

UNIT-2

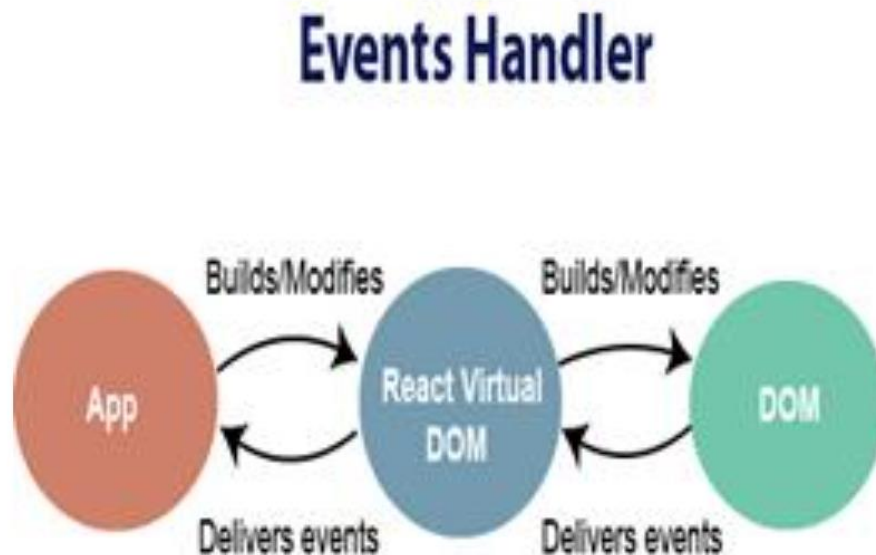
REACT ADVANCED CONCEPTS

2.1.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.

React Events



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

React events are named as camelCase instead of lowercase.

With JSX, a function is passed as the event handler instead of a string. For example:

Event declaration in plain HTML:

```
<button onclick="showMessage()">
  Hello Skyeagle
</button>
```

Event declaration in React:

```
<button onClick={showMessage}>
  Hello Skyeagle
</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:

```
<a href="#" onclick="console.log('You had clicked a Link.');" return false">
  Click_Me
</a>
```

In React, we can write it as:

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('You had clicked a Link.');"
  }
  return (
    <a href="#" onClick={handleClick}>
      Click_Me
    </a>
  );
}
```

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.

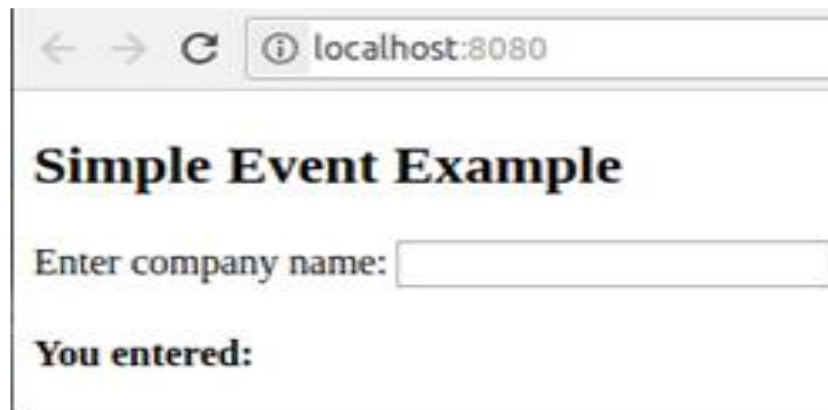
Example

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.

```
import React, { Component } from 'react';
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      companyName: ""
    };
  }
  changeText(event) {
    this.setState({
      companyName: event.target.value
    });
  }
  render() {
    return (
      <div>
        <h2>Simple Event Example</h2>
        <label htmlFor="name">Enter company name: </label>
        <input type="text" id="companyName" onChange={this.changeText.bind(this)} />
        <h4>You entered: { this.state.companyName }</h4>
      </div>
    );
  }
}
export default App;
```

Output

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



The screenshot shows a web browser window with the address bar displaying 'localhost:8080'. The page title is 'Simple Event Example'. Below the title, there is a text input field with the placeholder text 'Enter company name:'. Below the input field, the text 'You entered:' is displayed.

2.2.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.

Uncontrolled component

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.

```
import React, { Component } from 'react';
class App extends React.Component {
  constructor(props) {
    super(props);
    this.updateSubmit = this.updateSubmit.bind(this);
    this.input = React.createRef();
  }
  updateSubmit(event) {
    alert('You have entered the UserName and CompanyName successfully.');
```

```
render() {
  return (
    <form onSubmit={this.updateSubmit}>
      <h1>Uncontrolled Form Example</h1>
      <label>Name:
        <input type="text" ref={this.input} />
      </label>
      <label>
        CompanyName:
        <input type="text" ref={this.input} />
      </label>
      <input type="submit" value="Submit" />
    </form>
  );
}
export default App;
```

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

```
import React, { Component } from 'react';
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: '' };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleChange(event) {
    this.setState({ value: event.target.value });
  }
  handleSubmit(event) {
    alert('You have submitted the input successfully: ' + this.state.value);
    event.preventDefault();
  }
  render() {
```

```

return (
  <form onSubmit={this.handleSubmit}>
    <h1>Controlled Form Example</h1>
    <label>
      Name:
      <input type="text" value={this.state.value} onChange={this.handleChange} />
    </label>
    <input type="submit" value="Submit" />
  </form>
);
}
}
export default App;

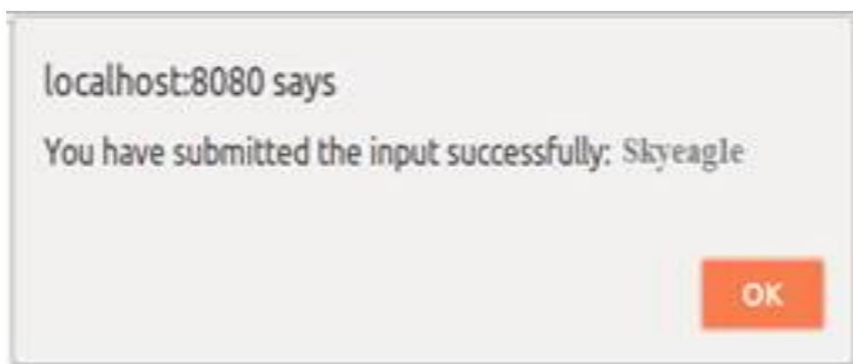
```

Output

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



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

```

import React, { Component } from 'react';
class App extends React.Component {

```

```

constructor(props) {
  super(props);
  this.state = {
    personGoing: true,
    numberOfPersons: 5
  };
  this.handleInputChange = this.handleInputChange.bind(this);
}
handleInputChange(event) {
  const target = event.target;
  const value = target.type === 'checkbox' ? target.checked : target.value;
  const name = target.name;
  this.setState({
    [name]: value
  });
}
render() {
  return (
    <form>
      <h1>Multiple Input Controlled Form Example</h1>
      <label>
        Is Person going:
        <input
          name="personGoing"
          type="checkbox"
          checked={this.state.personGoing}
          onChange={this.handleInputChange} />
      </label>
      <br />
      <label>
        Number of persons:
        <input
          name="numberOfPersons"
          type="number"
          value={this.state.numberOfPersons}
          onChange={this.handleInputChange} />
      </label>
    </form>
  );
}
}
export default App;

```

Output



Multiple Input Controlled Form

Is Person going: ☒

Number of persons:

2.3. Scaling React components:

1. Setting default properties for components
2. Understanding React property types and validation
3. Rendering children
4. Creating higher-order components for code reuse
5. Presentational versus container components

1. Setting Default properties in components

Imagine that you're building a `Datepicker` component that takes a few required properties such as number of rows, locale, and current date:

```
<Datepicker currentDate={Date()} locale="US" rows={4}/>
```

The key benefit of `defaultProps` is that *if a property is missing, a default value is rendered*.

To set a default property value on the component class, you define `defaultProps`. For example, in the aforementioned `Datepicker` component definition, you can add a static class attribute (not an instance attribute, because that won't work—instance attributes are set in `constructor()`):

```
class Datepicker extends React.Component {
  ...
}

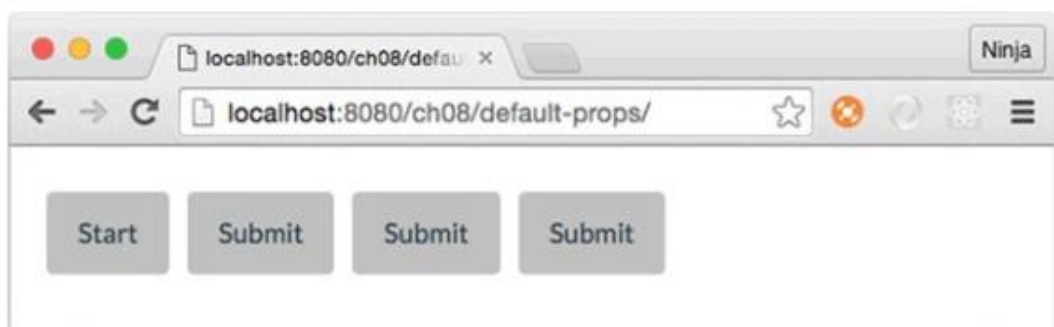
Datepicker.defaultProps = {
  currentDate: Date(),
  rows: 4,
  locale: 'US'
}
```

```
class Button extends React.Component {
  render() {
    return <button className="btn" >{this.props.buttonLabel}</button>
  }
}
```

```
Button.defaultProps = { buttonLabel: 'Submit' }
```

```
class Content extends React.Component {
  render() {
    return (
      <div>
        <Button buttonLabel="Start"/>
        <Button />
        <Button />
        <Button />
      </div>
    )
  }
}
```

Figure 8.1. The first button has a label that's set on creation. The other elements don't and thus fall back to the default property value.



2. React property types and validation

Going back to the earlier example with the `Datepicker` component and coworkers who aren't aware of property types (`"5"` versus `5`), you can set property types to use with React.js component classes. You do so via the `propTypes` static attribute. This feature of property types doesn't enforce data types on property values and instead gives you a warning. That is, if you're in development mode, and a type doesn't match, you'll get a warning message in the console

and in production; nothing will be done to prevent the wrong type from being used. In essence, React.js suppresses this warning in production mode. Thus, `propTypes` is mostly a convenience feature to warn you about mismatches in data types at a developmental stage.

```
class DatePicker extends React.Component {
  ...
}
DatePicker.propTypes = {
  currentDate: PropTypes.string,
  rows: PropTypes.number,
  locale: PropTypes.oneOf(['US', 'CA', 'MX', 'EU'])
}
```

← window.PropTypes
because the script
includes prop-types.js

3. Rendering children

Let's continue with the fictional React project; but instead of a `DatePicker` (which is now robust and warns you about any missing or incorrect properties), you're tasked with creating a component that's universal enough to use with any children you pass to it. It's a blog post `Content` component that may consist of a heading and a paragraph of text:

```
<Content>
```

```
<h1>React.js</h1>
```

```
<p>Rocks</p>
```

```
</Content>
```

The `children` property is an easy way to render all children with `{this.props.children}`. You can also do more than rendering. For example, add a `<div>` and pass along child elements:

```
class Content extends React.Component {
```

```
  render() {
```

```
    return (
```

```

    <div className="content">

      {this.props.children}

    </div>
  )
}
}

```

The parent of `Content` has the children `<h1>` and `<p>` :

```

ReactDOM.render(

  <div>

    <Content>

      <h1>React</h1>

      <p>Rocks</p>

    </Content>

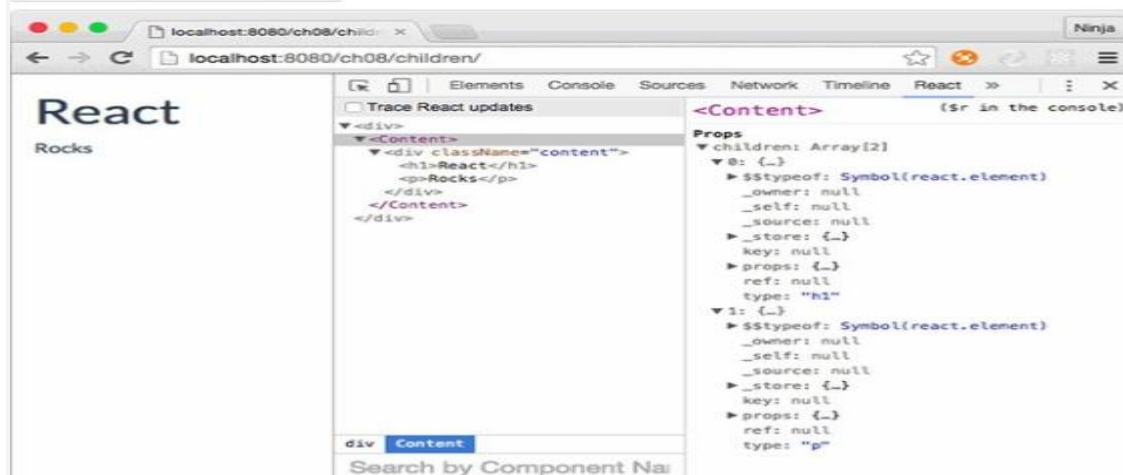
  </div>,

  document.getElementById('content')

)

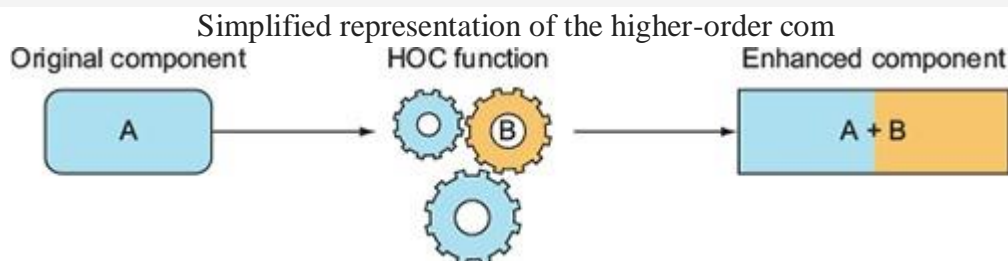
```

Figure 8.5. Rendering a single `Content` component with a heading and paragraph using `this.props.children`, which shows two items



4. Creating React higher-order components for code reuse

A higher-order component (HOC) lets you enhance a component with additional logic. You can think of this pattern as components inheriting functionality when used with HOCs. In other words, *HOCs let you reuse code*. This allows you and your team to share functionality among React.js components.



5. Presentational versus container components:

Presentational components typically only add structure to DOM and styling. They take properties but often don't have their own states. Most of the time, you can use functions for stateless presentational components.

2.4. REACT ROUTING:

React Router is a standard library for routing in React. It enables the navigation among views of various components in a React Application, allows changing the browser URL, and keeps the UI in sync with the URL.

Let us create a simple application to React to understand how the React Router works. The application will contain three components: home component, about a component, and contact component. We will use React Router to navigate between these components. After installing `react-router-dom`, add its components to your React application.

Adding React Router Components: The main Components of React Router are:

- **BrowserRouter:** BrowserRouter is a router implementation that uses the HTML5 history API(pushState, replaceState and the popstate event) to keep your UI in sync with the URL. It is the parent component that is used to store all of the other components.
- **Routes:** It's a new component introduced in the v6 and a upgrade of the component. The main advantages of Routes over Switch are:
 - Relative s and s
 - Routes are chosen based on the best match instead of being traversed in order.
- **Route:** Route is the conditionally shown component that renders some UI when its path matches the current URL.
- **Link:** Link component is used to create links to different routes and implement navigation around the application. It works like HTML anchor tag.

Using React Router: To use React Router, let us first create few components in the react application. In your project directory, create a folder named **component** inside the src folder and now add 3 files named **home.js**, **about.js** and **contact.js** to the component folder.

Let us add some code to our 3 components:

Home.js

```
import React from 'react';

function Home (){
    return <h1>Welcome to the world of Geeks!</h1>
}

export default Home;
```

About.js

```
import React from 'react';

function About () {
    return <div>
        <h2>shri Vishnu engineering college for women</h2>

        Read more about us at :
        <a href="https://www.svecw.edu.in/about/">
        </a>

    </div>
}

export default About;
```

Contact.js

```
import React from 'react';

function Contact (){
    return <address>
        You can find us here:<br />
        SVECW<br />
        VISHNUPUR,BHIMAVARAM <br />
        PIN-534201
    </address>
```

```
}
```

export default Contact;

BrowserRouter: Add BrowserRouter aliased as Router to your app.js file in order to wrap all the other components. BrowserRouter is a parent component and can have only single child.

```
class App extends Component {
  render() {
    return (
      <Router>
        <div className="App">
          </div>
        </Router>
      </Router>
    );
  }
}
```

Link: Let us now create links to our components. Link component uses the **to** prop to describe the location where the links should navigate to.

```
<div className="App">
  <ul>
    <li>
      <Link to="/">Home</Link>
    </li>
    <li>
      <Link to="/about">About Us</Link>
    </li>
    <li>
      <Link to="/contact">Contact Us</Link>
    </li>
  </ul>
</div>
```

Route: Route component will now help us to establish the link between component's UI and the URL. To include routes to the application, add the code give below to your app.js.

```
<Route exact path="/" element={< Home />}></Route>
<Route exact path="/about" element={< About />}></Route>
<Route exact path="/contact" element={< Contact />}></Route>
```

Routes: To render a single component, wrap all the routes inside the Routes Component.

```
<Routes>
  <Route exact path="/" element={< Home />}></Route>
  <Route exact path="/about" element={< About />}></Route>
  <Route exact path="/contact" element={< Contact />}></Route>
</Routes>
```

After adding all the components here is our complete source code:

```
import React, { Component } from 'react';
import { BrowserRouter as Router, Routes, Route, Link } from 'react-router-dom';
import Home from './component/home';
import About from './component/about';
import Contact from './component/contact';
import './App.css';
```

```

class App extends Component {
  render() {
    return (
      <Router>
        <div className="App">
          <ul className="App-header">
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/about">About Us</Link>
            </li>
            <li>
              <Link to="/contact">Contact Us</Link>
            </li>
          </ul>
          <Routes>
            <Route exact path="/" element={< Home />}></Route>
            <Route exact path="/about" element={< About />}></Route>
            <Route exact path="/contact" element={< Contact />}></Route>
          </Routes>
        </div>
      </Router>
    );
  }
}

export default App;

```

2.5.REDUX:

What is Redux?

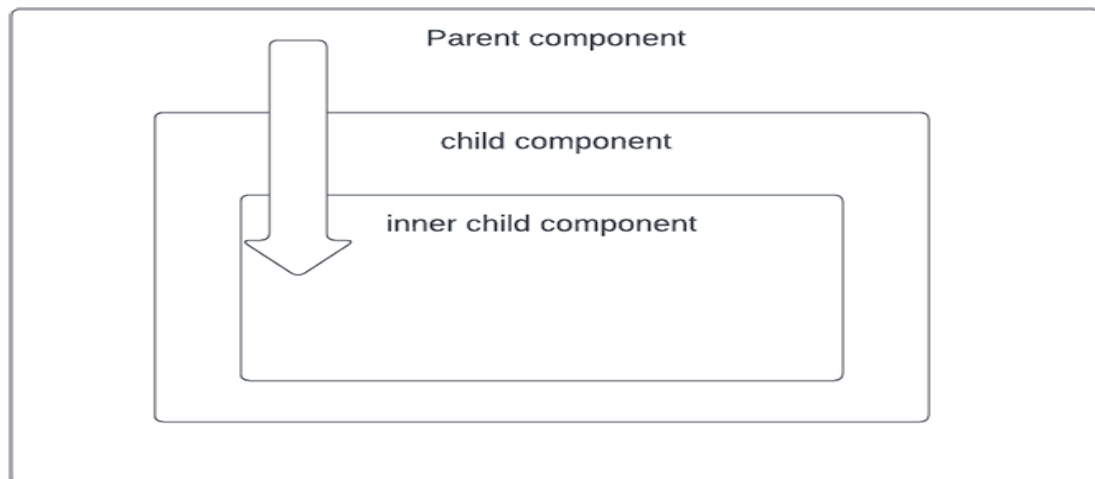
Redux is a predictable state container designed to help you write JavaScript apps that behave consistently across client, server, and native environments, and are easy to test.

With Redux, the state of your application is kept in a store, and each component can access any state that it needs from this store.

What is Redux used for?

Simply put, Redux is used to maintain and update data across your applications for multiple components to share, all while remaining independent of the components.

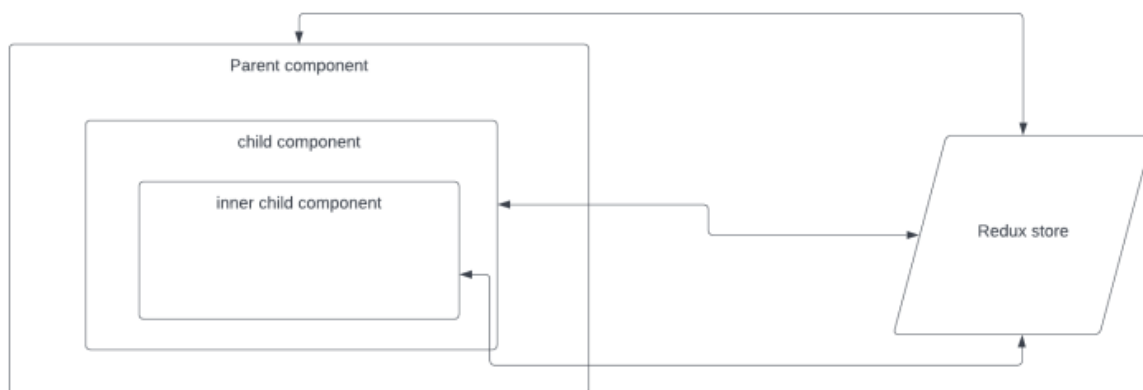
Without Redux, you would have to make data dependent on the components and pass it through different components to where it's needed.



WITHOUT REDUX

In our example above, we only need some data in the parent and inner child components, but we're forced to pass it to all the components (including the child component that doesn't need it) to get it to where it's needed.

With Redux, we can make state data independent of the components, and when needed, a component can access or update through the Redux store.



WITH REDUX

What is state management in Redux?

State management is essentially a way to facilitate communication and sharing of data across components. It creates a tangible data structure to represent the state of your app that you can read from and write to. That way, you can see otherwise invisible states while you're working with them.

Most libraries, such as React, Angular, etc. are built with a way for components to internally manage their state without any need for an external library or tool. It does well for applications with few components, but as the application grows bigger, managing states shared across components becomes a chore.

In an app where data is shared among components, it might be confusing to actually know where a state should live. Ideally, the data in a component should live in just one component, so sharing data among sibling components becomes difficult.

For instance, in React, to share data among siblings, a state has to live in the parent component. A method for updating this state is provided by the parent component and passed as props to these sibling components.

Here's a simple example of a login component in React. The input of the login component affects what is displayed by its sibling component, the status component:

```
class App extends React.Component {
  constructor(props) {
    super(props);
    // First the Parent creates a state for what will be passed
    this.state = { userStatus: "NOT LOGGED IN"}
    this.setStatus = this.setStatus.bind(this);
  }
  // A method is provided for the child component to update the
  // state of the
  // userStatus
  setStatus(username, password) {
    const newUsers = users;
    newUsers.map(user => {
      if (user.username == username && user.password === password) {
        this.setState({
          userStatus: "LOGGED IN"
        })
      }
    });
  }

  render() {
    return (
      <div>
        // the state is passed to the sibling as a props as is
        // updated whenever
        // the child component changes the input
        <Status status={this.state.userStatus} />
        // this method is passed to the child component as a props
        // which it
        // uses to change the state of the userStatus
        <Login handleSubmit={this.setStatus} />
      </div>
    );
  }
}
```

Basically, the state will have to be lifted up to the nearest parent component and to the next until it gets to an ancestor that is common to both components that need the state, and then it

is passed down. This makes the state difficult to maintain and less predictable. It also means passing data to components that do not need it.

It's clear that state management gets messy as the app gets complex. This is why you need a state management tool like Redux that makes it easier to maintain these states. Let's get a good overview of Redux concepts before considering its benefits.

What are Redux actions?

Simply put, Redux actions are events.

They are the only way you can send data from your application to your Redux store. The data can be from user interactions, API calls, or even form submissions.

Actions are plain JavaScript objects that must have

- a **type** property to indicate the type of action to be carried out, and
- a **payload** object that contains the information that should be used to change the state.

```
{
  type: "LOGIN",
  payload: {
    username: "foo",
    password: "bar"
  }
}
```

What are Redux reducers?

Reducers are pure functions that take the current state of an application, perform an action, and return a new state. The reducer handles how the state (application data) will change in response to an action.

```
const LoginComponent = (state = initialState, action) => {
  switch (action.type) {

    // This reducer handles any action with type "LOGIN"
    case "LOGIN":
      return state.map(user => {
        if (user.username !== action.username) {
          return user;
        }

        if (user.password === action.password) {
          return {
            ...user,
            login_status: "LOGGED IN"
          }
        }
      });
      default:
        return state;
  }
}
```

```
} };
```

What is Redux Store?

The store is a “container” (really a JavaScript object) that holds the application state, and the only way the state can change is through actions dispatched to the store. Redux allows individual components connect to the store and apply changes to it by dispatching actions.

It is highly recommended to keep only one store in any Redux application. You can access the state stored, update the state, and register or unregister listeners via helper methods.

Let’s create a store for our login app:

```
const store = createStore(LoginComponent);
```

Why use Redux?

1. Redux makes the state predictable
2. Redux is maintainable
3. Debugging is easy in Redux
4. Performance benefits
5. Ease of testing
6. State persistence
7. Server-side rendering

2.6.Unit Testing of React Apps using JEST:

Testing software is as important as developing it, since, testing helps find out whether the software meets the actual requirements or not. A thoroughly tested software product ensures dependability, security, and high performance, which leads to time-saving, customer satisfaction, and cost-effectiveness.

ReactJS is a popular JavaScript library that is used for building highly rich user interfaces. A few reasons that make React a popular framework among developers are:

- **Ease of Learning:** React JS has a short learning curve as compared to other front-end libraries or frameworks, hence any developer with a basic knowledge of JavaScript can start learning and building apps using React JS in a short period of time.
- **Quick Rendering:** React JS uses virtual DOM and algorithms like diffing, which makes the development process fast and efficient.
- **One-way data-binding:** React JS follows one-way data binding, which means you get more control over the flow of the application.
- **High Performance:** The best advantage of React JS is its performance. There are many features that make it possible:
 - React uses virtual DOM, which enables the re-rendering of nodes only when they are required to.

- React JS also supports bundling and tree shaking, which minimizes the end user's resource load.

This tutorial deep dives into performing unit testing of React Apps using JEST.

What is Unit Testing for React Apps? Why is it important?

Unit Testing is a testing method that tests an individual unit of software in isolation. Unit testing for React Apps means testing an individual React Component.

“Unit testing is a great discipline, which can lead to 40% – 80% reductions in bug density.” –

Unit Testing is important for React Apps, as it helps in [testing the individual functionality of React components](#). Moreover, any error in code can be identified at the beginning itself, saving time to rectify it at later stages. Some of the core benefits of Unit Testing are:

- **Process Becomes Agile:** Agile Testing process is the main advantage of unit testing. When you add more features to the software, it might affect the older designs and you might need to make changes to the old design and code later. This can be expensive and require extra effort. But if you do unit testing, the whole process becomes much faster and easier.
- **Quality of code:** Unit testing significantly improves the quality of the code. It helps developers to identify the smallest defects that can be present in the units before they go for the integration testing.
- **Facilitates change:** Refactoring the code or updating the system library becomes much easier when you test each component of the app individually.
- **Smooth Debugging:** The debugging process is very simplified by doing unit testing. If a certain test fails, then only the latest changes that have been made to the code need to be debugged.
- **Reduction in cost:** When bugs are detected at an early stage, through unit testing, they can be fixed at almost no cost as compared to a later stage, let's say during production, which can be really expensive.

How to perform Unit testing of React Apps using JEST?

[Jest](#) is a JavaScript testing framework that allows developers to run tests on JavaScript and TypeScript code and can be easily integrated with React JS.

Step 1: Create a new react app

For unit testing a react app, let's create one using the command given below:

```
npx create-react-app react-testing-tutorial
```

Open the **package.json**, and you will find that when you use **create-react-app** for creating a react project, it has default support for jest and react testing library. This means that we do not have to install them manually.

Step 2: Create a component

Let's create a component called Counter, which simply increases and decreases a numeric value at the click of respective buttons.

```
import React, { useState } from "react";

const Counter = () => {

  const [counter, setCounter] = useState(0);


  const incrementCounter = () => {

    setCounter((prevCounter) => prevCounter + 1);

  };

  const decrementCounter = () => {

    setCounter((prevCounter) => prevCounter - 1);

  };

  return (

    <>

    <button data-testid="increment" onClick={incrementCounter}>

    +

    </button>

    <p data-testid="counter">{counter}</p>

    <button disabled data-testid="decrement" onClick={decrementCounter}>

    -

    </button>

    </>

  )
}
```

```
);

};

export default Counter;
```

Step 3: Write a unit test for the react component

Before writing an actual unit test, let's understand the general structure of a test block:

- A test is usually written in a test block.
- Inside the test block, the first thing we do is to render the component that we want to test.
- Select the elements that we want to interact with
- Interact with those elements
- Assert that the results are as expected.

The unit test of react component can be written as seen in the code snippet below:

```
import { render, fireEvent, screen } from "@testing-library/react";

import Counter from "../components/Counter";

//test block

test("increments counter", () => {

  // render the component on virtual dom

  render(<Counter />);

  //select the elements you want to interact with

  const counter = screen.getByTestId("counter");

  const incrementBtn = screen.getByTestId("increment");

  //interact with those elements

  fireEvent.click(incrementBtn);

  //assert the expected result

  expect(counter).toHaveTextContent("1");

});
```

Note: In order to let jest know about this test file, it's important to use the extension **.test.js**. The above test can be described as:

- The test block can be written either using **test()** or **it()**. Either of the two methods takes two parameters:
 - The first parameter is to name the test. For example, **increments counter**.
 - The second parameter is a **callback** function, which describes the actual test.
- Using the **render()** method from the react testing library in the above test to render the Counter component in a virtual DOM.
- The **screen** property from the react testing library helps select the elements needed to test by the **test ids** provided earlier.
- To interact with the button, using the **fireEvent** property from the react testing library in the test.
- And finally, it is asserted that the counter element will contain a value '1'.

Step 4: Run the test

Run the test using the following command:

```
npm run test
```

Test Result

```
$ npm run test

PASS src/tests/Counter.test.js (15.279 s)
  ✓ increments counter (204 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        31.064 s
Ran all test suites related to changed files.
```