3. value: action.value
4. })

The first parameter to Object.assign is the target which is a new empty object, then we are mixing the new object together with our existing state and also changing the status and value properties. So the result is effectively a clone of our state object but with the state modified after login.

**Spread(...) operator**

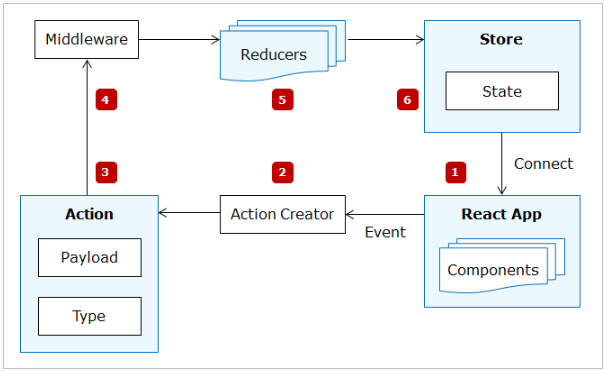
The spread operator is represented using 3 dots (...)

It copies own enumerable properties from a provided object onto a new object.

1. var obj1 = { foo: 'bar', x: 42 };
2. var obj2 = { foo: 'baz', y: 13 };
3. var clonedObj = { ...obj1 };
4. *// Object { foo: "bar", x: 42 }*
5. var mergedObj = { ...obj1, ...obj2 };
6. *// Object { foo: "baz", x: 42, y: 13 }*

Next, let us look at the overall Redux data flow.

**Redux Data Flow**



**React Components:** React components represents the UI rendered on the browser. Components would dispatch actions for events triggered within them and whenever state changes the component renders the current state by connecting to Redux store.

1. **Action Creators:** Action creators are functions which wraps the actual action object.
2. **Action:** Action is a plain JavaScript object. Action is a command to change the state when an event is triggered.
3. **Middleware:** Middleware is a mediator between the action and reducer. Its purpose is to intercept the actions before it reaches the reducer.
4. **Reducer:** Reducer changes the state of the application based on the action triggered. Reducer are functions which accepts action and current state and modify the current state by creating a copy of it based on the action.
5. **Store:** Store is responsible for managing the entire state of the application. State management is centralized in Redux. Action will be dispatched to store using dispatch method of store.

# Redux Data Flow explained with Login as an example

Let us observe the Redux data flow by considering Login as an example. Login form is represented by Login component.

1. When user clicks on the login button an action is dispatched.
2. Action reaches action creator which returns an action. Action object contains action type and payload information (if required)
3. Action reaches Reducer which modifies the state
4. If any operation needs to be performed before modifying state then action reaches middleware first and then reaches reducer
5. Reducer updates the modified state to the store
6. React components gets the updated state from the store

**NOTE:**We have not covered **middlewares** in this course.

Redux Store

Considering the given analogy of a visit to Library, the book racks/shelves is to the bank what the Redux Store is to Redux. Just like the book shelves in library keep the book safe and intact in the library, the state of your application is kept safe by something called a store. So, the ***store****keeps our 'books' or****state****intact*.

We learned that the entire state of the application is present in a single state object tree in Redux called as Store. Apart from this, store brings actions and reducers together to update any state.

Redux Store is not a class, rather it is an object consisting of a few methods as given below:

* **getState()**
  + It does not take any argument and returns the current state tree of your application.
  + It is equal to the last value returned by the store's reducers
* **dispatch(action)**
  + It takes an argument **i.e. action.**Action is a plain object that describes the changes to made in the application. The action object must have a type field to indicate the type of action being performed. You will see this in detail in redux actions module.
  + It dispatches an action for stimulating state changes
  + It invokes the store's reducing function with the values ofcurrent state (obtained by invoking **getState()** method) and the action.
  + The return value of dispatch(action) is considered as the next state and the change listeners are notified immediately
* **subscribe(listener)**
  + It is used to add a change listener.
  + **subscribe()** method will be called any time an action is dispatched.
  + It takes an argument **i.e. listener**which is a callback function that gets invoked every time an action is dispatched.
  + It returns a function that unsubscribes the change listener.
* **replaceReducer(nextReducer)**
  + It is called to replace the reducer function currently used by the store to calculate the state by the nextReducer .
  + It is an advance API that might be needed in case your app implements code splitting.

In brief, the purpose of redux store is to do following things:

* hold complete application state
* provide access to state by using *store.getState()*
* allow state to be updated by using *dispatch(action)*
* register and de-register listeners by using *subscribe(listener)*

# creating a redux store: Demo

Store is created by using the **createStore()** method. The **createStore()** method needs to be imported from Redux, and it takes a single reducer as parameter. We generally pass **root reducer** as parameter to the **createStore()** method.

1. *// Below code can be present in store.js file*
2. import { createStore } from 'redux';
3. *//initial state*
4. export const initialBookState = {
5. accountBooks: 666666
6. }
7. *//reducer config*
8. export const libraryReducers; *//all the reducers combined*
10. *//store config*
11. export function configureStore(){
12. const store = createStore(libraryReducers, initialBookState );
13. return store;
14. }
15. export const bookStore = configureStore();

In the above code:

* We have given an initial state for the app, i.e. initialBookState, this stores the initial number of books
* We have created a Redux store by using the createStore() method
* We have passed libraryReducers and initialBookState as parameters to the createStore() method

We have configured the libraryReducers, before creating a bookStore according to our analogy, they represent the Librarians, who are an agent of books or state management.

Next, let us understand redux Actions.

**Redux Actions**

As you know, In react, **state is read-only,**so the only way to change the state is to emit an action, an object describing what happened.

**Action**is a JavaScript object which describes an event.

When you walk to the library, you go there with a clear action in mind. In this example, you want to borrow some books. If we chose to represent that process in a simple Redux application, your action to the library may be represented by an object.

One that looks like this:

1. { type: 'BORROW', qty: 6 }
2. { type: 'RETURN', qty: 1 }

In the context of a Redux application, this object is called an action! It always has a type field that describes the action you want to perform. In this case, it is *BORROW*.

Whenever you need to change/update the state of your Redux application, you need to dispatch an action.

**Creating an Action**

**Action object** must contain a **type** property which describes the type of the action triggered and optionally can have ***payload*** information.

The payload information can be:

* number
* a boolean value
* a complex object
* or any other value that is serializable to JSON.
* The payload information should **NOT** be a functions or Promises.

# Action creators

Action creators are the functions that return the action object. We can easily perform an action by invoking action creator.

For example,

1. *//can be present in action.js file*
2. *//returns an object which contains a type attribute and some payload*
3. export const borrowAction = (quantity) => ({
4. type: 'BORROW',
5. qty: quantity
6. });
7. export const returnAction = (quantity) => ({
8. type: 'RETURN',
9. qty: quantity
10. });

Redux Reducers

Let us look at the next part of redux **i.e. Reducers**

Considering our visit to library analogy, The **Librarian**to the library is what the **Reducer** is to redux.

Remember that, in the library analogy, we couldn’t just go straight into the book shelves to take a book. We had to get it through the Librarian.

Well, we had an action in mind, but we had to convey that action to someone — the Librarian— who in turn communicated (in whatever way they did) with the book shelves that holds all the books.

Like we made our action known to the Librarian, we have to do the same in your Redux application. If we want to **update**the state of our application, you convey your **action**to the **reducer** — our own Librarian.

This process is mostly called **dispatching**an action. It means sending off the action to the reducers. The reducer knows what to do. For a library:

* it will take your action to BORROW and
* it will ensure that you get the required books borrowed.

In Redux terms, the books in the library is our state. So, your *reducer knows what to do*, and it always *returns your new state*.

Hence, To specify how the state tree is transformed by actions, we write pure reducers.

Let us see how to write a reducer.

# Redux Reducers: Demo

Since**state is immutable**, we can not change it directly. Actions only describe what happened, but they tell nothing about how state will change.

Reducers are responsible for state change and they are pure functions, which means they never modify state. They always return new state.

Reducers handles application state changes based on actions dispatched. Reducers are responsible for copying the current state and modifying the copy and returning the new state.

For example,

1. *// can be present in reducers.js file*
2. ​​​​import { createStore, combineReducers } from 'redux';
4. const initialBookState = { accountBooks: 666666 };
6. *//creating a single reducer which can change the accountBooks present in initialBookState*
7. export const bookReducer = (initialState = initialBookState.accountBooks, action) => {
9. switch(action.type){
10. case 'BORROW':
11. let newBooksBal = initialState - action.qty;
12. return newBooksBal;
13. case 'RETURN':
14. let newBooksBal = initialState + (-(-action.qty));
15. return newBooksBal;
16. default:
17. return initialState;
18. }
19. };
20. *//combining all the reducers to a single libraryReducers*
21. export const libraryReducers = combineReducers({accountBooks: bookReducer });

A typical Reducer would contain ***switch***statements where it would check for the actions and whichever action matches the state is changed according to the action and a new copy of state is returned.

Since reducers are pure functions, you should take care loading initial state and not doing something which makes reducer impure functions:

# Initializing state

Reducer accepts the current state and changes the state based on the action and updates the state to the store. We should initialize initial state to reducers, if we do not initialize initial value for state in reducers, it will be **undefined**by default.

Initial state of the application should be passed to the Reducer as a best practice. Initial state in our library analogy has accountBooks pre-initialized.

# Forbidden in Reducers

* Mutating the arguments passed
* Performing side effects like API call, HTTP call etc.
* Calling other non pure functions (Ex: date.now(), math.random())

Redux Components

As react application grow bigger, we have multiple components in application. Writing UI and business logic like updating state in same component makes component cumbersome and it become harder to maintain. It is advised that we should create separate component for displaying UI and handling business logic.

It is recommended that we separate component into two part, **presentational** components and **container**components so that we can have view related components organised at one place and components dealing with data, state and logic organised at one place.

Let us take a look at the two redux components.

**Presentational vs Container components**

# Presentational components

Presentational components are mainly used to display content on page. So, they consist of markup and styles along with their props. They generally don't deal with data fetching and state updates. Also, they do not have their own state.

# Container components

Container components are responsible for connecting to the Redux store and fetching the updated state from Redux **store**. Once container components get the state from Redux store they pass the updated state to presentational components as props.

Container components can be written by developer by subscribing to the store using **store.subscribe()** and the current state can be fetched from the store using**store.getState()** method.

But as already discussed React Redux has already implemented the necessary performance optimizations. Hence we generate container components using the methods provided by react-redux library, rather than writing the container components by hand.

**Creating container components**

# ****Container Components: Demo****

Container components are used to**fetch the updated state** from redux **store**. Once container components get the state from Redux store, they pass the updated state to presentational components as props. These props can have their state, data and call Redux actions.

container component can be generated by **connect()** function of react-redux library. To use **connect()** we need to define two functions:

* **mapStateToProps**- This function helps in transforming the current state present in the redux store as props and then the same is passed to it's corresponding presentational component.
* **mapDispatchToProps**- This function is used by container components to dispatch actions.

### React Redux Connect — Anatomy of connect()

The **connect()** function provided by **react-redux** can take up to four arguments, all of which are optional. Calling the **connect()** function returns a higher-order component which can be used to wrap any React component.

Since a higher-order component is returned by **connect()**,(A **higher**-**order** component is a **function** that takes a component and returns a new component ) it has to be invoked again with the base React component in order to convert it to a container component:

const ContainerComponent = connect()(BaseComponent);

Here is the signature of the **connect()** function:

connect([mapStateToProps], [mapDispatchToProps], [mergeProps], [options])

#### mapStateToProps(state, [ownProps]) => stateProps

This argument is a **function** that returns either a **plain object** or **another function**. Passing this argument subscribes the container component to the Redux store updates, which means that the **mapStateToProps** function will be invoked each time the store updates. If you are not interested in store updates, leave it as **undefined** or **null**.

**mapStateToProps** is declared with **two parameters**, the second one being optional. The first parameter is the current state of the Redux store. The second parameter, if passed, is an object of the props passed to the component

If a plain object is returned from **mapStateToProps**, the returned **stateProps** object is merged into the component’s props.

#### mapDispatchToProps(dispatch, [ownProps]) => dispatchProps

This argument can either be an **object** or a **function** that returns either a **plain object** or **another function**.

To better illustrate how **mapDispatchToProps** works, you will need to have some action creators.

For example, say you have the following action creators:

|  |  |
| --- | --- |
|  | export const writeComment = (comment) => ({ |
|  | comment, |
|  | type: 'WRITE\_COMMENT' |
|  | }); |
|  |  |
|  | export const updateComment = (id, comment) => ({ |
|  | id, |
|  | comment, |
|  | type: 'UPDATE\_COMMENT' |
|  | }); |
|  |  |
|  | export const deleteComment = (id) => ({ |
|  | id, |
|  | type: 'DELETE\_COMMENT' |
|  | }); |

**1. Default Implementation**

If you don’t supply your own **mapDispatchToProps** object or function, the default implementation will be used, which simply injects the store’s **dispatch** method as a prop to the component.

You can use the **dispatch** prop in your component as follows:

|  |  |
| --- | --- |
|  | import React from 'react'; |
|  | import { connect } from 'react-redux'; |
|  | import { updateComment, deleteComment } from './actions'; |
|  |  |
|  | function Comment(props) { |
|  | const { id, content } = props.comment; |
|  |  |
|  | // Invoking the actions via props.dispatch() |
|  | const editComment = () => props.dispatch(updateComment(id, content)); |
|  | const removeComment = () => props.dispatch(deleteComment(id)); |
|  |  |
|  | return ( |
|  | <div> |
|  | <p>{ content }</p> |
|  | <button type="button" onClick={editComment}>Edit Comment</button> |
|  | <button type="button" onClick={removeComment}>Remove Comment</button> |
|  | </div> |
|  | ) |
|  | } |
|  |  |
|  | export default connect()(Comment); |

**2. Passing an object**

If an **object** is passed for this argument, each function in the object will be taken to be a Redux action creator and will be wrapped into a call to the store’s **dispatch** method so that it can be called directly. The resulting **dispatchProps** object of action creators will be merged into the component’s props.

Let us have a look at the container for app component **i.e.** **AppContainer**.

1. *//can be present in AppContainer.js file*
2. import { borrowAction , returnAction } from './redux';
3. import AppRedux from './AppRedux'; *//presentational component that contains the view*
4. import { connect } from 'react-redux';
5. const mapStatetoProps = state => ({
6. pBookBalance : state.accountBooks
7. });
8. const mapDispatchToProps = {
9. borrowBooks : borrowAction ,
10. returnBooks : returnAction
11. }
12. export default connect(mapStatetoProps,mapDispatchToProps)(AppRedux);

**Creating the view or presentational components**

# ****Presentational Components: Demo****

Presentational components is mainly used to display content on the page, so they consist of markup and styles along with their props. They generally don't deal with data fetching and state updates. They generally do not have their own state.

Here we are going to create the view component:

1. import React, { Component } from 'react';
2. import 'bootstrap/dist/css/bootstrap.min.css';
3. *// a presentational component which has no state of its own*
4. export default class AppRedux extends Component {
5. userInputQty = 0;
6. render() {
7. return (<div>
8. <div className={'col-5 offset-1 card'}>
9. <br/>
10. <table>
11. <tbody>
12. <tr>
13. <th><h3>Books in library are: </h3></th>
14. <td><h3>{+this.props.pBookBalance }</h3></td>
15. </tr>
16. <tr>
17. <th>Enter the amount for transaction: </th>
18. <td><input onChange={(e)=>{ this.userInputQty = e.target.value;}} type='number' className={'form-control'}/></td>
19. </tr>
20. <tr>
21. <td>
22. <button onClick={()=>{this.props.debit(this.userInputQty)}} className={'btn btn-primary'}>
23. Borrow </button>
24. </td>
25. <td>
26. <button onClick={()=>{this.props.credit(this.userInputQty)}} className={'btn btn-success'}>
27. Debit </button>
28. </td>
29. </tr>
30. </tbody>
31. </table>
32. <br/>
33. </div>
34. </div>);
35. }
36. }

Next we will a <Provider>, which will provide data from the store into the application.

**Providers**

In order to subscribe to store and get state from the store, container components need access to the store. To pass store to React, react-redux library provides an in-built **Provider** component which provides access to the store for all the container components in an application.

When rendering the root component to the DOM, the root component should be wrapped within the *<Provider>* component as shown below.

1. *//other imports if any*
2. import React from 'react';
3. import ReactDOM from 'react-dom';
4. import AppRedux from './AppContainer';
5. import { Provider } from 'react-redux';
6. import { bankStore } from './redux';
7. ReactDOM.render(
8. <Provider store={ bankStore }>
9. <AppRedux />
10. </Provider>,
11. document.getElementById("root")
12. );

By wrapping our root component with Provider component and passing store to provider component, we can access store anywhere in out react application.

**Advanced Redux**

Combining Multiple Reducers

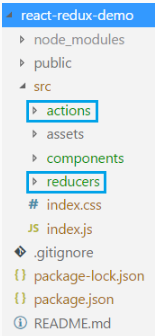
**A real Redux application - folder structure organisation**

Now that we have looked at basic elements of Redux and seen a small app using Redux, let us look at the folder structure that we are going to use moving forward.

Right now, all of our app's Redux related code is located in a single file. This isn't a very realistic way to build a Redux app.

To fix this, we're going to use Create React App to scaffold a React app for us. Then we can add a few extra folders inside the src. i.e -  *src/****reducers***and *src/****actions***

Observe the app *folder structure* as shown below:



We have grouped our resources in the application based on capability or type: any action will be found in the Actions folder, any reducer will be found in Reducers, and so on.

Using this directory structure, if we wanted to import all actions into a component, we can get them all in a single import!

**Understanding combine reducers**

Let's compare the ***object****passed* to *combineReducers(),*with the original state object.

We passed the following object:

1. {
2. users: users,
3. settings: settings,
4. products: products,
5. }

Using the latest object syntax, if the key-name and the value variable names are the same, the above object can be written as:

1. {
2. users, *// key is users, value is the users Reducer*
3. settings, *// key is settings, value is the settings Reducer*
4. products, *// key is products, value is the products Reducer*
5. }

# Final comparison:

| **State Tree** | **Combined Reducers** |
| --- | --- |
| 1. { 2. users: { ... }, 3. settings: { ... }, 4. products: { 5. btyxlj666: { 6. id: 'btyxlj666', 7. text: 'HotWheels car', 8. seller: { 9. name: 'HotWheels', 10. id: 'htwh666', 11. img: 'tw2t.com/t2m.png' 12. } 13. } 14. } 15. } | 1. const reducer = combineReducers({ 2. users, 3. settings, 4. products, 5. }); |

combineReducers() is responsible for:

* invoking all the other reducers
* passing these child reducers, the portion of their state, that they care about.

**Note**: In the object passed to combineReducers( { } ), the key name should be same as child reducer name. Else, pass the object as key-value pairs.

We're making one root reducer, by composing a bunch of other reducers together. We are combining the **users**, **settings**, and the **products** Reducers.

Next let's see the benefits of splitting up the state into slices. 