UNIT-2

Updating React Components, Creating a Newsfeed.
Forms, **Libraries & Routing**: Working with Forms & Third Party libraries, Routing.
Redux: Application Architecture, Integrating Redux with React.

Q) Explain updating phase in React component lifecycle with a suitable example.

At updating phase of a component, below methods are executed based on props or state being received and updates happens in a component:

- static getDerivedStateFromProps()
- shouldComponentUpdate()
- render()
- getSnapshotBeforeUpdate()
- componentDidUpdate()

static getDerivedStateFromProps(props,state):

- getDerivedStateFromProps is invoked just before calling the render method, it happens both on initial and subsequent mount. This method should return an object to update the state or null to update nothing.
- The main purpose of this method is to make sure that the state and props are in sync.

Eg.

```
const MyComponent extends React.Component {
   ...

   static getDerivedStateFromProps(props,state){
      return {
         amount : 10; //state will be updated to this
         //return null; if don't want to update the state
      }
   }
}
```

shouldComponentUpdate(nextProps, nextState):

- Executed before rendering when new state or props are being received
- By default it will return true by calling componentDidUpdate() method. If it returns false componentDidUpdate() method will not be invoked
- In case we want to skip re-rendering, this method could be used by returning false, so this would skip re-rendering
- This method only exists as a performance optimization

getSnapshotBeforeUpdate(prevProps, prevState):

- This method is called immediately after the render and before the changes are uodated to the DOM.

- This method is the last chance to get the prevState and prevProps values before the component is updated. The method can return values based on the prevState and prevProps values.
- Once the updates flushed to the DOM, immediately next lifecycle method will be invoked

Eg. One of the use case of this method is to save the scroll position in chat application.

```
getSnapshotBeforeUpdate(prevProps, prevState) {
    if (prevState.chats.length < this.state.chats.length) {
      const layout = this.layout.current;
      const isAtBottomOfChatWindow =
        window.innerHeight + window.pageYOffset === layout.scrollHeight;
      return { isAtBottomOfGrid };
    }
    return null;
}
```

The above method checks whether the user is at the bottom of the chat window. If so, then the user should be scrolled to the bottom of the chat window, when new messages comes up in the chat window.
If user is not at the bottom of the chat window, then keep them there. Hence we return null

render():
This method is called to re-render the component with the updated state and/or props.

componentDidUpdate(prevProps, prevState, snapshot):

- Invoked immediately after the changes are updated

Eg.
The snapshot returned from getSnapshotBeforeUpdate method will be passed as a third argument to the componentDidUpdate method as shown below

```
componentDidUpdate(prevProps, prevState, snapshot) {

  if (snapshot.isAtBottomOfChatWindow) {
    window.scrollTo({
      top: this.layout.current.scrollHeight
    });
  }
}
```

In the componentDidUpdate method, we check whether user was at the bottom on chat window using snapshot argument. If yes, we scroll the user to the bottom of the new messages that has come.

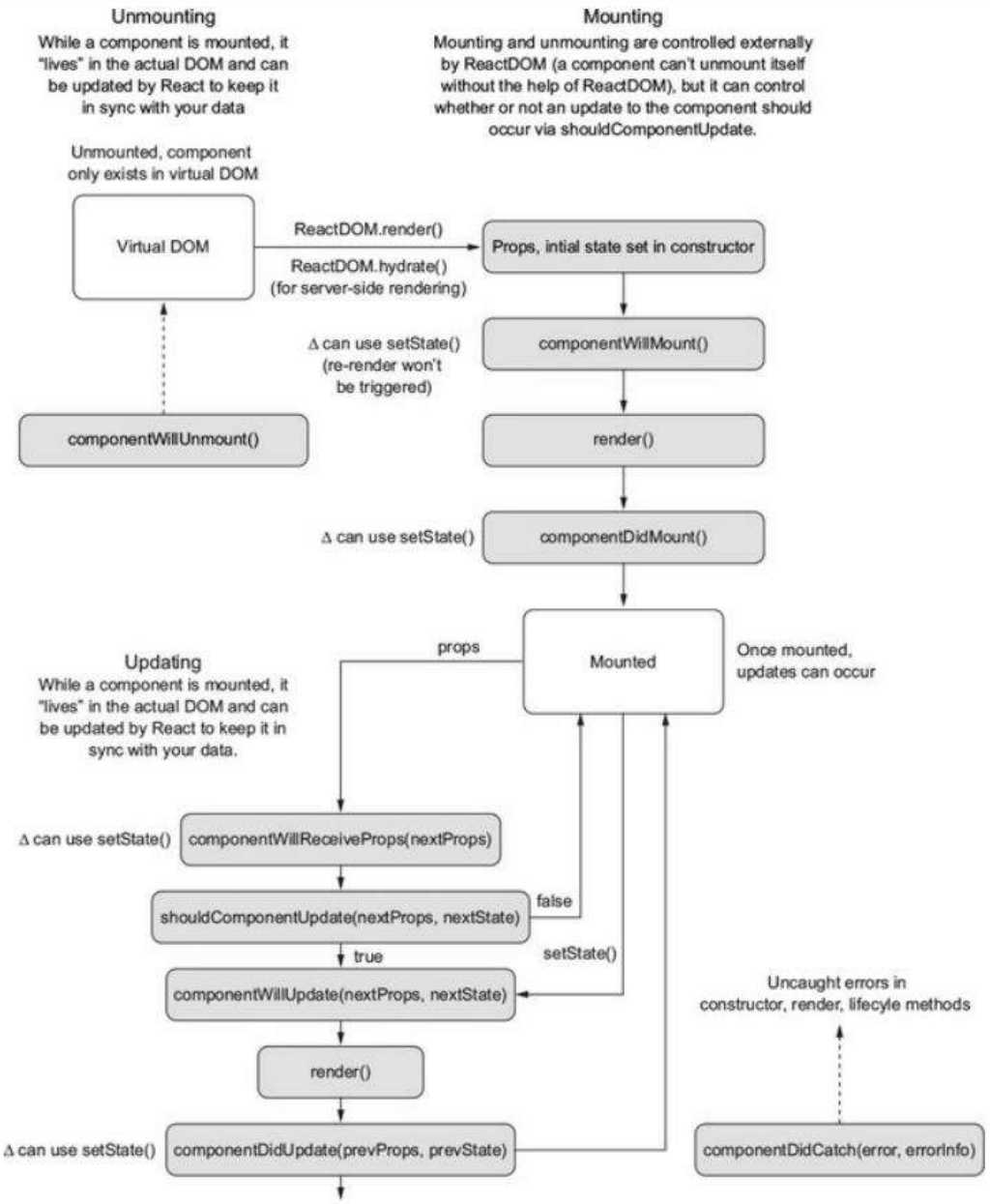


Fig. Overview of React Component Lifecycle

Eg. Create a timer application that updates the time after 4 seconds on the webpage.

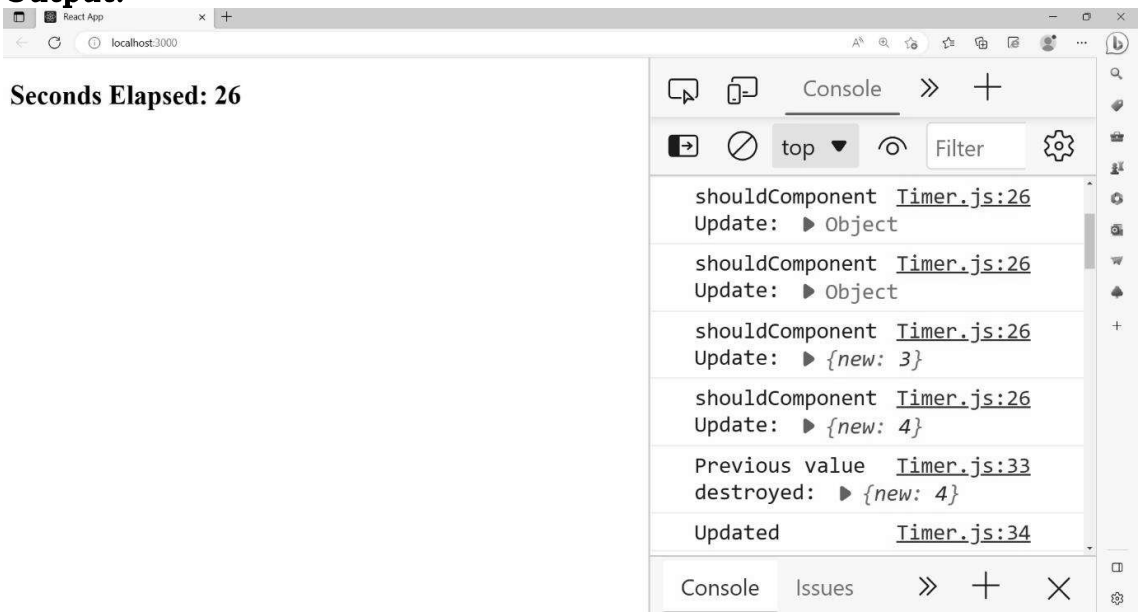
index.js

```
import React from "react";
import {createRoot} from 'react-dom/client';
import Timer from "./components/Timer";


const root = createRoot(document.getElementById('root'));
root.render(
  <Timer />
);
```

Timer.js

```
import React from 'react';
class Timer extends React.Component {
    constructor(props) {
    super(props);
    this.state = {
        count: 0
        };
    }
    start = () => {
        this.setState({
        count: this.state.count + 1
        });
    }
    componentDidMount() {
        this.interval = setInterval(this.start, 1000);
    }
    render() {
        return ( <React.Fragment>
                    <Updates new = {this.state.count} />
                </React.Fragment>);
            }
        }
class Updates extends React.Component {
    shouldComponentUpdate(newProps, newState) {
        if (this.props.new <= 3) {
            console.log('shouldComponentUpdate:', newProps);
                return false;
            } else {
                 return true;
            }
    }
    componentDidUpdate(prevProps, prevState, snapshot) {
            console.log('Previous value destroyed:', prevProps);
            console.log('Updated');
    }
    render() {
        return ( <React.Fragment>
                    <h2> Seconds Elapsed: {this.props.new} </h2>
                </React.Fragment>);
            }
}
export default Timer;
```

Output:



In the above code, componentDidMount() called multiple times as the state value for the component changes.

Q) Explain Controlled and Uncontrolled Form Components with example.

In most of the web applications, forms are required for taking user input, placing an order, booking tickets, etc.

We can make form elements interactive by setting a callback to the onChange prop. Form components listen to changes and they are fired when,

- The value of <input> or <textarea> changes
- Checked state of <input> changes
- Selected state of <input> changes

We should maintain a state for every form field.

We can create two different ways of forms in React:

1. Controlled component
2. Uncontrolled component

Controlled component: In HTML, form elements such as <input>, <textarea>, and <select> typically maintains their own state and that gets updated based on user input. But in React, we handle it using state and state gets updated using setState() method. So an input form element whose value is controlled by React in this way will be called a **controlled component.**

Below is an example of a controlled component where it has a value attribute and a handler function.

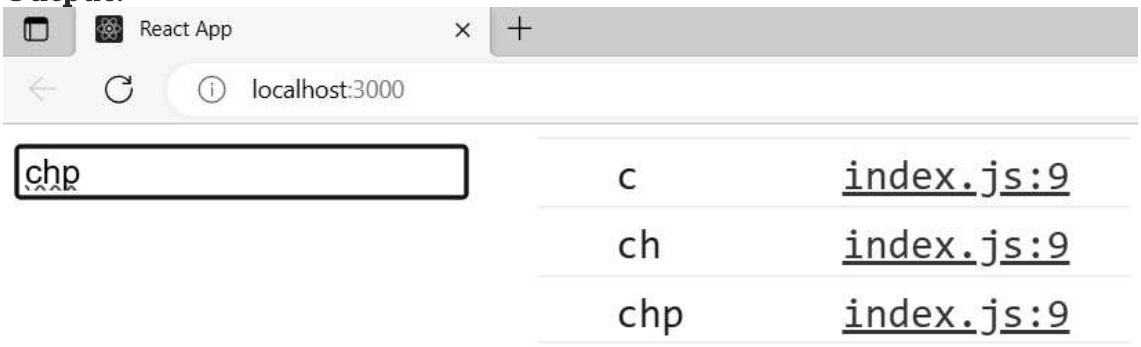
Eg.

<input type="text" value={this.state.name} onChange={this.handleData}/>

index.js

```
import {createRoot} from 'react-dom/client';
import React, { useState } from 'react';

function ControlledFormDemo() {
  const [value, setValue] = useState('');

  function handleChange(event) {
    setValue(event.target.value);
    console.log(event.target.value)

  }

  return (
    <input type="text" value={value} onChange={handleChange} />
  );
}

const root = createRoot(document.getElementById('root'));
root.render(<ControlledFormDemo />);
```

Output:



Eg.2:

```
import { useState } from 'react';
import ReactDOM from 'react-dom/client';

function MyForm() {
  const [inputs, setInputs] = useState({});

  const handleChange = (event) => {
    const name = event.target.name;
    const value = event.target.value;
    setInputs(values => ({...values, [name]: value}))

  }
```

```
  const handleSubmit = (event) => {
    event.preventDefault();
    console.log(inputs)
    alert(JSON.stringify(inputs));
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>Enter your name:
      <input
        type="text"
        name="username"
        value={inputs.username || ""}
        onChange={handleChange}
      />
      </label>
      <label>Enter your age:
        <input
          type="number"
          name="age"
          value={inputs.age || ""}
          onChange={handleChange}
        />
        </label>
        <input type="submit" />
    </form>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);
```

In above example, We control the values of more than one input field by adding a name attribute to each element. We initialize our state with an empty object.

To access the fields in the event handler, we use event.target.name and event.target.value .

To update the state, use square brackets [bracket notation] around the property name.

Output:

(a) Before clicking on submit

(b) After clicking on submit

Uncontrolled component: The traditional HTML input elements are uncontrolled component, where the form data is handled by DOM itself. In uncontrolled component, we do not use the value attributes.

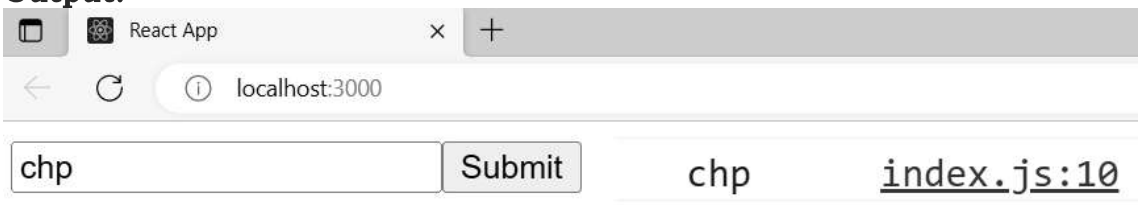
Below is the example for an uncontrolled component:

<input type="text" />
Here, input will not have value attribute and event handler, instead to get the value from the DOM **ref** is used as follows:
<input type="text" ref={this.input}/>

index.js

```
import {createRoot} from 'react-dom/client';
import React, { useRef } from 'react';

function UncontrolledFormEg() {
  const inputRef = useRef();

  function handleSubmit(event) {
    event.preventDefault();
    const value = inputRef.current.value;
    console.log(value);
  }

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" ref={inputRef} />
      <button type="submit">Submit</button>
    </form>
  );
}

const root = createRoot(document.getElementById('root'));
root.render(<UncontrolledFormEg />);
```

Output:



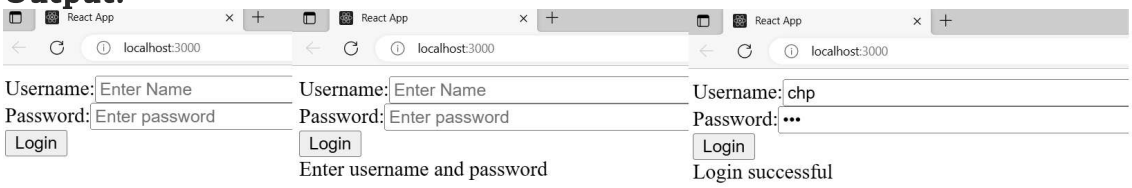
Q) Create a login page using forms in React.js

index.js

```
import {createRoot} from 'react-dom/client';
import React,{useState} from 'react';

const Login = () => {
    const [username,setUsername] = useState('')
    const [password,setPassword] = useState('')
    const [success,setSuccess] = useState('')
    const [error,setError]= useState('')

        const handleSubmit = (event) => {
        event.preventDefault()

      if(username===""||password==="") {
            setError("Enter username and password")
            setSuccess("")
        }
        else {
            setSuccess("Login successful")
            setError("")
        }
    }

  return (
    <>
      <div className="container">
        <form onSubmit={handleSubmit}>
          <div className="form-group">
            <label htmlFor="name">Username:</label>
            <input
              style={{ width: "40%" }}
              type="text"
              id="name"
              value={username}
              onChange={(event)=>setUsername(event.target.value)}
              className="form-control"
              placeholder="Enter Name"
            />
```

```
        </div>
        <div className="form-group">
          <label htmlFor="pwd">Password:</label>
          <input
            style={{ width: "40%" }}
            type="password"
            id="pwd"
            value={password}
            onChange={(event)=>setPassword(event.target.value)}
            className="form-control"
            placeholder="Enter password"
          />
        </div>
        <button type="submit" >
          Login
        </button>
        {success?<div className="text-success">{success}</div>:null}
        {error?<div className="text-danger">{error}</div>:null}
      </form>
    </div>
  </>
 );
};

const root = createRoot(document.getElementById('root'));
root.render(<Login />);
```

Output:



Q) Explain about data validation and sanitization with an example. (or) Explain how to create a newsfeed/post comment in a social network application.

For any social network application creating post is a basic component to express and share ideas. We add validation and sanitization to the post inorder to restrict the length of the post and remove bad words.

Data validation is the process of checking whether the data entered by a user is in the correct format or meets certain criteria before it is accepted.

Data sanitization is the process of cleaning and removing unwanted characters/words from data input.

Eg. index.js

```
import React from 'react';
import Filter from 'bad-words';
import {createRoot} from 'react-dom/client';

const filter = new Filter();

class CreatePost extends React.Component {

    constructor(props) {
        super(props);
        this.state = {
        content: '',
        valid: false,
    };
        this.handleSubmit = this.handleSubmit.bind(this);
        this.handlePostChange = this.handlePostChange.bind(this);
    }

    handlePostChange(event) {
    const content = filter.clean(event.target.value);

    this.setState(() => {
    return {
        content,
        valid: content.length <= 20
        };
    });
  }

  handleSubmit(){
    if (!this.state.valid) {
        return;
    }
    const newPost = {
    content: this.state.content,
    };
    console.log(this.state);
    }

    render() {
        return (
            <div className="createpost">
            <button onClick={this.handleSubmit}>Post</button>
            <textarea
                value={this.state.content}
                onChange={this.handlePostChange}
                placeholder="Post your content"
            />
            </div>
            );
        }
}
const root = createRoot(document.getElementById('root'));
root.render(<CreatePost />);
```

In the above example, we have kept a validation on the content that it is posted only if the length of the post is less than or equal to 20 characters. If the content contains any bad words like hell, sadist etc., they are sanitized i.e.; characters are changed to *.

Output:

Q) What is the need of Routing? Explain routing with an example.

As you know, these days all the web applications are single page applications (SPA) because in multi-page applications:

- Every request will be sent to the server from the client
- The Server responds to the client with new HTML content
- Every time page reload will happen for every request
- This would increase the round trips to the server and cause delay in response

SPA overcomes the limitations of multi-page application as described below:

- Rather than loading a new page from the server on every user interaction (such as clicking on the login button), it loads an entire web page containing all views from the server when the application starts
- As a result, after the initial page load, no server communication is required for further page updates upon user interaction

So here, we have to navigate from one view to another without hitting server. For this React JS provides the react-router-dom library.

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.

Defining routes is as simple as defining a single Route component for each route in your application and then putting all those Route components in a single Routes component. Whenever URL changes React Router will look at the routes defined in your Routes component and it will render the content in the element prop of the Route that has a path that matches the URL.

Eg.

Create Header,About,Home,ContactUs components within the directory "components" as shown below:

Header.js

```
import React from 'react';
import { Link, Outlet} from 'react-router-dom';
const Header = () => {
     return (<nav>
         <Link to="home">Home</Link>
         <Link to="about"> About Us </Link>
         <Link to="contact"> Contact Us </Link>
         <Outlet/>
       </nav>)
   }
export default Header;
```

About.js

```
const About = () => {
       return <>
              <h2> Inside About Us</h2>
          </>
   }
   export default About;
```

Contact.js

```
const ContactUs = () => {
       return <>
              <h2> Inside ContactUs</h2>
          </>
   }
export default ContactUs;
```

Home.js

```
const Home = () => {
    return <>
      <h2> Inside home</h2>
       </>
    }
 export default Home;
```

index.js

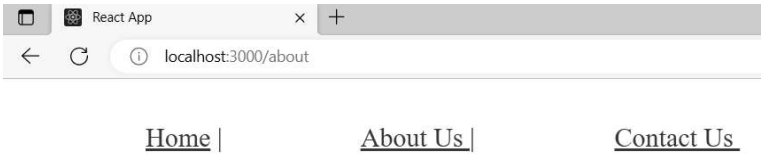
```
import React from 'react';
import ReactDOM from 'react-dom/client';
import {BrowserRouter as Router, Routes, Route } from "react-router-dom";
import Header from './components/Header';
import Home from './components/Home';
import About from './components/About';
import ContactUs from './components/Contact';
```

```
class RoutingDemo extends React.Component{
  render(){
  return(
    <Router>
      <Routes>
        <Route path="/" element={<Header />}>
          <Route index element={<Home />}/>
          <Route path="home" element={<Home />}/>
          <Route path="about" element={<About />}/>
          <Route path="contact" element={<ContactUs />}/>
        </Route>
      </Routes>
    </Router>
  )
 }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <RoutingDemo />
);
```

Output:

Inside home

(a) Displaying content from Home component when the application loads



Inside About Us

(b) Displaying the content from AboutUs component when clicking the link AboutUs

Inside ContactUs

(c) Displaying the content from ContactUs component when clicking the link ContactUs

Q) Explain Route Parameters with a suitable example.

When a user navigates to a specific URL in a single-page application, there may be parts of the URL that change dynamically based on user input or other factors. These dynamic parts of the URL are known as **parameters**. Parameters passed along with URL are called **route parameters**.

Eg.

<Link to="/display/React"> React </Link>

In the above code snippet, on click of React link, URL will be updated as localhost:3000/display/React.

To render a component, based on the URL changes route can be configured as follows:

<Route path="/display/:topic" component={Display}/>

In the above code snippet, the Display component gets rendered, only when a parameter is passed to the path.

Here, the topic is the route param.

In Display component, route param can be accessed as follows:

this.props.match.params.topic

match: react-router-dom passes in a prop called match into every route that is rendered. Inside this match object, we have another object called params.
params: It is an object containing URL parameters

Eg. index.js

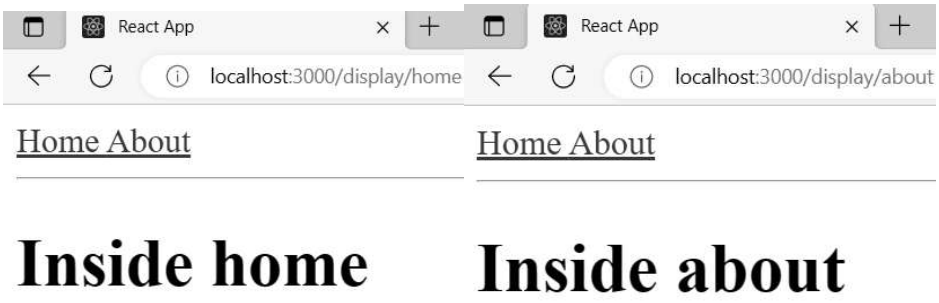
```
import React from "react";
import {createRoot} from 'react-dom/client';
import {BrowserRouter as Router,Routes,Route,Link} from "react-router-
dom";
import Display from "./components/Display";

class Demo extends React.Component {
  render(){
    return (
      <Router>
        <div>
            <Link to="/display/home">Home   </Link>
            <Link to="/display/about">  About</Link>
             <hr />
```

```
        <Routes>
          <Route path="/display/:page" element={<Display />} />
        </Routes>
      </div>
    </Router>
  );
  }
}

const root = createRoot(document.getElementById('root'));
root.render(<Demo />);
```

Display.js

```
import React from "react";
import { useParams } from "react-router-dom";
export default function Display() {
   const { page } = useParams();
    return (
         <h1>Inside {page}</h1>
    );
}
```

The **useParams()** hook is a built-in hook provided by the react-router-dom library in React. It is used to extract dynamic parameters from the current URL path.



Q) Explain Redux Architecture with a neat sketch.

Redux is an application data-flow architecture which is used for predicting state changes in JavaScript applications. Redux is not a framework. Redux can be used with any other View library but mostly used with React.

The main concepts of Redux architecture are:
Store: The place where the entire state of the application is stored
Actions: JS objects which encapsulate the events triggered within the application.

Reducers: Pure functions that modify the state of the application according to the actions triggered.
In Redux state is an immutable object. The state object cannot be changed directly in Redux. Triggering actions are the only way to modify the state. Every time the state is modified a new object should be returned which represents the modifications that are made to the state object.

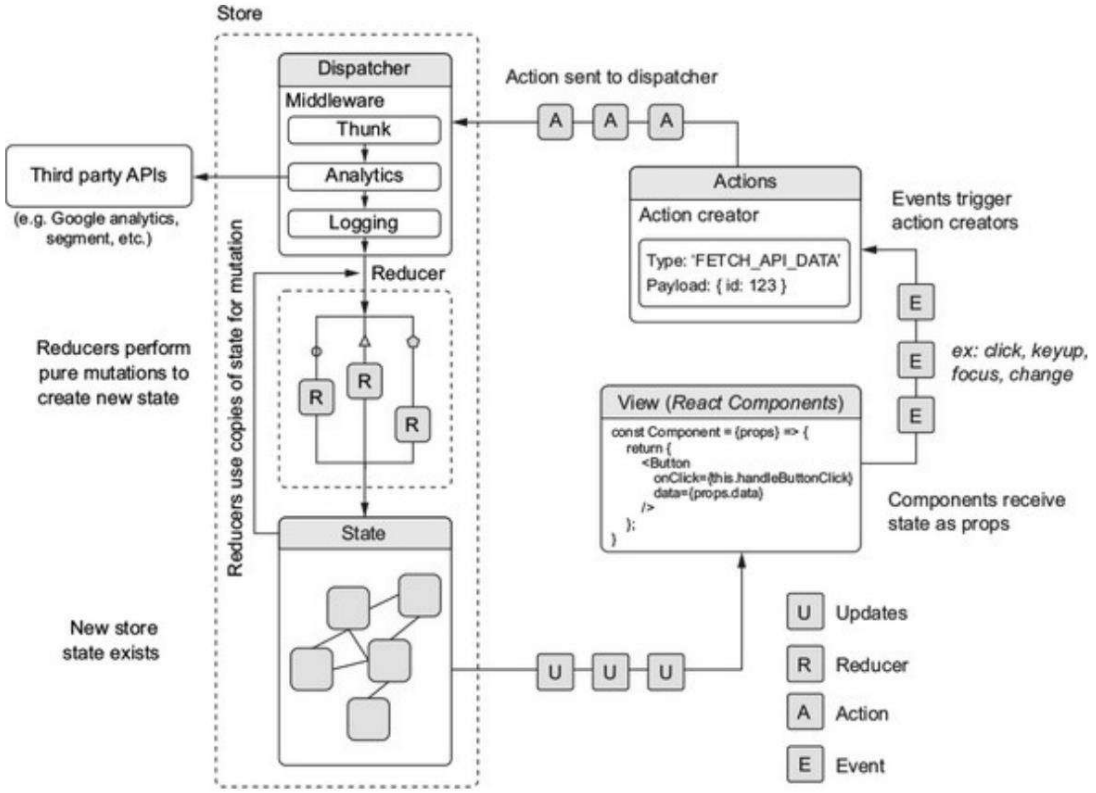


Fig. Overview of Redux Architecture

React Components: React components represent the UI rendered on the browser. Components would dispatch actions for events triggered within them and whenever the state changes the component renders the current state by connecting to the Redux store.
Action Creators: Action creators also called convenient functions that wrap the actual action object. It is not mandatory to define action creators to return the action, rather you can just use the action object for dispatching an action, but action creators helps us in achieving maintainability, abstraction, encapsulation, and testability.
Eg.

```
const login = (isAuthenticated) => {
    return {
        type: "LOGIN",
        isAuthenticated
    }
}
```

Action: Action is a plain JavaScript object. Action is a command to change the state when an event is triggered.

Middleware: Middleware is a mediator between the action and reducer. Its purpose is to intercept the actions before it reaches the reducer.

Reducer: Reducer changes the state of the application based on the action triggered. Reducer is a function that accepts action and current state and modifies the current state by creating a copy of it based on the action.
To achieve modularity, you can write multiple reducers in Redux.

Though there can be multiple reducers as your application grows, ultimately, you can pass only one reducer object to the createStore method. We can use the combineReducers() method of Redux to combine multiple reducers into a single reducing function and then pass it to the createStore method.

Whenever an action is triggered the action would reach the root reducer first and then all the reducers get the action from root reducer.

The root reducer will combine the state from all the reducers and create a single new state object and will update the store with the new state.

Eg.

```
import LoginReducer from './LoginReducer';
import CounterReducer from './CounterReducer';
import { combineReducers } from 'redux';

const rootReducer = combineReducers({
   CounterReducer,
   LoginReducer,
})
export default rootReducer;
```

Store: Store is responsible for managing the entire state of the application. State management is centralized in Redux. Action will be dispatched to the store using the dispatch method of the store.
A Store can be created as shown below

Eg.

```
import {createStore} from 'redux';
function myReducer() {
}
const store = createStore(myReducer)
```

The reducer is passed to the store. Since the state changes are done by reducer, the reducer would update the store with the updated state. Every time reducer changes the state it updates the store and the store would contain the latest state.

Store Methods

1. getState(): This method can be used to get the current state from the store.

2. dispatch(action): React components should use this method to dispatch an action whenever an event occurs within the component. This method dispatches an action and then the reducer takes care of updating the state

3. subscribe(listener): Used for registering the listeners

We don't have any API's to change the data in the store. The only way to change the state present in the store is by dispatching actions. The store would handle actions using reducers.

Q) Create a react-redux application that increments and decrements state value.

index.js

```
import React from "react";
import {createRoot} from 'react-dom/client';
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import App from './App';

const initialState = {
  count: 0
};

function reducer(state = initialState, action) {
  switch(action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    default:
      return state;
  }
}

const store = createStore(reducer);

const root = createRoot(document.getElementById('root'));
root.render(<Provider store={store}>
  <App />
</Provider>);
```

App.js

```
import React from 'react';
import { connect } from 'react-redux';

function Counter(props) {
  return (
```

```
    <div>
      <h1>Counter: {props.count}</h1>
      <button onClick={props.increment}>Increment</button>
      <button onClick={props.decrement}>Decrement</button>
    </div>
  );
}

function mapStateToProps(state) {
  return {
    count: state.count
  };
}

function mapDispatchToProps(dispatch) {
  return {
    increment: () => dispatch({ type: 'INCREMENT' }),
    decrement: () => dispatch({ type: 'DECREMENT' })
  };
}
export default connect(mapStateToProps, mapDispatchToProps)(Counter);
```

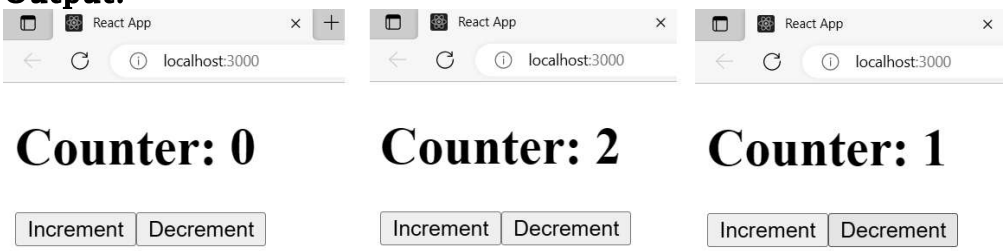
In the above example, the index.js file sets up the Redux store, while the App.js file defines the counter component and connects it to the store using the connect function.

The createStore() is used to create a store that holds the state of the application and the reducer function is passed to it. The Provider component is used to make the Redux store available to the rest of the application by passing it as a prop to the Provider component.

The mapStateToProps() takes the current state of the application as an argument and returns an object that contains the count property of the state.

The mapDispatchToProps() takes the dispatch function as an argument and returns an object that contains two functions: increment and decrement. These functions dispatch the INCREMENT and DECREMENT actions, respectively.

Output:



Q) Explain how to use middleware in react-redux applications with an example.

Redux thunk middleware allows the developer to write action creators which returns a function instead of an action object. The middleware can be used to delay the dispatch of action or dispatch the actions based on certain conditions.

Functionality that could not be included within the reducer, can be included in middleware.

The most common requirement to use middleware is to support asynchronous actions. Middleware lets you dispatch async actions in addition to your regular actions. Middleware lets you wrap the dispatch method of the store.

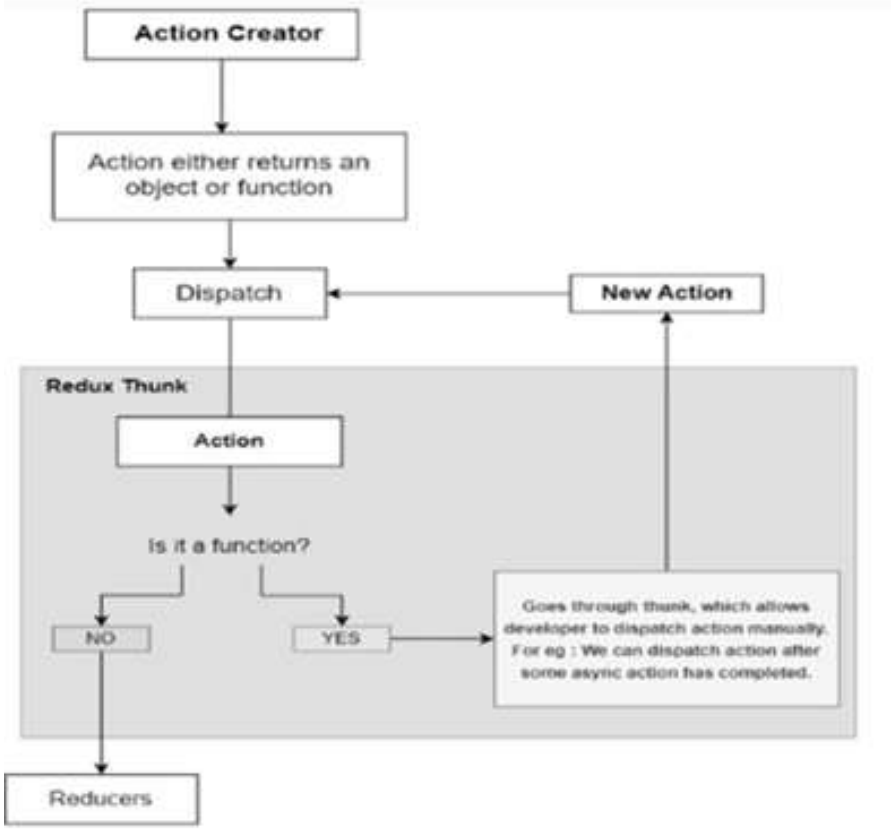


Fig. Redux with Middleware

Advantages of middleware:

- It provides functionality between action and reducer. After an action is triggered and before its reduced, middleware can take action.
- Middleware allows pre-processing of actions. We can check for correct syntax, make sure that they conform to your standards or edit them in some way like wrapping them in a function
- Perfect for debugging an application
- API calls are happening through middleware so if there is a mistake, you can check middleware first.

Eg. Create a react-redux application that increments and decrements state value using middleware thunk.

index.js

```
import React from "react";
import {createRoot} from 'react-dom/client';
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import { Provider } from 'react-redux';
import App from './App';

const initialState = {
  count: 0
};

function reducer(state = initialState, action) {
  switch(action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    default:
      return state;
  }
}
const store = createStore(reducer, applyMiddleware(thunk));

const root = createRoot(document.getElementById('root'));
root.render(<Provider store={store}>
  <App />
</Provider>);
```

App.js

```
import React from 'react';
import { connect } from 'react-redux';

function Counter(props) {
  return (
    <div>
      <h1>Counter: {props.count}</h1>
      <button onClick={props.increment}>Increment</button>
      <button onClick={props.decrement}>Decrement</button>
    </div>
  );
}
```

```
function mapStateToProps(state) {
  return {
    count: state.count
  };
}

function mapDispatchToProps(dispatch) {
  return {
    increment: () => dispatch((dispatch) => {
      setTimeout(() => {
        dispatch({ type: 'INCREMENT' });
      }, 1000);
    }),
    decrement: () => dispatch((dispatch) => {
      setTimeout(() => {
        dispatch({ type: 'DECREMENT' });
      }, 1000);
    })
  };
}
export default connect(mapStateToProps, mapDispatchToProps)(Counter);
```

Output:

Here the action creators return functions that take in the dispatch method as an argument. These functions are used to create a delay using setTimeout before dispatching the actual action object.

Q) Explain about mapbox library with an example.

Mapbox is a mapping and geoservices platform that provides an incredible variety of map and locationrelated services. You can customize maps with data, create different styles of maps and overlays, do geographic search, add navigation.

index.js

```
import React from "react";
import {createRoot} from 'react-dom/client';
import { useRef, useEffect } from 'react';
import mapboxgl from 'mapbox-gl';
```

```
mapboxgl.accessToken = 'Use your access token';

function Map() {
  const mapContainer = useRef(null);

  useEffect(() => {
    // Get the user's current location
    navigator.geolocation.getCurrentPosition(position => {
      const { latitude, longitude } = position.coords;

      // Initialize the map with the user's location as the center point
      const map = new mapboxgl.Map({
        container: mapContainer.current,
        style: 'mapbox://styles/mapbox/streets-v11',
        center: [longitude, latitude],
        zoom: 9
      });

      // Add a marker at the user's location
      new mapboxgl.Marker().setLngLat([longitude, latitude]).addTo(map);

      return () => map.remove();
    });
  }, []);

  return <div ref={mapContainer} />;
}
const root = createRoot(document.getElementById('root'));
root.render(
  <Map />
);
```

Output:

Here, the user's current location is retrieved using the navigator.geolocation.getCurrentPosition method, and initializes the map using the Mapbox GL library with the user's location as the center point. We also add a marker to the map at the user's location. Finally map is removed when the component unmounts.