

Introduction

ReactJS is a declarative, efficient, and flexible JavaScript library for building reusable UI components. It is an open-source, component-based front end library responsible only for the view layer of the application. It was created by **Jordan Walke**, who was a software engineer at **Facebook**. It was initially developed and maintained by Facebook and was later used in its products like **WhatsApp & Instagram**. Facebook developed ReactJS in **2011** in its newsfeed section, but it was released to the public in the month of **May 2013**.

Today, most of the websites are built using MVC (model view controller) architecture. In MVC architecture, React is the 'V' which stands for view, whereas the architecture is provided by the Redux or Flux.

A ReactJS application is made up of multiple components, each component responsible for outputting a small, reusable piece of HTML code. The components are the heart of all React applications. These Components can be nested with other components to allow complex applications to be built of simple building blocks. ReactJS uses virtual DOM based mechanism to fill data in HTML DOM. The virtual DOM works fast as it only changes individual DOM elements instead of reloading complete DOM every time.

To create React app, we write React components that correspond to various elements. We organize these components inside higher level components which define the application structure. For example, we take a form that consists of many elements like input fields, labels, or buttons. We can write each element of the form as React components, and then we combine it into a higher-level component, i.e., the form component itself. The form components would specify the structure of the form along with elements inside of it.

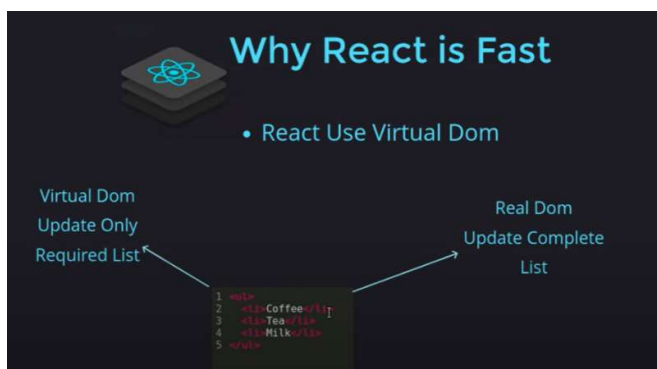
Why learn ReactJS?

Today, many JavaScript frameworks are available in the market (like angular, node), but still, React came into the market and gained popularity amongst them. The previous frameworks follow the traditional data flow structure, which uses the DOM (Document Object Model). DOM is an object which is created by the browser each time a web page is loaded. It dynamically adds or removes the data at the back end and when any modifications were done, then each time

a new DOM is created for the same page. This repeated creation of DOM makes unnecessary memory wastage and reduces the performance of the application.

Therefore, a new technology ReactJS framework invented which remove this drawback. ReactJS allows you to divide your entire application into various components. ReactJS still used the same traditional data flow, but it is not directly operating on the browser's Document Object Model (DOM) immediately; instead, it operates on a virtual DOM. It means rather than manipulating the document in a browser after changes to our data, it resolves changes on a DOM built and run entirely in memory. After the virtual DOM has been updated, React determines what changes made to the actual browser's DOM. The React Virtual DOM exists entirely in memory and is a representation of the web browser's

DOM. Due to this, when we write a React component, we did not write directly to the DOM; instead, we are writing virtual components that react will turn into -the DOM.



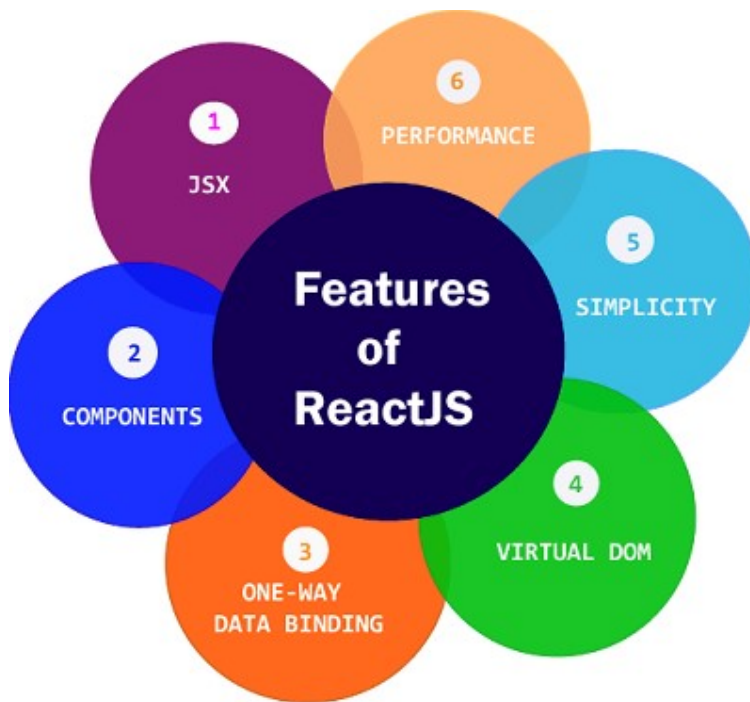
```
function MyComponent({ name }) {  
  return <div>Hello, {name}!</div>;  
}
```

React - Declarative

JS - Imperative

```
function MyComponent(name) {  
  const element = document.createElement('div'); // Create a new div element  
  element.textContent = `Hello, ${name}!`; // Set text content manually  
  return element; // Return the created element  
}
```

React Features



Currently, ReactJS gaining quick popularity as the best JavaScript framework among web developers. It is playing an essential role in the front-end ecosystem. The important features of ReactJS are as following.

- JSX
- Components
- One-way Data Binding
- Virtual DOM
- Simplicity
- Performance

JSX

JSX stands for JavaScript XML. It is a JavaScript syntax extension. Its an XML or HTML like syntax used by ReactJS. This syntax is processed into JavaScript calls of React Framework. It extends the ES6 so that HTML like text can co-exist with JavaScript react code. It is not necessary to use JSX, but it is recommended to use in ReactJS.

Components

ReactJS is all about components. ReactJS application is made up of multiple components, and each component has its own logic and controls. These

components can be reusable which help you to maintain the code when working on larger scale projects.

One-way Data Binding

ReactJS is designed in such a manner that follows unidirectional data flow or one-way data binding. The benefits of one-way data binding give you better control throughout the application. If the data flow is in another direction, then it requires additional features. It is because components are supposed to be immutable and the data within them cannot be changed. Flux is a pattern that helps to keep your data unidirectional. This makes the application more flexible that leads to increase efficiency.

Virtual DOM

A virtual DOM object is a representation of the original DOM object. It works like a one-way data binding. Whenever any modifications happen in the web application, the entire UI is re-rendered in virtual DOM representation. Then it checks the difference between the previous DOM representation and new DOM. Once it has done, the real DOM will update only the things that have actually changed. This makes the application faster, and there is no wastage of memory.

Simplicity

ReactJS uses JSX file which makes the application simple and to code as well as understand. We know that ReactJS is a component-based approach which makes the code reusable as your need. This makes it simple to use and learn.

Performance

ReactJS is known to be a great performer. This feature makes it much better than other frameworks out there today. The reason behind this is that it manages a virtual DOM. The DOM is a cross-platform and programming API which deals with HTML, XML or XHTML. The DOM exists entirely in memory. Due to this, when we create a component, we did not write directly to the DOM. Instead, we are writing virtual components that will turn into the DOM leading to smoother and faster performance.

	AngularJS	ReactJS
Author	Google	Facebook Community
Developer	Misko Hevery	Jordan Walke
Initial Release	October 2010	March 2013
Latest Version	Angular 1.7.8 on 11 March 2019.	React 16.8.6 on 27 March 2019
Language	JavaScript, HTML	JSX
Type	Open Source MVC Framework	Open Source JS Framework
Rendering	Client-Side	Server-Side
Packaging	Weak	Strong
Data-Binding	Bi-directional	Uni-directional
DOM	Regular DOM	Virtual DOM
Testing	Unit and Integration Testing	Unit Testing
App Architecture	MVC	Flux
Dependencies	It manages dependencies automatically.	It requires additional tools to manage dependencies.
Routing	It requires a template or controller to its router configuration, which has to be managed manually.	It doesn't handle routing but has a lot of modules for routing, eg., react-router.
Performance	Slow	Fast, due to virtual DOM.
Best For	It is best for single page applications that update a single view at a time.	It is best for single page applications that update multiple views at a time.

React JSX

All of the React components have a **render** function. The render function specifies the HTML output of a React component. JSX(JavaScript Extension), is a React extension which allows writing JavaScript code that looks like HTML. In other words, JSX is an HTML-like syntax used by React that extends ECMAScript so that **HTML-like** syntax can co-exist with JavaScript/React code. The syntax is used by **preprocessors** (i.e., transpilers like babel) to transform HTML-like syntax into standard JavaScript objects that a JavaScript engine will parse.

JSX provides you to write HTML/XML-like structures (e.g., DOM-like tree structures) in the same file where you write JavaScript code, then preprocessor will transform these expressions into actual JavaScript code. Just like XML/HTML, JSX tags have a tag name, attributes, and children.

```
<div>Hello JavaTpoint</div>
```

Corresponding Output

```
React.createElement("div", null, "Hello JavaTpoint");
```

The above line creates a **react element** and passing **three arguments** inside where the first is the name of the element which is **div**, second is the **attributes** passed in the **div** tag, and last is the **content** you pass which is the "Hello JavaTpoint."

Why use JSX?

- It is faster than regular JavaScript because it performs optimization while translating the code to JavaScript.
- Instead of separating technologies by putting markup and logic in separate files, React uses components that contain both. We will learn components in a further section.
- It is type-safe, and most of the errors can be found at compilation time.
- It makes easier to create templates.

Nested Elements in JSX

To use more than one element, you need to wrap it with one container element. Here, we use **div** as a container element which has **three** nested elements inside it.

App.JSX

```
1. import React, { Component } from 'react';
2. class App extends Component {
3.   render() {
4.     return(
5.       <div>
6.         <h1>JavaScript</h1>
7.         <h2>Training Institutes</h2>
8.         <p>This is the Website.</p>
9.       </div>
10.    );
11.  }
```

```
12.}
13.export default App;
```

O/P:

JSX Attributes

JSX use attributes with the HTML elements same as regular HTML. JSX uses **camelcase** naming convention for attributes rather than standard naming convention of HTML such as a class in HTML becomes **className** in JSX because the class is the reserved keyword in JavaScript. We can also use our own custom attributes in JSX. For custom attributes, we need to use **data-prefix**. In the below example, we have used a custom attribute **data-demoAttribute** as an attribute for the `<p>` tag.

In JSX, we can specify attribute values in two ways:

1. As String Literals: We can specify the values of attributes in double quotes: In JSX, we can specify attribute values in two ways:

1. As String Literals: We can specify the values of attributes in double quotes:

```
1. import React, { Component } from 'react';
2. class App extends Component{
3.   render(){
4.     return(
5.       <div>
6.         <h1>CBIT</h1>
7.         <h2>Engineering Institute</h2>
8.         <p data-demoAttribute = "demo">AIDS Dept.</p>
9.       </div>
10.    );
11.  }
12.}
13.export default App;
```

In JSX, we can specify attribute values in two ways:

1. **As String Literals:** We can specify the values of attributes in double quotes:

```
var element = <h2 className = "firstAttribute">Hello JavaTpoint</h2>;
```

```
1. import React, { Component } from 'react';
2. class App extends Component{
3.   render(){
4.     return(
5.       <div>
6.         <h1 className = "hello" >JavaTpoint</h1>
7.         <p data-
demoAttribute = "demo">This website contains the best CS tutorials.</p>
8.       </div>
9.     );
10.  }
11.}
12.export default App;
```

2. **As Expressions:** We can specify the values of attributes as expressions using curly braces {}:

```
var element = <h2 className = {varName}>Hello JavaTpoint</h2>;
```

```
1. import React, { Component } from 'react';
2. class App extends Component{
3.   render(){
4.     return(
5.       <div>
6.         <h1 className = "hello" >{25+20}</h1>
7.       </div>
8.     );
9.   }
10.}
11.export default App;
```


JSX Comments

JSX allows us to use comments that begin with `/*` and ends with `*/` and wrapping them in curly braces `{}` just like in the case of JSX expressions. Below example shows how to use comments in JSX.

```
import React, { Component } from 'react';
class App extends Component{
  render(){
    return(
      <div>
        <h1 className = "hello" >Hello JavaTpoint</h1>
        { /* This is a comment in JSX */ }
      </div>
    );
  }
}
export default App;
```

JSX Styling

React always recommends to use **inline** styles. To set inline styles, you need to use **camelCase** syntax. React automatically allows appending **px** after the number value on specific elements. The following example shows how to use styling in the element.

```
import React, { Component } from 'react';
class App extends Component{
  render(){
    var myStyle = {
      fontSize: 80,
      fontFamily: 'Courier',
      color: '#003300'
    }
    return (
      <div>
        <h1 style = {myStyle}>www.javatpoint.com</h1>
      </div>
    );
  }
}
```

```
}  
export default App;
```

```
import React, { Component } from 'react';  
class App extends Component{  
  render(){  
    var i = 5;  
    return (  
      <div>  
        <h1>{i == 1 ? 'True!' : 'False!'}</h1>  
      </div>  
    );  
  }  
}  
export default App;
```

Output:

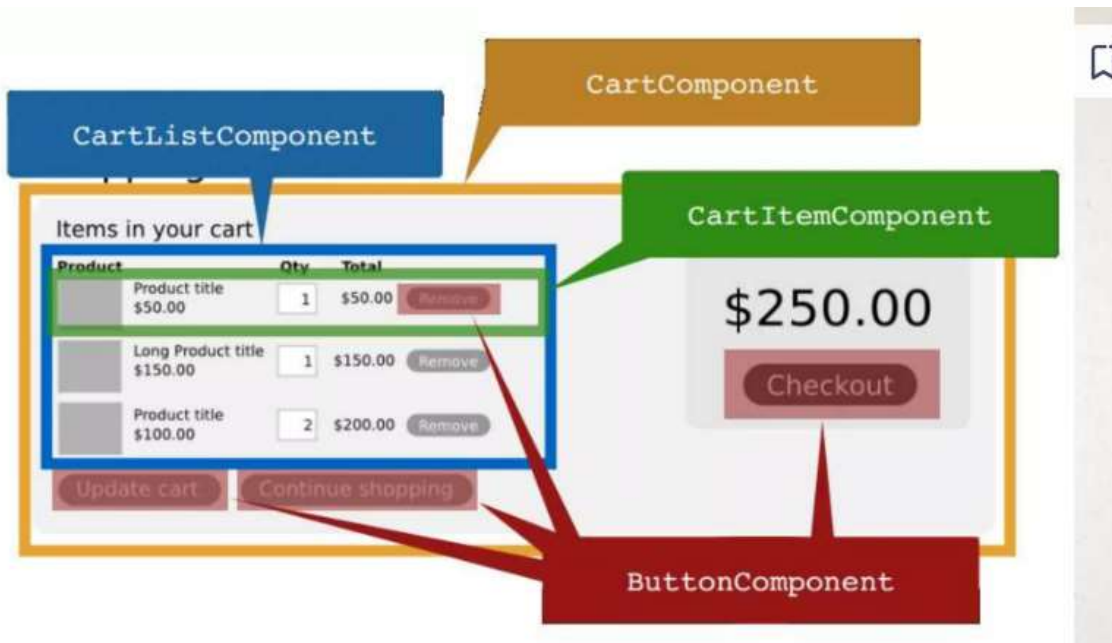
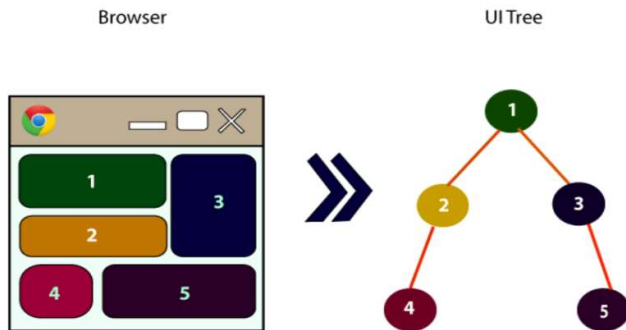
```
False!
```

React Components

Earlier, the developers write more than thousands of lines of code for developing a single page application. These applications follow the traditional DOM structure, and making changes in them was a very challenging task. If any mistake found, it manually searches the entire application and update accordingly. The component-based approach was introduced to overcome an issue. In this approach, the entire application is divided into a small logical group of code, which is known as components.

A Component is considered as the core building blocks of a React application. It makes the task of building UIs much easier. Each component exists in the same space, but they work independently from one another and merge all in a parent component, which will be the final UI of your application.

Every React component have their own structure, methods as well as APIs. They can be reusable as per your need. For better understanding, consider the entire UI as a tree. Here, the root is the starting component, and each of the other pieces becomes branches, which are further divided into sub-branches.



1. Functional Components
2. Class Components

Functional Components

In React, function components are a way to write components that only contain a render method and don't have their own state. They are simply JavaScript functions that may or may not receive data as parameters. We can create a function that takes props(properties) as input and returns what should be rendered. A valid functional component can be shown in the below example.

1. `function WelcomeMessage(props) {`
2. `return <h1>Welcome to the , {props.name}</h1>;`
3. `}`

The functional component is also known as a stateless component because they do not hold or manage state. It can be explained in the below example.

Example

```
1. import React, { Component } from 'react';
2. class App extends React.Component {
3.   render() {
4.     return (
5.       <div>
6.         <First/>
7.         <Second/>
8.       </div>
9.     );
10.  }
11.}
12.class First extends React.Component {
13.  render() {
14.    return (
15.      <div>
16.        <h1>Welcome</h1>
17.      </div>
18.    );
19.  }
20.}
21.class Second extends React.Component {
22.  render() {
23.    return (
24.      <div>
25.        <h2>Full Stack Development</h2>
26.        <p>Welcome To React</p>
27.      </div>
28.    );
29.  }
30.}
31.export default App;
```

Class Components

Class components are more complex than functional components. It requires you to extend from `React.Component` and create a render function which returns a React element. You can pass data from one class to other class components. You can create a class by defining a class that extends `Component` and has a render function. Valid class component is shown in the below example.

```
1. class MyComponent extends React.Component {  
2.   render() {  
3.     return (  
4.       <div>This is main component.</div>  
5.     );  
6.   }  
7. }
```

The class component is also known as a stateful component because they can hold or manage local state. It can be explained in the below example.

Example

In this example, we are creating the list of unordered elements, where we will dynamically insert StudentName for every object from the data array. Here, we are using ES6 arrow syntax (=>) which looks much cleaner than the old JavaScript syntax. It helps us to create our elements with fewer lines of code. It is especially useful when we need to create a list with a lot of items.

```
1. import React, { Component } from 'react';  
2. class App extends React.Component {  
3.   constructor() {  
4.     super();  
5.     this.state = {  
6.       data:  
7.       [  
8.         {  
9.           "name":"Abhishek"  
10.        },  
11.        {  
12.          "name":"Saharsh"  
13.        },  
14.        {  
15.          "name":"Ajay"  
16.        }  
17.      ]  
18.    }  
19.  }  
20.  render() {  
21.    return (  
22.      <div>  
23.        <StudentName/>
```

```

24.     <ul>
25.         {this.state.data.map((item) => <List data = {item} />)}
26.     </ul>
27. </div>
28. );
29. }
30.}
31.class StudentName extends React.Component {
32.  render() {
33.      return (
34.          <div>
35.              <h1>Student Name Detail</h1>
36.          </div>
37.      );
38.  }
39.}
40.class List extends React.Component {
41.  render() {
42.      return (
43.          <ul>
44.              <li>{this.props.data.name}</li>
45.          </ul>
46.      );
47.  }
48.}
49.export default App;

```

React Component Life-Cycle

In ReactJS, every component creation process involves various lifecycle methods. These lifecycle methods are termed as component's lifecycle. These lifecycle methods are not very complicated and called at various points during a component's life. The lifecycle of the component is divided into **four phases**. They are:

1. Initial Phase
2. Mounting Phase (adding nodes to the DOM)
3. Updating Phase (altering existing nodes in the DOM)
4. Unmounting Phase (removing nodes from the DOM)
5. **Error handling** (verifying that your code works and is bug-free)

1. Initial Phase

It is the **birth** phase of the lifecycle of a ReactJS component. Here, the component starts its journey on a way to the DOM. In this phase, a component contains the default Props and initial State. These default properties are done in the constructor of a component. The initial phase only occurs once and consists of the following methods.

- **getDefaultProps()**
It is used to specify the default value of this.props. It is invoked before the creation of the component or any props from the parent is passed into it.
- **getInitialState()**
It is used to specify the default value of this.state. It is invoked before the creation of the component.

2. Mounting Phase

In this phase, the instance of a component is created and inserted into the DOM. It consists of the following methods.

- **componentWillMount()**
This is invoked immediately before a component gets rendered into the DOM. In the case, when you call **setState()** inside this method, the component will not **re-render**.
- **componentDidMount()**
This is invoked immediately after a component gets rendered and placed on the DOM. Now, you can do any DOM querying operations.

render()

This method is defined in each and every component. It is responsible for returning a single root **HTML node** element. If you don't want to render anything, you can return a **null** or **false** value.

3. Updating Phase

It is the next phase of the lifecycle of a react component. Here, we get new **Props** and change **State**. This phase also allows to handle user interaction and provide communication with the components hierarchy. The main aim of this phase is to ensure that the component is displaying the latest version of 'itself'. Unlike the Birth or Death phase, this phase repeats again and again. This phase consists of the following methods.

- **componentWillReceiveProps()**
It is invoked when a component receives new props. If you want to update the state in response to prop changes, you should compare `this.props` and `nextProps` to perform state transition by using **`this.setState()`** method.
- **shouldComponentUpdate()**
It is invoked when a component decides any changes/updation to the DOM. It allows you to control the component's behavior of updating itself. If this method returns true, the component will update. Otherwise, the component will skip the updating.
- **componentWillUpdate()**
It is invoked just before the component updating occurs. Here, you can't change the component state by invoking **`this.setState()`** method. It will not be called, if **`shouldComponentUpdate()`** returns false.
- **render()**
It is invoked to examine **`this.props`** and **`this.state`** and return one of the following types: React elements, Arrays and fragments, Booleans or null, String and Number. If **`shouldComponentUpdate()`** returns false, the code inside **`render()`** will be invoked again to ensure that the component displays itself properly.
- **componentDidUpdate()**
It is invoked immediately after the component updating occurs. In this method, you can put any code inside this which you want to execute once the updating occurs. This method is not invoked for the initial render.

4. Unmounting Phase

It is the final phase of the react component lifecycle. It is called when a component instance is **destroyed** and **unmounted** from the DOM. This phase contains only one method and is given below.

- **componentWillUnmount()**
This method is invoked immediately before a component is destroyed and unmounted permanently. It performs any necessary **cleanup** related task such as invalidating timers, event listener, canceling network requests, or cleaning up DOM elements. If a component instance is unmounted, you cannot mount it again.

Example

1. **import** React, { Component } from 'react';
- 2.
3. **class** App **extends** React.Component {


```
4.   constructor(props) {
5.     super(props);
6.     this.state = {hello: "React"};
7.     this.changeState = this.changeState.bind(this)
8.   }
9.   render() {
10.    return (
11.      <div>
12.        <h1>ReactJS component's Lifecycle</h1>
13.        <h3>Hello {this.state.hello}</h3>
14.        <button onClick = {this.changeState}>Click Here!</button>
15.      </div>
16.    );
17.  }  componentWillMount() {
18.    console.log('Component Will MOUNT!')
19.  }
20.  componentDidMount() {
21.    console.log('Component Did MOUNT!')
22.  }
23.  changeState(){
24.    this.setState( {hello:"All!!- Its a great reactjs tutorial."});
25.  }
26.  componentWillReceiveProps(newProps) {
27.    console.log('Component Will Recieve Props!')
28.  }
29.  shouldComponentUpdate(newProps, newState) {
30.    return true;
31.  }
32.  componentWillUpdate(nextProps, nextState) {
33.    console.log('Component Will UPDATE!');
34.  }
35.  componentDidUpdate(prevProps, prevState) {
36.    console.log('Component Did UPDATE!')
37.  }
38.  componentWillUnmount() {
39.    console.log('Component Will UNMOUNT!')
40.  }
41.}
42.export default App;
```

React State

The state is an updatable structure that is used to contain data or information about the component. The state in a component can change over time. The change in state over time can happen as a response to user action or system event. A component with the state is known as stateful components. It is the heart of the react component which determines the behavior of the component and how it will render. They are also responsible for making a component dynamic and interactive.

A state must be kept as simple as possible. It can be set by using the **setState()** method and calling **setState()** method triggers UI updates. A state represents the component's local state or information. It can only be accessed or modified inside the component or by the component directly. To set an initial state before any interaction occurs, we need to use the **getInitialState()** method.

For example, if we have five components that need data or information from the state, then we need to create one container component that will keep the state for all of them.

Defining State

To define a state, you have to first declare a default set of values for defining the component's initial state. To do this, add a class constructor which assigns an initial state using **this.state**. The **'this.state'** property can be rendered inside **render()** method.

Example

The below sample code shows how we can create a stateful component using ES6 syntax.

```
1. import React, { Component } from 'react';
2. class App extends React.Component {
3.   constructor() {
4.     super();
5.     this.state = { displayBio: true };
6.   }
7.   render() {
8.     const bio = this.state.displayBio ? (
9.       <div>
```

```

10.         <p><h3>CBIT is one of the best Java training institute in Hyderabad
    . We have a team of experienced Java developers and trainers from multinational
    l companies to teach our campus students.</h3></p>
11.     </div>
12.     ) : null;
13.     return (
14.         <div>
15.             <h1> Welcome to ReactJs!! </h1>
16.             { bio }
17.         </div>
18.     );
19. }
20.}
21.export default App;

```

To set the state, it is required to call the `super()` method in the constructor. It is because `this.state` is uninitialized before the `super()` method has been called.

Changing the State

We can change the component state by using the `setState()` method and passing a new state object as the argument. Now, create a new method `toggleDisplayBio()` in the above example and bind `this` keyword to the `toggleDisplayBio()` method otherwise we can't access `this` inside `toggleDisplayBio()` method.

```
1. this.toggleDisplayBio = this.toggleDisplayBio.bind(this);
```

Example

In this example, we are going to add a **button** to the **render()** method. Clicking on this button triggers the `toggleDisplayBio()` method which displays the desired output.

```

1. import React, { Component } from 'react';
2. class App extends React.Component {
3.   constructor() {
4.     super();
5.     this.state = { displayBio: false };
6.     console.log('Component this', this);
7.     this.toggleDisplayBio = this.toggleDisplayBio.bind(this);
8.   }

```

```

9.     toggleDisplayBio(){
10.         this.setState({displayBio: !this.state.displayBio});
11.     }
12.     render() {
13.         return (
14.             <div>
15.                 <h1>Welcome to JavaTpoint!!</h1>
16.                 {
17.                     this.state.displayBio ? (
18.                         <div>
19.                             <p><h4> CBIT is one of the best Java training institute in
Hyderabad. We have a team of experienced Java developers and trainers from
multinational companies to teach our campus students.</h4></p>
20.                             <button onClick={ this.toggleDisplayBio }> Show Less </button>
21.                         </div>
22.                     ) : (
23.                         <div>
24.                             <button onClick={ this.toggleDisplayBio }> Read More </
button>
25.                         </div>
26.                     )
27.                 }
28.             </div>
29.         )
30.     }
31. }
32. export default App;

```

When you click the Read More button, you will get the below output, and when you click the Show Less button, you will get the output as shown in the above image.

React Props

Props stand for "**Properties**." They are **read-only** components. It is an object which stores the value of attributes of a tag and work similar to the HTML attributes. It gives a way to pass data from one component to other components.

It is similar to function arguments. Props are passed to the component in the same way as arguments passed in a function.

Props are **immutable** so we cannot modify the props from inside the component. Inside the components, we can add attributes called props. These attributes are available in the component as **this.props** and can be used to render dynamic data in our render method.

When you need immutable data in the component, you have to add props to **ReactDOM.render()** method in the **main.js** file of your ReactJS project and used it inside the component in which you need. It can be explained in the below example.

Example

App.js

```
1. import React, { Component } from 'react';
2. class App extends React.Component {
3.   render() {
4.     return (
5.       <div>
6.         <h1> Welcome to { this.props.name } </h1>
7.         <p> <h4> CBIT is one of the best Java training institute in Hyderabad.
           We have a team of experienced Java developers and trainers from multinational
           companies to teach our campus students. </h4> </p>
8.       </div>
9.     );
10.  }
11.}
12.export default App;
```

Main.js

```
1. import React from 'react';
2. import ReactDOM from 'react-dom';
3. import App from './App.js';
4.
5. ReactDOM.render(<App name = "JavaTpoint!!" />, document.getElementById(
  'root'));
```

Default Props

It is not necessary to always add props in the `ReactDOM.render()` element. You can also set **default** props directly on the component constructor. It can be explained in the below example.

Example

App.js

```
1. import React, { Component } from 'react';
2. class App extends React.Component {
3.   render() {
4.     return (
5.       <div>
6.         <h1>Default Props Example</h1>
7.         <h3>Welcome to {this.props.name}</h3>
8.         <p> CBIT is one of the best Java training institute in Hyderabad. We
           have a team of experienced Java developers and trainers from multinational
           companies to teach our campus students.</p>
9.       </div>
10.    );
11.  }
12.}
13.App.defaultProps = {
14.  name: "CBIT"
15.}
16.export default App;
```

Main.js

```
1. import React from 'react';
2. import ReactDOM from 'react-dom';
3. import App from './App.js';
4.
5. ReactDOM.render(<App/>, document.getElementById('app'));
```

Output

State and Props

It is possible to combine both state and props in your app. You can set the state in the parent component and pass it in the child component using props. It can be shown in the below example.

Example

App.js

```
1. import React, { Component } from 'react';
2. class App extends React.Component {
3.   constructor(props) {
4.     super(props);
5.     this.state = {
6.       name: "JavaTpoint",
7.     }
8.   }
9.   render() {
10.    return (
11.      <div>
12.        <JTP jtpProp = {this.state.name}/>
13.      </div>
14.    );
15.  }
16.}
17.class JTP extends React.Component {
18.  render() {
19.    return (
20.      <div>
21.        <h1>State & Props Example</h1>
22.        <h3>Welcome to {this.props.jtpProp}</h3>
23.        <p>Javatpoint is one of the best Java training institute in Noida, Delhi,
        Gurugram, Ghaziabad and Faridabad.</p>
24.      </div>
25.    );
26.  }
27.}
28.export default App;
```

Main.js

```
1. import React from 'react';
2. import ReactDOM from 'react-dom';
```

3. `import App from './App.js';`
- 4.
5. `ReactDOM.render(<App/>, document.getElementById('app'));`

Output:

React Fragments

In React, whenever you want to render something on the screen, you need to use a render method inside the component. This render method can return **single** elements or **multiple** elements. The render method will only render a single root node inside it at a time. However, if you want to return multiple elements, the render method will require a '**div**' tag and put the entire content or elements inside it. This extra node to the DOM sometimes results in the wrong formatting of your HTML output and also not loved by the many developers.

Example

1. `// Rendering with div tag`
2. `class App extends React.Component {`
3. `render() {`
4. `return (`
5. `//Extraneous div element`
6. `<div>`
7. `<h2> Hello World! </h2>`
8. `<p> Welcome to the JavaTpoint. </p>`
9. `</div>`
10. `);`
11. `}`
12. `}`

To solve this problem, React introduced **Fragments** from the **16.2** and above version. Fragments allow you to group a list of children without adding extra nodes to the DOM.

Syntax

1. `<React.Fragment>`
2. `<h2> child1 </h2>`
3. `<p> child2 </p>`
4. `..`

5. `</React.Fragment>`

Example

```
1. // Rendering with fragments tag
2. class App extends React.Component {
3.   render() {
4.     return (
5.       <React.Fragment>
6.         <h2> Hello World! </h2>
7.         <p> Welcome to the JavaTpoint. </p>
8.       </React.Fragment>
9.     );
10.  }
11.}
```

Why we use Fragments?

The main reason to use Fragments tag is:

1. It makes the execution of code faster as compared to the div tag.
2. It takes less memory.

Fragments Short Syntax

There is also another shorthand exists for declaring fragments for the above method. It looks like **empty** tag in which we can use of '`<>`' and '' instead of the '`React.Fragment`'.

Example

```
1. //Rendering with short syntax
2. class Columns extends React.Component {
3.   render() {
4.     return (
5.       <>
6.         <h2> Hello World! </h2>
7.         <p> Welcome to the JavaTpoint </p>
8.       </>
9.     );
10.  }
11.}
```

Keyed Fragments

The shorthand syntax does not accept key attributes. You need a key for mapping a collection to an array of fragments such as to create a description list. If you need to provide keys, you have to declare the fragments with the explicit `<React.Fragment>` syntax.

Example

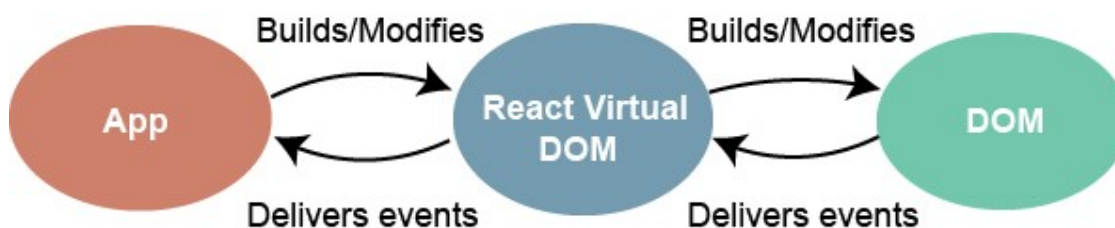
```
1. Function = (props) {  
2.   return (  
3.     <Fragment>  
4.       {props.items.data.map(item => (  
5.         // Without the 'key', React will give a key warning  
6.         <React.Fragment key={item.id}>  
7.           <h2>{item.name}</h2>  
8.           <p>{item.url}</p>  
9.           <p>{item.description}</p>  
10.        </React.Fragment>  
11.      )})  
12.    </Fragment>  
13.  )  
14.}
```

React Events

An event is an action that could be triggered as a result of the user action or system generated event. For example, a mouse click, loading of a web page, pressing a key, window resizes, and other interactions are called events.

React has its own event handling system which is very similar to handling events on DOM elements. The react event handling system is known as Synthetic Events. The synthetic event is a cross-browser wrapper of the browser's native event.

Events Handler



Handling events with react have some syntactic differences from handling events on DOM. These are:

1. React events are named as **camelCase** instead of **lowercase**.
2. With JSX, a function is passed as the **event handler** instead of a **string**.
For example:

Event declaration in plain HTML:

1. `<button onclick="showMessage()">`
2. Hello CBIT
3. `</button>`

Event declaration in React:

1. `<button onClick={showMessage}>`
2. Hello CBIT
3. `</button>`

3. In react, we cannot return **false** to prevent the **default** behavior. We must call **preventDefault** event explicitly to prevent the default behavior. For example:

In plain HTML, to prevent the default link behavior of opening a new page, we can write:

1. ``
2. Click_Me
3. ``

In React, we can write it as:

1. `function ActionLink() {`

```

2.   function handleClick(e) {
3.       e.preventDefault();
4.       console.log('You had clicked a Link.');
```

```

5.   }
6.   return (
7.       <a href="#" onClick={handleClick}>
8.           Click_Me
9.       </a>
10.  );
11.}
```

In the above example, e is a **Synthetic Event** which defines according to the **W3C** spec.

Now let us see how to use Event in React.

Example

In the below example, we have used only one component and adding an onChange event. This event will trigger the **changeText** function, which returns the company name.

```

1. import React, { Component } from 'react';
2. class App extends React.Component {
3.   constructor(props) {
4.     super(props);
5.     this.state = {
6.       companyName: "
7.     };
8.   }
9.   changeText(event) {
10.    this.setState({
11.      companyName: event.target.value
12.    });
13.  }
14.  render() {
15.    return (
16.      <div>
17.        <h2>Simple Event Example</h2>
18.        <label htmlFor="name">Enter company name: </label>
19.        <input type="text" id="companyName" onChange={this.changeText.
bind(this)} />
20.        <h4>You entered: { this.state.companyName } </h4>
21.      </div>
```

```
22.    );  
23.  }  
24.}  
25.export default App;
```

Output

When you execute the above code, you will get the following output.

Output

When you execute the above code, you will get the following output.



A screenshot of a web browser window at localhost:8080. The page title is "Simple Event Example". It contains a form with a label "Enter company name:" followed by a text input field. Below the input field, it says "You entered:".

After entering the name in the textbox, you will get the output as like below screen.



A screenshot of the same web browser window. The text input field now contains "www.javatpoint.com". The output below the input field now reads "You entered: www.javatpoint.com".

React Router

Routing is a process in which a user is directed to different pages based on their action or request. ReactJS Router is mainly used for developing Single Page Web Applications. React Router is used to define multiple routes in the application. When a user types a specific URL into the browser, and if this URL path matches any 'route' inside the router file, the user will be redirected to that particular route.

React Router is a standard library system built on top of the React and used to create routing in the React application using React Router Package. It provides the synchronous URL on the browser with data that will be displayed on the web page. It maintains the standard structure and behavior of the application and mainly used for developing single page web applications.

Need of React Router

React Router plays an important role to display multiple views in a single page application. Without React Router, it is not possible to display multiple views in React applications. Most of the social media websites like Facebook, Instagram uses React Router for rendering multiple views.

React Router Installation

React contains three different packages for routing. These are:

1. **react-router:** It provides the core routing components and functions for the React Router applications.
2. **react-router-native:** It is used for mobile applications.
3. **react-router-dom:** It is used for web applications design.

It is not possible to install react-router directly in your application. To use react routing, first, you need to install react-router-dom modules in your application. The below command is used to install react router dom.

1. `$ npm install react-router-dom --save`

Components in React Router

There are two types of router components:

- **<BrowserRouter>:** It is used for handling the dynamic URL.
- **<HashRouter>:** It is used for handling the static request.

Example

Step-1: In our project, we will create two more components along with **App.js**, which is already present.

About.js

1. **import** React from 'react'
2. **class** About **extends** React.Component {
3. render() {
4. **return** <h1>About</h1>
5. }
6. }
7. **export default** About

Contact.js

1. **import** React from 'react'
2. **class** Contact **extends** React.Component {
3. render() {
4. **return** <h1>Contact</h1>
5. }

6. }
7. export **default** Contact

App.js

1. **import** React from 'react'
2. **class** App **extends** React.Component {
3. render() {
4. **return** (
5. <div>
6. <h1>Home</h1>
7. </div>
8.)
9. }
10. }
11. export **default** App

Step-2: For Routing, open the index.js file and import all the three component files in it. Here, you need to import line: **import { Route, Link, BrowserRouter as Router } from 'react-router-dom'** which helps us to implement the Routing. Now, our index.js file looks like below.

What is Route?

It is used to define and render component based on the specified path. It will accept components and render to define what should be rendered.

Index.js

1. **import** React from 'react';
2. **import** ReactDOM from 'react-dom';
3. **import** { Route, Link, BrowserRouter as Router } from 'react-router-dom'
4. **import** './index.css';
5. **import** App from './App';
6. **import** About from './about'
7. **import** Contact from './contact'
- 8.
9. **const** routing = (
10. <Router>
11. <div>
12. <h1>React Router Example</h1>
13. <Route path="/" component={App} />
14. <Route path="/about" component={About} />
15. <Route path="/contact" component={Contact} />

16. `</div>`
17. `</Router>`
- 18.)
19. `ReactDOM.render(routing, document.getElementById('root'));`

Step-3: Open **command prompt**, go to your project location, and then type **npm start**. You will get the following screen.



Now, if you enter **manually** in the browser: **localhost:3000/about**, you will see **About** component is rendered on the screen.



Step-4: In the above screen, you can see that **Home** component is still rendered. It is because the home path is '/' and about path is '/about', so you can observe that **slash** is common in both paths which render both components. To stop this behavior, you need to use the **exact** prop. It can be seen in the below example.

Index.js

1. **import** React from 'react';
2. **import** ReactDOM from 'react-dom';
3. **import** { Route, Link, BrowserRouter as Router } from 'react-router-dom'
4. **import** './index.css';
5. **import** App from './App';
6. **import** About from './about'


```
7. import Contact from './contact'
8.
9. const routing = (
10. <Router>
11.   <div>
12.     <h1>React Router Example</h1>
13.     <Route exact path="/" component={App} />
14.     <Route path="/about" component={About} />
15.     <Route path="/contact" component={Contact} />
16.   </div>
17. </Router>
18.)
19.ReactDOM.render(routing, document.getElementById('root'));
```

Output



Adding Navigation using Link component

Sometimes, we want to need **multiple** links on a single page. When we click on any of that particular **Link**, it should load that page which is associated with that path without **reloading** the web page. To do this, we need to import **<Link>** component in the **index.js** file.

What is < Link> component?

This component is used to create links which allow to **navigate** on different **URLs** and render its content without reloading the webpage.

ADVERTISEMENT

Example

Index.js

```
1. import React from 'react';
```

```
2. import ReactDOM from 'react-dom';
3. import { Route, Link, BrowserRouter as Router } from 'react-router-dom'
4. import './index.css';
5. import App from './App';
6. import About from './about'
7. import Contact from './contact'
8.
9. const routing = (
10. <Router>
11.   <div>
12.     <h1>React Router Example</h1>
13.     <ul>
14.       <li>
15.         <Link to="/">Home</Link>
16.       </li>
17.       <li>
18.         <Link to="/about">About</Link>
19.       </li>
20.       <li>
21.         <Link to="/contact">Contact</Link>
22.       </li>
23.     </ul>
24.     <Route exact path="/" component={App} />
25.     <Route path="/about" component={About} />
26.     <Route path="/contact" component={Contact} />
27.   </div>
28. </Router>
29.)
30.ReactDOM.render(routing, document.getElementById('root'));
```

Output



After adding Link, you can see that the routes are rendered on the screen. Now, if you click on the **About**, you will see URL is changing and About component is rendered.



React Router Switch

The `<Switch>` component is used to render components only when the path will be **matched**. Otherwise, it returns to the **not found** component.

To understand this, first, we need to create a **notfound** component.

notfound.js

1. **import** React from 'react'
2. **const** Notfound = () => <h1>Not found</h1>
3. **export default** Notfound

Now, import component in the index.js file. It can be seen in the below code.

Index.js

1. **import** React from 'react';
2. **import** ReactDOM from 'react-dom';
3. **import** { BrowserRouter as Router, Route, Link, NavLink, Switch } from 'react-router-dom'
4. **import** './index.css';
5. **import** App from './App';
6. **import** About from './about'
7. **import** Contact from './contact'
8. **import** Notfound from './notfound'
- 9.
10. **const** routing = (
11. <Router>
12. <div>
13. <h1>React Router Example</h1>
14.
15.
16. <NavLink to="/" exact activeClassName={
17. {color:'red'}
18. }>Home</NavLink>
19.

```

20.   <li>
21.     <NavLink to="/about" exact activeStyle={
22.       {color:'green'}
23.     }>About</NavLink>
24.   </li>
25.   <li>
26.     <NavLink to="/contact" exact activeStyle={
27.       {color:'magenta'}
28.     }>Contact</NavLink>
29.   </li>
30. </ul>
31. <Switch>
32.   <Route exact path="/" component={App} />
33.   <Route path="/about" component={About} />
34.   <Route path="/contact" component={Contact} />
35.   <Route component={NotFound} />
36. </Switch>
37. </div>
38. </Router>
39.)
40.ReactDOM.render(routing, document.getElementById('root'));

```

Output

If we manually enter the wrong path, it will give the not found error.



React Router <Redirect>

A <Redirect> component is used to redirect to another route in our application to maintain the old URLs. It can be placed anywhere in the route hierarchy.

Nested Routing in React

Nested routing allows you to render **sub-routes** in your application. It can be understood in the below example.

Example

index.js

```
1. import React from 'react';
2. import ReactDOM from 'react-dom';
3. import { BrowserRouter as Router, Route, Link, NavLink, Switch } from 'react-
  -router-dom'
4. import './index.css';
5. import App from './App';
6. import About from './about'
7. import Contact from './contact'
8. import NotFound from './notfound'
9.
10. const routing = (
11.   <Router>
12.     <div>
13.       <h1>React Router Example</h1>
14.       <ul>
15.         <li>
16.           <NavLink to="/" exact activeStyle={
17.             {color:'red'}
18.           }>Home</NavLink>
19.         </li>
20.         <li>
21.           <NavLink to="/about" exact activeStyle={
22.             {color:'green'}
23.           }>About</NavLink>
24.         </li>
25.         <li>
26.           <NavLink to="/contact" exact activeStyle={
27.             {color:'magenta'}
28.           }>Contact</NavLink>
29.         </li>
30.       </ul>
31.       <Switch>
32.         <Route exact path="/" component={App} />
33.         <Route path="/about" component={About} />
34.         <Route path="/contact" component={Contact} />
35.         <Route component={NotFound} />
36.       </Switch>
37.     </div>
38.   </Router>
39.)
40. ReactDOM.render(routing, document.getElementById('root'));
```

In the **contact.js** file, we need to import the **React Router** component to implement the **subroutes**.

contact.js

```
1. import React from 'react'
2. import { Route, Link } from 'react-router-dom'
3.
4. const Contacts = ({ match }) => <p>{match.params.id}</p>
5.
6. class Contact extends React.Component {
7.   render() {
8.     const { url } = this.props.match
9.     return (
10.      <div>
11.        <h1>Welcome to Contact Page</h1>
12.        <strong>Select contact Id</strong>
13.        <ul>
14.          <li>
15.            <Link to="/contact/1">Contacts 1 </Link>
16.          </li>
17.          <li>
18.            <Link to="/contact/2">Contacts 2 </Link>
19.          </li>
20.          <li>
21.            <Link to="/contact/3">Contacts 3 </Link>
22.          </li>
23.          <li>
24.            <Link to="/contact/4">Contacts 4 </Link>
25.          </li>
26.        </ul>
27.        <Route path="/contact/:id" component={Contacts} />
28.      </div>
29.    )
30.  }
31.}
32.export default Contact
```

Output

When we execute the above program, we will get the following output.



After clicking the **Contact** link, we will get the contact list. Now, selecting any contact, we will get the corresponding output. It can be shown in the below example.



Benefits Of React Router

The benefits of React Router is given below:

- In this, it is not necessary to set the browser history manually.
- Link uses to navigate the internal links in the application. It is similar to the anchor tag.

- It uses Switch feature for rendering.
- The Router needs only a Single Child element.
- In this, every component is specified in .

React Forms

Forms are an integral part of any modern web application. It allows the users to interact with the application as well as gather information from the users. Forms can perform many tasks that depend on the nature of your business requirements and logic such as authentication of the user, adding user, searching, filtering, booking, ordering, etc. A form can contain text fields, buttons, checkbox, radio button, etc.

Creating Form

React offers a stateful, reactive approach to build a form. The component rather than the DOM usually handles the React form. In React, the form is usually implemented by using controlled components.

There are mainly two types of form input in React.

1. Uncontrolled component
2. Controlled component

Uncontrolled component

The uncontrolled input is similar to the traditional HTML form inputs. The DOM itself handles the form data. Here, the HTML elements maintain their own state that will be updated when the input value changes. To write an uncontrolled component, you need to use a ref to get form values from the DOM. In other words, there is no need to write an event handler for every state update. You can use a ref to access the input field value of the form from the DOM.

Example

In this example, the code accepts a field **username** and **company name** in an uncontrolled component.

1. `import React, { Component } from 'react';`
2. `class App extends React.Component {`
3. `constructor(props) {`
4. `super(props);`
5. `this.updateSubmit = this.updateSubmit.bind(this);`


```
6.   this.input = React.createRef();
7.   }
8.   updateSubmit(event) {
9.     alert('You have entered the UserName and CompanyName successfully.');
```

10. event.preventDefault();

11. }

12. render() {

13. return (

14. <form onSubmit={this.updateSubmit}>

15. <h1>Uncontrolled Form Example</h1>

16. <label>Name:

17. <input type="text" ref={this.input} />

18. </label>

19. <label>

20. CompanyName:

21. <input type="text" ref={this.input} />

22. </label>

23. <input type="submit" value="Submit" />

24. </form>

25.);

26. }

27.}

28.export default App;

Output

When you execute the above code, you will see the following screen.



The screenshot shows a web browser window with the address bar set to 'localhost:8080'. The page displays a heading 'Uncontrolled Form Example'. Below the heading, there is a form consisting of two text input fields. The first field is preceded by the label 'Name:' and the second by 'CompanyName:'. To the right of these fields is a button labeled 'Submit'.

After filling the data in the field, you get the message that can be seen in the below screen.



localhost:8080 says
You have entered the UserName an successfully.

Controlled Component

In HTML, form elements typically maintain their own state and update it according to the user input. In the controlled component, the input form element is handled by the component rather than the DOM. Here, the mutable state is kept in the state property and will be updated only with **setState()** method.

Controlled components have functions that govern the data passing into them on every **onChange event**, rather than grabbing the data only once, e.g., when you click a **submit button**. This data is then saved to state and updated with `setState()` method. This makes component have better control over the form elements and data.

A controlled component takes its current value through **props** and notifies the changes through **callbacks** like an `onChange` event. A parent component "controls" this changes by handling the callback and managing its own state and then passing the new values as props to the controlled component. It is also called as a "dumb component."

Example

1. **import** React, { Component } from 'react';
2. **class** App **extends** React.Component {
3. constructor(props) {
4. **super**(props);
5. **this**.state = {value: ""};
6. **this**.handleChange = **this**.handleChange.bind(**this**);
7. **this**.handleSubmit = **this**.handleSubmit.bind(**this**);
8. }
9. handleChange(event) {
10. **this**.setState({value: event.target.value});
11. }
12. handleSubmit(event) {

```
13.   alert('You have submitted the input successfully: ' + this.state.value);
14.   event.preventDefault();
15. }
16. render() {
17.   return (
18.     <form onSubmit={this.handleSubmit}>
19.       <h1>Controlled Form Example</h1>
20.       <label>
21.         Name:
22.         <input type="text" value={this.state.value} onChange={this.handleC
hange} />
23.       </label>
24.       <input type="submit" value="Submit" />
25.     </form>
26.   );
27. }
28.}
29.export default App;
```

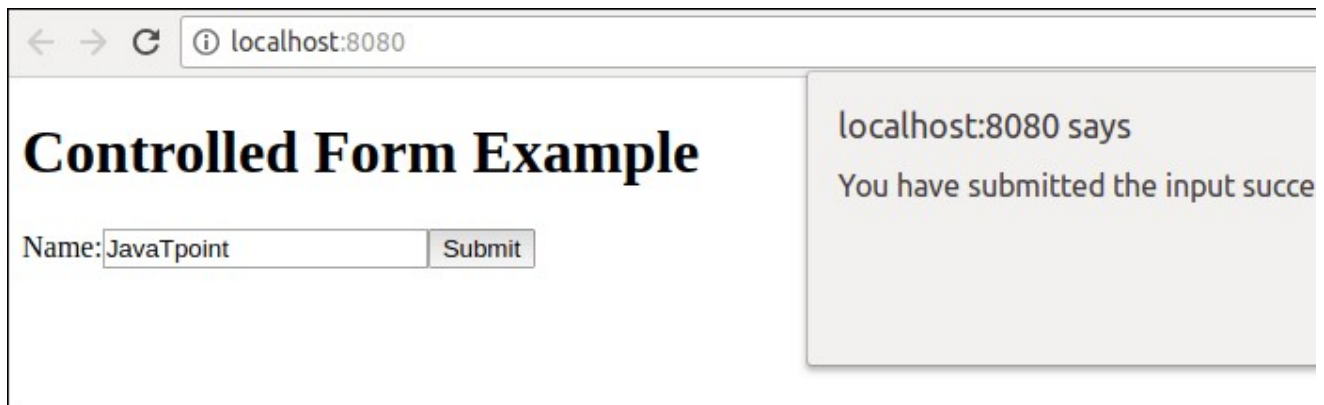
Output

When you execute the above code, you will see the following screen.



The screenshot shows a web browser window with the address bar displaying 'localhost:8080'. The main content area features a heading 'Controlled Form Example' in a large, bold, black serif font. Below the heading, there is a form with the label 'Name:' followed by a text input field and a 'Submit' button. The input field is currently empty.

After filling the data in the field, you get the message that can be seen in the below screen.



Handling Multiple Inputs in Controlled Component

If you want to handle multiple controlled input elements, add a **name** attribute to each element, and then the handler function decided what to do based on the value of **event.target.name**.

Example

```
1. import React, { Component } from 'react';
2. class App extends React.Component {
3.   constructor(props) {
4.     super(props);
5.     this.state = {
6.       personGoing: true,
7.       numberOfPersons: 5
8.     };
9.     this.handleChange = this.handleChange.bind(this);
10.  }
11.  handleChange(event) {
12.    const target = event.target;
13.    const value = target.type === 'checkbox' ? target.checked : target.value;
14.    const name = target.name;
15.    this.setState({
16.      [name]: value
17.    });
18.  }
19.  render() {
20.    return (
21.      <form>
22.        <h1>Multiple Input Controlled Form Example</h1>
23.        <label>
24.          Is Person going:
```

```
25.     <input
26.       name="personGoing"
27.       type="checkbox"
28.       checked={this.state.personGoing}
29.       onChange={this.handleChange} />
30.   </label>
31.   <br />
32.   <label>
33.     Number of persons:
34.     <input
35.       name="numberOfPersons"
36.       type="number"
37.       value={this.state.numberOfPersons}
38.       onChange={this.handleChange} />
39.   </label>
40. </form>
41. );
42. }
43. }
44. export default App;
```

Output



Multiple Input Controlled Form Example

Is Person going: ☒

Number of persons:

React-Paginate

The Reactjs pagination library will be used directly for the pagination functionality of any list of items. The props required here are an array of list items to render and an onChange callback function, which informs the parent component about the page change.

But pagination is important by filtering and displaying only relevant data, for example, Google search engine.

Therefore, Reactjs pagination becomes crucial when users search for certain information and do not consume random information.

Some built-in libraries are used to handle pagination in web development, in the case of React. You directly use many resources to handle pagination in the application.

Some of The NPM And Other Pagination Libraries



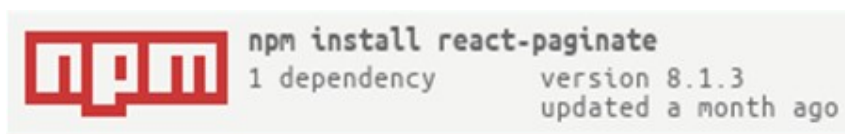
- **react-js-paginate**
- **react-paginate**
- **react-boot/paginate**

Instead of rewriting your own, it's ideal to use these existing libraries, as there's no need to reinvent the wheel; instead, you can focus on other things.

But at the same time, it is very important to know what is going on behind the scenes to build and customize the app to your needs without compromising any requirements.

There is also a possibility that different logic is used for different reactjs pagination packages. To implement react pagination in your app, read this excellent react-jw-pagination article.

Here we talk only about react-paginate (the second one):



A ReactJS component to render the pagination.

By installing the component and writing just a bit of CSS, you can get this:
Note: You must write your CSS to get this UI. This package does not provide any CSS.



React Table

A table is an arrangement which organizes information into rows and columns. It is used to store and display data in a structured format.

The react-table is a lightweight, fast, fully customizable (JSX, templates, state, styles, callbacks), and extendable Datagrid built for React. It is fully controllable via optional props and callbacks.

Features

1. It is lightweight at 11kb (and only need 2kb more for styles).
2. It is fully customizable (JSX, templates, state, styles, callbacks).
3. It is fully controllable via optional props and callbacks.
4. It has client-side & Server-side pagination.
5. It has filters.
6. Pivoting & Aggregation
7. Minimal design & easily themeable

Installation

Let us create a **React app** using the following command.

1. `$ npx create-react-app myreactapp`

Next, we need to install **react-table**. We can install react-table via npm command, which is given below.

1. `$ npm install react-table`

Once, we have installed react-table, we need to **import** the react-table into the react component. To do this, open the **src/App.js** file and add the following snippet.

1. **import** ReactTable from `"react-table"`;

Let us assume we have data which needs to be rendered using react-table.

```
1. const data = [{
2.   name: 'Ayaan',
3.   age: 26
4. }, {
5.   name: 'Ahana',
6.   age: 22
7. }, {
8.   name: 'Peter',
9.   age: 40
10. }, {
11.  name: 'Virat',
12.  age: 30
13. }, {
14.  name: 'Rohit',
15.  age: 32
16. }, {
17.  name: 'Dhoni',
18.  age: 37
19. }]
```

Along with data, we also need to specify the **column info** with **column attributes**.

```
1. const columns = [{
2.   Header: 'Name',
3.   accessor: 'name'
4. }, {
5.   Header: 'Age',
6.   accessor: 'age'
7. }]
```

Inside the render method, we need to bind this data with react-table and then returns the react-table.

```
1. return (
2.   <div>
3.     <ReactTable
4.       data={data}
5.       columns={columns}
6.       defaultPageSize = {2}
7.       pageSizeOptions = {[2,4, 6]}
8.     />
```


9. </div>
10.)

Now, our **src/App.js** file looks like as below.

```
1. import React, { Component } from 'react';
2. import ReactTable from "react-table";
3. import "react-table/react-table.css";
4.
5. class App extends Component {
6.   render() {
7.     const data = [{
8.       name: 'Ayaan',
9.       age: 26
10.    }, {
11.      name: 'Ahana',
12.      age: 22
13.    }, {
14.      name: 'Peter',
15.      age: 40
16.    }, {
17.      name: 'Virat',
18.      age: 30
19.    }, {
20.      name: 'Rohit',
21.      age: 32
22.    }, {
23.      name: 'Dhoni',
24.      age: 37
25.    }]
26.    const columns = [{
27.      Header: 'Name',
28.      accessor: 'name'
29.    }, {
30.      Header: 'Age',
31.      accessor: 'age'
32.    }]
33.    return (
34.      <div>
35.        <ReactTable
36.          data={data}
37.          columns={columns}
38.          defaultPageSize = {2}
```

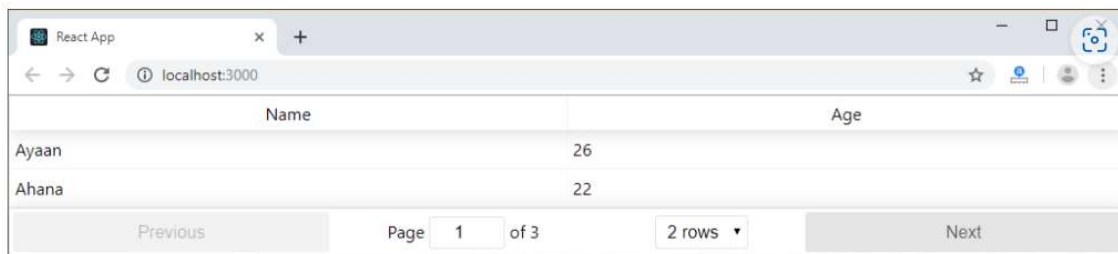
```

39.         pageSizeOptions = {[2,4, 6]}
40.     />
41. </div>
42. )
43. }
44.}
45.export default App;

```

Output

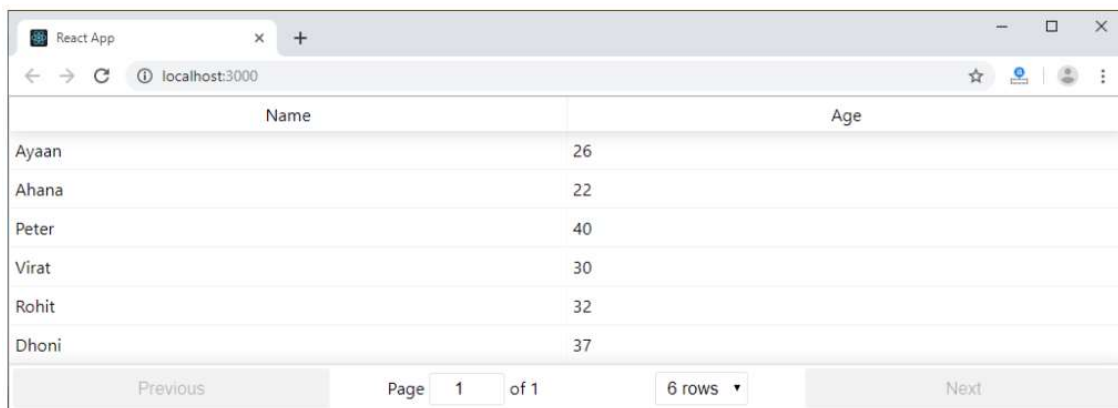
When we execute the React app, we will get the output as below.



Name	Age
Ayaan	26
Ahana	22

Previous Page 1 of 3 2 rows Next

Now, change the rows dropdown menu, we will get the output as below.



Name	Age
Ayaan	26
Ahana	22
Peter	40
Virat	30
Rohit	32
Dhoni	37

Previous Page 1 of 1 6 rows Next

React Portals

The **React 16.0** version introduced React portals in **September 2017**. A React portal provides a way to render an element outside of its component hierarchy, i.e., in a separate component.

Before React 16.0 version, it is very tricky to render the child component outside of its parent component hierarchy. If we do this, it breaks the convention where a component needs to render as a new element and follow a **parent-child** hierarchy. In React, the parent component always wants to go where its child component goes. That's why React portal concept comes in.

Syntax

1. ReactDOM.createPortal(child, container)

Here, the first argument (child) is the component, which can be an element, string, or fragment, and the second argument (container) is a DOM element.

Example before React v16

Generally, when you want to return an element from a component's render method, it is mounted as a new div into the DOM and render the children of the closest parent component.

```
1. render() {  
2.   // React mounts a new div into the DOM and renders the children into it  
3.   return (  
4.     <div>  
5.       {this.props.children}  
6.     </div>  
7.   );  
8. }
```

Example using portal

But, sometimes we want to insert a child component into a different location in the DOM. It means React does not want to create a new div. We can do this by creating React portal.

```
1. render() {  
2.   return ReactDOM.createPortal(  
3.     this.props.children,  
4.     myNode,  
5.   );  
6. }
```

Features

- It uses React version 16 and its official API for creating portals.
- It has a fallback for React version 15.
- It transports its children component into a new React portal which is appended by default to document.body.
- It can also target user specified DOM element.
- It supports server-side rendering
- It supports returning arrays (no wrapper div's needed)

- It uses `<Portal />` and `<PortalWithState />` so there is no compromise between flexibility and convenience.
- It doesn't produce any DOM mess.
- It has no dependencies, minimalistic.

When to use?

The common use-cases of React portal include:

- Modals
- Tooltips
- Floating menus
- Widgets

Installation

We can install React portal using the following command.

1. `$ npm install react-portal --save`

Explanation of React Portal

Create a new React project using the following command.

1. `$ npx create-react-app reactapp`

Open the App.js file and insert the following code snippet.

App.js

ADVERTISEMENT

1. `import React, {Component} from 'react';`
2. `import './App.css'`
3. `import PortalDemo from './PortalDemo.js';`
- 4.
5. `class App extends Component {`
6. `render () {`
7. `return (`
8. `<div className='App'>`
9. `<PortalDemo />`
10. `</div>`
11. `);`
12. `}`

```
13.}
14.export default App;
```

The next step is to create a **portal** component and import it in the App.js file.

PortalDemo.js

```
1. import React from 'react'
2. import ReactDOM from 'react-dom'
3.
4. function PortalDemo(){
5.   return ReactDOM.createPortal(
6.     <h1>Portals Demo</h1>,
7.     document.getElementById('portal-root')
8.   )
9. }
10.export default PortalDemo
```

Now, open the Index.html file and add a <div id="portal-root"></div> element to access the child component outside the root node.

Index.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3.   <head>
4.     <meta charset="utf-8" />
5.     <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico" />
6.     <meta name="viewport" content="width=device-width, initial-scale=1" />
7.     <meta name="theme-color" content="#000000" />
8.     <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
9.     <title>React App</title>
10.  </head>
11.  <body>
12.    <noscript>It is required to enable JavaScript to run this app.</noscript>
13.    <div id="root"></div>
14.    <div id="portal-root"></div>
15.  </body>
16.</html>
```

Output:



React Hooks

Hooks are the new feature introduced in the React 16.8 version. It allows you to use state and other React features without writing a class. Hooks are the functions which "hook into" React state and lifecycle features from function components. It does not work inside classes.

Hooks are backward-compatible, which means it does not contain any breaking changes. Also, it does not replace your knowledge of React concepts.

When to use a Hooks

If you write a function component, and then you want to add some state to it, previously you do this by converting it to a class. But, now you can do it by using a Hook inside the existing function component.

Rules of Hooks

Hooks are similar to JavaScript functions, but you need to follow these two rules when using them. Hooks rule ensures that all the stateful logic in a component is visible in its source code. These rules are:

1. Only call Hooks at the top level

Do not call Hooks inside loops, conditions, or nested functions. Hooks should always be used at the top level of the React functions. This rule ensures that Hooks are called in the same order each time a component renders.

2. Only call Hooks from React functions

You cannot call Hooks from regular JavaScript functions. Instead, you can call Hooks from React function components. Hooks can also be called from custom Hooks.

Pre-requisites for React Hooks

1. Node version 6 or above
2. NPM version 5.2 or above
3. Create-react-app tool for running the React App

React Hooks Installation

To use React Hooks, we need to run the following commands:

1. `$ npm install react@16.8.0-alpha.1 --save`
2. `$ npm install react-dom@16.8.0-alpha.1 --save`

The above command will install the latest React and React-DOM alpha versions which support React Hooks. Make sure the **package.json** file lists the React and React-DOM dependencies as given below.

1. `"react": "^16.8.0-alpha.1",`
2. `"react-dom": "^16.8.0-alpha.1",`

Hooks State

Hook state is the new way of declaring a state in React app. Hook uses `useState()` functional component for setting and retrieving state. Let us understand Hook state with the following example.

App.js

1. `import React, { useState } from 'react';`
- 2.
3. `function CountApp() {`
4. `// Declare a new state variable, which we'll call "count"`
5. `const [count, setCount] = useState(0);`
- 6.
7. `return (`
8. `<div>`
9. `<p>You clicked {count} times</p>`
10. `<button onClick={() => setCount(count + 1)}>`
11. `Click me`
12. `</button>`

```

13. </div>
14. );
15.}
16.export default CountApp;

```



In the above example, `useState` is the Hook which needs to call inside a function component to add some local state to it. The `useState` returns a pair where the first element is the current state value/initial value, and the second one is a function which allows us to update it. After that, we will call this function from an event handler or somewhere else. The `useState` is similar to `this.setState` in class. The equivalent code without Hooks looks like as below.

App.js

```

1. import React, { useState } from 'react';
2.
3. class CountApp extends React.Component {
4.   constructor(props) {
5.     super(props);
6.     this.state = {
7.       count: 0
8.     };
9.   }
10.  render() {
11.    return (
12.      <div>
13.        <p><b>You clicked {this.state.count} times</b></p>
14.        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
15.          Click me
16.        </button>
17.      </div>
18.    );
19.  }
20.}
21.export default CountApp;

```


Hooks Effect

The Effect Hook allows us to perform side effects (an action) in the function components. It does not use components lifecycle methods which are available in class components. In other words, Effects Hooks are equivalent to `componentDidMount()`, `componentDidUpdate()`, and `componentWillUnmount()` lifecycle methods.

Side effects have common features which the most web applications need to perform, such as:

- Updating the DOM,
- Fetching and consuming data from a server API,
- Setting up a subscription, etc.

Let us understand Hook Effect with the following example.

```
1. import React, { useState, useEffect } from 'react';
2.
3. function CounterExample() {
4.   const [count, setCount] = useState(0);
5.
6.   // Similar to componentDidMount and componentDidUpdate:
7.   useEffect(() => {
8.     // Update the document title using the browser API
9.     document.title = `You clicked ${count} times`;
10.  });
11.
12.  return (
13.    <div>
14.      <p>You clicked {count} times</p>
15.      <button onClick={() => setCount(count + 1)}>
16.        Click me
17.      </button>
18.    </div>
19.  );
20.}
21. export default CounterExample;
```

The above code is based on the previous example with a new feature which we set the document title to a custom message, including the number of clicks.

Output:



In React component, there are two types of side effects:

1. Effects Without Cleanup
2. Effects With Cleanup

Effects without Cleanup

It is used in `useEffect` which does not block the browser from updating the screen. It makes the app more responsive. The most common example of effects which don't require a cleanup are manual DOM mutations, Network requests, Logging, etc.

Effects with Cleanup

Some effects require cleanup after DOM updation. For example, if we want to set up a subscription to some external data source, it is important to clean up memory so that we don't introduce a memory leak. React performs the cleanup of memory when the component unmounts. However, as we know that, effects run for every render method and not just once. Therefore, React also cleans up effects from the previous render before running the effects next time.

Custom Hooks

A custom Hook is a JavaScript function. The name of custom Hook starts with "use" which can call other Hooks. A custom Hook is just like a regular function, and the word "use" in the beginning tells that this function follows the rules of Hooks. Building custom Hooks allows you to extract component logic into reusable functions.

Let us understand how custom Hooks works in the following example.

1. `import React, { useState, useEffect } from 'react';`
- 2.

```

3. const useDocumentTitle = title => {
4.   useEffect() => {
5.     document.title = title;
6.   }, [title])
7. }
8.
9. function CustomCounter() {
10.  const [count, setCount] = useState(0);
11.  const incrementCount = () => setCount(count + 1);
12.  useDocumentTitle(`You clicked ${count} times`);
13.  // useEffect() => {
14.  //   document.title = `You clicked ${count} times`
15.  // });
16.
17.  return (
18.    <div>
19.      <p>You clicked {count} times</p>
20.      <button onClick={incrementCount}>Click me</button>
21.    </div>
22.  )
23.}
24.export default CustomCounter;

```

In the above snippet, useDocumentTitle is a custom Hook which takes an argument as a string of text which is a title. Inside this Hook, we call useEffect Hook and set the title as long as the title has changed. The second argument will perform that check and update the title only when its local state is different than what we are passing in.

Built-in Hooks

Here, we describe the APIs for the built-in Hooks in React. The built-in Hooks can be divided into two parts, which are given below.

Basic Hooks

- useState
- useEffect
- useContext

Additional Hooks

- useReducer

- useCallback
- useMemo
- useRef
- useImperativeHandle
- useEffect
- useDebugValue

React Redux

Redux is an open-source JavaScript library used to manage application state. React uses Redux for building the user interface. It was first introduced by **Dan Abramov** and **Andrew Clark** in **2015**.

React Redux is the official React binding for Redux. It allows React components to read data from a Redux Store, and dispatch **Actions** to the **Store** to update data. Redux helps apps to scale by providing a sensible way to manage state through a unidirectional data flow model. React Redux is conceptually simple. It subscribes to the Redux store, checks to see if the data which your component wants have changed, and re-renders your component.

Redux was inspired by Flux. Redux studied the Flux architecture and omitted unnecessary complexity.

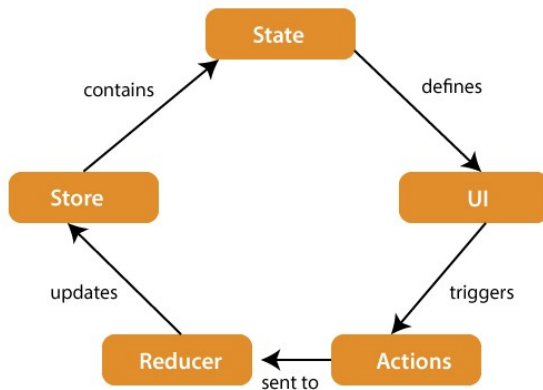
- Redux does not have Dispatcher concept.
- Redux has an only Store whereas Flux has many Stores.
- The Action objects will be received and handled directly by Store.

Why use React Redux?

The main reason to use React Redux are:

- React Redux is the official **UI bindings** for react Application. It is kept up-to-date with any API changes to ensure that your React components behave as expected.
- It encourages good 'React' architecture.
- It implements many performance optimizations internally, which allows to components re-render only when it actually needs.

Redux Architecture



he components of Redux architecture are explained below.

STORE: A Store is a place where the entire state of your application lists. It manages the status of the application and has a `dispatch(action)` function. It is like a brain responsible for all moving parts in Redux.

ACTION: Action is sent or dispatched from the view which are payloads that can be read by Reducers. It is a pure object created to store the information of the user's event. It includes information such as type of action, time of occurrence, location of occurrence, its coordinates, and which state it aims to change.

REDUCER: Reducer read the payloads from the actions and then updates the store via the state accordingly. It is a pure function to return a new state from the initial state.

Redux Installation

Requirements: React Redux requires React 16.8.3 or later version.

To use React Redux with React application, you need to install the below command.

```
$ npm install redux react-redux --save
```

Reducer

Introduction

We have already come across Redux and its state management solution for React. As soon as the rise of Redux, Reducers gained immense popularity amongst developers. We don't need to learn Redux to understand Reducers. Its

knowledge might help us get adapted quickly. In this article, we would be learning about Reducers and what they are, so let's dive in.

A Reducer can be termed as a pure function that returns a new state whenever we take the state of an application and its action as the argument. In general terms, Reducers are the state managers in an application. Consider an instance where we code something in [HTML](#) for an input field. The Reducers thereby manages the UI state of an application or controlled components.

Consider another instance, if we design an authentication reducer that can take an empty object in the initial phase of the application. We can notify the user that he has logged in and had returned to the state, where he is being marked as a logged-in user. It is simple as that.

Implementation

Before implementing the Reducers, we might need to start from the very basics of pure functions. Pure functions are nothing but those functions that do not have any side effects and they return the exact result provided the same arguments are passed in. State management is also a form of pure functions and the actions associated with can be also be termed as a pure function. Consider the below code:

1. `const multiply = (x, y) => x * y;`
- 2.
3. `multiply(2, 3);`

The example above returns a value based on the input fields provided by us i.e. 2 and 3 and we would always get 6 as the output as long as the same input is provided. Nothing else can affect the output we will get. Similarly, a reducer can be easily implemented in the same concept. Consider the below code:

1. `const initialState = {};`
2. `const cartReducer = (state = initialState, action) =>`
3. `{`
4. `// your_code_goes_here`
5. `}`

In the above code, we have defined state and action property associated with Reducers. Let's understand them in depth-first.

State

A state can be considered as the data component that **holds the data** required by the component and instructs the component to render and re-render if the object changes its state. In a Redux managed application, the reducer is where the **change of state** occurs.

Action

An action can be defined as an object that **contains the payload** of information or the responsible component that triggers the changes in the Redux application state. Actions are by default the states in JavaScript that tell Redux to perform the specific type of task. Consider the below code:

```
1. const action = {  
2.   type: 'ADD_TO_BAG',  
3.   payload: {  
4.     product: 'Apples',  
5.     quantity: 6  
6.   }  
7. }
```

An action can be defined as an object that contains the payload of information or the responsible component that triggers the changes in the Redux application state. Actions are by default the states in JavaScript that tell Redux to perform the specific type of task.

A reducer function takes two arguments i.e. the current state and the actions and displays result based on both the arguments to a new state. Consider the below syntax for the same.

```
1. (state,action) => newState
```

In the similar sense, a reducer function is implemented. Consider the below code:

```
1. function ReducerCounter(state,action)  
2. {  
3.   return state+1;  
4. }  
5.  
6. Or  
7.  
8. function ReducerCounter = (count,action) =>  
9. {  
10.   return count+1;
```

11.}

In this case, the reducer function increases the count by one since the current state is an integer. Renaming state to count does no harm since the argument becomes more readable and a better approach for beginners. This makes Reducer functions the perfect fit for reasoning about the state changes and has enough room to test them in an isolated environment. You can repeat the same test with the same input and the same output would be displayed.

Moreover, we can also perform conditional state transitions as we do with the core programs. You can do the conditional state transitions with the same concept of increasing and decreasing the count as we did in the example code. Consider the below code:

```
1. const counterReducer = (count, action) =>
2. {
3.   if (action.type === 'Increment')
4.   {
5.     return count + 1;
6.   }
7.
8.   if (action.type === 'Decrement')
9.   {
10.    return count - 1;
11.  }
12.
13.  return count;
14.};
```

Conversely, it can be stated that reducers in JavaScript are one of the most useful methods of arrays that should always be in the arsenal of developer community. Reducers can be indirectly called as map methods that are used with arrays to improve their simplicity and performance in specific situations. As discussed earlier, a reduce methods invokes a callback function to each of the array stored element and outputs the final values as operation generator. Reducers deliver a cleaner way to iterate over objects and process the data stored in the array blocks efficiently.

To understand more about the reducers, we can create our own reducer with the help the below example shown. In this example, we are implementing a reducer function to observe how it works under the hood. This example would give you a better idea when to use the Reducer in JavaScript for optimizing the performance of our program.


```

1. array.prototype.reduceConceptual = function(callback, initial) {
2.   if (!this) {
3.     throw Error('Array.prototype.forEach called on null or undefined');
4.   }
5.
6.   if (!callback || typeof callback !== 'function') {
7.     throw Error('callback is not a function');
8.   }
9.
10.  let acc = initial;
11.
12.  for(i=0;i<this.length;i++) {
13.    acc = callback(acc, this[i], i, this);
14.  }
15.
16.  return acc;
17.}

```

First, we can check if the reduce method was called on a **null** or **undefined** object. Then we check if the passed callback is a function.

In the above code, we first check if the reducer method is called by an undefined or null object. Then we check if the passed callback can be considered as function.

After these initial type checks, we assign the past initial value to the accumulator. We then iterate over the array using loops to call the callback for every array item. After the execution ends, we have to return the value of the accumulator.

We are using this implementation only to help you understand how the reduce method actually works. For example, you can see that it uses **for loop** to iterate through the array under the hood.

Note: We are using the above coding sample only to make the monolithic concepts of reducers and how the methods work. Hence, the above example is not a production quality code and is used just for the creation of prototyping the reducer method in standard JavaScript.

Conclusion

Reducers are an integral part of state management in Redux. With the help of Reducers, we can easily manage the asynchronous transfer of data into different modes and configure it. Another advantage of using Reducers is that we can

write pure functions to update only specific areas of our Redux application without creating any side effects in other areas. In this tutorial, we learned the basics and core of Reducers and how it extends its support while managing states in the Redux application. We also learned the core of reducers i.e. state and actions with the help of various examples.