Introduction to React

Example 1: Setting Up a React App

bash

```
npx create-react-app my-app

cd my-app
npm start
```

Explanation: This sets up a new React project using Create React App and starts the development server.

Example 2: JSX Syntax

```
function App() {

  return <h1>Hello, World!</h1>;
}
export default App;
```

Explanation: allows writing HTML-like syntax in JavaScript, which React transforms into React elements.

Example 3: Functional Component

```
function Welcome(props) {

  return <h1>Hello, {props.name}</h1>;
}
```

Explanation: A functional component is a JavaScript function that returns a React element.

Example 4: Class Component

```
class Welcome extends React.Component {

  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Explanation: A class component is a JavaScript class that extends React.Component and must include a rendermethod.

Example 5: Props

```
function Greeting(props) {

  return <h1>Hello, {props.name}!</h1>;
}

function App() {
  return <Greeting name="Alice" />;
}
export default App;
```

Explanation: Props are used to pass data from parent to child components.

2. State and Lifecycle

Example 1: State in Class Components

```
class Clock extends React.Component {

  constructor(props) {
    super(props);
```

```
    this.state = { date: new Date() };
  }

  render() {
    return <h2>It is {this.state.date.toLocaleTimeString()}.</h2>;
  }
}
```

Explanation: State is a JavaScript object managed within the component, often representing data that changes over time.

Example 2: useState Hook in Functional Components

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


function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}
```

Explanation: The useState hook allows functional components to have state.

Example 3: Component Lifecycle Methods

```
class Example extends React.Component {

  componentDidMount() {
    console.log("Component mounted!");
  }
```

```
  componentWillUnmount() {
    console.log("Component will unmount!");
  }

  render() {
    return <h1>Hello, World!</h1>;
  }
}
```

Explanation: Lifecycle methods are special methods in class components that run at different stages of the component's lifecycle.

Example 4: useEffect Hook

```
import React, { useEffect } from 'react';

function Example() {
  useEffect(() => {
    console.log("Component mounted or updated!");

    return () => {
      console.log("Component will unmount!");
    };
  }, []);

  return <h1>Hello, World!</h1>;
}
```

Explanation: The useEffect hook runs side effects in functional components.

Example 5: State Updates and Re-rendering

```
class Example extends React.Component {
```

```
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  increment = () => {
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    return (
      <div>
        <p>{this.state.count}</p>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
```

Explanation: State updates trigger re-rendering of the component with the new state.

3. Handling Events

Example 1: Handling Click Events

```
function Button() {

  function handleClick() {
    alert("Button clicked!");
  }

  return <button onClick={handleClick}>Click me</button>;
}
```

Explanation: Events in React are handled by adding event handlers like onClick to elements.

Example 2: Passing Arguments to Event Handlers

```
function Button() {

  function handleClick(id) {
    alert(`Button ${id} clicked!`);
  }

  return <button onClick={() => handleClick(1)}>Click me</button>;
}
```

Explanation: You can pass arguments to event handlers by using an arrow function.

Example 3: Preventing Default Behavior

```
function Link() {

  function handleClick(e) {
    e.preventDefault();
    alert("Link clicked!");
  }

  return <a href="#" onClick={handleClick}>Click me</a>;
}
```

Explanation: You can prevent the default behavior of events, like preventing a link from navigating.

Example 4: Handling Form Submissions

```
function Form() {

  function handleSubmit(e) {
    e.preventDefault();
    alert("Form submitted!");
```

```
  }

  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Submit</button>
    </form>
  );
}
```

Explanation: Handling form submissions involves preventing the default form behavior and processing the data.

Example 5: Binding 'this' in Class Components

```
class Button extends React.Component {

  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    alert("Button clicked!");
  }

  render() {
    return <button onClick={this.handleClick}>Click me</button>;
  }
}
```

Explanation: In class components, you often need to bind this to event handler methods.

4. Conditional Rendering

Example 1: Using if-else

```
function Greeting(props) {

  if (props.isLoggedIn) {
    return <h1>Welcome back!</h1>;
  } else {
    return <h1>Please sign up.</h1>;
  }
}
```

Explanation: Conditional rendering can be done using standard JavaScript control structures.

Example 2: Ternary Operator

```
function Greeting(props) {

  return (
    <h1>{props.isLoggedIn ? "Welcome back!" : "Please sign up."}</h1>
  );
}
```

Explanation: The ternary operator provides a concise way to conditionally render elements.

Example 3: Logical && Operator

```
function WarningBanner(props) {

  if (!props.warn) {
    return null;
  }

  return <div className="warning">Warning!</div>;
}
```

Explanation: The && operator can be used to render an element

conditionally based on a boolean expression.

Example 4: Inline If with Logical &&

```
function Message(props) {
  return (
    <div>
      {props.messages.length > 0 &&
        <h2>You have {props.messages.length} unread messages.</h2>
      }
    </div>
  );
}
```

Explanation: Only render the element if the condition is true.

Example 5: Preventing Component Rendering

```
function Greeting(props) {
  if (!props.isLoggedIn) {
    return null;
  }

  return <h1>Welcome back!</h1>;
}
```

Explanation: Returning null prevents rendering a component.

5. Lists and Keys

Example 1: Rendering a List

```
function NumberList(props) {
```

```
  const numbers = props.numbers;
  return (
    <ul>
      {numbers.map((number) => (
        <li key={number.toString()}>{number}</li>
      ))}
    </ul>
  );
}
```

Explanation: Use map() to create a list of React elements from an array.

Example 2: Keys in Lists

```
function ListItem(props) {
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  return (
    <ul>
      {numbers.map((number) => (
        <ListItem key={number.toString()} value={number} />
      ))}
    </ul>
  );
}
```

Explanation: Keys help React identify which items have changed, are added, or removed.

Example 3: Keys Must Be Unique

```
function ListWithKeys() {
```

```
  const items = ['a', 'b', 'c'];
  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>{item}</li>
      ))}
    </ul>
  );
}
```

Explanation: Keys should be unique among sibling elements.

Example 4: Using Keys for Components

```
function Blog(props) {
  const sidebar = (
    <ul>
      {props.posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
  const content = props.posts.map((post) => (
    <div key={post.id}>
      <h3>{post.title}</h3>
      <p>{post.content}</p>
    </div>
  ));
  return (
    <div>
      {sidebar}
      {content}
    </div>
  );
}
```

Explanation: Keys are used to give elements a stable identity.

Example 5: Extracting Components with Keys

```
function ListItem(props) {
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <ListItem key={number.toString()} value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

Explanation: When extracting components, keys should be kept on the elements inside the array.

6. Forms and Controlled Components

Example 1: Controlled Input Component

```
class NameForm 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('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" value={this.state.value}
onChange={this.handleChange} />
        </label>
        <button type="submit">Submit</button>
      </form>
    );
  }
}
```

Explanation: In a controlled component, form data is handled by a React component.

Example 2: Controlled Textarea

```
class EssayForm extends React.Component {

  constructor(props) {
    super(props);
    this.state = { value: 'Please write an essay about your favorite DOM
element.' };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({ value: event.target.value });
  }
```

```
  handleSubmit(event) {
    alert('An essay was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Essay:
          <textarea value={this.state.value}
onChange={this.handleChange} />
        </label>
        <button type="submit">Submit</button>
      </form>
    );
  }
}
```

Explanation: Textareas in controlled components also use the value attribute.

Example 3: Controlled Select

```
class FlavorForm extends React.Component {

  constructor(props) {
    super(props);
    this.state = { value: 'coconut' };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({ value: event.target.value });
  }

  handleSubmit(event) {
```

```
    alert('Your favorite flavor is: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Pick your favorite flavor:
          <select value={this.state.value} onChange={this.handleChange}>
            <option value="grapefruit">Grapefruit</option>
            <option value="lime">Lime</option>
            <option value="coconut">Coconut</option>
            <option value="mango">Mango</option>
          </select>
        </label>
        <button type="submit">Submit</button>
      </form>
    );
  }
}
```

Explanation: Select elements in controlled components use the value attribute.

Example 4: Handling Multiple Inputs

```
class Reservation extends React.Component {

  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      numberOfGuests: 2
    };

    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>
        <label>
          Is going:
          <input
            name="isGoing"
            type="checkbox"
            checked={this.state.isGoing}
            onChange={this.handleInputChange} />
        </label>
        <br />
        <label>
          Number of guests:
          <input
            name="numberOfGuests"
            type="number"
            value={this.state.numberOfGuests}
            onChange={this.handleInputChange} />
        </label>
      </form>
    );
  }
}
```

Explanation: For multiple controlled inputs, you can use a single change handler that identifies the target input by its name.

Example 5: Uncontrolled Components

```
class UncontrolledForm extends React.Component {

  constructor(props) {
    super(props);
    this.input = React.createRef();
  }

  handleSubmit = (event) => {
    event.preventDefault();
    alert('A name was submitted: ' + this.input.current.value);
  };

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" ref={this.input} />
        </label>
        <button type="submit">Submit</button>
      </form>
    );
  }
}
```

Explanation: Uncontrolled components use refs to access form values directly from the DOM.

7. Lifting State Up

Example 1: Shared State Example

```
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>The water would boil.</p>;
  }
  return <p>The water would not boil.</p>;
}

class Calculator extends React.Component {
  constructor(props) {
```

```
    super(props);
    this.state = { temperature: '' };

    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.setState({ temperature: e.target.value });
  }

  render() {
    const temperature = this.state.temperature;
    return (
      <fieldset>
        <legend>Enter temperature in Celsius:</legend>
        <input
          value={temperature}
          onChange={this.handleChange} />

        <BoilingVerdict
          celsius={parseFloat(temperature)} />
      </fieldset>
    );
  }
}
```

Explanation: Lifting state up involves moving state to a common ancestor component to share data among multiple child components.

8. Composition vs. Inheritance

Example 1: Containment

```
function FancyBorder(props) {

  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
```

```
}

function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        Welcome
      </h1>
      <p className="Dialog-message">
        Thank you for visiting our spacecraft!
      </p>
    </FancyBorder>
  );
}
```

Explanation: Composition involves containing components within other components using props.children.

Example 2: Specialization

```
function Dialog(props) {

  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
    </FancyBorder>
  );
}

function WelcomeDialog() {
  return (
    <Dialog
      title="Welcome"
      message="Thank you for visiting our spacecraft!" />
  );
```

```
}
```

Explanation: Specialization involves creating specialized components by passing specific props to a generic component.

Example 3: Composition with Props

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
      <div className="SplitPane-right">
        {props.right}
      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane
      left={<Contacts />}
      right={<Chat />} />
  );
}
```

Explanation: Composition can involve passing components as props to other components.

9. React Router

Example 1: Basic Routing

```
import { BrowserRouter as Router, Route, Link } from 'react-router-dom';


function Home() {
  return <h2>Home</h2>;
}

function About() {
  return <h2>About</h2>;
}

function App() {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/about">About</Link>
            </li>
          </ul>
        </nav>
        <Route path="/" exact component={Home} />
        <Route path="/about" component={About} />
      </div>
    </Router>
  );
}
```

Explanation: React Router is used for navigation in React applications.

Example 2: Route Parameters

```
import { BrowserRouter as Router, Route, Link } from 'react-router-dom';


function User({ match }) {
```

```
  return <h2>User ID: {match.params.id}</h2>;
}

function App() {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li>
              <Link to="/user/1">User 1</Link>
            </li>
            <li>
              <Link to="/user/2">User 2</Link>
            </li>
          </ul>
        </nav>
        <Route path="/user/:id" component={User} />
      </div>
    </Router>
  );
}
```

Explanation: Route parameters can be used to pass data through the URL.

Example 3: Nested Routes

```
import { BrowserRouter as Router, Route, Link } from 'react-router-dom';


function Topic({ match }) {
  return <h3>Requested topic ID: {match.params.topicId}</h3>;
}

function Topics({ match }) {
  return (
    <div>
      <h2>Topics</h2>
      <ul>
```

```
        <li>
          <Link to={`${match.url}/components`}>Components</Link>
        </li>
        <li>
          <Link to={`${match.url}/props-v-state`}>Props v. State</Link>
        </li>
      </ul>

      <Route path={`${match.path}/:topicId`} component={Topic} />
      <Route
        exact
        path={match.path}
        render={() => <h3>Please select a topic.</h3>}
      />
    </div>
  );
}

function App() {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li>
              <Link to="/topics">Topics</Link>
            </li>
          </ul>
        </nav>
        <Route path="/topics" component={Topics} />
      </div>
    </Router>
  );
}
```

Explanation: Nested routes allow defining routes within routes, enabling a hierarchical navigation structure.

Example 4: Redirects

```
import { BrowserRouter as Router, Route, Redirect } from 'react-router-dom';


function Home() {
  return <h2>Home</h2>;
}

function App() {
  return (
    <Router>
      <div>
        <Route path="/" exact component={Home} />
        <Route path="/old-home">
          <Redirect to="/" />
        </Route>
      </div>
    </Router>
  );
}
```

Explanation: Redirects can be used to navigate to a different route programmatically.

Example 5: Programmatic Navigation

```
import { BrowserRouter as Router, Route, useHistory } from 'react-router-dom';


function Home() {
  let history = useHistory();

  function handleClick() {
    history.push("/about");
  }

  return (
    <div>
      <h2>Home</h2>
```

```
      <button onClick={handleClick}>Go to About</button>
    </div>
  );
}

function About() {
  return <h2>About</h2>;
}

function App() {
  return (
    <Router>
      <Route path="/" exact component={Home} />
      <Route path="/about" component={About} />
    </Router>
  );
}
```

Explanation: Programmatic navigation allows navigating to different routes based on actions, like button clicks.

10. Context API

Example 1: Creating a Context

```
const ThemeContext = React.createContext('light');
```

Explanation: Create a context with a default value.

Example 2: Providing a Context

```
function App() {

  return (
    <ThemeContext.Provider value="dark">
      <Toolbar />
```

```
    </ThemeContext.Provider>
  );
}
```

Explanation: Use Provider to pass the context value down the component tree.

Example 3: Consuming a Context (Class Component)

```
class ThemedButton extends React.Component {
  static contextType = ThemeContext;
  render() {
    return <button theme={this.context}>Button</button>;
  }
}
```

Explanation: Access the context value in class components using contextType.

Example 4: Consuming a Context (Function Component)

```
function ThemedButton() {
  const theme = React.useContext(ThemeContext);
  return <button theme={theme}>Button</button>;
}
```

Explanation: Access the context value in functional components using useContext.

Example 5: Nested Contexts

```
const UserContext = React.createContext();

const ThemeContext = React.createContext();

function App() {
  return (
    <UserContext.Provider value="Alice">
      <ThemeContext.Provider value="dark">
        <Content />
      </ThemeContext.Provider>
    </UserContext.Provider>
  );
}

function Content() {
  const user = React.useContext(UserContext);
  const theme = React.useContext(ThemeContext);
  return (
    <div>
      <p>User: {user}</p>
      <p>Theme: {theme}</p>
    </div>
  );
}
```

Explanation: Multiple contexts can be nested and consumed by components.

11. React Hooks

Example 1: useState Hook

```
function Counter() {

  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
```

```
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}
```

Explanation: The useState hook lets you add state to functional components.

Example 2: useEffect Hook

```
function Example() {

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
}
```

Explanation: The useEffect hook lets you perform side effects in functional components.

Example 3: useContext Hook

```
const ThemeContext = React.createContext('light');


function ThemedButton() {
  const theme = useContext(ThemeContext);
  return <button style={{ background: theme }}>Button</button>;
}
```

Explanation: The useContext hook lets you access context values in functional components.

Example 4: useRef Hook

```
function TextInputWithFocusButton() {

  const inputEl = useRef(null);
  const onButtonClick = () => {
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </>
  );
}
```

Explanation: The useRef hook lets you create mutable references to DOM elements.

Example 5: useReducer Hook

```
function reducer(state, action) {

  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0 });
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
    </>
```

```
  );
}
```

Explanation: The useReducer hook is useful for complex state logic involving multiple sub-values or when the next state depends on the previous one.

12. Higher-Order Components (HOCs)

Example 1: Basic HOC

```
function withExtraProps(WrappedComponent) {

  return function EnhancedComponent(props) {
    return <WrappedComponent {...props} extra="Some extra prop" />;
  };
}
```

Explanation: HOCs are functions that take a component and return a new component with additional props or behavior.

Example 2: Conditional Rendering with HOC

```
function withAdminPermission(WrappedComponent) {

  return function EnhancedComponent(props) {
    if (props.isAdmin) {
      return <WrappedComponent {...props} />;
    } else {
      return <h2>Access Denied</h2>;
    }
  };
}
```

Explanation: HOCs can conditionally render components based on

props.

Example 3: Reusing Component Logic with HOC

```
function withLoadingIndicator(WrappedComponent) {
  return function EnhancedComponent({ isLoading, ...props }) {
    if (isLoading) {
      return <div>Loading...</div>;
    }
    return <WrappedComponent {...props} />;
  };
}
```

Explanation: HOCs can be used to add reusable logic to components.

13. Portals

Example 1: Basic Portal

```
import ReactDOM from 'react-dom';

function MyPortal() {
  return ReactDOM.createPortal(
    <div>
      This is rendered outside the main DOM hierarchy.
    </div>,
    document.getElementById('portal-root')
  );
}
```

Explanation: Portals allow rendering components outside the main DOM hierarchy.

14. Error Boundaries

Example 1: Error Boundary Component

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // Log the error to an error reporting service
    logErrorToMyService(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

Explanation: Error boundaries catch JavaScript errors in their child components and display a fallback UI.

15. Code Splitting

Example 1: Dynamic Import

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
```

```
    </div>
  );
}
```

Explanation: Code splitting allows loading components asynchronously, improving the app's performance.

16. Server-Side Rendering (SSR)

Example 1: Basic SSR Setup

```
import React from 'react';
import ReactDOMServer from 'react-dom/server';
import App from './App';

const express = require('express');
const app = express();

app.get('*', (req, res) => {
  const html = ReactDOMServer.renderToString(<App />);
  res.send(`
    <html>
      <head>
        <title>SSR with React</title>
      </head>
      <body>
        <div id="root">${html}</div>
        <script src="/bundle.js"></script>
      </body>
    </html>
  `);
});

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Explanation: SSR allows rendering React components on the server, providing better SEO and faster initial load times.

17. Testing in React

Example 1: Unit Testing with Jest

```
import { render } from '@testing-library/react';

import App from './App';

test('renders learn react link', () => {
  const { getByText } = render(<App />);
  const linkElement = getByText(/learn react/i);
  expect(linkElement).toBeInTheDocument();
});
```

Explanation: Jest is a testing framework used for unit testing React components.

Example 2: Snapshot Testing

```
import renderer from 'react-test-renderer';

import App from './App';

test('renders correctly', () => {
  const tree = renderer.create(<App />).toJSON();
  expect(tree).toMatchSnapshot();
});
```

Explanation: Snapshot testing captures the component's output and compares it with a reference snapshot.

Example 3: End-to-End Testing with Cypress

```
describe('My First Test', () => {
```

```
  it('Does not do much!', () => {
    cy.visit('https://Example.cypress.io');
    cy.contains('type').click();
    cy.url().should('include', '/commands/actions');
    cy.get('.action-email')
      .type('fake@email.com')
      .should('have.value', 'fake@email.com');
  });
});
```

Explanation: Cypress is used for end-to-end testing, simulating user interactions with the application.

1. and Rendering

Example 1: Conditional Rendering with Ternary Operator

```
function Greeting({ isLoggedIn }) {
  return (
    <div>
      {isLoggedIn ? <h1>Welcome back!</h1> : <h1>Please sign up.</
h1>}
    </div>
  );
}
```

Explanation: This Example demonstrates using the ternary operator for conditional rendering based on the isLoggedIn prop.

Example 2: Rendering Lists with Keys

```
function ItemList({ items }) {
```

```
  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>{item}</li>
      ))}
    </ul>
  );
}
```

Explanation: When rendering lists, it's important to use a unique key prop to help React identify which items have changed, are added, or are removed.

Example 3: Fragment Syntax

```
function FragmentExample() {
  return (
    <>
      <h1>Title</h1>
      <p>This is a description</p>
    </>
  );
}
```

Explanation: React Fragments (<></>) allow grouping a list of children without adding extra nodes to the DOM.

2. Components and Props

Example 1: Default Props

```
function Button({ color, text }) {
  return <button style={{ backgroundColor: color }}>{text}</button>;
}
```

```
Button.defaultProps = {
  color: 'blue',
  text: 'Click Me'
};
```

Explanation: Default props allow setting default values for props, which can be useful when a prop is not provided.

Example 2: Prop Types Validation

```
import PropTypes from 'prop-types';

function User({ name, age }) {
  return (
    <div>
      <h2>{name}</h2>
      <p>Age: {age}</p>
    </div>
  );
}

User.propTypes = {
  name: PropTypes.string.isRequired,
  age: PropTypes.number.isRequired
};
```

Explanation: Prop types help validate the props passed to a component, ensuring they match the expected types.

3. State and Lifecycle

Example 1: State Management in Class Components

```
class Counter extends React.Component {

  constructor(props) {
```

```
    super(props);
    this.state = { count: 0 };
  }

  increment = () => {
    this.setState(prevState => ({ count: prevState.count + 1 }));
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
```

Explanation: setState is used to update the component's state, and the state changes trigger a re-render.

Example 2: ComponentDidMount for Fetching Data

```
class DataFetcher extends React.Component {

  state = { data: null };

  componentDidMount() {
    fetch('https://api.Example.com/data')
      .then(response => response.json())
      .then(data => this.setState({ data }));
  }

  render() {
    return this.state.data ? <div>{this.state.data}</div> : <p>Loading...</
p>;
  }
}
```

Explanation: componentDidMount is a lifecycle method used for executing code, like fetching data, after the component is mounted.

4. Handling Events

Example 1: Passing Arguments to Event Handlers

```
function ListItem({ item, onDelete }) {
  return (
    <li>
      {item}
      <button onClick={() => onDelete(item)}>Delete</button>
    </li>
  );
}
```

Explanation: Event handlers can receive additional arguments by wrapping them in an arrow function.

Example 2: Handling Form Submissions

```
class Form extends React.Component {
  state = { value: '' };

  handleChange = (e) => {
    this.setState({ value: e.target.value });
  };

  handleSubmit = (e) => {
    e.preventDefault();
    console.log('Form submitted:', this.state.value);
  };

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
```

```
        <input type="text" value={this.state.value}
onChange={this.handleChange} />
        <button type="submit">Submit</button>
      </form>
    );
  }
}
```

Explanation: Handling form submissions involves preventing the default form behavior and using state to manage form data.

5. Conditional Rendering

Example 1: Inline If with Logical && Operator

```
function Mailbox({ unreadMessages }) {

  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>You have {unreadMessages.length} unread messages.</h2>}
    </div>
  );
}
```

Explanation: The logical && operator can be used for conditional rendering, where the second expression is rendered if the first is true.

Example 2: Preventing Component Rendering

```
function WarningBanner({ warn }) {

  if (!warn) {
    return null;
  }
```

```
  return <div className="warning">Warning!</div>;
}
```

Explanation: Returning null from a component's render method prevents it from rendering.

6. Lists and Keys

Example 1: Generating Components from Lists

```
function NumberList({ numbers }) {
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>{number}</li>
  );
  return <ul>{listItems}</ul>;
}
```

Explanation: Keys help React identify which items have changed, are added, or are removed, and should be unique among siblings.

Example 2: Controlled Components

```
class NameForm 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('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" value={this.state.value}
onChange={this.handleChange} />
        </label>
        <button type="submit">Submit</button>
      </form>
    );
  }
}
```

Explanation: In a controlled component, the form data is handled by the state, not the DOM.

7. Forms

Example 1: Handling Multiple Input

```
class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      numberOfGuests: 2
    };

    this.handleInputChange = this.handleInputChange.bind(this);
  }

  handleInputChange(event) {
    const target = event.target;
    const value = target.name === 'isGoing' ? target.checked :
target.value;
```

```
    const name = target.name;

    this.setState({
      [name]: value
    });
  }

  render() {
    return (
      <form>
        <label>
          Is going:
          <input
            name="isGoing"
            type="checkbox"
            checked={this.state.isGoing}
            onChange={this.handleInputChange} />
        </label>
        <br />
        <label>
          Number of guests:
          <input
            name="numberOfGuests"
            type="number"
            value={this.state.numberOfGuests}
            onChange={this.handleInputChange} />
        </label>
      </form>
    );
  }
}
```

Explanation: For handling multiple controlled inputs, you can add a name attribute to each input element and let the handler function choose what to do based on the value of event.target.name.

8. Lifecycle Methods

Example 1: componentDidUpdate

```
class Timer extends React.Component {
```

```
  constructor(props) {
    super(props);
    this.state = { seconds: 0 };
  }

  tick() {
    this.setState(state => ({
      seconds: state.seconds + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentDidUpdate(prevProps, prevState) {
    if (this.state.seconds !== prevState.seconds) {
      console.log(`Timer updated: ${this.state.seconds} seconds`);
    }
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>
        Seconds: {this.state.seconds}
      </div>
    );
  }
}
```

Explanation: componentDidUpdate is invoked immediately after updating occurs, allowing for additional data fetching or operations based on the change.

9. Hooks

Example 1: Custom Hook for Fetching Data

```
import { useState, useEffect } from 'react';


function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch(url)
      .then(response => response.json())
      .then(data => {
        setData(data);
        setLoading(false);
      });
  }, [url]);

  return { data, loading };
}
```

Explanation: Custom hooks allow you to extract and reuse logic. This useFetch hook handles data fetching.

Example 2: useReducer for Complex State

```
import { useReducer } from 'react';


const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
```

```
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
    </div>
  );
}
```

Explanation: useReducer is a hook that is used for state management, useful when state logic is complex.

10. Context API

Example 1: Context with Default Value

```
const ThemeContext = React.createContext('light');

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}
```

```
function ThemedButton() {
  const theme = React.useContext(ThemeContext);
  return <button style={{ background: theme === 'dark' ? '#333' : '#fff' }}
>Themed Button</button>;
}
```

Explanation: The Context API allows passing data through the component tree without having to pass props down manually at every level.

11. Refs

Example 1: Managing Focus

```
class TextInput extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }

  componentDidMount() {
    this.textInput.current.focus();
  }

  render() {
    return <input type="text" ref={this.textInput} />;
  }
}
```

Explanation: Refs provide a way to access DOM nodes or React elements created in the render method.

12. Higher-Order Components (HOCs)

Example 1: HOC to Add Logging

```
function withLogger(WrappedComponent) {
  return class extends React.Component {
```

```
    componentDidMount() {
      console.log(`Component ${WrappedComponent.name} mounted`);
    }

    componentWillUnmount() {
      console.log(`Component ${WrappedComponent.name} will
unmount`);
    }

    render() {
      return <WrappedComponent {...this.props} />;
    }
  };
}

const EnhancedComponent = withLogger(SomeComponent);
```

Explanation: HOCs can be used to wrap components and add functionality, such as logging lifecycle events.

13. Portals

Example 1: Modal Implementation with Portals

```
import ReactDOM from 'react-dom';

function Modal({ isOpen, children }) {
  if (!isOpen) return null;

  return ReactDOM.createPortal(
    <div className="modal">
      <div className="modal-content">{children}</div>
    </div>,
    document.getElementById('modal-root')
  );
}
```

Explanation: Portals are useful for rendering components that should not disrupt the parent component's styling, such as modals.

14. Error Boundaries

Example 1: Catching Errors in Specific Components

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    console.error('Error caught:', error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}

function MyApp() {
  return (
    <ErrorBoundary>
      <ComponentThatMayError />
    </ErrorBoundary>
  );
}
```

Explanation: Error boundaries can catch errors in the components below them in the tree, logging the error and displaying a fallback UI.

15. Code Splitting

Example 1: Using React.lazy for Code Splitting

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```

Explanation: React.lazy allows loading components lazily, which can help reduce the bundle size and improve the performance of your application.

16. Server-Side Rendering (SSR)

Example 1: SSR with Data Fetching

```
import express from 'express';
import React from 'react';
import ReactDOMServer from 'react-dom/server';
import App from './App';

const app = express();

app.get('*', (req, res) => {
  const initialData = fetchDataSomehow();
  const html = ReactDOMServer.renderToString(<App
initialData={initialData} />);
  res.send(`
    <html>
      <head>
        <title>SSR with React</title>
      </head>
      <body>
```

```
        <div id="root">${html}</div>
        <script>window.__INITIAL_DATA__ = ${JSON.stringify(initialData)}
</script>
        <script src="/bundle.js"></script>
      </body>
    </html>
  `);
});

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Explanation: SSR can be combined with data fetching to pre-populate the application state on the server, improving the initial load experience.

17. Testing in React

Example 1: Mocking Fetch Requests with Jest

```
import { render, screen } from '@testing-library/react';
import App from './App';

global.fetch = jest.fn(() =>
  Promise.resolve({
    json: () => Promise.resolve({ data: '12345' }),
  })
);

test('fetches and displays data', async () => {
  render(<App />);
  const dataElement = await screen.findByText(/12345/i);
  expect(dataElement).toBeInTheDocument();
});
```

Explanation: Mocking fetch requests in tests can simulate server responses, allowing you to test how components handle data fetching.