|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **Cues**  NoteGem Horizontal Line          what is reactjs  what does it do  who developed it  features      what's a virtual dom    what is reconciliation                    what is the anatomy of a react component?      what is jsx?      what are the two major pillars of react's design philosophy?        what is state?        what is an export?                            what strategy does react use when passing data?    what are props?          what is a stateful component?                        what is best practice when data is consumed in multiple components?            how are components rendered in a stateful component?            why pass via props method?        what is a side effect?        what are some examples of side effects?            what are the categories of side effects?      what is a pure component?                what is a react hook?          what are some examples of effects with no clean-up?                what is static code analysis?                                                                          what are important parts to focus on when testing?                                          what is an identity object proxy?                                                        what is shallow rendering?                                                      what is reactjs?                what does it implement and why is this good?              what are some features of react?                        what are some advantages of react?                what are some limitations of react?                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    what are some pros of JSX?                                                                                                                              how do we use custom attributes?                                    how to use javascript expressions?                                                  what can't you use inside JSX and what can you use in its place?                                                    what styling does react recommend and in what syntax?                                                          how to write comments in react?                                      what's the naming convention in react?                                                                                                                                                                                                                                                                                                                                                                  what is state?                                                                                                          what is props and what is the difference between it and state?      what to do when you need immutable data in your component?                                                                                how to set default props?                                                                                                                                                                                                                                          how to validate props?                                                                                                                                                                how to set state?                                                                                                how to force an update?                                                                            how to find a DOM node?                                                                    what is the react component life cycle?                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      how to handle child events?                                                                                                                  what is a ref in react and when do you use them?                                                                                                                      what is a react key and what does it do?                                                                                                                                                                                                                                                                                                                                                                      what is flux?              what are the elements of flux?                                                  what are the pros of using flux?                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          what are higher-order components?  why are they pure?                                                                                                                                    what are some best practices in react? | **Notes**  NoteGem Horizontal Line    **reactjs** = user interface component library     * library built in javascript to build out user interfaces by Facebook to solve problems when dealing with complex state, what user is going to be interacting with on web page     scalable, reusable, fun to write, everything is a component    **virtual dom** = engine on virtual representation of what the dom will be  need to offload state apps need to use from dom  **reconciliation** = react will detect that state of app has changed, then update virtual dom     * once it knows what's changed, it will update only those dom nodes     components are building blocks to any react app     * few different types of react components     anatomy of a react component:     * import React from 'react' * const Intro = ()=>{return(<div><h1>Hi Lambda</h1></div>);};     the html portion is a JS object = **JSX**  returning a JS object that describes html we want to make  {expression} will evaluate any javascript expression  two major pillars of react's design philosophy  imperative programming v declarative programming  as a programmer most of your time is spent reading other people's code and trying to understand it, not writing it yourself    **state** = data    import {useState} from 'React';    **exports** = exports a single function that can now be imported into another module with import    **export default \_\_\_** = exported by default     * others must be exported by name     import examples    import {a} from './dir/fileName'; (current dir)     * ../ = one higher dir     import{F1,F2,F3} from '.dir.js'    react uses top-to-bottom strategy when passing data    **props** = read only; when we want to pass info held on state inside one component to another component     * can never make changes   + helps ensure data flow stays clean, organized     **stateful component** = one that holds state data     * either   + as object placed inside constructor OR   + as function component that includes useState function from React 16.8     when data comes into application it is loaded and stored on state either in centralized component specifically for state management OR component rendering other components    other data specific to that component can be kept in that component    when data is consumed in multiple components, it's best to centralize that data in state in top-level component    components rendered in stateful component can receive that data via a props attribute     * sent down to child component * can be accessed there like any other object * if change, do not make change to prop data itself   + send changes back to state-holding component   + often stored changes are sent to parent container as enclosed info in called function     why pass into via props method?     * control * readability * maintenance * reusability     **side effect** = anything that affects something outside scope of function being executed     * examples      * fetching data from api * timers * logging * manually manipulating dom      * two categories      * those that require clean-up * those that don't     react component without side effects = **pure component**  if it renders same output for same state and props  **side effect** = something that can cause component to return different output for same state and props  react has tools to deal with them  one is effect hook, useEffect()    effect hook tells react component needs to run or execute     * some side effect takes in two parameters * callback function to run(?) side effect * dependency array     examples of effects with no clean-up     * network requests * manual dom manipulations * logging     can run and will not cause problems if component mounts unmounts and remounts again  other effects will continue to execute  forms of testing    **static code analysis** = examples eslint and flow    don't have to run your code  don't have to really write these  do not enforce style with eslint  can reactdom.render right in test  can simulate dom via node or jsdom in jest    unit test will mock out all dependencies  smaller test, testing something in particular  integration tests = rerenders app    have to mock out API calls  rendering more of the component working together  more coverage than unit test  can be slower and harder to set up  end-to-end tests = not jest but cypress  treating app as black box from a user's perspective, clicking and typing, then verifying app state, how assert would    in javascript, fundamental concept of test is something that throws an error when things are wrong    goal with testing is to use code in same way that it will be used in app    yarn run test react    check out puppeteer, glamorous, visual regression    test("title", ()=>{// test here: arrange, act, assert});  expect().toBe('')  expect().toMatch('')(strings)    when testing, focus on what are the important parts we want to verify about this     * what's the user's experience * how do I want to verify that user's experience * .not.toMatch('')     jsdom is memory heavy, so it adds to the start time  don't need it if you don't need to access dom  node can't import css so use     * converts css to json/js * module.exports={   testEnvironment:"JSDOM", (is default option)  moduleNameWrapper:{  '[\\.css$':require.resolve('test/style-mock')](file://.css$':require.resolve('test/style-mock')),  },'.module.css$': identity-obj-proxy'  }    cra v2 supports css modules by adding module before css will treat it as css module    will get back object with keys for every class name  import styles from './auto-scaling-text.module.css'    **identity object proxy** = when you put dot on an object it will take that and consider that to be the thing it's returning    can check that d3 is loaded where appropriate, like not on login screen    module.exports ={setupTestFrameworkScriptFile: require.resolve('./test/setup-testFramework.js')}  **code coverage** = "jest --coverage"  "test: watch": "jest --watch"    **test.only** = only run certain tests    const submit = new window.event('submit')  form.dispatchEvent(submit)  throw error('here')  jest.mock('../../utils/API', ()=>{  return {  posts{  create: jest.fn(()=>promise.resolve());  },  }  })    expect().toHaveBeenCalledTimes(1)  expect().toHaveBeenCalledWith('/')  expect().toBeGreaterThanOrEqual()  expect().toBeLessThanOrEqual()    **shallow rendering** = render every component that isn't a composite component  no custom component    getByLabelText(container, 'username')  getByText(container, 'print username').click()  expect(queryByTestId(container, 'printed-username')).toBeInTheDOM()  expect(queryByTestId(container, 'printed-username')).toHaveTextContent('text content here')  expect().toMatchSnapshot()  simulate.click(getByText('Load Greeting'))  simulate.click(getByText('Load Greeting').toHaveTextContent())  simulate.click(getByText('Load Greeting').toHaveAttribute('disabled')  .getByAltText()  renderIntoDocument(<div>)  fireEvent(node: HTMLelement, event:event)  import{render, simulate} from 'react-testing-library'  or cleanup, fireEvent, wait, waitForElement  .toMatchObject()                ReactJS - Overview  ReactJS is JavaScript library used for building reusable UI components. According to React official documentation, following is the definition:    React is a library for building composable user interfaces. It encourages the creation of reusable UI components, which present data that changes over time. Lots of people use React as the V in MVC. React abstracts away the DOM from you, offering a simpler programming model and better performance. React can also render on the server using Node, and it can power native apps using React Native. React implements one-way reactive data flow, which reduces the boilerplate and is easier to reason about than traditional data binding.    React Features     * JSX − JSX is JavaScript syntax extension. It isn't necessary to use JSX in React development, but it is recommended. * Components − React is all about components. You need to think of everything as a component. This will help you maintain the code when working on larger scale projects. * Unidirectional data flow and Flux − React implements one-way data flow which makes it easy to reason about your app. Flux is a pattern that helps keeping your data unidirectional. * License − React is licensed under the Facebook Inc. Documentation is licensed under CC BY 4.0.     React Advantages     * Uses virtual DOM which is a JavaScript object. This will improve apps performance, since JavaScript virtual DOM is faster than the regular DOM. * Can be used on client and server side as well as with other frameworks. * Component and data patterns improve readability, which helps to maintain larger apps.     React Limitations     * Covers only the view layer of the app, hence you still need to choose other technologies to get a complete tooling set for development. * Uses inline templating and JSX, which might seem awkward to some developers.     ReactJS - Environment Setup  In this chapter, we will show you how to set up an environment for successful React development. Notice that there are many steps involved but this will help speed up the development process later. We will need NodeJS, so if you don't have it installed, check the link from the following table.     |  |  | | --- | --- | | **Sr.No.** | **Software & Description** | | 1 | NodeJS and NPM  NodeJS is the platform needed for the ReactJS development. Checkout our [NodeJS Environment Setup](https://www.tutorialspoint.com/nodejs/nodejs_environment_setup.htm). |     After successfully installing NodeJS, we can start installing React upon it using npm. You can install ReactJS in two ways     * Using webpack and babel. * Using the create-react-app command.     Installing ReactJS using webpack and babel  Webpack is a module bundler (manages and loads independent modules). It takes dependent modules and compiles them to a single (file) bundle. You can use this bundle while developing apps using command line or, by configuring it using webpack.config file.    Babel is a JavaScript compiler and transpiler. It is used to convert one source code to other. Using this you will be able to use the new ES6 features in your code where, babel converts it into plain old ES5 which can be run on all browsers.    **Step 1 - Create the Root Folder**  Create a folder with name reactApp on the desktop to install all the required files, using the mkdir command.    C:\Users\username\Desktop>mkdir reactApp C:\Users\username\Desktop>cd reactApp    To create any module, it is required to generate the package.json file. Therefore, after Creating the folder, we need to create a package.json file. To do so you need to run the npm init command from the command prompt.    C:\Users\username\Desktop\reactApp>npm init    This command asks information about the module such as packagename, description, author etc. you can skip these using the –y option.    C:\Users\username\Desktop\reactApp>npm init -y Wrote to C:\reactApp\package.json: {  "name": "reactApp",  "version": "1.0.0",  "description": "",  "main": "index.js",  "scripts": {  "test": "echo \"Error: no test specified\" && exit 1"  },  "keywords": [],  "author": "",  "license": "ISC" }    **Step 2 - install React and react dom**  Since our main task is to install ReactJS, install it, and its dom packages, using install react and react-dom commands of npm respectively. You can add the packages we install, to package.json file using the --save option.    C:\Users\Tutorialspoint\Desktop\reactApp>npm install react --save C:\Users\Tutorialspoint\Desktop\reactApp>npm install react-dom --save    Or, you can install all of them in single command as:    C:\Users\username\Desktop\reactApp>npm install react react-dom --save    **Step 3 - Install webpack**  Since we are using webpack to generate bundler install webpack, webpack-dev-server and webpack-cli.    C:\Users\username\Desktop\reactApp>npm install webpack --save C:\Users\username\Desktop\reactApp>npm install webpack-dev-server --save C:\Users\username\Desktop\reactApp>npm install webpack-cli --save    Or, you can install all of them in single command as:    C:\Users\username\Desktop\reactApp>npm install webpack webpack-dev-server webpack-cli --save    **Step 4 - Install babel**  Install babel, and its plugins babel-core, babel-loader, babel-preset-env, babel-preset-react and, html-webpack-plugin    C:\Users\username\Desktop\reactApp>npm install babel-core --save-dev C:\Users\username\Desktop\reactApp>npm install babel-loader --save-dev C:\Users\username\Desktop\reactApp>npm install babel-preset-env --save-dev C:\Users\username\Desktop\reactApp>npm install babel-preset-react --save-dev C:\Users\username\Desktop\reactApp>npm install html-webpack-plugin --save-dev    Or, you can install all of them in single command as:    C:\Users\username\Desktop\reactApp>npm install babel-core babel-loader babel-preset-env   babel-preset-react html-webpack-plugin --save-dev    **Step 5 - Create the Files**  To complete the installation, we need to create certain files namely, index.html, App.js, main.js, webpack.config.js and, .*babelrc*. You can create these files manually or, using command prompt.    C:\Users\username\Desktop\reactApp>type nul > index.html C:\Users\username\Desktop\reactApp>type nul > App.js C:\Users\username\Desktop\reactApp>type nul > main.js C:\Users\username\Desktop\reactApp>type nul > webpack.config.js C:\Users\username\Desktop\reactApp>type nul > .babelrc    **Step 6 - Set Compiler, Server and Loaders**  Open webpack-config.js file and add the following code. We are setting webpack entry point to be main.js. Output path is the place where bundled app will be served. We are also setting the development server to 8001 port. You can choose any port you want.    webpack.config.js    const path = require('path'); const HtmlWebpackPlugin = require('html-webpack-plugin');  module.exports = {  entry: './main.js',  output: {  path: path.join(\_\_dirname, '/bundle'),  filename: 'index\_bundle.js'  },  devServer: {  inline: true,  port: 8001  },  module: {  rules: [  {  test: /\.jsx?$/,  exclude: /node\_modules/,  loader: 'babel-loader',  query: {  presets: ['es2015', 'react']  }  }  ]  },  plugins:[  new HtmlWebpackPlugin({  template: './index.html'  })  ] }    Open the package.json and delete "test" "echo \"Error: no test specified\" && exit 1" inside "scripts" object. We are deleting this line since we will not do any testing in this tutorial. Let's add the start and build commands instead.    "start": "webpack-dev-server --mode development --open --hot", "build": "webpack --mode production"    **Step 7 - index.html**  This is just regular HTML. We are setting div id = "app" as a root element for our app and adding index\_bundle.js script, which is our bundled app file.    <!DOCTYPE html> <html lang = "en">  <head>  <meta charset = "UTF-8">  <title>React App</title>  </head>  <body>  <div id = "app"></div>  <script src = 'index\_bundle.js'></script>  </body> </html>    **Step 8 − App.jsx and main.js**  This is the first React component. We will explain React components in depth in a subsequent chapter. This component will render Hello World.    App.js    import React, { Component } from 'react'; class App extends Component{  render(){  return(  <div>  <h1>Hello World</h1>  </div>  );  } } export default App;    We need to import this component and render it to our root App element, so we can see it in the browser.    main.js    import React from 'react'; import ReactDOM from 'react-dom'; import App from './App.js';  ReactDOM.render(<App />, document.getElementById('app'));    Note − Whenever you want to use something, you need to import it first. If you want to make the component usable in other parts of the app, you need to export it after creation and import it in the file where you want to use it.    Create a file with name .babelrc and copy the following content to it.    {  "presets":["env", "react"] }    **Step 9 - Running the Server**  The setup is complete and we can start the server by running the following command.    C:\Users\username\Desktop\reactApp>npm start    It will show the port we need to open in the browser. In our case, it is <http://localhost:8001/>. After we open it, we will see the following output.    Running the Server    **Step 10 - Generating the bundle**  Finally, to generate the bundle you need to run the build command in the command prompt as:    C:\Users\Tutorialspoint\Desktop\reactApp>npm run build    This will generate the bundle in the current folder as shown below.    Generating the bundle    Using the create-react-app command  Instead of using webpack and babel you can install ReactJS more simply by installing create-react-app.    **Step 1 - install create-react-app**  Browse through the desktop and install the Create React App using command prompt as shown below:    C:\Users\Tutorialspoint>cd C:\Users\Tutorialspoint\Desktop\ C:\Users\Tutorialspoint\Desktop>npx create-react-app my-app    This will create a folder named my-app on the desktop and installs all the required files in it.    **Step 2 - Delete all the source files**  Browse through the src folder in the generated my-app folder and remove all the files in it as shown below:    C:\Users\Tutorialspoint\Desktop>cd my-app/src C:\Users\Tutorialspoint\Desktop\my-app\src>del \* C:\Users\Tutorialspoint\Desktop\my-app\src\\*, Are you sure (Y/N)? y    **Step 3 - Add files**  Add files with names index.css and index.js in the src folder as:    C:\Users\Tutorialspoint\Desktop\my-app\src>type nul > index.css C:\Users\Tutorialspoint\Desktop\my-app\src>type nul > index.js    In the index.js file add the following code:    import React from 'react'; import ReactDOM from 'react-dom'; import './index.css';    **Step 4 - Run the project**  Finally, run the project using the start command.    npm start    Run the Project    ReactJS - JSX  React uses JSX for templating instead of regular JavaScript. It is not necessary to use it, however, following are some pros that come with it.     * It is faster because it performs optimization while compiling code to JavaScript. * It is also type-safe and most of the errors can be caught during compilation. * It makes it easier and faster to write templates, if you are familiar with HTML.     Using JSX  JSX looks like a regular HTML in most cases. We already used it in the Environment Setup chapter. Look at the code from App.jsx where we are returning div.    App.jsx    import React from 'react';  class App extends React.Component {  render() {  return (  <div>  Hello World!!!  </div>  );  } } export default App;    Even though it's similar to HTML, there are a couple of things we need to keep in mind when working with JSX.    Nested Elements  If we want to return more elements, we need to wrap it with one container element. Notice how we are using div as a wrapper for h1, h2 and p elements.    App.jsx    import React from 'react';  class App extends React.Component {  render() {  return (  <div>  <h1>Header</h1>  <h2>Content</h2>  <p>This is the content!!!</p>  </div>  );  } } export default App;    React JSX Wrapper    Attributes  We can use our own custom attributes in addition to regular HTML properties and attributes. When we want to add custom attribute, we need to use data- prefix. In the following example, we added data-myattribute as an attribute of p element.    import React from 'react';  class App extends React.Component {  render() {  return (  <div>  <h1>Header</h1>  <h2>Content</h2>  <p data-myattribute = "somevalue">This is the content!!!</p>  </div>  );  } } export default App;    JavaScript Expressions  JavaScript expressions can be used inside of JSX. We just need to wrap it with curly brackets {}. The following example will render 2.    import React from 'react';  class App extends React.Component {  render() {  return (  <div>  <h1>{1+1}</h1>  </div>  );  } } export default App;    React JSX Inline Javascript    We cannot use if else statements inside JSX, instead we can use conditional (ternary) expressions. In the following example, variable i equals to 1 so the browser will render true, If we change it to some other value, it will render false.    import React from 'react';  class App extends React.Component {  render() {  var i = 1;  return (  <div>  <h1>{i == 1 ? 'True!' : 'False'}</h1>  </div>  );  } } export default App;    React JSX Ternary Expression    Styling  React recommends using inline styles. When we want to set inline styles, we need to use camelCase syntax. React will also automatically append px after the number value on specific elements. The following example shows how to add myStyle inline to h1 element.    import React from 'react';  class App extends React.Component {  render() {  var myStyle = {  fontSize: 100,  color: '#FF0000'  }  return (  <div>  <h1 style = {myStyle}>Header</h1>  </div>  );  } } export default App;    React JSX Inline Style    Comments  When writing comments, we need to put curly brackets {} when we want to write comment within children section of a tag. It is a good practice to always use {} when writing comments, since we want to be consistent when writing the app.    import React from 'react';  class App extends React.Component {  render() {  return (  <div>  <h1>Header</h1>  {//End of the line Comment...}  {/\*Multi line comment...\*/}  </div>  );  } } export default App;    Naming Convention  HTML tags always use lowercase tag names, while React components start with Uppercase.    Note − You should use className and htmlFor as XML attribute names instead of class and for.    This is explained on React official page as:  Since JSX is JavaScript, identifiers such as class and for are discouraged as XML attribute names. Instead, React DOM components expect DOM property names such as className and htmlFor, respectively.    ReactJS - Components  In this chapter, we will learn how to combine components to make the app easier to maintain. This approach allows to update and change your components without affecting the rest of the page.    Stateless Example  Our first component in the following example is App. This component is owner of Header and Content. We are creating Header and Content separately and just adding it inside JSX tree in our App component. Only App component needs to be exported.    App.jsx    import React from 'react';  class App extends React.Component {  render() {  return (  <div>  <Header/>  <Content/>  </div>  );  } } class Header extends React.Component {  render() {  return (  <div>  <h1>Header</h1>  </div>  );  } } class Content extends React.Component {  render() {  return (  <div>  <h2>Content</h2>  <p>The content text!!!</p>  </div>  );  } } export default App;    To be able to render this on the page, we need to import it in main.js file and call reactDOM.render(). We already did this while setting the environment.    main.js    import React from 'react'; import ReactDOM from 'react-dom'; import App from './App.jsx';  ReactDOM.render(<App />, document.getElementById('app'));    The above code will generate the following result.    React Component Stateless    Stateful Example  In this example, we will set the state for owner component (App). The Header component is just added like in the last example since it doesn't need any state. Instead of content tag, we are creating table and tbody elements, where we will dynamically insert TableRow for every object from the data array.    It can be seen that we are using EcmaScript 2015 arrow syntax (=>) which looks much cleaner than the old JavaScript syntax. This will help us create our elements with fewer lines of code. It is especially useful when we need to create a list with a lot of items.    App.jsx  import React from 'react';  class App extends React.Component {  constructor() {  super();  this.state = {  data:   [  {  "id":1,  "name":"Foo",  "age":"20"  },  {  "id":2,  "name":"Bar",  "age":"30"  },  {  "id":3,  "name":"Baz",  "age":"40"  }  ]  }  }  render() {  return (  <div>  <Header/>  <table>  <tbody>  {this.state.data.map((person, i) => <TableRow key = {i}   data = {person} />)}  </tbody>  </table>  </div>  );  } } class Header extends React.Component {  render() {  return (  <div>  <h1>Header</h1>  </div>  );  } } class TableRow extends React.Component {  render() {  return (  <tr>  <td>{this.props.data.id}</td>  <td>{this.props.data.name}</td>  <td>{this.props.data.age}</td>  </tr>  );  } } export default App;    main.js    import React from 'react'; import ReactDOM from 'react-dom'; import App from './App.jsx';  ReactDOM.render(<App/>, document.getElementById('app'));    Note − Notice that we are using key = {i} inside map() function. This will help React to update only the necessary elements instead of re-rendering the entire list when something changes. It is a huge performance boost for larger number of dynamically created elements.    React Component Statefull    ReactJS - State  State is the place where the data comes from. We should always try to make our state as simple as possible and minimize the number of stateful components. If we have, for example, ten components that need data from the state, we should create one container component that will keep the state for all of them.    Using State  The following sample code shows how to create a stateful component using EcmaScript2016 syntax.    App.jsx    import React from 'react';  class App extends React.Component {  constructor(props) {  super(props);                   this.state = {  header: "Header from state...",  content: "Content from state..."  }  }  render() {  return (  <div>  <h1>{this.state.header}</h1>  <h2>{this.state.content}</h2>  </div>  );  } } export default App;    main.js    import React from 'react'; import ReactDOM from 'react-dom'; import App from './App.jsx';  ReactDOM.render(<App />, document.getElementById('app'));    This will produce the following result.    React State Simple    ReactJS - Props Overview  The main difference between state and props is that props are immutable. This is why the container component should define the state that can be updated and changed, while the child components should only pass data from the state using props.    Using Props  When we need immutable data in our component, we can just add props to reactDOM.render() function in main.js and use it inside our component.    App.jsx    import React from 'react';  class App extends React.Component {  render() {  return (  <div>  <h1>{this.props.headerProp}</h1>  <h2>{this.props.contentProp}</h2>  </div>  );  } } export default App;    main.js    import React from 'react'; import ReactDOM from 'react-dom'; import App from './App.jsx';  ReactDOM.render(<App headerProp = "Header from props..." contentProp = "Content  from props..."/>, document.getElementById('app'));  export default App;    This will produce the following result.    React Props Example    Default Props  You can also set default property values directly on the component constructor instead of adding it to the reactDom.render() element.    App.jsx    import React from 'react';  class App extends React.Component {  render() {  return (  <div>  <h1>{this.props.headerProp}</h1>  <h2>{this.props.contentProp}</h2>  </div>  );  } } App.defaultProps = {  headerProp: "Header from props...",  contentProp:"Content from props..." } export default App;    main.js    import React from 'react'; import ReactDOM from 'react-dom'; import App from './App.jsx';  ReactDOM.render(<App/>, document.getElementById('app'));    Output is the same as before.    React Props Example    State and Props  The following example shows how to combine state and props in your app. We are setting the state in our parent component and passing it down the component tree using props. Inside the render function, we are setting headerProp and contentProp used in child components.    App.jsx    import React from 'react';  class App extends React.Component {  constructor(props) {  super(props);  this.state = {  header: "Header from props...",  content: "Content from props..."  }  }  render() {  return (  <div>  <Header headerProp = {this.state.header}/>  <Content contentProp = {this.state.content}/>  </div>  );  } } class Header extends React.Component {  render() {  return (  <div>  <h1>{this.props.headerProp}</h1>  </div>  );  } } class Content extends React.Component {  render() {  return (  <div>  <h2>{this.props.contentProp}</h2>  </div>  );  } } export default App;    main.js    import React from 'react'; import ReactDOM from 'react-dom'; import App from './App.jsx';  ReactDOM.render(<App/>, document.getElementById('app'));    The result will again be the same as in the previous two examples, the only thing that is different is the source of our data, which is now originally coming from the state. When we want to update it, we just need to update the state, and all child components will be updated. More on this in the Events chapter.    React Props Example    ReactJS - Props Validation  Properties validation is a useful way to force the correct usage of the components. This will help during development to avoid future bugs and problems, once the app becomes larger. It also makes the code more readable, since we can see how each component should be used.    Validating Props  In this example, we are creating App component with all the props that we need. App.propTypes is used for props validation. If some of the props aren't using the correct type that we assigned, we will get a console warning. After we specify validation patterns, we will set App.defaultProps.    App.jsx    import React from 'react';  class App extends React.Component {  render() {  return (  <div>  <h3>Array: {this.props.propArray}</h3>  <h3>Bool: {this.props.propBool ? "True..." : "False..."}</h3>  <h3>Func: {this.props.propFunc(3)}</h3>  <h3>Number: {this.props.propNumber}</h3>  <h3>String: {this.props.propString}</h3>  <h3>Object: {this.props.propObject.objectName1}</h3>  <h3>Object: {this.props.propObject.objectName2}</h3>  <h3>Object: {this.props.propObject.objectName3}</h3>  </div>  );  } }  App.propTypes = {  propArray: React.PropTypes.array.isRequired,  propBool: React.PropTypes.bool.isRequired,  propFunc: React.PropTypes.func,  propNumber: React.PropTypes.number,  propString: React.PropTypes.string,  propObject: React.PropTypes.object }  App.defaultProps = {  propArray: [1,2,3,4,5],  propBool: true,  propFunc: function(e){return e},  propNumber: 1,  propString: "String value...",    propObject: {  objectName1:"objectValue1",  objectName2: "objectValue2",  objectName3: "objectValue3"  } } export default App;    main.js    import React from 'react'; import ReactDOM from 'react-dom'; import App from './App.jsx';  ReactDOM.render(<App/>, document.getElementById('app'));    Props are Valid    ReactJS - Component API  In this chapter, we will explain React component API. We will discuss three methods: setState(), forceUpdate and ReactDOM.findDOMNode(). In new ES6 classes, we have to manually bind this. We will use this.method.bind(this) in the examples.    Set State  setState() method is used to update the state of the component. This method will not replace the state, but only add changes to the original state.    import React from 'react';  class App extends React.Component {  constructor() {  super();                   this.state = {  data: []  }           this.setStateHandler = this.setStateHandler.bind(this);  };  setStateHandler() {  var item = "setState..."  var myArray = this.state.data.slice();          myArray.push(item);  this.setState({data: myArray})  };  render() {  return (  <div>  <button onClick = {this.setStateHandler}>SET STATE</button>  <h4>State Array: {this.state.data}</h4>  </div>  );  } } export default App;    We started with an empty array. Every time we click the button, the state will be updated. If we click five times, we will get the following output.    React Component API Set State    Force Update  Sometimes we might want to update the component manually. This can be achieved using the forceUpdate() method.    import React from 'react';  class App extends React.Component {  constructor() {  super();  this.forceUpdateHandler = this.forceUpdateHandler.bind(this);  };  forceUpdateHandler() {  this.forceUpdate();  };  render() {  return (  <div>  <button onClick = {this.forceUpdateHandler}>FORCE UPDATE</button>  <h4>Random number: {Math.random()}</h4>  </div>  );  } } export default App;    We are setting a random number that will be updated every time the button is clicked.    React Component API Force Update    Find Dom Node  For DOM manipulation, we can use ReactDOM.findDOMNode() method. First we need to import react-dom.    import React from 'react'; import ReactDOM from 'react-dom';  class App extends React.Component {  constructor() {  super();  this.findDomNodeHandler = this.findDomNodeHandler.bind(this);  };  findDomNodeHandler() {  var myDiv = document.getElementById('myDiv');  ReactDOM.findDOMNode(myDiv).style.color = 'green';  }  render() {  return (  <div>  <button onClick = {this.findDomNodeHandler}>FIND DOME NODE</button>  <div id = "myDiv">NODE</div>  </div>  );  } } export default App;    The color of myDiv element changes to green, once the button is clicked.    React Component API Find Dom Node    Note − Since the 0.14 update, most of the older component API methods are deprecated or removed to accommodate ES6.    ReactJS - Component Life Cycle  In this chapter, we will discuss component lifecycle methods.    Lifecycle Methods     * componentWillMount is executed before rendering, on both the server and the client side. * componentDidMount is executed after the first render only on the client side. This is where AJAX requests and DOM or state updates should occur. This method is also used for integration with other JavaScript frameworks and any functions with delayed execution such as setTimeout or setInterval. We are using it to update the state so we can trigger the other lifecycle methods. * componentWillReceiveProps is invoked as soon as the props are updated before another render is called. We triggered it from setNewNumber when we updated the state. * shouldComponentUpdate should return true or false value. This will determine if the component will be updated or not. This is set to true by default. If you are sure that the component doesn't need to render after state or props are updated, you can return false value. * componentWillUpdate is called just before rendering. * componentDidUpdate is called just after rendering. * componentWillUnmount is called after the component is unmounted from the dom. We are unmounting our component in main.js.     In the following example, we will set the initial state in the constructor function. The setNewnumber is used to update the state. All the lifecycle methods are inside the Content component.    App.jsx    import React from 'react';  class App extends React.Component {  constructor(props) {  super(props);    this.state = {  data: 0  }  this.setNewNumber = this.setNewNumber.bind(this)  };  setNewNumber() {  this.setState({data: this.state.data + 1})  }  render() {  return (  <div>  <button onClick = {this.setNewNumber}>INCREMENT</button>  <Content myNumber = {this.state.data}></Content>  </div>  );  } } class Content extends React.Component {  componentWillMount() {  console.log('Component WILL MOUNT!')  }  componentDidMount() {  console.log('Component DID MOUNT!')  }  componentWillReceiveProps(newProps) {   console.log('Component WILL RECIEVE PROPS!')  }  shouldComponentUpdate(newProps, newState) {  return true;  }  componentWillUpdate(nextProps, nextState) {  console.log('Component WILL UPDATE!');  }  componentDidUpdate(prevProps, prevState) {  console.log('Component DID UPDATE!')  }  componentWillUnmount() {  console.log('Component WILL UNMOUNT!')  }  render() {  return (  <div>  <h3>{this.props.myNumber}</h3>  </div>  );  } } export default App;    main.js    import React from 'react'; import ReactDOM from 'react-dom'; import App from './App.jsx';  ReactDOM.render(<App/>, document.getElementById('app'));  setTimeout(() => {  ReactDOM.unmountComponentAtNode(document.getElementById('app'));}, 10000);    After the initial render, we will get the following screen.    React Component Lifecycle Initial Screen    ReactJS - Forms  In this chapter, we will learn how to use forms in React.    **Simple Example**  In the following example, we will set an input form with value = {this.state.data}. This allows to update the state whenever the input value changes. We are using onChange event that will watch the input changes and update the state accordingly.    App.jsx    import React from 'react';  class App extends React.Component {  constructor(props) {  super(props);    this.state = {  data: 'Initial data...'  }  this.updateState = this.updateState.bind(this);  };  updateState(e) {  this.setState({data: e.target.value});  }  render() {  return (  <div>  <input type = "text" value = {this.state.data}   onChange = {this.updateState} />  <h4>{this.state.data}</h4>  </div>  );  } } export default App;    main.js    import React from 'react'; import ReactDOM from 'react-dom'; import App from './App.jsx';  ReactDOM.render(<App/>, document.getElementById('app'));    When the input text value changes, the state will be updated.    React Forms Simple    Complex Example  In the following example, we will see how to use forms from child component. onChange method will trigger state update that will be passed to the child input value and rendered on the screen. A similar example is used in the Events chapter. Whenever we need to update state from child component, we need to pass the function that will handle updating (updateState) as a prop (updateStateProp).    App.jsx    import React from 'react';  class App extends React.Component {  constructor(props) {  super(props);    this.state = {  data: 'Initial data...'  }  this.updateState = this.updateState.bind(this);  };  updateState(e) {  this.setState({data: e.target.value});  }  render() {  return (  <div>  <Content myDataProp = {this.state.data}   updateStateProp = {this.updateState}></Content>  </div>  );  } } class Content extends React.Component {  render() {  return (  <div>  <input type = "text" value = {this.props.myDataProp}   onChange = {this.props.updateStateProp} />  <h3>{this.props.myDataProp}</h3>  </div>  );  } } export default App;    main.js    import React from 'react'; import ReactDOM from 'react-dom'; import App from './App.jsx';  ReactDOM.render(<App/>, document.getElementById('app'));    This will produce the following result.    React Forms Complex    ReactJS - Events  In this chapter, we will learn how to use events.    **Simple Example**  This is a simple example where we will only use one component. We are just adding onClick event that will trigger updateState function once the button is clicked.    App.jsx    import React from 'react';  class App extends React.Component {  constructor(props) {  super(props);    this.state = {  data: 'Initial data...'  }  this.updateState = this.updateState.bind(this);  };  updateState() {  this.setState({data: 'Data updated...'})  }  render() {  return (  <div>  <button onClick = {this.updateState}>CLICK</button>  <h4>{this.state.data}</h4>  </div>  );  } } export default App;    main.js  import React from 'react'; import ReactDOM from 'react-dom'; import App from './App.jsx';  ReactDOM.render(<App/>, document.getElementById('app'));    This will produce the following result.    React Events Simple    Child Events  When we need to update the state of the parent component from its child, we can create an event handler (updateState) in the parent component and pass it as a prop (updateStateProp) to the child component where we can just call it.    App.jsx  import React from 'react';  class App extends React.Component {  constructor(props) {  super(props);    this.state = {  data: 'Initial data...'  }  this.updateState = this.updateState.bind(this);  };  updateState() {  this.setState({data: 'Data updated from the child component...'})  }  render() {  return (  <div>  <Content myDataProp = {this.state.data}   updateStateProp = {this.updateState}></Content>  </div>  );  } } class Content extends React.Component {  render() {  return (  <div>  <button onClick = {this.props.updateStateProp}>CLICK</button>  <h3>{this.props.myDataProp}</h3>  </div>  );  } } export default App;    main.js    import React from 'react'; import ReactDOM from 'react-dom'; import App from './App.jsx';  ReactDOM.render(<App/>, document.getElementById('app'));    This will produce the following result.    React Events Child    ReactJS - Refs  The ref is used to return a reference to the element. Refs should be avoided in most cases, however, they can be useful when we need DOM measurements or to add methods to the components.    Using Refs  The following example shows how to use refs to clear the input field. ClearInput function searches for element with ref = "myInput" value, resets the state, and adds focus to it after the button is clicked.    App.jsx    import React from 'react'; import ReactDOM from 'react-dom';  class App extends React.Component {  constructor(props) {  super(props);                   this.state = {  data: ''  }  this.updateState = this.updateState.bind(this);  this.clearInput = this.clearInput.bind(this);  };  updateState(e) {  this.setState({data: e.target.value});  }  clearInput() {  this.setState({data: ''});  ReactDOM.findDOMNode(this.refs.myInput).focus();  }  render() {  return (  <div>  <input value = {this.state.data} onChange = {this.updateState}   ref = "myInput"></input>  <button onClick = {this.clearInput}>CLEAR</button>  <h4>{this.state.data}</h4>  </div>  );  } } export default App;    main.js    import React from 'react'; import ReactDOM from 'react-dom'; import App from './App.jsx';  ReactDOM.render(<App/>, document.getElementById('app'));    Once the button is clicked, the input will be cleared and focused.    React Refs Example    ReactJS - Keys  React keys are useful when working with dynamically created components or when your lists are altered by the users. Setting the key value will keep your components uniquely identified after the change.    Using Keys  Let's dynamically create Content elements with unique index (i). The map function will create three elements from our data array. Since the key value needs to be unique for every element, we will assign i as a key for each created element.    App.jsx    import React from 'react';  class App extends React.Component {  constructor() {  super();                   this.state = {  data:[  {  component: 'First...',  id: 1  },  {  component: 'Second...',  id: 2  },  {  component: 'Third...',  id: 3  }  ]  }  }  render() {  return (  <div>  <div>  {this.state.data.map((dynamicComponent, i) => <Content   key = {i} componentData = {dynamicComponent}/>)}  </div>  </div>  );  } } class Content extends React.Component {  render() {  return (  <div>  <div>{this.props.componentData.component}</div>  <div>{this.props.componentData.id}</div>  </div>  );  } } export default App;    main.js    import React from 'react'; import ReactDOM from 'react-dom'; import App from './App.jsx';  ReactDOM.render(<App/>, document.getElementById('app'));    We will get the following result for the Key values of each element.    React Keys Example    If we add or remove some elements in the future or change the order of the dynamically created elements, React will use the key values to keep track of each element.    ReactJS - Router  In this chapter, we will learn how to set up routing for an app.    **Step 1 - Install React Router**    A simple way to install the react-router is to run the following code snippet in the command prompt window.    C:\Users\username\Desktop\reactApp>npm install react-router    **Step 2 - Create Components**  In this step, we will create four components. The App component will be used as a tab menu. The other three components (Home), (About) and (Contact) are rendered once the route has changed.    main.js  import React from 'react'; import ReactDOM from 'react-dom'; import { Router, Route, Link, browserHistory, IndexRoute } from 'react-router'  class App extends React.Component {  render() {  return (  <div>  <ul>  <li>Home</li>  <li>About</li>  <li>Contact</li>  </ul>  {this.props.children}  </div>  )  } } export default App;  class Home extends React.Component {  render() {  return (  <div>  <h1>Home...</h1>  </div>  )  } } export default Home;  class About extends React.Component {  render() {  return (  <div>  <h1>About...</h1>  </div>  )  } } export default About;  class Contact extends React.Component {  render() {  return (  <div>  <h1>Contact...</h1>  </div>  )  } } export default Contact;    **Step 3 - Add a Router**  Now, we will add routes to the app. Instead of rendering App element like in the previous example, this time the Router will be rendered. We will also set components for each route.    main.js    ReactDOM.render((  <Router history = {browserHistory}>  <Route path = "/" component = {App}>  <IndexRoute component = {Home} />  <Route path = "home" component = {Home} />  <Route path = "about" component = {About} />  <Route path = "contact" component = {Contact} />  </Route>  </Router> ), document.getElementById('app'))    When the app is started, we will see three clickable links that can be used to change the route.    Clickable Links    ReactJS - Flux Concept  Flux is a programming concept, where the data is uni-directional. This data enters the app and flows through it in one direction until it is rendered on the screen.    Flux Elements  Following is a simple explanation of the flux concept. In the next chapter, we will learn how to implement this into the app.     * Actions − Actions are sent to the dispatcher to trigger the data flow. * Dispatcher − This is a central hub of the app. All the data is dispatched and sent to the stores. * Store − Store is the place where the application state and logic are held. Every store is maintaining a particular state and it will update when needed. * View − The view will receive data from the store and re-render the app.     The data flow is depicted in the following image.    React Flux Concept Image    Flux Pros     * Single directional data flow is easy to understand. * The app is easier to maintain. * The app parts are decoupled.     ReactJS - Using Flux  In this chapter, we will learn how to implement flux pattern in React applications. We will use Redux framework. The goal of this chapter is to present the simplest example of every piece needed for connecting Redux and React.    **Step 1 - Install Redux**  We will install Redux via the command prompt window.    C:\Users\username\Desktop\reactApp>npm install --save react-redux    **Step 2 - Create Files and Folders**  In this step, we will create folders and files for our actions, reducers, and components. After we are done with it, this is how the folder structure will look like.  C:\Users\Tutorialspoint\Desktop\reactApp>mkdir actions C:\Users\Tutorialspoint\Desktop\reactApp>mkdir components C:\Users\Tutorialspoint\Desktop\reactApp>mkdir reducers C:\Users\Tutorialspoint\Desktop\reactApp>type nul > actions/actions.js C:\Users\Tutorialspoint\Desktop\reactApp>type nul > reducers/reducers.js C:\Users\Tutorialspoint\Desktop\reactApp>type nul > components/AddTodo.js C:\Users\Tutorialspoint\Desktop\reactApp>type nul > components/Todo.js C:\Users\Tutorialspoint\Desktop\reactApp>type nul > components/TodoList.js    React Redux Folder Structure    **Step 3 - Actions**  Actions are JavaScript objects that use type property to inform about the data that should be sent to the store. We are defining ADD\_TODO action that will be used for adding new item to our list. The addTodo function is an action creator that returns our action and sets an id for every created item.    actions/actions.js    export const ADD\_TODO = 'ADD\_TODO'  let nextTodoId = 0;  export function addTodo(text) {  return {  type: ADD\_TODO,  id: nextTodoId++,  text  }; }    **Step 4 - Reducers**  While actions only trigger changes in the app, the reducers specify those changes. We are using switch statement to search for a ADD\_TODO action. The reducer is a function that takes two parameters (state and action) to calculate and return an updated state.    The first function will be used to create a new item, while the second one will push that item to the list. Towards the end, we are using combineReducers helper function where we can add any new reducers we might use in the future.    reducers/reducers.js    import { combineReducers } from 'redux' import { ADD\_TODO } from '../actions/actions'  function todo(state, action) {  switch (action.type) {  case ADD\_TODO:  return {  id: action.id,  text: action.text,  }  default:  return state  } } function todos(state = [], action) {  switch (action.type) {  case ADD\_TODO:  return [  ...state,  todo(undefined, action)  ]  default:  return state  } } const todoApp = combineReducers({  todos }) export default todoApp    Step 5 - Store  The store is a place that holds the app's state. It is very easy to create a store once you have reducers. We are passing store property to the provider element, which wraps our route component.    main.js    import React from 'react'  import { render } from 'react-dom' import { createStore } from 'redux' import { Provider } from 'react-redux'  import App from './App.jsx' import todoApp from './reducers/reducers'  let store = createStore(todoApp) let rootElement = document.getElementById('app')  render(  <Provider store = {store}>  <App />  </Provider>,           rootElement )    **Step 6 - Root Component**  The App component is the root component of the app. Only the root component should be aware of a redux. The important part to notice is the connect function which is used for connecting our root component App to the store.    This function takes select function as an argument. Select function takes the state from the store and returns the props (visibleTodos) that we can use in our components.    App.jsx    import React, { Component } from 'react' import { connect } from 'react-redux' import { addTodo } from './actions/actions'  import AddTodo from './components/AddTodo.js' import TodoList from './components/TodoList.js'  class App extends Component {  render() {  const { dispatch, visibleTodos } = this.props    return (  <div>  <AddTodo onAddClick = {text =>dispatch(addTodo(text))} />  <TodoList todos = {visibleTodos}/>  </div>  )  } } function select(state) {  return {  visibleTodos: state.todos  } } export default connect(select)(App);    **Step 7 - Other Components**  These components shouldn't be aware of redux.    components/AddTodo.js    import React, { Component, PropTypes } from 'react'  export default class AddTodo extends Component {  render() {  return (  <div>  <input type = 'text' ref = 'input' />                                   <button onClick = {(e) => this.handleClick(e)}>  Add  </button>  </div>  )  }  handleClick(e) {  const node = this.refs.input  const text = node.value.trim()  this.props.onAddClick(text)  node.value = ''  } }    components/Todo.js    import React, { Component, PropTypes } from 'react'  export default class Todo extends Component {  render() {  return (  <li>  {this.props.text}  </li>  )  } }    components/TodoList.js    import React, { Component, PropTypes } from 'react' import Todo from './Todo.js'  export default class TodoList extends Component {  render() {  return (  <ul>  {this.props.todos.map(todo =>  <Todo  key = {todo.id}  {...todo}  />  )}  </ul>  )  } }    When we start the app, we will be able to add items to our list.    React Redux Example    ReactJS - Animations  In this chapter, we will learn how to animate elements using React.    **Step 1 - Install React CSS Transitions Group**  This is React add-on used for creating basic CSS transitions and animations. We will install it from the command prompt window:    C:\Users\username\Desktop\reactApp>npm install react-addons-css-transition-group    **Step 2 - Add a CSS file**  Let's create a new file style.css.    C:\Users\Tutorialspoint\Desktop\reactApp>type nul > css/style.css    To be able to use it in the app, we need to link it to the head element in index.html.    <!DOCTYPE html> <html lang = "en">  <head>  <link rel = "stylesheet" type = "text/css" href = "./style.css">  <meta charset = "UTF-8">  <title>React App</title>  </head>  <body>  <div id = "app"></div>  <script src = 'index\_bundle.js'></script>  </body> </html>    **Step 3 - Appear Animation**  We will create a basic React component.    The ReactCSSTransitionGroup element will be used as a wrapper of the component we want to animate. It will use transitionAppear and transitionAppearTimeout, while transitionEnter and transitionLeave are false.    App.jsx    import React from 'react'; var ReactCSSTransitionGroup = require('react-addons-css-transition-group');  class App extends React.Component {  render() {  return (  <div>  <ReactCSSTransitionGroup transitionName = "example"  transitionAppear = {true} transitionAppearTimeout = {500}  transitionEnter = {false} transitionLeave = {false}>                                           <h1>My Element...</h1>  </ReactCSSTransitionGroup>  </div>   );  } } export default App;    main.js    import React from 'react' import ReactDOM from 'react-dom'; import App from './App.jsx';  ReactDOM.render(<App />, document.getElementById('app'));    The CSS animation is very simple.    css/style.css    .example-appear {  opacity: 0.04; } .example-appear.example-appear-active {  opacity: 2;  transition: opacity 50s ease-in; }    Once we start the app, the element will fade in.    React Animations Appear    **Step 4 - Enter and Leave Animations**  Enter and leave animations can be used when we want to add or remove elements from the list.    App.jsx  import React from 'react'; var ReactCSSTransitionGroup = require('react-addons-css-transition-group');  class App extends React.Component {  constructor(props) {  super(props);                   this.state = {  items: ['Item 1...', 'Item 2...', 'Item 3...', 'Item 4...']  }  this.handleAdd = this.handleAdd.bind(this);  };  handleAdd() {  var newItems = this.state.items.concat([prompt('Create New Item')]);  this.setState({items: newItems});  }  handleRemove(i) {  var newItems = this.state.items.slice();  newItems.splice(i, 1);  this.setState({items: newItems});  }  render() {  var items = this.state.items.map(function(item, i) {  return (  <div key = {item} onClick = {this.handleRemove.bind(this, i)}>  {item}  </div>  );  }.bind(this));    return (  <div>   <button onClick = {this.handleAdd}>Add Item</button>                                   <ReactCSSTransitionGroup transitionName = "example"   transitionEnterTimeout = {500} transitionLeaveTimeout = {500}>  {items}  </ReactCSSTransitionGroup>  </div>  );  } } export default App;    main.js    import React from 'react' import ReactDOM from 'react-dom'; import App from './App.jsx';  ReactDOM.render(<App />, document.getElementById('app'));    css/style.css    .example-enter {  opacity: 0.04; } .example-enter.example-enter-active {  opacity: 5;  transition: opacity 50s ease-in; } .example-leave {  opacity: 1; } .example-leave.example-leave-active {  opacity: 0.04;  transition: opacity 50s ease-in; }    When we start the app and click the Add Item button, the prompt will appear.    React Animations Prompt    Once we enter the name and press OK, the new element will fade in.    React Animations Enter    Now we can delete some of the items (Item 3...) by clicking it. This item will fade out from the list.    React Animations Leave    ReactJS - Higher Order Components  Higher order components are JavaScript functions used for adding additional functionalities to the existing component. These functions are pure, which means they are receiving data and returning values according to that data. If the data changes, higher order functions are re-run with different data input. If we want to update our returning component, we don't have to change the HOC. All we need to do is change the data that our function is using.    Higher Order Component (HOC) is wrapping around "normal" component and provide additional data input. It is actually a function that takes one component and returns another component that wraps the original one.    Let us take a look at a simple example to easily understand how this concept works. The MyHOC is a higher order function that is used only to pass data to MyComponent. This function takes MyComponent, enhances it with newData and returns the enhanced component that will be rendered on the screen.    import React from 'react';  var newData = {  data: 'Data from HOC...', }  var MyHOC = ComposedComponent => class extends React.Component {  componentDidMount() {  this.setState({  data: newData.data  });  }  render() {  return <ComposedComponent {...this.props} {...this.state} />;  } }; class MyComponent extends React.Component {  render() {  return (  <div>  <h1>{this.props.data}</h1>  </div>  )  } }  export default MyHOC(MyComponent);    If we run the app, we will see that data is passed to MyComponent.    React HOC Output    Note − Higher order components can be used for different functionalities. These pure functions are the essence of functional programming. Once you are used to it, you will notice how your app is becoming easier to maintain or to upgrade.    ReactJS - Best Practices  In this chapter, we will list React best practices, methods, and techniques that will help us stay consistent during the app development.     * **State** − The state should be avoided as much as possible. It is a good practice to centralize the state and pass it down the component tree as props. Whenever we have a group of components that need the same data, we should set a container element around them that will hold the state. Flux pattern is a nice way of handling the state in React apps. * **PropTypes** − The PropTypes should always be defined. This will help is track all props in the app and it will also be useful for any developer working on the same project. * **Render** − Most of the app's logic should be moved inside the render method. We should try to minimize logic in component lifecycle methods and move that logic in the render method. The less state and props we use, the cleaner the code will be. We should always make the state as simple as possible. If we need to calculate something from the state or props, we can do it inside the render method. * **Composition** − React team suggests to use a single responsibility principle. This means that one component should only be responsible for one functionality. If some of the components have more than one functionality, we should refactor and create a new component for every functionality. * **Higher Order Components (HOC)** − Former React versions offered mixins for handling reusable functionalities. Since mixins are now deprecated, one of the solutions is to use HOC. | |
| |  | | --- | | **Summary** | |