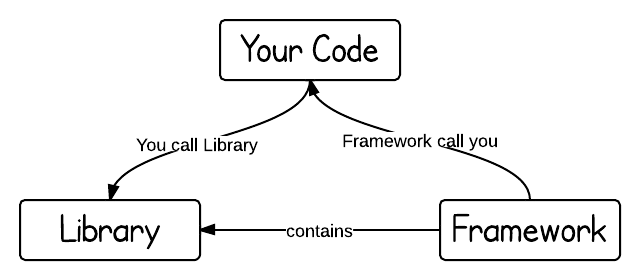
**React**

**Library vs Framework**

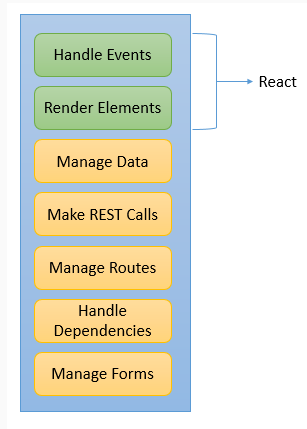
<https://www.programcreek.com/2011/09/what-is-the-difference-between-a-java-library-and-a-framework/>



<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?

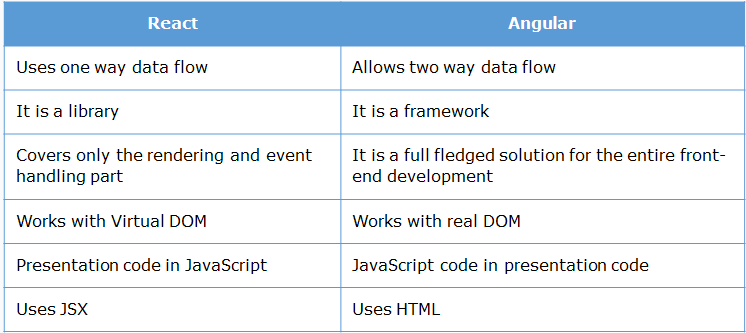
A virtual DOM is a in-memory representation of the actual DOM. The way virtual DOM works is:

* 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:



# 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");**

# 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

**<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}.

**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.**

**Component - constructor, methods and variables**

* 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.

# What is state?

Data in the Virtual DOM is state and only when this Virtual DOM is updated, React will update the actual DOM.

 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*".

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 });
4. this.setState({counter : this.state.counter + 1 });
5. console.log("Button Clicked," this.state.counter);
6. };

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:

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

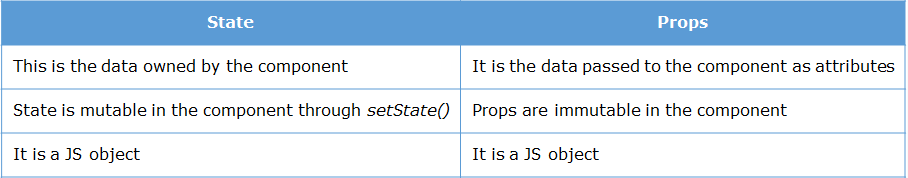
# Props

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 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:

* **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

**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 method *after mounting*of components.

The methods which gets called in this phase are  *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.

Graphical user interface, timeline

Description automatically generated

<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)

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)

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>

Basics of routing

npm install react-router-dom –save

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:

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.

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;

# <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.

**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.

1. <Route exact path="/" render={() => (<Redirect to="/home">)} />

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.

# <Redirect />

We will see the following props associated with Redirect:

1. **to**: string :- The URL to redirect to. Any valid URL path.

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

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}/>

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.

**Switch - for rendering components**

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.

# <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" \l "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.

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.*

# 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.

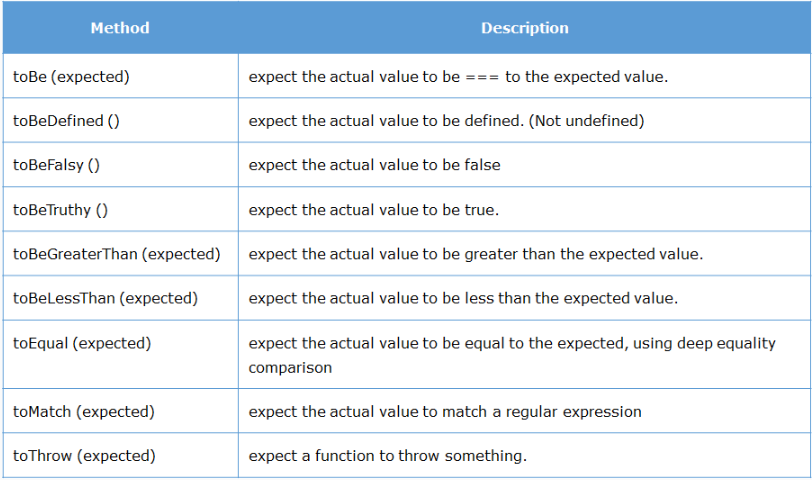
Jest -> Test Runner

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.

**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.

**Jest Matchers**



Enzyme -> Test Environment

# 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

**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 />);

**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() and setProps:**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**

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. });

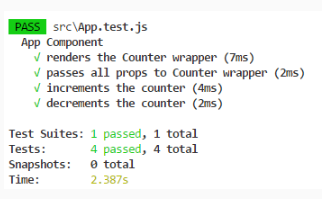
In this file, we have one test suite and four test cases which we have created in previous step.

**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.

# toMatchSnapshot()

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

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

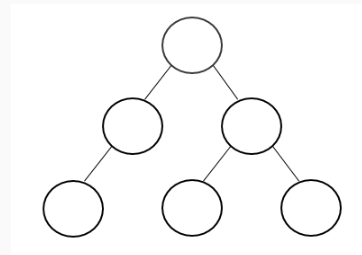
**Redux**is a component based library developed by **Dan Abramov** and **Andrew Clark**in 2015 with an inspiration from **Facebook's Flux**and **Elm**(functional programming language) for easing the development of react applications. It became popular in the developers' community very soon due to its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

# 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 alter the global state of the application and takes at least one argument 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()**

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**

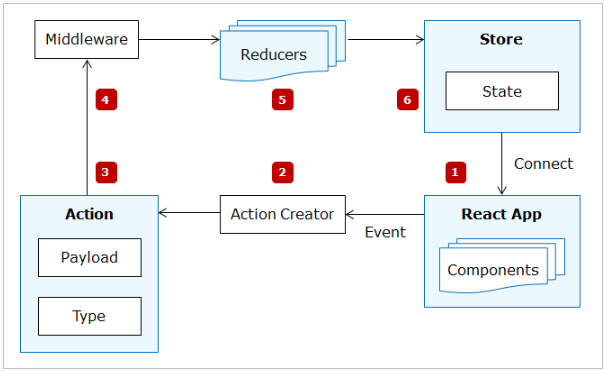
The spread operator is represented using 3 dots (...)

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

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.

One that looks like this:

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

**Creating an Action**

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

# 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

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

There are two main ways to initialize state for your application. The createStore method can accept an optional preloadedState value as its second argument. Reducers can also specify an initial value by looking for an incoming state argument that is undefined, and returning the value they'd like to use as a default. This can either be done with an explicit check inside the reducer, or by using the ES6 default argument value syntax: function myReducer(state = someDefaultValue, action).

It's not always immediately clear how these two approaches interact. Fortunately, the process does follow some predictable rules. Here's how the pieces fit together.

Summary

Without combineReducers() or similar manual code, preloadedState always wins over state = ... in the reducer because the state passed to the reducer is preloadedState and is not undefined, so the ES6 argument syntax doesn't apply.

With combineReducers() the behavior is more nuanced. Those reducers whose state is specified in preloadedState will receive that state. Other reducers will receive undefined and because of that will fall back to the state = ... default argument they specify.

In general, preloadedState wins over the state specified by the reducer. This lets reducers specify initial data that makes sense to them as default arguments, but also allows loading existing data (fully or partially) when you're hydrating the store from some persistent storage or the server.

Note: Reducers whose initial state is populated using preloadedState will still need to provide a default value to handle when passed a state of undefined. All reducers are passed undefined on initialization, so they should be written such that when given undefined, some value should be returned. This can be any non-undefined value; there's no need to duplicate the section of preloadedState here as the default.

**In Depth**[**#**](https://redux.js.org/usage/structuring-reducers/initializing-state#in-depth)

**Single Simple Reducer**[**#**](https://redux.js.org/usage/structuring-reducers/initializing-state#single-simple-reducer)

First let's consider a case where you have a single reducer. Say you don't use combineReducers().

Then your reducer might look like this:

1. function counter(state = 0, action) {
2. switch (action.type) {
3. case 'INCREMENT': return state + 1
4. case 'DECREMENT': return state - 1 default: return state
5. }
6. }

Now let's say you create a store with it.

1. import { createStore } from 'redux'
2. const store = createStore(counter)
3. console.log(store.getState()) // 0

The initial state is zero. Why? Because the second argument to createStore was undefined. This is the state passed to your reducer the first time. When Redux initializes it dispatches a "dummy" action to fill the state. So your counter reducer was called with state equal to undefined. **This is exactly the case that "activates" the default argument.** Therefore, state is now 0 as per the default state value (state = 0). This state (0) will be returned.

Let's consider a different scenario:

1. import { createStore } from 'redux'const
2. store = createStore(counter, 42)
3. console.log(store.getState()) // 42

Why is it 42, and not 0, this time? Because createStore was called with 42 as the second argument. This argument becomes the state passed to your reducer along with the dummy action. **This time, state is not undefined (it's 42!), so ES6 default argument syntax has no effect.** The state is 42, and 42 is returned from the reducer.

# 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.

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**.

**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.

**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**

**Understanding combine reducers**

# 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.

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

# Why Do We Write super(props)?

**Why do we call super? Can we *not* call it? If we have to call it, what happens if we don’t pass props? Are there any other arguments?** Let’s find out.

In JavaScript, super refers to the parent class constructor. (In our example, it points to the React.Component implementation.)

Importantly, you can’t use this in a constructor until *after* you’ve called the parent constructor. JavaScript won’t let you:

class Checkbox extends React.Component {

constructor(props) {

// 🔴 Can’t use `this` yet

super(props);

// ✅ Now it’s okay though

this.state = { isOn: true };

}

// ...

}

There’s a good reason for why JavaScript enforces that parent constructor runs before you touch this. Consider a class hierarchy:

class Person {

constructor(name) {

this.name = name;

}

}

class PolitePerson extends Person {

constructor(name) {

this.greetColleagues(); // 🔴 This is disallowed, read below why

super(name);

}

greetColleagues() {

alert('Good morning folks!');

}

}

Imagine using this before super call *was* allowed. A month later, we might change greetColleagues to include the person’s name in the message:

greetColleagues() {

alert('Good morning folks!');

alert('My name is ' + this.name + ', nice to meet you!');

}

But we forgot that this.greetColleagues() is called before the super() call had a chance to set up this.name. So this.name isn’t even defined yet! As you can see, code like this can be very difficult to think about.

To avoid such pitfalls, **JavaScript enforces that if you want to use this in a constructor, you *have to* call super first.** Let the parent do its thing! And this limitation applies to React components defined as classes too:

This leaves us with another question: why pass props?

You might think that passing props down to super is necessary so that the base React.Component constructor can initialize this.props:

And that’s not far from truth — indeed, that’s [what it does](https://github.com/facebook/react/blob/1d25aa5787d4e19704c049c3cfa985d3b5190e0d/packages/react/src/ReactBaseClasses.js#L22).

But somehow, even if you call super() without the props argument, you’ll still be able to access this.props in the render and other methods. (If you don’t believe me, try it yourself!)

How does that work? It turns out that **React also assigns props on the instance right after calling your constructor:**

// Inside React

const instance = new YourComponent(props);

instance.props = props;

So even if you forget to pass props to super(), React would still set them right afterwards. There is a reason for that.

When React added support for classes, it didn’t just add support for ES6 classes alone. The goal was to support as wide range of class abstractions as possible. It was [not clear](https://reactjs.org/blog/2015/01/27/react-v0.13.0-beta-1.html#other-languages) how relatively successful would ClojureScript, CoffeeScript, ES6, Fable, Scala.js, TypeScript, or other solutions be for defining components. So React was intentionally unopinionated about whether calling super() is required — even though ES6 classes are.

So does this mean you can just write super() instead of super(props)?

**Probably not because it’s still confusing.** Sure, React would later assign this.props *after* your constructor has run. But this.props would still be undefined *between* the super call and the end of your constructor:

// Inside React

class Component {

constructor(props) {

this.props = props;

// ...

}

}

// Inside your code

class Button extends React.Component {

constructor(props) {

super(); // 😬 We forgot to pass props

console.log(props); // ✅ {}

console.log(this.props); // 😬 undefined }

// ...

}

It can be even more challenging to debug if this happens in some method that’s called *from* the constructor. **And that’s why I recommend always passing down super(props), even though it isn’t strictly necessary:**

class Button extends React.Component {

constructor(props) {

super(props); // ✅ We passed props

console.log(props); // ✅ {}

console.log(this.props); // ✅ {}

}

// ...

}

This ensures this.props is set even before the constructor exits.

# You Probably Don't Need Derived State

For a long time, the lifecycle componentWillReceiveProps was the only way to update state in response to a change in props without an additional render. In version 16.3, [we introduced a replacement lifecycle, getDerivedStateFromProps](https://reactjs.org/blog/2018/03/29/react-v-16-3.html#component-lifecycle-changes) to solve the same use cases in a safer way.

## When to Use Derived State

getDerivedStateFromProps exists for only one purpose. It enables a component to update its internal state as the result of **changes in props**. Our previous blog post provided some examples, like [recording the current scroll direction based on a changing offset prop](https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html#updating-state-based-on-props) or [loading external data specified by a source prop](https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html#fetching-external-data-when-props-change).

We did not provide many examples, because as a general rule, **derived state should be used sparingly**. All problems with derived state that we have seen can be ultimately reduced to either (1) unconditionally updating state from props or (2) updating state whenever props and state don’t match. (We’ll go over both in more detail below.)

* If you’re using derived state to memoize some computation based only on the current props, you don’t need derived state. See [What about memoization?](https://reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html#what-about-memoization) below.
* If you’re updating derived state unconditionally or updating it whenever props and state don’t match, your component likely resets its state too frequently. Read on for more details.

## Common Bugs When Using Derived State

The terms [“controlled”](https://reactjs.org/docs/forms.html#controlled-components) and [“uncontrolled”](https://reactjs.org/docs/uncontrolled-components.html) usually refer to form inputs, but they can also describe where any component’s data lives. Data passed in as props can be thought of as **controlled** (because the parent component controls that data). Data that exists only in internal state can be thought of as **uncontrolled** (because the parent can’t directly change it).

The most common mistake with derived state is mixing these two; when a derived state value is also updated by setState calls, there isn’t a single source of truth for the data.

The [external data loading example](https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html#fetching-external-data-when-props-change) mentioned above may sound similar, but it’s different in a few important ways. In the loading example, there is a clear source of truth for both the “source” prop and the “loading” state. When the source prop changes, the loading state should **always** be overridden. Conversely, the state is overridden only when the prop **changes** and is otherwise managed by the component.

Problems arise when any of these constraints are changed. This typically comes in two forms. Let’s take a look at both.

### Anti-pattern: Unconditionally copying props to state

A common misconception is that getDerivedStateFromProps and componentWillReceiveProps are only called when props “change”. These lifecycles are called any time a parent component rerenders, regardless of whether the props are “different” from before. Because of this, it has always been unsafe to unconditionally override state using either of these lifecycles. **Doing so will cause state updates to be lost.**

Let’s consider an example to demonstrate the problem. Here is an EmailInput component that “mirrors” an email prop in state:

class EmailInput extends Component {

state = { email: this.props.email };

render() {

return <input onChange={this.handleChange} value={this.state.email} />;

}

handleChange = event => {

this.setState({ email: event.target.value });

};

componentWillReceiveProps(nextProps) {

// This will erase any local state updates!

// Do not do this.

this.setState({ email: nextProps.email });

}

}

Let’s consider an example to demonstrate the problem. Here is an EmailInput component that “mirrors” an email prop in state:

class EmailInput extends Component {

state = { email: this.props.email };

render() {

return <input onChange={this.handleChange} value={this.state.email} />;

}

handleChange = event => {

this.setState({ email: event.target.value });

};

componentWillReceiveProps(nextProps) {

// This will erase any local state updates!

// Do not do this.

this.setState({ email: nextProps.email });

}

}

At first, this component might look okay. State is initialized to the value specified by props and updated when we type into the <input>. But if our component’s parent rerenders, anything we’ve typed into the <input> will be lost! ([See this demo for an example.](https://codesandbox.io/s/m3w9zn1z8x)) This holds true even if we were to compare nextProps.email !== this.state.email before resetting.

In this simple example, adding shouldComponentUpdate to rerender only when the email prop has changed could fix this.

However in practice, components usually accept multiple props; another prop changing would still cause a rerender and improper reset. Function and object props are also often created inline, making it hard to implement a shouldComponentUpdate that reliably returns true only when a material change has happened. [Here is a demo that shows that happening.](https://codesandbox.io/s/jl0w6r9w59) As a result, shouldComponentUpdate is best used as a performance optimization, not to ensure correctness of derived state.

### Anti-pattern: Erasing state when props change

Continuing the example above, we could avoid accidentally erasing state by only updating it when props.email changes:

class EmailInput extends Component {

state = {

email: this.props.email

};

componentWillReceiveProps(nextProps) {

// Any time props.email changes, update state.

if (nextProps.email !== this.props.email) {

this.setState({

email: nextProps.email

});

}

}

// ...

}

**Note**

Even though the example above shows componentWillReceiveProps, the same anti-pattern applies to getDerivedStateFromProps.

We’ve just made a big improvement. Now our component will erase what we’ve typed only when the props actually change.

There is still a subtle problem. Imagine a password manager app using the above input component. When navigating between details for two accounts with the same email, the input would fail to reset. This is because the prop value passed to the component would be the same for both accounts! This would be a surprise to the user, as an unsaved change to one account would appear to affect other accounts that happened to share the same email.

This design is fundamentally flawed, but it’s also an easy mistake to make.

Fortunately there are two alternatives that work better. The key to both is that **for any piece of data, you need to pick a single component that owns it as the source of truth, and avoid duplicating it in other components.** Let’s take a look at each of the alternatives.

## Preferred Solutions

### Recommendation: Fully controlled component

One way to avoid the problems mentioned above is to remove state from our component entirely. If the email address only exists as a prop, then we don’t have to worry about conflicts with state. We could even convert EmailInput to a lighter-weight function component:

function EmailInput(props) {

return <input onChange={props.onChange} value={props.email} />;

}

### Recommendation: Fully uncontrolled component with a key

Another alternative would be for our component to fully own the “draft” email state. In that case, our component could still accept a prop for the initial value, but it would ignore subsequent changes to that prop:

class EmailInput extends Component {

state = { email: this.props.defaultEmail };

handleChange = event => {

this.setState({ email: event.target.value });

};

render() {

return <input onChange={this.handleChange} value={this.state.email} />;

}

}

In order to reset the value when moving to a different item (as in our password manager scenario), we can use the special React attribute called key. When a key changes, React will [create a new component instance rather than update the current one](https://reactjs.org/docs/reconciliation.html#keys). Keys are usually used for dynamic lists but are also useful here. In our case, we could use the user ID to recreate the email input any time a new user is selected:

<EmailInput

defaultEmail={this.props.user.email}

key={this.props.user.id}

/>

Each time the ID changes, the EmailInput will be recreated and its state will be reset to the latest defaultEmail value. ([Click here to see a demo of this pattern.](https://codesandbox.io/s/6v1znlxyxn)) With this approach, you don’t have to add key to every input. It might make more sense to put a key on the whole form instead. Every time the key changes, all components within the form will be recreated with a freshly initialized state.

#### Alternative 1: Reset uncontrolled component with an ID prop

If key doesn’t work for some reason (perhaps the component is very expensive to initialize), a workable but cumbersome solution would be to watch for changes to “userID” in getDerivedStateFromProps:

class EmailInput extends Component {

state = {

email: this.props.defaultEmail,

prevPropsUserID: this.props.userID

};

static getDerivedStateFromProps(props, state) {

// Any time the current user changes,

// Reset any parts of state that are tied to that user.

// In this simple example, that's just the email.

if (props.userID !== state.prevPropsUserID) {

return {

prevPropsUserID: props.userID,

email: props.defaultEmail

};

}

return null;

}

// ...

}

## What about memoization?

We’ve also seen derived state used to ensure an expensive value used in render is recomputed only when the inputs change. This technique is known as [memoization](https://en.wikipedia.org/wiki/Memoization" \t "_blank).

Using derived state for memoization isn’t necessarily bad, but it’s usually not the best solution. There is inherent complexity in managing derived state, and this complexity increases with each additional property. For example, if we add a second derived field to our component state then our implementation would need to separately track changes to both.

Let’s look at an example of one component that takes one prop—a list of items—and renders the items that match a search query entered by the user. We could use derived state to store the filtered list:

class Example extends Component {

state = {

filterText: "",

};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// NOTE: this example is NOT the recommended approach.

// See the examples below for our recommendations instead.

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

static getDerivedStateFromProps(props, state) {

// Re-run the filter whenever the list array or filter text change.

// Note we need to store prevPropsList and prevFilterText to detect changes.

if (

props.list !== state.prevPropsList ||

state.prevFilterText !== state.filterText

) {

return {

prevPropsList: props.list,

prevFilterText: state.filterText,

filteredList: props.list.filter(item => item.text.includes(state.filterText))

};

}

return null;

}

handleChange = event => {

this.setState({ filterText: event.target.value });

};

render() {

return (

<Fragment>

<input onChange={this.handleChange} value={this.state.filterText} />

<ul>{this.state.filteredList.map(item => <li key={item.id}>{item.text}</li>)}</ul>

</Fragment>

);

}

}

This implementation avoids recalculating filteredList more often than necessary. But it is more complicated than it needs to be, because it has to separately track and detect changes in both props and state in order to properly update the filtered list. In this example, we could simplify things by using PureComponent and moving the filter operation into the render method:

// PureComponents only rerender if at least one state or prop value changes.

// Change is determined by doing a shallow comparison of state and prop keys.

class Example extends PureComponent {

// State only needs to hold the current filter text value:

state = {

filterText: ""

};

handleChange = event => {

this.setState({ filterText: event.target.value });

};

render() {

// The render method on this PureComponent is called only if

// props.list or state.filterText has changed.

const filteredList = this.props.list.filter(

item => item.text.includes(this.state.filterText)

)

return (

<Fragment>

<input onChange={this.handleChange} value={this.state.filterText} />

<ul>{filteredList.map(item => <li key={item.id}>{item.text}</li>)}</ul>

</Fragment>

);

}

}

The above approach is much cleaner and simpler than the derived state version. Occasionally, this won’t be good enough—filtering may be slow for large lists, and PureComponent won’t prevent rerenders if another prop were to change. To address both of these concerns, we could add a memoization helper to avoid unnecessarily re-filtering our list:

import memoize from "memoize-one";

class Example extends Component {

// State only needs to hold the current filter text value:

state = { filterText: "" };

// Re-run the filter whenever the list array or filter text changes:

filter = memoize(

(list, filterText) => list.filter(item => item.text.includes(filterText))

);

handleChange = event => {

this.setState({ filterText: event.target.value });

};

render() {

// Calculate the latest filtered list. If these arguments haven't changed

// since the last render, `memoize-one` will reuse the last return value.

const filteredList = this.filter(this.props.list, this.state.filterText);

return (

<Fragment>

<input onChange={this.handleChange} value={this.state.filterText} />

<ul>{filteredList.map(item => <li key={item.id}>{item.text}</li>)}</ul>

</Fragment>

);

}

}

This is much simpler and performs just as well as the derived state version!

When using memoization, remember a couple of constraints:

1. In most cases, you’ll want to **attach the memoized function to a component instance**. This prevents multiple instances of a component from resetting each other’s memoized keys.
2. Typically you’ll want to use a memoization helper with a **limited cache size** in order to prevent memory leaks over time. (In the example above, we used memoize-one because it only caches the most recent arguments and result.)
3. None of the implementations shown in this section will work if props.list is recreated each time the parent component renders. But in most cases, this setup is appropriate.