Testing is a very important part of application development. Testing helps in identifying the bugs at the earlier stage and ensure the proper working of the application.

It is important to write unit test cases to all the React components that we create as a part of a React application.

There are a lot of libraries available in the market to test React applications, but most popularly used libraries are Jest and enzyme.

**Jest:**

Jest is a javascript testing library which runs in the node. It was introduced by Facebook. It can be used to test any UI libraries but shines well when used to test React applications.

**Advantages of Jest:**

1. JEST is faster because jest can run test cases parallel
2. Testing DOM is very much simpler with JEST as it is bundled with JSDOM
3. Provides awesome support for mocking,
4. Able to mock dependencies using simple mock functions,
5. Able to  mock modules
6. Able to mock timer related functions like setTimeout, setInterval
7. Provides built-in support for coverage reports
8. Supports snapshot testing. This snapshot testing is a major advantage we will discuss more it in further videos
9. Support for the promise

**Enzyme:**

In order to test React applications along with JEST, we need enzyme. Enzyme is a JS testing utility for React  
Enzyme helps in asserting, manipulating and traversing through the components. We will be able to get the output of the component easily.  
We can simulate mounting of a component, simulate changing the state of a component using enzyme very easily.  
Hence we use enzyme along with JEST 

Installing jest is taken care of by create-react-app tool, so we need not install jest manually. But enzyme is not installed by create-react-app, so we need to install the enzyme.

Enzyme needs enzyme-adapter to be configured, hence we need to install enzyme-adapter. The version of the enzyme-adapter should match with the react version installed. The latest version of React is 16, hence, we need to install enzyme-adapter-react-16.

Now we have to configure enzyme-adapter. Create a file setupTests.js within the src folder and write the below code

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

We configure the enzyme adapter by creating a new adapter and passing it to the configure method.

The command to run the test cases in React application is npm test. You can observe the command in the scripts section of the package.json file

1. "scripts": {
2. "start": "react-scripts start",
3. "build": "react-scripts build",
4. "test": "react-scripts test",
5. "eject": "react-scripts eject"
6. }

When we run the command npm test, jest runs all the test cases present in all the files within the src folder with.test.js or .spec.js extension. If we are writing test cases for App component present inside the App.js file, then the common naming convention for the test file would be App.test.js or App.spec.js

The test cases can also be written inside \_\_tests\_\_ folder within the src folder.

The syntax of Jest is similar to Jasmine. We use the describe method to group the test cases and it() method to write the test cases as shown below:

1. describe('title',()=>{
2. it('title',()=>{
3. *// test case*
4. })
5. })

Within a test case, we use expect function, which returns the actual value. The expect function is chained with matcher. Matcher returns the expected value.

**Jest Matchers**

Some of the built-in matchers supported by jest are:

**toBe:** checks the equality using (===) operator.

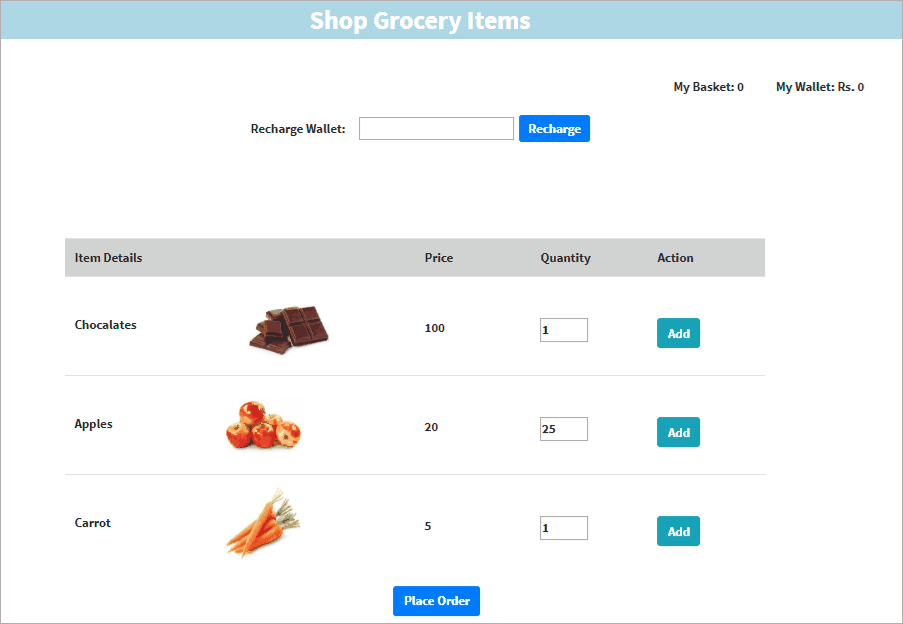
**toEqual:** checks the equality using (==) operator. Used for checking object equality

**toBeDefined:** checks whether the value is defined

**toBeTruthy:** checks whether the value is true

There are many more matchers available in JEST. Refer to the JEST official documentation to know about matchers.

Consider the shopping cart application which works as shown below



In this application

1. Items are displayed, the user can add items to the cart using the add button
2. The user can specify the quantity required for each item and then add the item to the cart
3. Once the user adds items to the cart, My Basket gets updated. MyBasket indicates the number of items added to the cart and the Total price also gets displayed.
4. Similarly, user can add other items as well to the cart by specifying the quantity and my basket and the total price gets updated accordingly
5. Users can place the order by clicking on the place order button.
6. Users can place the order if there is sufficient balance in his/her wallet. Hence the user has to recharge some amount to his/her account using the recharge button by entering the amount in the input field
7. Once user recharges, mywallet gets updated and the user will be able to place the order. Once the user places an order, the order placed successfully message is displayed

The application contains 2 components ItemDetails and Item component.

Item component displays the item name, image, price, quantity input field and Add button

ItemDetails component displays all the items, items added to the cart, Wallet balance, recharge button, Total amount and Place Order button

**Highlights:**

1. Understanding the Shopping cart application

**Demo Steps:**

1. Create a new application Shopping Cart using create-react-app as shown below

1. npx create-react-app shoppingcart

2. Create Item.js file within the src folder as shown below

1. import React from 'react';
2. import 'bootstrap/dist/css/bootstrap.min.css';
3. class Item extends React.Component {
4. constructor({quantity}){
5. super()
6. this.state= {
7. itemids:[],
8. quant:quantity,
9. errorMessage:"",
10. name:""
11. }
12. }
13. handleQuantityChange(event,maxquantity,name) {
14. const newQuantity = event.target.value
15. if(newQuantity > parseInt(maxquantity)) {
16. var message="You cannot buy more than " +maxquantity+" "+name
17. this.setState({errorMessage:"You cannot buy more than " +maxquantity+" "+name})
18. }
19. this.setState({quant:newQuantity})
20. this.props.quantityChange(message)
21. }
23. addItem(id,price) {
24. const newQuantity = this.state.quant
25. this.props.callbackParent(id,newQuantity,price)
26. }
28. render() {
29. var id=this.props.itemid
30. var maxquantity = this.props.maxquantity
31. return(
33. <tr>
34. <td style={{width:"30%"}}>
35. <div className="inline-block">{this.props.name}</div>
36. <div className="inline-block spacing">
37. <img width="100px" height="100px" src={this.props.img} />
38. </div>
39. </td>
40. <td style={{width:"10%"}}>
41. <div class="align-center">{this.props.price}</div>
42. </td>
43. <td style={{width:"10%"}}>
44. <div className="align-center"><input className="input-width" id="quantity" type="number" value={this.state.quant} onChange = {(e)=> this.handleQuantityChange(e,maxquantity,this.props.name)}/></div>
45. </td>
46. <td style={{width:"10%"}}>
47. <div className="align-center">
48. <button className="btn btn-info" id="addButton" type="button" onClick={() => this.addItem(id,this.props.price)}>Add</button>
49. </div>
50. </td>
51. </tr>
52. )
53. }
54. }
55. export default Item;

3. Create ItemDetails.js component as shown below

1. import React, {Component} from 'react';
2. import Item from './Item';
3. import 'bootstrap/dist/css/bootstrap.min.css';
4. const API = '/products.json';
5. class ItemDetails extends Component {
6. constructor(props) {
7. super(props);
8. this.placeOrder = this.placeOrder.bind(this)
9. this.resetErrorMessage = this.resetErrorMessage.bind(this)
10. this.fetchData = this.fetchData.bind(this);
11. this.state = {
12. itemDetails: [],
13. error:null,
14. quantityArr: [],
15. wallet: 0,
16. totalPrice: 0,
17. errorMessage:""
18. };
20. }
21. fetchData() {
22. fetch(API)
23. .then(response => {
24. if (response.ok) {
25. return response.json();
26. } else {
27. throw new Error('Something went wrong ...');
28. }
29. })
30. .then(data => {
31. this.setState({ itemDetails: data})
32. })
33. .catch(error => {
34. this.setState({ error })
35. });
36. }
37. componentDidMount() {
38. this.fetchData()
39. }
40. resetErrorMessage() {
41. this.setState({errorMessage:""})
42. }
43. resetValues() {
44. this.setState({errorMessage:"",quantityArr:[],totalPrice:0})
45. }
47. handleRecharge(recharge) {
48. if (recharge == "" || recharge == 0) {
49. this.setState({
50. errorMessage: "Please enter valid amount to recharge"
51. })
52. setTimeout(()=>{this.resetErrorMessage()},5000);
53. } else {
54. var wallet = recharge
55. this.setState({
56. wallet: recharge
57. })
58. this.setState({
59. errorMessage: "Rs." + wallet + " " + "added to the wallet"
60. })
61. setTimeout(()=>{this.resetErrorMessage()},5000);
62. }
63. }
64. placeOrder() {
65. if (this.state.quantityArr.length == 0) {
66. this.setState({
67. errorMessage: "Please add items to cart"
68. })
69. setTimeout(()=>{this.resetErrorMessage()},5000);
70. }
72. else if((this.state.wallet < this.state.totalPrice)|| (this.state.wallet==0)) {
73. this.setState({errorMessage:"You do not have sufficient balance to place the order. Please recharge your account"})
74. setTimeout(()=>{this.resetErrorMessage()},10000);
75. }
76. else if(this.state.wallet >= this.state.totalPrice) {
77. var balance = this.state.wallet-this.state.totalPrice
78. this.setState({errorMessage:"Your order placed successfully",wallet:balance})
79. setTimeout(()=>{this.resetValues()},5000)
80. }
81. }
82. onChildChanged(id, newQuantity, rprice) {
83. var index = this.state.quantityArr.findIndex(el => el[id] != undefined)
85. if (index > -1) {
86. this.state.totalPrice += (-this.state.quantityArr[index][id] \* rprice) + (newQuantity \* rprice)
87. this.state.quantityArr[index][id] = newQuantity
88. } else {
89. var obj = {}
90. obj[id] = newQuantity
91. this.state.quantityArr.push(obj)
92. this.state.totalPrice += (newQuantity \* rprice)
93. }
94. this.setState({
95. quantityArr: this.state.quantityArr,
96. totalPrice: this.state.totalPrice
97. })
98. }
99. onQuanityChange(message) {
100. this.setState({errorMessage: message})
101. setTimeout(()=>{this.resetErrorMessage()},5000)
102. }
103. render() {
104. var items = this.state.itemDetails
105. return (
106. <div className="container"><br/>
107. <div id="heading">Shop Grocery Items</div><br/><br/><br/>
108. <div className="alignRight"> My Basket: {this.state.quantityArr.length} &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp;My Wallet: Rs. {this.state.wallet} </div><br/><br/>
109. <div className="textAlign"> Recharge Wallet: &nbsp;&nbsp;&nbsp;&nbsp;<input type="text"
110. ref="recharge"/>&nbsp;&nbsp;
111. <button onClick={()=>this.handleRecharge(this.refs.recharge.value)} type="button" id="recharge" className="btn btn-primary"> Recharge </button></div><br/><br/>
112. <span id="error">{this.state.errorMessage}</span><br/>
113. <div className={this.state.totalPrice ? 'alignTableRight':'alignTableRight hidden'}> Total Price: {this.state.totalPrice}</div> <br/><br/>
114. <table className="custom-table table table-responsive">
115. <thead className="header">
116. <tr>
117. <th>Item Details</th>
118. <th>Price</th>
119. <th>Quantity</th>
120. <th>Action</th>
121. </tr>
122. </thead>
123. <tbody>{this.state.itemDetails.map(itemDetail =>
124. <Item key={itemDetail.itemId}
125. itemid={itemDetail.itemId}
126. name={itemDetail.itemName}
127. img={itemDetail.ItemImg}
128. price={itemDetail.itemPrice}
129. quantity={itemDetail.itemQuantity}
130. maxquantity={itemDetail.MaxQuantity}
131. callbackParent={(id, quant, price) => this.onChildChanged(id, quant, price)}
132. quantityChange={(message) => this.onQuanityChange(message)}/>)}
133. </tbody>
134. </table >
135. <button className="marginLeft btn btn-primary" type="button" onClick = {this.placeOrder}>Place Order</button></div>
136. )
137. }
138. }
139. export default ItemDetails;

4. Render ItemDetails component within the index.js file as shown below

1. import React from 'react';
2. import ReactDOM from 'react-dom';
3. import './index.css';
4. import App from './App';
5. import ItemDetails from './ItemDetails'
6. import \* as serviceWorker from './serviceWorker';
7. ReactDOM.render(<ItemDetails />, document.getElementById('root'));
8. serviceWorker.unregister();

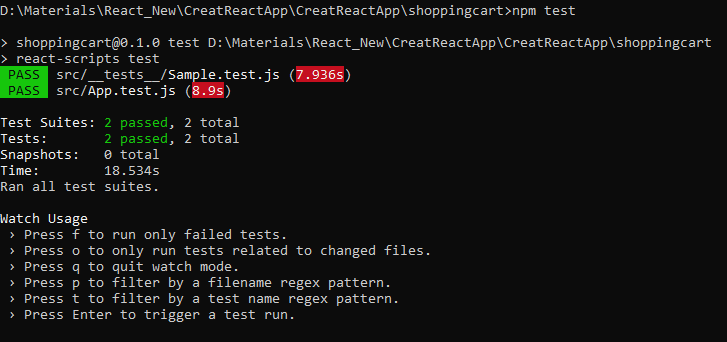
5. Include images and products.json within the public folder and include font.woff file within src. The images, products.json, and font can be downloaded from [here](https://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_012792610464833536403_shared/web-hosted/assets/shoppingcart.zip)

6. Create folder \_\_tests\_\_ within src and create a sample.test.js file and write a sample test case as shown below

1. describe('sample test case',()=>{
2. it('sample test case',()=>{
3. expect('hello').toEqual('hello')
4. })
5. })

By default, create-react-app would have App.test.js file for the App component. You can retain it or delete it.

7. Run the test cases using npm test command and the output will be as shown below



npm test command will run all the test cases in files with .test.js or .spec.js extension and displays how many test cases have passed and how many have failed.

Let us observe the test cases for Item component

In order to render the component within the test case, we can use the mount or shallow method of the enzyme. The shallow method renders only the component to the DOM, but the mount method renders the child components.

The shallow method returns the wrapper instance around the rendered output. The render method returns the ReactWrapper.

Using the wrapper that shallow and mount method returns, we can invoke state, setState, find methods

1. describe('quantity input', () => {
2. let quantityChange;
3. let wrapper;
4. let input;
5. beforeEach(()=>{
6. quantityChange = jest.fn();
7. wrapper = shallow (<Item quantityChange={quantityChange} />);
8. input = wrapper.find('#quantity')
9. input.simulate('change', {target: {value:3}});
10. })
11. it('should respond to the change event and change the state of Item component', () => {
12. expect(wrapper.state('quant')).toEqual(3);
13. });

The above test case checks whether the quantity state changes appropriately after the user enters the value to the quantity input field.

To achieve the above requirement

1. We need to mount the component using a shallow method when we mount the component, we need to pass the quantityChange method as a prop. The method is mocked using jest.fn()

2. After mounting the component, the quantity input field is found and the change event is simulated with value 3.

3. The within it method we check whether the quant state value is 3.

1. it('should respond to the change event and change the state of Item component', () => {
2. wrapper.instance().handleQuantityChange({target:{value:1}},2,'roopan')
3. expect(quantityChange).toHaveBeenCalledWith(undefined)
4. wrapper.instance().handleQuantityChange({target:{value:3}},2,'roopan')
5. expect(quantityChange).toHaveBeenCalledWith('You cannot buy more than 2 roopan')
6. wrapper.instance().handleQuantityChange({target:{value:2}},2,'roopan')
7. expect(quantityChange).toHaveBeenCalledWith(undefined)
8. });

In the above test case, the handleQuantityChangemethod is invoked using instance method and then we test whether the quantityChange method has been called with correct arguments.

The beforeEach method gets executed before executing each test case. We can use this method to do the initial setup.

Let us observe the test cases for the Item Details component

1. describe('Testing ItemDetails component',()=>{
2. describe('testing the fetch API call with mock response',()=>{
3. let body;
4. let init;
5. let wrapper;
6. beforeEach( done =>{
7. body = [{
8. "itemId": "1",
9. "itemName": "Chocalates",
10. "ItemImg": "/Images/chocalates.png",
11. "itemPrice": "100",
12. "itemQuantity": "1",
13. "MaxQuantity": "10"
14. }]
15. init = {
16. status: 200
17. }
18. window.fetch = jest.fn().mockImplementation(() => Promise.resolve(new Response (JSON.stringify(body), init)));
19. wrapper = mount( < ItemDetails/> );
20. done();
21. })
22. it('testing fetch API call for success',async ()=>{
23. expect(wrapper.state('itemDetails')).toEqual([{
24. "itemId": "1",
25. "itemName": "Chocalates",
26. "ItemImg": "/Images/chocalates.png",
27. "itemPrice": "100",
28. "itemQuantity": "1",
29. "MaxQuantity": "10"
30. }])
31. })
32. })
33. describe('testing for fetch failure', ()=>{
34. let wrapper;
35. beforeEach( done =>{
36. window.fetch = jest.fn().mockImplementation(() => Promise.resolve({ok:false}));
37. wrapper = mount( < ItemDetails/> );
38. done();
39. })
40. it('testing for fetch failure',async ()=>{
41. expect(wrapper.state('error')).toBeDefined()
42. })
43. })

In the above test case, the API response is mocked and then tested against the expected value

1. it('testing success message is set for successfully placing the order',()=>{
2. let wrapper = mount (<ItemDetails />);
3. wrapper.setState({quantityArr: [{
4. "itemId":"1",
5. "itemName":"Chocalates",
6. "ItemImg":"/Images/chocalates.png",
7. "itemPrice":"100",
8. "itemQuantity":"1",
9. "MaxQuantity":"10"
10. }],wallet:5,totalPrice:5})
11. wrapper.instance().placeOrder()
12. console.log(wrapper.state())
13. expect(wrapper.state('errorMessage')).toEqual('Your order placed successfully')
14. expect(wrapper.state('wallet')).toEqual(5-5)
15. })

In the above test case, the quantityArr state is set to a mock value and then the placeOrder method is invoked. Then we test the errorMessage and wallet state values are changed appropriately.

**Highlights:**

* Write unit test cases for Item and ItemDetails component

**Demo Steps**

1. Install enzyme and enzyme-adapter-react-16 using the below command

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

2. Create a setupTests.js file within the src folder and write the below code

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

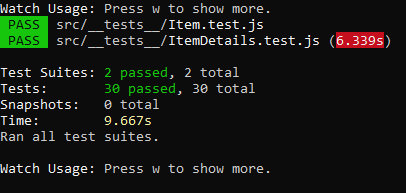
3. Install isomorphic-fetch, which is used for mocking the fetch AJAX call

1. npm install isomorphic-fetch

4. Write unit test cases for Item and ItemDetails component within \_\_tests\_\_ folder

The test cases can be downloaded [here](https://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_auth_012792610464833536403_shared/web-hosted/assets/testsCases.zip)

4. Run the command npm test and you can see the below output



Snapshot Testing is used to ensure that the UI of the component does not change. Once we create a snapshot for any component, then whenever we run the npm test command, Jest will compare the created snapshot against the component's output, if there is any change in the UI of the component then the test case fails.

To generate a snapshot, we need to install react-test-renderer package.

1. npm install react-test-renderer

 To perform snapshot testing we need to write a test case as shown below

1. import Item from '../Item.js';
2. import renderer from 'react-test-renderer';
3. import React from 'react';
4. test('Item component renders the props correctly', () => {
5. const itemDetail = {
6. "itemId":"1",
7. "itemName":"Chocalates",
8. "ItemImg":"/Images/chocalates.png",
9. "itemPrice":"100",
10. "itemQuantity":"1",
11. "MaxQuantity":"10"
12. }
13. const myFun1 = jest.fn()
14. const myFun2 = jest.fn()
15. const rendered = renderer.create(
16. <Item itemid={itemDetail.itemId}
17. name={itemDetail.itemName}
18. img={itemDetail.ItemImg}
19. price={itemDetail.itemPrice}
20. quantity={itemDetail.itemQuantity}
21. maxquantity={itemDetail.MaxQuantity}
22. callbackParent={(id, quant, price) => myFun1(id, quant, price)}
23. quantityChange={(message) => myFun2(message)} />
24. );
25. expect(rendered.toJSON()).toMatchSnapshot();
26. });

The above test case generates a snapshot for Item component after running npm test command inside \_\_snapshots\_\_ folder. If the component is updated then snapshot doesn't match and test case fails.

**Highlights:**

* Create a snapshot for Item component
* Check whether the snapshot matches with the Item component UI

**Demo steps:**

1. Install react-test-renderer using the below command

1. npm install react-test-renderer

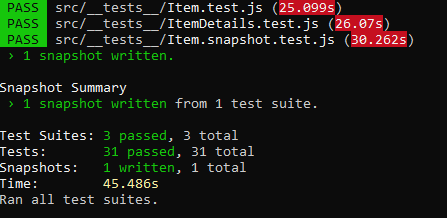
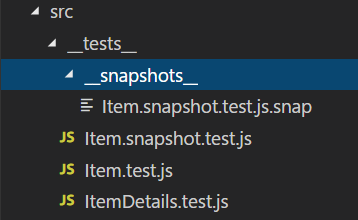
2. Create Item.snapshot.test.js inside \_\_tests\_\_ file as shown below

1. import Item from '../Item.js';
2. import renderer from 'react-test-renderer';
3. import React from 'react';
4. test('Item component renders the props correctly', () => {
5. const itemDetail = {
6. "itemId":"1",
7. "itemName":"Chocalates",
8. "ItemImg":"/Images/chocalates.png",
9. "itemPrice":"100",
10. "itemQuantity":"1",
11. "MaxQuantity":"10"
12. }
13. const myFun1 = jest.fn()
14. const myFun2 = jest.fn()
15. const rendered = renderer.create(
16. <Item itemid={itemDetail.itemId}
17. name={itemDetail.itemName}
18. img={itemDetail.ItemImg}
19. price={itemDetail.itemPrice}
20. quantity={itemDetail.itemQuantity}
21. maxquantity={itemDetail.MaxQuantity}
22. callbackParent={(id, quant, price) => myFun1(id, quant, price)}
23. quantityChange={(message) => myFun2(message)} />
24. );
25. expect(rendered.toJSON()).toMatchSnapshot();
26. });

3. Run the test cases using below command

1. npm test

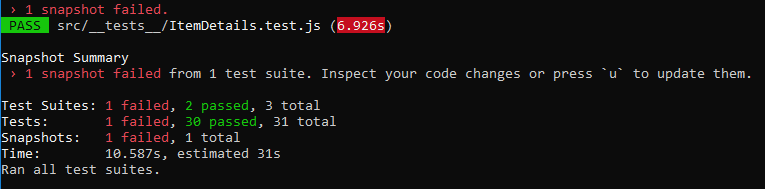
The above command will create a snapshot inside \_\_snapshots\_\_ folder as shown below

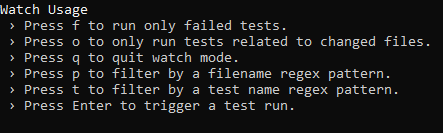
4. Now add an h1 element to the Item component as shown below

2. render() {
3. var id=this.props.itemid
4. var maxquantity = this.props.maxquantity
5. return(
6. <React.Fragment>
7. <h1>Item Component</h1>
8. <tr>
9. <td style={{width:"30%"}}>
10. <div className="inline-block">{this.props.name}</div>
11. <div className="inline-block spacing">
12. <img width="100px" height="100px" src={this.props.img} />
13. </div>
14. </td>
15. <td style={{width:"10%"}}>
16. <div class="align-center">{this.props.price}</div>
17. </td>
18. <td style={{width:"10%"}}>
19. <div className="align-center"><input className="input-width" id="quantity" type="number" value={this.state.quant} onChange = {(e)=> this.handleQuantityChange(e,maxquantity,this.props.name)}/></div>
20. </td>
21. <td style={{width:"10%"}}>
22. <div className="align-center">
23. <button className="btn btn-info" id="addButton" type="button" onClick={() => this.addItem(id,this.props.price)}>Add</button>
24. </div>
25. </td>
26. </tr>
27. </React.Fragment>
28. )
29. }

5. Now when we run the npm test command the test case fails as shown below



6. To match the updated UI just press option u. Option u will update the snapshot.



To check the code coverage add --coverage option to the test command in the package.json file as shown below

1. "scripts": {
2. "start": "react-scripts start",
3. "build": "react-scripts build",
4. "test": "react-scripts test --coverage",
5. "eject": "react-scripts eject"
6. }

After adding --coverage option, when we run the npm test command the coverage report is displayed as shown below.