Core React Concepts

Components: Functional vs. Class Components

React applications are built using components, which are independent, reusable pieces of UI.

Functional Components

These are plain JavaScript functions that accept props as an argument and return JSX. With the introduction of Hooks, functional components can now manage state and lifecycle, making them the preferred choice for most new development.

Pros:

- **Simpler Syntax:** More concise and easier to read.
- **Better Performance (with Hooks):** React can optimize functional components more effectively.
- **Easier Testing:** Pure functions are generally easier to test.
- **Hooks:** Access to state and lifecycle features without classes.

Cons:

- Before Hooks, couldn't manage state or lifecycle methods directly (now largely mitigated).

Example:

JavaScript

```
// src/components/ProductCard.js

import React from 'react';


function ProductCard(props) {
  const { product } = props; // Destructuring props for cleaner access


  return (
    <div className="product-card">
      <img src={product.imageUrl} alt={product.name} />
      <h3>{product.name}</h3>
      <p>Price: ₹{product.price.toFixed(2)}</p>
      <button onClick={() => props.onAddToCart(product.id)}>Add to Cart</button>
    </div>
  );
}
```

export default ProductCard;

Class Components

These are ES6 classes that extend React.Component and have their own internal state and lifecycle methods. They were the original way to create stateful components in React.

Pros:

- **State and Lifecycle Methods:** Built-in mechanisms for managing state and reacting to component lifecycle events (before Hooks).

Cons:

- **More Boilerplate:** Require more code (constructor, render() method, this binding).
- **this Context Issues:** Can lead to confusion with this binding in event handlers.
- **Harder to Test:** Often involves mocking lifecycle methods.

Example:

JavaScript

```
// src/components/ProductDetails.js

import React from 'react';


class ProductDetails extends React.Component {
 constructor(props) {
  super(props);
  this.state = {
   quantity: 1,
   isLoading: true,
   productData: null,
   error: null,
  };
  this.handleQuantityChange = this.handleQuantityChange.bind(this); // Binding 'this'
 }


 // Lifecycle method: Called after the component mounts
 async componentDidMount() {
  try {
   const response = await fetch(`/api/products/${this.props.productId}`);
   if (!response.ok) {
```

```
      throw new Error(` HTTP error! status: ${response.status}` );
    }
    const data = await response.json();
    this.setState({ productData: data, isLoading: false });
  } catch (error) {
    this.setState({ error: error.message, isLoading: false });
  }
}

// Lifecycle method: Called after component updates
componentDidUpdate(prevProps, prevState) {
  // Example: Fetch new data if productId changes
  if (this.props.productId !== prevProps.productId) {
    // Logic to refetch data for new product
    console.log('Product ID changed, fetching new data...');
  }
}

// Lifecycle method: Called before component unmounts
componentWillUnmount() {
  // Clean up subscriptions or timers here
  console.log('ProductDetails component unmounting...');
}

handleQuantityChange(event) {
  this.setState({ quantity: parseInt(event.target.value) });
}

render() {
  const { productData, isLoading, error, quantity } = this.state;

  if (isLoading) {
```

```
    return <div>Loading product details...</div>;
  }

  if (error) {
    return <div>Error: {error}</div>;
  }

  return (
    <div className="product-details">
      <h2>{productData.name}</h2>
      <p>{productData.description}</p>
      <p>Price: ₹{productData.price.toFixed(2)}</p>
      <input
        type="number"
        min="1"
        value={quantity}
        onChange={this.handleQuantityChange}
      />
      <button onClick={() => this.props.onAddToCart(productData.id, quantity)}>
        Add to Cart
      </button>
    </div>
  );
 }
}

export default ProductDetails;
```

props (Properties)

props are read-only properties passed from a parent component to a child component. They are how data flows down the component tree.

- **Immutable:** Components should never modify their own props.
- **Communication:** Primary way for parent components to communicate with children.

Example (already seen in ProductCard and ProductDetails): In ProductCard, product and onAddToCart are props. In ProductDetails, productId and onAddToCart are props.

state

state is an object that holds data that can change over time within a component. When the state changes, the component re-renders.

- **Mutable (within the component):** Can be updated using setState in class components or useState hook in functional components.
- **Local to Component:** State is managed within the component where it's defined.

Example (already seen in ProductDetails class component): quantity, isLoading, productData, error are part of the component's state.

JSX (JavaScript XML)

JSX is a syntax extension for JavaScript that allows you to write HTML-like code directly within your JavaScript files. React uses JSX to describe what the UI should look like. It's not HTML or a string; it's syntactical sugar for React.createElement().

Example:

JavaScript

```
// A simple JSX expression
const element = <h1>Hello, Flipkart Customer!</h1>;
```

Embedding Expressions

You can embed any valid JavaScript expression inside JSX using curly braces {}. This includes variables, function calls, and more.

Example:

JavaScript

```
const userName = "Alice";
const greeting = <p>Welcome, {userName}!</p>; // Embedding a variable


function formatPrice(price) {
  return `₹${price.toFixed(2)}`;
}


const productPrice = 1299.50;
const priceDisplay = <span>Price: {formatPrice(productPrice)}</span>; // Embedding a function call
```

Conditional Rendering

Displaying different UI based on certain conditions.

1. **if statements (outside JSX):**

JavaScript

```
function UserGreeting(props) {
  if (props.isLoggedIn) {
    return <h1>Welcome back!</h1>;
  }
  return <h1>Please log in.</h1>;
}
```

2. **Ternary Operator (condition ? true : false) (inside JSX):**

JavaScript

```
function AuthButton(props) {
  return (
    <button>
      {props.isLoggedIn ? 'Logout' : 'Login'}
    </button>
  );
}
```

3. **Logical && Operator (for rendering nothing when false):**

JavaScript

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

List Rendering

Rendering lists of items typically involves using the map() array method to transform an array of data into an array of JSX elements. **Always provide a unique key prop** when rendering lists to help React efficiently update the UI.

Example:

```
JavaScript

function ProductList(props) {
  const products = [
   { id: 'p1', name: 'Laptop', price: 65000, imageUrl: '/images/laptop.jpg' },
   { id: 'p2', name: 'Smartphone', price: 30000, imageUrl: '/images/smartphone.jpg' },
   { id: 'p3', name: 'Headphones', price: 2500, imageUrl: '/images/headphones.jpg' },
  ];

  return (
   <div className="product-list">
    <h2>Featured Products</h2>
    {products.map(product => (
     // Key is crucial for performance and correct rendering
     <ProductCard key={product.id} product={product} onAddToCart={props.onAddToCart} />
    ))}
   </div>
  );
}
```

Virtual DOM

The **Virtual DOM (VDOM)** is a lightweight, in-memory representation of the actual **Document Object Model (DOM)**. React uses the Virtual DOM to optimize UI updates and improve performance.

How it works:

1. **Initial Render:** When a React component is rendered for the first time, React builds a Virtual DOM tree from your JSX and then uses this tree to construct the actual DOM.
2. **State/Prop Change:** When the state or props of a component change, React creates a **new Virtual DOM tree** representing the updated UI.
3. **Diffing (Reconciliation):** React's **"diffing algorithm"** then compares the new Virtual DOM tree with the previous one. This comparison identifies the minimal set of changes required to update the actual DOM.
4. **Batching Updates:** Instead of updating the real DOM for every single change, React batches multiple updates together.
5. **Actual DOM Update:** Finally, React applies only the identified minimal changes to the actual DOM. This is much faster than directly manipulating the entire DOM every time something changes.

Why it's efficient for large-scale applications like Flipkart:

- **Reduced DOM Manipulations:** Direct DOM manipulation is slow. By minimizing the number of actual DOM operations, React significantly boosts performance.
- **Batching:** Multiple state updates across different components can be batched into a single render cycle, reducing overhead.
- **Abstracted Away:** Developers don't need to manually optimize DOM updates; React handles it efficiently behind the scenes. This allows UI engineers to focus on application logic rather than imperative DOM manipulation.
- **Cross-Browser Compatibility:** React's Virtual DOM ensures consistent behavior across different browsers, abstracting away browser-specific DOM quirks.

Hooks

Hooks are functions that let you "hook into" React state and lifecycle features from functional components. They were introduced in React 16.8 to allow functional components to be just as powerful as class components, but with a simpler and more intuitive API.

useState

Allows functional components to have state. It returns a stateful value and a function to update it.

Purpose: Manage component-specific state. **When to use:** When you need a piece of data within your component to change and trigger a re-render.

Example: Simple Counter

JavaScript

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


function Counter() {
 const [count, setCount] = useState(0); // Initialize count to 0


 const increment = () => {
  setCount(count + 1);
 };


 const decrement = () => {
  setCount(prevCount => prevCount - 1); // Using functional update for safety
 };


 return (
  <div>
```

```
    <p>Count: {count}</p>
    <button onClick={increment}>Increment</button>
    <button onClick={decrement}>Decrement</button>
   </div>
  );
}

export default Counter;
```

useEffect

Enables functional components to perform side effects (data fetching, subscriptions, manual DOM manipulations) after every render. It acts as a combination of componentDidMount, componentDidUpdate, and componentWillUnmount for functional components.

Purpose: Handle side effects that occur after the component renders. **When to use:**

- Fetching data from an API.
- Setting up subscriptions (e.g., real-time chat, WebSocket).
- Manually changing the DOM.
- Timers (setTimeout, setInterval).

Syntax: useEffect(callback, [dependencies])

- callback: The function containing the side effect logic.
- [dependencies]: An optional array of values. The effect will re-run only if any of these values change between renders.
    - **Empty array []:** Effect runs once after the initial render (like componentDidMount).
    - **No array:** Effect runs after every render (like componentDidMount + componentDidUpdate).
    - **With dependencies:** Effect runs after the initial render and whenever any of the dependencies change.

Example: Data Fetching

JavaScript

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

function ProductFetcher({ productId }) {
  const [product, setProduct] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
```

```
useEffect(() => {
  const fetchProduct = async () => {
    setLoading(true);
    setError(null);
    try {
      const response = await fetch(` /api/products/${productId}` );
      if (!response.ok) {
        throw new Error('Failed to fetch product.');
      }
      const data = await response.json();
      setProduct(data);
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  };

  fetchProduct();

  // Cleanup function: runs before the component unmounts or before the effect re-runs
  return () => {
    // For example, cancel ongoing fetch requests or clear timers
    console.log('Cleaning up ProductFetcher effect for:', productId);
  };
}, [productId]); // Re-run effect when productId changes

if (loading) return <div>Loading product...</div>;
if (error) return <div>Error: {error}</div>;
if (!product) return <div>No product found.</div>;

return (
```

```
    <div>
      <h3>{product.name}</h3>
      <p>{product.description}</p>
      <p>Price: ₹{product.price.toFixed(2)}</p>
    </div>
  );
}

export default ProductFetcher;
```

useContext

Allows functional components to consume values from a React Context.

Purpose: Avoid "prop drilling" (passing props down through many levels of nested components) for global data like themes, user authentication status, or language settings. **When to use:** When you have data that needs to be accessible by many components at different nesting levels.

Example: Theme Toggle

JavaScript

```
// src/contexts/ThemeContext.js
import React, { createContext, useState, useContext } from 'react';

// 1. Create a Context
export const ThemeContext = createContext(null);

// 2. Create a Provider Component
export const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState('light');

  const toggleTheme = () => {
    setTheme(prevTheme => (prevTheme === 'light' ? 'dark' : 'light'));
  };

  const contextValue = { theme, toggleTheme };

  return (
```

```
    <ThemeContext.Provider value={contextValue}>
      {children}
    </ThemeContext.Provider>
  );
};

// src/components/ThemedButton.js
// 3. Consume the Context
function ThemedButton() {
  const { theme, toggleTheme } = useContext(ThemeContext);

  const buttonStyle = {
    backgroundColor: theme === 'light' ? '#eee' : '#333',
    color: theme === 'light' ? '#333' : '#eee',
    padding: '10px 20px',
    border: 'none',
    borderRadius: '5px',
    cursor: 'pointer',
  };

  return (
    <button style={buttonStyle} onClick={toggleTheme}>
      Toggle Theme ({theme})
    </button>
  );
}

export default ThemedButton;

// src/App.js
import React from 'react';
import { ThemeProvider } from './contexts/ThemeContext';
```

```
import ThemedButton from './components/ThemedButton';


function App() {
  return (
    <ThemeProvider>
      <div style={{ padding: '20px', border: '1px solid #ccc' }}>
        <h1>My Flipkart App</h1>
        <ThemedButton />
        <p>This paragraph will also react to the theme if it consumed the context.</p>
      </div>
    </ThemeProvider>
  );
}
```

useRef

Returns a mutable ref object whose .current property is initialized to the passed argument (initialValue). The returned object will persist for the full lifetime of the component.

Purpose:

- Accessing the DOM elements directly (e.g., focusing an input, playing media).
- Storing a mutable value that doesn't cause a re-render when it changes (unlike useState).

When to use:

- Managing focus, text selection, or media playback.
- Triggering imperative animations.
- Integrating with third-party DOM libraries.
- Storing any mutable value that needs to persist across renders but doesn't need to trigger a re-render.

Example: Focus Input

JavaScript

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


function FocusInput() {
  const inputRef = useRef(null); // Initialize with null


  const handleClick = () => {
```

```
  // Access the DOM element through .current
  inputRef.current.focus();
 };

 return (
  <div>
   <input type="text" ref={inputRef} />
   <button onClick={handleClick}>Focus Input</button>
  </div>
 );
}

export default FocusInput;
```

useCallback

Returns a memoized callback function. It's useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders.

Purpose: Prevent unnecessary re-renders of child components that receive callbacks as props. **When to use:** When you pass a callback function to a child component that is wrapped in React.memo (or is a class component implementing shouldComponentUpdate).

Example:

JavaScript

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

// Child component that re-renders only if its props change
const Button = memo(({ onClick, children }) => {
 console.log('Button rendered:', children);
 return <button onClick={onClick}>{children}</button>;
});

function ParentComponent() {
 const [count, setCount] = useState(0);
 const [text, setText] = useState('');
```

```
  // This function will only be re-created if 'count' changes
  const handleClick = useCallback(() => {
    setCount(prevCount => prevCount + 1);
  }, [count]); // Dependency array: only recreate if count changes

  // This function will be recreated on every render if not memoized
  const handleInputChange = (e) => {
    setText(e.target.value);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <input type="text" value={text} onChange={handleInputChange} placeholder="Type something..." />
      {/* Button component will only re-render when handleClick reference changes (i.e., when count changes) */}
      <Button onClick={handleClick}>Increment Count</Button>
      <Button onClick={() => console.log('Another button clicked')}>Another Button</Button>
    </div>
  );
}

export default ParentComponent;
```

useMemo

Returns a memoized value. It's useful for optimizing expensive computations by preventing them from being re-calculated on every render if their dependencies haven't changed.

Purpose: Optimize performance by caching the result of an expensive computation. **When to use:** When you have a complex calculation that takes time to complete and its result is only dependent on specific values.

Example: Expensive Calculation

JavaScript

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

function ProductFilter({ products }) {
  const [searchTerm, setSearchTerm] = useState('');
```

```
const [showInStock, setShowInStock] = useState(false);

// Expensive calculation: filtering products
const filteredProducts = useMemo(() => {
 console.log('Filtering products...'); // This will only log when dependencies change
 let result = products;

 if (searchTerm) {
  result = result.filter(product =>
   product.name.toLowerCase().includes(searchTerm.toLowerCase())
  );
 }

 if (showInStock) {
  result = result.filter(product => product.inStock);
 }

 return result;
}, [products, searchTerm, showInStock]); // Dependencies: re-calculate if these change

return (
 <div>
  <input
   type="text"
   placeholder="Search products..."
   value={searchTerm}
   onChange={(e) => setSearchTerm(e.target.value)}
  />
  <label>
   <input
    type="checkbox"
    checked={showInStock}
```

```
        onChange={(e) => setShowInStock(e.target.checked)}
      />
      Show In Stock
    </label>

    <h2>Filtered Products:</h2>
    <ul>
     {filteredProducts.map(product => (
       <li key={product.id}>
         {product.name} - ₹{product.price.toFixed(2)} {product.inStock ? '(In Stock)' : ''}
       </li>
     ))}
    </ul>
   </div>
  );
}

export default ProductFilter;

// Example usage in App.js
/*
const allProducts = [
 { id: 'a1', name: 'Laptop', price: 65000, inStock: true },
 { id: 'a2', name: 'Mouse', price: 800, inStock: true },
 { id: 'a3', name: 'Keyboard', price: 1500, inStock: false },
 { id: 'a4', name: 'Monitor', price: 12000, inStock: true },
];

function App() {
 return <ProductFilter products={allProducts} />;
}
*/
```

Context API

The Context API provides a way to pass data through the component tree without having to pass props down manually at every level (known as "prop drilling"). It's ideal for "global" data that many components in your application might need, such as:

- User authentication status
- Theme (light/dark mode)
- Preferred language
- Shopping cart details (in a simpler application)

How it works:

1. **createContext:** Creates a Context object.
2. **Provider:** A React component that allows consuming components to subscribe to context changes. It accepts a value prop to be passed to its descendants.
3. **useContext (or Consumer for class components):** Used by child components to read the context value.

When to use: For application-wide data that rarely changes or for prop drilling that becomes unwieldy. For complex state management with frequent updates, consider libraries like Redux or Zustand.

Example (already seen in useContext section): The ThemeContext and ThemeProvider demonstrate how to use the Context API for theme management.

Higher-Order Components (HOCs) and Render Props

These are advanced React patterns for **code reusability** and sharing logic between components, especially before the widespread adoption of Hooks. While Hooks are now often preferred for their simpler approach, understanding HOCs and Render Props is valuable, especially when working with older codebases or specific design requirements.

Higher-Order Components (HOCs)

A HOC is a **function that takes a component as an argument and returns a new component** with enhanced functionality. It's a pattern derived from higher-order functions in functional programming.

Purpose: Add common behavior or data to multiple components without duplicating code. **When to use:**

- Authentication (e.g., withAuth HOC to check user login status).
- Logging.
- Data fetching (e.g., withData HOC to fetch data and pass it as props).
- Theming.

Example: withAuth HOC

JavaScript

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

```
// HOC: Takes a Component and returns a new Component
function withAuth(WrappedComponent) {
  return function AuthenticatedComponent(props) {
    const [isAuthenticated, setIsAuthenticated] = useState(false);
    const [isLoading, setIsLoading] = useState(true);

    useEffect(() => {
      // Simulate checking authentication status
      setTimeout(() => {
        const token = localStorage.getItem('authToken');
        setIsAuthenticated(!!token); // Convert to boolean
        setIsLoading(false);
      }, 1000);
    }, []);

    if (isLoading) {
      return <div>Checking authentication...</div>;
    }

    if (!isAuthenticated) {
      return <div>Please log in to access this page.</div>;
    }

    // Render the wrapped component with its original props and any new props from HOC
    return <WrappedComponent {...props} userRole="admin" />;
  };
}

// Component to be wrapped
function Dashboard(props) {
  return (
```

```
    <div>
      <h2>Welcome to the Dashboard!</h2>
      <p>Your user ID: {props.userId}</p>
      <p>Your role: {props.userRole}</p>
    </div>
  );
}

// Export the enhanced component
export const AuthenticatedDashboard = withAuth(Dashboard);

// Usage: <AuthenticatedDashboard userId="user123" />
```

Pros:

- **Separation of Concerns:** Logic is cleanly separated from the UI.
- **Reusability:** Easily apply the same logic to many components.
- **No Prop Drilling:** Can inject props directly.

Cons:

- **Prop Collisions:** Can lead to prop name clashes if HOCs inject props with the same names as existing props.
- **Complicated Debugging:** Nested HOCs can create a "wrapper hell" in the React DevTools, making it harder to trace component hierarchy and data flow.
- **Implicit Dependencies:** It's not always immediately obvious what props a HOC adds to the wrapped component.

Render Props

The Render Props pattern involves passing a **function as a prop** to a component, and that function dictates what the component should render. The component with the render prop passes its internal state or logic to the render prop function, which then uses that data to render JSX.

Purpose: Share code between components using a prop whose value is a function. **When to use:**

- Sharing dynamic behavior or data (e.g., mouse position tracking, feature toggles).
- When the rendering logic is highly dynamic and depends on the internal state of the reusable component.

Example: DataLoader with Render Prop

JavaScript

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

```
// Component that encapsulates data fetching logic
function DataLoader(props) {
  const { url, render } = props;
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      setLoading(true);
      setError(null);
      try {
        const response = await fetch(url);
        if (!response.ok) {
          throw new Error('Failed to fetch data.');
        }
        const json = await response.json();
        setData(json);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    };
    fetchData();
  }, [url]);

  // Call the render prop with the data, loading, and error states
  return render({ data, loading, error });
}

// Usage of DataLoader with different render props
```

```
function ProductDisplay() {
  return (
    <DataLoader
      url="/api/products/1"
      render={({ data, loading, error }) => {
        if (loading) return <p>Loading product data...</p>;
        if (error) return <p>Error: {error}</p>;
        if (!data) return <p>No product data.</p>;
        return (
          <div>
            <h3>Product: {data.name}</h3>
            <p>Description: {data.description}</p>
          </div>
        );
      }}
    />
  );
}

function UserProfileDisplay() {
  return (
    <DataLoader
      url="/api/users/profile"
      render={({ data, loading, error }) => {
        if (loading) return <p>Loading user profile...</p>;
        if (error) return <p>Error loading profile.</p>;
        if (!data) return <p>No profile data.</p>;
        return (
          <div>
            <h3>User: {data.name}</h3>
            <p>Email: {data.email}</p>
          </div>
```

```
      );
    }}
  />
 );
}


// In your App component:
/*
function App() {
 return (
  <div>
   <ProductDisplay />
   <hr />
   <UserProfileDisplay />
  </div>
 );
}
*/
```

Pros:

- **Explicit Data Flow:** Clear what data is being passed from the DataLoader to the consumer.
- **Flexibility:** Provides great flexibility in how the shared logic is rendered.
- **Avoids Prop Collisions:** No risk of prop name clashes as you define the arguments to the render prop function.

Cons:

- **Nested JSX:** Can lead to deeply nested JSX structures, sometimes called "callback hell" or "render prop hell," which can reduce readability.
- **Performance:** Creating new inline functions for the render prop on every render can sometimes lead to unnecessary re-renders of the child component if not optimized with React.memo and useCallback.

HOCs vs. Render Props (Flipkart UI Engineer context):

- **HOCs** are more suited for cross-cutting concerns like authentication, logging, or adding common props to many components. They *enhance* components.
- **Render Props** are better when the shared logic directly influences *how* a component renders, especially for dynamic rendering based on internal state (e.g., a MouseTracker component passing x, y coordinates to a render prop to display a custom cursor).

2. State Management

State management refers to how an application handles, stores, and updates its data, and how that data is passed around to the parts of the UI that need it. Effective state management is crucial for large-scale applications like Flipkart, ensuring data consistency and predictable behavior.

Local Component State: useState or this.state

This is the simplest form of state management, where data is confined to a single component.

- **useState (for Functional Components):** This Hook is the modern way to add state to functional components.
    - **Purpose:** Manages data that is relevant only to the component itself and directly affects its rendering.
    - **Example Use Case (Flipkart):**
        - The **quantity** of an item selected on a product details page.
        - The **visibility** of a modal (e.g., "Add to Cart" confirmation).
        - The **search input value** within a search bar component.

JavaScript

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


function QuantitySelector() {
  const [quantity, setQuantity] = useState(1); // Local state for quantity


  const handleIncrement = () => setQuantity(prev => prev + 1);
  const handleDecrement = () => setQuantity(prev => (prev > 1 ? prev - 1 : 1));


  return (
    <div>
      <button onClick={handleDecrement}>-</button>
      <span>{quantity}</span>
      <button onClick={handleIncrement}>+</button>
      <p>Selected Quantity: {quantity}</p>
    </div>
  );
}
```

- **this.state (for Class Components):** The traditional way to manage state in class components. State is initialized in the constructor and updated using this.setState().
    - **Purpose:** Same as useState, but within the class component paradigm.
    - **Example:** A toggle for a filter section's expanded/collapsed state.

JavaScript

```
import React from 'react';


class FilterSection extends React.Component {
 constructor(props) {
  super(props);
  this.state = {
   isExpanded: false, // Local state for expansion
  };
 }


 toggleExpand = () => {
  this.setState(prevState => ({
   isExpanded: !prevState.isExpanded,
  }));
 };


 render() {
  return (
   <div>
    <h3 onClick={this.toggleExpand}>
     Filters {this.state.isExpanded ? '▲' : '▼'}
    </h3>
    {this.state.isExpanded && (
     <ul>
      <li>Price</li>
      <li>Brand</li>
      <li>Rating</li>
```

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

Global State Management: Redux, Zustand, Jotai

For data that needs to be accessible across many components or has complex update logic, **global state management** solutions are employed. These prevent "prop drilling" and offer centralized control.

Redux

Redux is a predictable state container for JavaScript apps. It follows a strict **unidirectional data flow**. It's widely used in large, complex applications due to its robust ecosystem and debugging tools.

- **Core Concepts:**
  - **Store:** A single JavaScript object that holds the entire application's global state. It's immutable.
  - **Actions:** Plain JavaScript objects that describe *what happened*. They are the only way to trigger a state change.
    - Example: { type: 'ADD_TO_CART', payload: { productId: 'p123', quantity: 1 } }
  - **Reducers:** Pure functions that take the current state and an action as arguments, and return a *new* state. They must not mutate the original state.
  - **Dispatch:** The method used to send an action to the store. store.dispatch(action).
  - **Middleware:** Extensible layer between dispatching an action and the reducer receiving it. Used for side effects (e.g., API calls with Redux-Thunk or Redux-Saga), logging, crash reporting.
- **Redux Flow:**
  1. **User Interaction:** A user clicks "Add to Cart."
  2. **Dispatch Action:** A component dispatches an ADD_TO_CART action.
  3. **Middleware (Optional):** If configured, middleware intercepts the action (e.g., Redux-Thunk handles asynchronous API calls before passing the action to reducers).
  4. **Reducer Update:** The appropriate reducer receives the action, creates a *new* state object based on the action's type and payload, and returns it.
  5. **Store Update:** The Redux store is updated with the new state.
  6. **Component Re-render:** Connected React components detect the state change and re-render with the new data.
- **Trade-offs (for Flipkart UI Engineer):**
  - **Pros:**
    - **Centralized State:** Single source of truth makes debugging easier.

        - **Predictable State Changes:** Strict rules make state changes transparent and testable.
        - **Powerful DevTools:** Redux DevTools offer time-travel debugging and replay features, invaluable for complex UIs.
        - **Scalability:** Well-suited for large applications with many interconnected data points.
        - **Rich Ecosystem:** Plenty of middleware and helper libraries.
    - **Cons:**
        - **Boilerplate:** Requires a significant amount of setup code (actions, reducers, constants) even for simple state. (mitigated by Redux Toolkit).
        - **Learning Curve:** Concepts like immutability, pure functions, and middleware can be challenging for beginners.
        - **Overkill for Simple Apps:** Unnecessary complexity for smaller applications.

- **Example (Simplified Cart using Redux Toolkit):**

JavaScript

```
// src/store/cartSlice.js
import { createSlice } from '@reduxjs/toolkit';

const cartSlice = createSlice({
 name: 'cart', // Slice name
 initialState: {
  items: [],
  totalItems: 0,
  totalPrice: 0,
 },
 reducers: {
  // Reducer functions directly mutate state (immer handles immutability under the hood)
  addItemToCart: (state, action) => {
   const newItem = action.payload;
   const existingItem = state.items.find(item => item.id === newItem.id);

   if (existingItem) {
    existingItem.quantity += newItem.quantity;
   } else {
    state.items.push(newItem);
```

```
    }
    state.totalItems += newItem.quantity;
    state.totalPrice += newItem.price * newItem.quantity;
   },
   removeItemFromCart: (state, action) => {
    const idToRemove = action.payload;
    const itemToRemove = state.items.find(item => item.id === idToRemove);

    if (itemToRemove) {
     state.totalItems -= itemToRemove.quantity;
     state.totalPrice -= itemToRemove.price * itemToRemove.quantity;
     state.items = state.items.filter(item => item.id !== idToRemove);
    }
   },
   clearCart: (state) => {
    state.items = [];
    state.totalItems = 0;
    state.totalPrice = 0;
   },
  },
});

export const { addItemToCart, removeItemFromCart, clearCart } = cartSlice.actions;
export default cartSlice.reducer;

// src/store/index.js
import { configureStore } from '@reduxjs/toolkit';
import cartReducer from './cartSlice';

const store = configureStore({
  reducer: {
   cart: cartReducer, // Combine reducers
```

```
  // other: otherReducer,
 },
 // Middleware is automatically set up by configureStore (e.g., Redux Thunk)
});

export default store;

// src/index.js (or App.js)
import React from 'react';
import ReactDOM from 'react-dom/client';
import { Provider } from 'react-redux';
import store from './store';
import CartComponent from './components/CartComponent';
import ProductList from './components/ProductList'; // Assuming this dispatches add to cart

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
 <Provider store={store}>
  <ProductList /> {/* Products can dispatch addItemToCart */}
  <CartComponent /> {/* Displays cart state */}
 </Provider>
);

// src/components/CartComponent.js
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { removeItemFromCart, clearCart } from '../store/cartSlice';

function CartComponent() {
 // Select data from the Redux store
 const cartItems = useSelector(state => state.cart.items);
 const totalItems = useSelector(state => state.cart.totalItems);
```

```
  const totalPrice = useSelector(state => state.cart.totalPrice);
  const dispatch = useDispatch();

  return (
    <div style={{ border: '1px solid #ccc', padding: '10px', marginTop: '20px' }}>
      <h2>Shopping Cart</h2>
      <p>Total Items: {totalItems}</p>
      <p>Total Price: ₹{totalPrice.toFixed(2)}</p>
      {cartItems.length === 0 ? (
        <p>Your cart is empty.</p>
      ) : (
        <ul>
          {cartItems.map(item => (
            <li key={item.id}>
              {item.name} x {item.quantity} - ₹{(item.price * item.quantity).toFixed(2)}
              <button onClick={() => dispatch(removeItemFromCart(item.id))}>Remove</button>
            </li>
          ))}
        </ul>
      )}
      <button onClick={() => dispatch(clearCart())} disabled={cartItems.length === 0}>
        Clear Cart
      </button>
    </div>
  );
}

export default CartComponent;
```

Alternatives: Zustand, Jotai

These are more lightweight and often simpler alternatives to Redux, particularly good for local-first, global state or for scenarios where Redux's full power isn't needed. They typically offer less boilerplate.

- **Zustand:** A small, fast, and scalable bear-necessities state management solution. It uses React Hooks for a more direct and less opinionated API.
    - **Trade-offs:**

        - **Pros:** Minimal boilerplate, intuitive API, good performance, no need for Provider components.
        - **Cons:** Less strict on data flow compared to Redux, which can be less predictable in very large teams without careful conventions. Smaller ecosystem than Redux.
    - **Example (Cart with Zustand):**

JavaScript

```
// src/store/useCartStore.js
import { create } from 'zustand';

// Create a store
const useCartStore = create((set) => ({
 items: [],
 totalItems: 0,
 totalPrice: 0,
 addItem: (newItem) => set((state) => {
  const existingItem = state.items.find(item => item.id === newItem.id);
  if (existingItem) {
   existingItem.quantity += newItem.quantity;
  } else {
   state.items.push(newItem);
  }
  return {
   items: [...state.items], // Ensure immutability for re-render detection
   totalItems: state.totalItems + newItem.quantity,
   totalPrice: state.totalPrice + (newItem.price * newItem.quantity),
  };
 }),
 removeItem: (idToRemove) => set((state) => {
  const itemToRemove = state.items.find(item => item.id === idToRemove);
  if (itemToRemove) {
   return {
    items: state.items.filter(item => item.id !== idToRemove),
    totalItems: state.totalItems - itemToRemove.quantity,
```

```
      totalPrice: state.totalPrice - (itemToRemove.price * itemToRemove.quantity),
    };
   }
   return state;
  }),
  clearCart: () => set({ items: [], totalItems: 0, totalPrice: 0 }),
}));

export default useCartStore;

// src/components/CartComponentZustand.js
import React from 'react';
import useCartStore from '../store/useCartStore';

function CartComponentZustand() {
  // Select only the parts of the state you need
  const { items, totalItems, totalPrice, removeItem, clearCart } = useCartStore();

  return (
    <div style={{ border: '1px solid #007bff', padding: '10px', marginTop: '20px' }}>
      <h2>Shopping Cart (Zustand)</h2>
      <p>Total Items: {totalItems}</p>
      <p>Total Price: ₹{totalPrice.toFixed(2)}</p>
      {items.length === 0 ? (
        <p>Your cart is empty.</p>
      ) : (
        <ul>
          {items.map(item => (
            <li key={item.id}>
              {item.name} x {item.quantity} - ₹{(item.price * item.quantity).toFixed(2)}
              <button onClick={() => removeItem(item.id)}>Remove</button>
            </li>
```

```
        ))}
      </ul>
    )}
    <button onClick={clearCart} disabled={items.length === 0}>
      Clear Cart
    </button>
  </div>
 );
}
export default CartComponentZustand;
```

- **Jotai:** A primitive and flexible state management library for React. It focuses on "atoms" – independent, isolated pieces of state – providing a fine-grained, bottom-up approach.
    - **Trade-offs:**
        - **Pros:** Extremely lightweight, minimal re-renders, excellent performance, highly flexible and composable.
        - **Cons:** Can be less intuitive for beginners when dealing with derived state or complex asynchronous flows compared to Redux's explicit middleware pattern. Newer, so smaller community resources.
    - **Example (Counter with Jotai):**

JavaScript

```
// src/store/atoms.js
import { atom } from 'jotai';

export const countAtom = atom(0);
export const doubledCountAtom = atom((get) => get(countAtom) * 2);

// src/components/CounterJotai.js
import React from 'react';
import { useAtom } from 'jotai';
import { countAtom, doubledCountAtom } from '../store/atoms';

function CounterJotai() {
  const [count, setCount] = useAtom(countAtom);
  const [doubledCount] = useAtom(doubledCountAtom); // Read-only atom
```

```
  return (
    <div style={{ border: '1px solid #28a745', padding: '10px', marginTop: '20px' }}>
      <h2>Counter (Jotai)</h2>
      <p>Count: {count}</p>
      <p>Doubled Count: {doubledCount}</p>
      <button onClick={() => setCount(c => c + 1)}>Increment</button>
      <button onClick={() => setCount(c => c - 1)}>Decrement</button>
    </div>
  );
}

export default CounterJotai;
```

- **Use Cases and Trade-offs for Flipkart:**
    - For a massive application like Flipkart, **Redux (with Redux Toolkit)** is often the default choice due to its robustness, extensive tooling for debugging, and strong conventions that help large teams maintain consistency. Critical data like user sessions, global product filters, and cart state would likely reside in Redux.
    - **Zustand or Jotai** could be excellent choices for more isolated, less "global" pieces of state, or for new features where rapid development and minimal boilerplate are priorities, especially if the data structure is simple or more local to a feature module. For instance, managing the state of a complex filter sidebar within a specific product listing page might be a good fit for Zustand/Jotai if that state doesn't need to be accessed by other, disparate parts of the application.

---

3. React Router

React Router is the standard library for client-side routing in React applications. It enables single-page applications (SPAs) to have multiple "pages" or views without requiring a full page reload, providing a seamless user experience.

Navigation: Setting up Routes, Nested Routes, Programmatic Navigation

- **Setting up Routes:** You define paths (URLs) and the components that should render when those paths are matched. BrowserRouter is commonly used for web applications. Routes (or Switch in older versions) ensures only one route matches at a time. Route defines the path and the element to render.

JavaScript

```
// src/App.js

import React from 'react';

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

import HomePage from './pages/HomePage';
```

```
import ProductsPage from './pages/ProductsPage';
import ProductDetailsPage from './pages/ProductDetailsPage';
import AboutPage from './pages/AboutPage';
import NotFoundPage from './pages/NotFoundPage'; // For 404

function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Home</Link> |{' '}
        <Link to="/products">Products</Link> |{' '}
        <Link to="/about">About Us</Link>
      </nav>
      <hr />
      <Routes>
        {/* Basic Routes */}
        <Route path="/" element={<HomePage />} />
        <Route path="/products" element={<ProductsPage />} />
        {/* Route with a parameter */}
        <Route path="/products/:productId" element={<ProductDetailsPage />} />
        <Route path="/about" element={<AboutPage />} />
        {/* Catch-all route for 404 */}
        <Route path="*" element={<NotFoundPage />} />
      </Routes>
    </Router>
  );
}

export default App;

// Example Page Components (simplified)
// src/pages/HomePage.js
```

```
function HomePage() { return <h2>Welcome to Flipkart!</h2>; }

// src/pages/ProductsPage.js

function ProductsPage() { return <h2>All Products</h2>; }

// src/pages/AboutPage.js

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

// src/pages/NotFoundPage.js

function NotFoundPage() { return <h2>404 - Page Not Found</h2>; }
```

- **Nested Routes:** Allows you to define routes within other routes, creating hierarchical UI structures. The parent route renders an Outlet component where nested routes will be rendered.

JavaScript

```
// src/pages/DashboardPage.js

import React from 'react';

import { Outlet, Link } from 'react-router-dom';


function DashboardPage() {

 return (

  <div>

   <h2>User Dashboard</h2>

   <nav>

    <Link to="profile">Profile</Link> |{' '}{/* Relative path */}

    <Link to="orders">My Orders</Link>

   </nav>

   <hr />

   {/* Outlet renders matched nested route's element */}

   <Outlet />

  </div>

 );

}


function UserProfile() { return <h3>User Profile Settings</h3>; }

function UserOrders() { return <h3>Your Recent Orders</h3>; }
```

```
// Update App.js Routes:
/*
<Routes>
 <Route path="/" element={<HomePage />} />
 <Route path="/dashboard" element={<DashboardPage />}>
  <Route path="profile" element={<UserProfile />} />  { If path is empty, it makes a default route for parent route}
  <Route path="orders" element={<UserOrders />} />
  <Route index element={<UserProfile />} /> {/* Default child route for /dashboard }
 </Route>
 <Route path="*" element={<NotFoundPage />} />
</Routes>
*/
```

- **Programmatic Navigation:** Using the useNavigate Hook (or history.push in older versions) to navigate imperatively (e.g., after a form submission, or a successful login).

```
JavaScript
import React, { useState } from 'react';
import { useNavigate } from 'react-router-dom';

function LoginForm() {
 const [username, setUsername] = useState('');
 const [password, setPassword] = useState('');
 const navigate = useNavigate(); // Get the navigate function

 const handleSubmit = (e) => {
  e.preventDefault();
  // Simulate login
  if (username === 'test' && password === 'password') {
   console.log('Login successful!');
   navigate('/dashboard'); // Navigate to dashboard on success
  } else {
   alert('Invalid credentials');
  }
```

```
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" placeholder="Username" value={username} onChange={(e) =>
setUsername(e.target.value)} />
      <input type="password" placeholder="Password" value={password} onChange={(e) =>
setPassword(e.target.value)} />
      <button type="submit">Login</button>
    </form>
  );
}
export default LoginForm;
```

Route Parameters: Handling Dynamic URLs

Route parameters allow you to capture dynamic segments from the URL. They are defined using a colon (:) followed by the parameter name in the path.

- **Accessing Parameters:** Use the useParams Hook in functional components.

JavaScript

```
// src/pages/ProductDetailsPage.js
import React from 'react';
import { useParams } from 'react-router-dom';

function ProductDetailsPage() {
  const { productId } = useParams(); // Get the productId from the URL

  // In a real app, you'd fetch product details using this productId
  const product = {
    'p123': { name: 'Samsung Galaxy F14', price: 12999, description: '5G Smartphone with Exynos 1330' },
    'p456': { name: 'LG 4K Smart TV', price: 45000, description: 'Stunning visuals with AI ThinQ' },
  }[productId]; // Simplified lookup

  if (!product) {
    return <h2>Product Not Found</h2>;
```

```
  }

  return (
    <div>
      <h2>{product.name}</h2>
      <p>Price: ₹{product.price.toFixed(2)}</p>
      <p>{product.description}</p>
    </div>
  );
}
export default ProductDetailsPage;

// Example URL: /products/p123
```

---

4. Performance Optimization

For a high-traffic e-commerce platform like Flipkart, performance is paramount. Slow loading times or unresponsive UIs directly impact user experience and conversion rates.

React.memo and useMemo/useCallback: Preventing Unnecessary Re-renders

These techniques are forms of **memoization**, which means caching the result of a function or component render and re-using it if the inputs haven't changed. This prevents costly re-computations or re-renders.

- **React.memo (for Components):** A Higher-Order Component that memoizes a functional component. It prevents a functional component from re-rendering if its props have not changed. It's like shouldComponentUpdate for functional components.

JavaScript

```
// src/components/ProductThumbnail.js
import React from 'react';

// This component will only re-render if its 'product' prop changes shallowly
const ProductThumbnail = React.memo(({ product, onClick }) => {
  console.log('Rendering ProductThumbnail:', product.name);
  return (
    <div style={{ border: '1px solid #eee', padding: '10px', margin: '5px' }} onClick={() => onClick(product.id)}>
      <img src={product.imageUrl} alt={product.name} width="100" />
      <p>{product.name}</p>
```

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

export default ProductThumbnail;

// Usage in ParentComponent:
/*
import React, { useState } from 'react';
import ProductThumbnail from './ProductThumbnail';

function ProductListingPage({ products }) { // products prop could come from API
 const [sortBy, setSortBy] = useState('price'); // State that changes often

 // This function reference would change on every render if not memoized, causing ProductThumbnail to re-render
 const handleProductClick = React.useCallback((productId) => {
  console.log('Clicked product:', productId);
  // Navigate or show product details
 }, []); // Dependency array: only recreate if needed

 return (
  <div>
   <h2>Browse Products</h2>
   <button onClick={() => setSortBy(sortBy === 'price' ? 'name' : 'price')}>
    Sort by {sortBy}
   </button>
   <div style={{ display: 'flex', flexWrap: 'wrap' }}>
    {products.map(product => (
     // If product prop doesn't change, ProductThumbnail won't re-render
     <ProductThumbnail key={product.id} product={product} onClick={handleProductClick} />
    ))}
```

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

- **useMemo (for Values):** Memoizes a computed value. It only re-computes the value when one of its dependencies changes.
    - **Purpose:** Avoid expensive calculations on every render.
    - **Example:** Calculating a heavily filtered/sorted product list, or parsing large data sets.

JavaScript

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

function ProductAggregator({ allProducts }) {
 const [minPrice, setMinPrice] = useState(0);

 // This expensive filter operation only runs when allProducts or minPrice changes
 const affordableProducts = useMemo(() => {
  console.log('Calculating affordable products...');
  return allProducts.filter(p => p.price >= minPrice);
 }, [allProducts, minPrice]); // Dependencies

 return (
  <div>
   <h3>Affordable Products (Min Price: {minPrice})</h3>
   <input
    type="range"
    min="0"
    max="10000"
    value={minPrice}
    onChange={(e) => setMinPrice(Number(e.target.value))}
   />
   <ul>
```

```
      {affordableProducts.map(p => (
        <li key={p.id}>{p.name} - ₹{p.price}</li>
      ))}
    </ul>
  </div>
 );
}
```

- **useCallback (for Functions):** Memoizes a callback function. It returns a memoized version of the callback that only changes if one of the dependencies has changed.
    - **Purpose:** Prevent unnecessary re-renders of child components that receive functions as props, especially those wrapped in React.memo.
    - **Example:** Passing an event handler to a memoized child component. (See ProductListingPage example above).

Lazy Loading and Code Splitting: Optimizing Bundle Size

For large applications like Flipkart, the JavaScript bundle size can become very large, leading to slow initial page loads. **Code splitting** allows you to split your code into smaller chunks that can be loaded on demand (lazy loading).

- **React.lazy:** A function that lets you render a dynamic import as a regular component. It automatically handles the loading of the component and suspense for the loading state.
- **Suspense:** A component that lets you "wait" for some code to load and declaratively specify a loading indicator (fallback) while it's loading.

How it works: Instead of loading all components upfront, you only load them when they are actually needed (e.g., when a user navigates to a specific route).

JavaScript

```
// src/App.js (Updated)
import React, { Suspense, lazy } from 'react';
import { BrowserRouter as Router, Routes, Route, Link } from 'react-router-dom';

// Lazy load components for better performance
const HomePage = lazy(() => import('./pages/HomePage'));
const ProductsPage = lazy(() => import('./pages/ProductsPage'));
const ProductDetailsPage = lazy(() => import('./pages/ProductDetailsPage'));
const AboutPage = lazy(() => import('./pages/AboutPage'));
const UserProfilePage = lazy(() => import('./pages/UserProfilePage'));
const CheckoutPage = lazy(() => import('./pages/CheckoutPage'));
```

```
function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Home</Link> |{' '}
        <Link to="/products">Products</Link> |{' '}
        <Link to="/about">About Us</Link> |{' '}
        <Link to="/profile">Profile</Link> |{' '}
        <Link to="/checkout">Checkout</Link>
      </nav>
      <hr />
      {/* Suspense fallback for showing a loading indicator while lazy components load */}
      <Suspense fallback={<div>Loading page...</div>}>
        <Routes>
          <Route path="/" element={<HomePage />} />
          <Route path="/products" element={<ProductsPage />} />
          <Route path="/products/:productId" element={<ProductDetailsPage />} />
          <Route path="/about" element={<AboutPage />} />
          <Route path="/profile" element={<UserProfilePage />} />
          <Route path="/checkout" element={<CheckoutPage />} />
          <Route path="*" element={<div>404 Not Found</div>} />
        </Routes>
      </Suspense>
    </Router>
  );
}

export default App;
```

Benefits for Flipkart:

- **Faster Initial Load:** Users see content quicker, even if the entire application isn't loaded yet.
- **Reduced Bandwidth Usage:** Only download code that's needed for the current view.

- **Improved User Experience:** A snappier and more responsive feel.

Keys in Lists: Importance for Efficient Rendering

When rendering lists of elements in React, the key prop is **absolutely essential**.

- **Purpose:** keys help React identify which items have changed, are added, or are removed. They provide a stable identity to each item in the list.
- **How it Works:** When a list updates, React uses the keys to efficiently reconcile the old list with the new list. Without keys, or with non-unique/unstable keys (like array indices), React has to re-render more components than necessary, leading to performance issues and potential bugs (e.g., incorrect state being associated with wrong items).
- **Rules for Keys:**
    - **Unique:** Each key must be unique among its siblings in the list.
    - **Stable:** The key should remain the same for a given item across re-renders.
    - **Avoid Array Index (if possible):** Using array indices (index) as keys is generally discouraged if the list items can change order, be added, or be removed, as this breaks stability and can lead to bugs. Use a stable ID from your data (e.g., product.id, user.uuid).

Example (Correct Key Usage):

JavaScript

```
function ProductGrid({ products }) {
 return (
  <div className="product-grid">
   {products.map(product => (
    // Use a unique and stable ID from your data as the key
    <div key={product.id} className="product-item">
     <img src={product.imageUrl} alt={product.name} />
     <h4>{product.name}</h4>
     <p>₹{product.price.toFixed(2)}</p>
    </div>
   ))}
  </div>
 );
}
```

---

5. Testing

Testing is a critical part of the software development lifecycle, especially for large and complex applications like Flipkart, to ensure correctness, prevent regressions, and build confidence in the codebase.

Unit Testing: Jest and React Testing Library

Unit testing focuses on testing individual, isolated units of code, typically components or pure functions, in isolation from the rest of the application.

- **Jest:** A popular JavaScript testing framework developed by Facebook. It's often used as the test runner, assertion library, and mocking library.
- **React Testing Library (RTL):** A library that encourages good testing practices by focusing on testing components the way users would interact with them. It provides utilities that query and assert DOM elements.

Philosophy of RTL: Test your components *as if* a user is interacting with them. Don't test internal component state or implementation details.

- **Example (Testing a simple Button component):**

JavaScript

```
// src/components/Button.js
import React from 'react';

function Button({ onClick, children, disabled }) {
  return (
    <button onClick={onClick} disabled={disabled}>
      {children}
    </button>
  );
}
export default Button;

// src/components/Button.test.js
import { render, screen, fireEvent } from '@testing-library/react';
import '@testing-library/jest-dom'; // For extended matchers like .toBeInTheDocument()
import Button from './Button';

test('renders button with correct text', () => {
  render(<Button onClick={() => {}}>Click Me</Button>);
  const buttonElement = screen.getByText(/Click Me/i); // Case-insensitive text match
  expect(buttonElement).toBeInTheDocument();
});
```

```
test('calls onClick prop when clicked', () => {
  const handleClick = jest.fn(); // Mock function
  render(<Button onClick={handleClick}>Submit</Button>);
  const buttonElement = screen.getByText(/Submit/i);

  fireEvent.click(buttonElement); // Simulate a click event

  expect(handleClick).toHaveBeenCalledTimes(1); // Assert mock function was called
});

test('button is disabled when disabled prop is true', () => {
  render(<Button onClick={() => {}} disabled={true}>Disabled Button</Button>);
  const buttonElement = screen.getByText(/Disabled Button/i);
  expect(buttonElement).toBeDisabled();
});
```

Integration Testing: Testing Interactions Between Components

Integration testing verifies that different modules or services of an application work together correctly. In React, this means testing how multiple components interact with each other, often by rendering a parent component and interacting with its children.

- **Approach with RTL:** RTL naturally lends itself to integration testing because it interacts with the DOM. By rendering a higher-level component that composes several smaller components, you can test their combined behavior.
- **Example (Testing ProductSearch and ProductList interaction):**

JavaScript

```
// src/components/ProductSearch.js
import React, { useState } from 'react';

function ProductSearch({ onSearch }) {
  const [term, setTerm] = useState('');
  const handleChange = (e) => setTerm(e.target.value);
  const handleSubmit = (e) => {
    e.preventDefault();
    onSearch(term);
```

```
  };
  return (
    <form onSubmit={handleSubmit}>
      <input type="text" value={term} onChange={handleChange} placeholder="Search products..." />
      <button type="submit">Search</button>
    </form>
  );
}
export default ProductSearch;

// src/components/ProductList.js
import React from 'react';
// Assume ProductCard is imported from earlier examples

function ProductList({ products }) {
  return (
    <div data-testid="product-list">
      {products.map(product => (
        <div key={product.id} data-testid="product-item">{product.name}</div> // Simplified
      ))}
    </div>
  );
}
export default ProductList;

// src/components/SearchableProductPage.js (Integration component)
import React, { useState, useEffect } from 'react';
import ProductSearch from './ProductSearch';
import ProductList from './ProductList';

const ALL_PRODUCTS = [
  { id: '1', name: 'Laptop', price: 50000 },
```

```
  { id: '2', name: 'Mouse', price: 500 },
  { id: '3', name: 'Keyboard', price: 1000 },
];

function SearchableProductPage() {
  const [filteredProducts, setFilteredProducts] = useState(ALL_PRODUCTS);

  const handleSearch = (searchTerm) => {
    const lowerCaseSearchTerm = searchTerm.toLowerCase();
    setFilteredProducts(
      ALL_PRODUCTS.filter(p => p.name.toLowerCase().includes(lowerCaseSearchTerm))
    );
  };

  return (
    <div>
      <ProductSearch onSearch={handleSearch} />
      <ProductList products={filteredProducts} />
    </div>
  );
}
export default SearchableProductPage;

// src/components/SearchableProductPage.test.js (Integration Test)
import { render, screen, fireEvent, waitFor } from '@testing-library/react';
import '@testing-library/jest-dom';
import SearchableProductPage from './SearchableProductPage';

test('filters product list based on search input', async () => {
  render(<SearchableProductPage />);

  // Check initial state (all products rendered)
```

```
  expect(screen.getAllByTestId('product-item')).toHaveLength(3);
  expect(screen.getByText(/Laptop/i)).toBeInTheDocument();
  expect(screen.getByText(/Mouse/i)).toBeInTheDocument();
  expect(screen.getByText(/Keyboard/i)).toBeInTheDocument();

  // Find the search input and type into it
  const searchInput = screen.getByPlaceholderText(/Search products.../i);
  fireEvent.change(searchInput, { target: { value: 'laptop' } });

  // Find and click the search button
  const searchButton = screen.getByRole('button', { name: /Search/i });
  fireEvent.click(searchButton);

  // Assert that only 'Laptop' is displayed
  await waitFor(() => {
    const productItems = screen.getAllByTestId('product-item');
    expect(productItems).toHaveLength(1);
    expect(screen.getByText(/Laptop/i)).toBeInTheDocument();
    expect(screen.queryByText(/Mouse/i)).not.toBeInTheDocument(); // Use queryBy to check for absence
  });
});
```

---

6. Advanced Topics (Good to know for larger applications)

These topics become increasingly important as a React application grows in size and complexity, especially for a platform like Flipkart that demands high performance, robustness, and maintainability.

Server-Side Rendering (SSR) / Next.js: For Improved Performance and SEO

Server-Side Rendering (SSR) is a technique where the initial rendering of a React application happens on the server, and the complete HTML is sent to the client. This differs from client-side rendering (CSR) where the browser downloads a minimal HTML file and then JavaScript bundles to render the UI.

- **How it works (simplified):**
    1. User requests a page.
    2. The server executes the React code, generating the initial HTML output.
    3. This HTML, along with minimal JavaScript, is sent to the browser.

4. The browser displays the HTML immediately.
5. The JavaScript then "hydrates" the HTML, making the application interactive (attaching event listeners, etc.).

- **Next.js:** A popular React framework that provides built-in support for SSR, Static Site Generation (SSG), and other features out of the box, simplifying the process of building production-ready React applications. It's heavily used in enterprise-level applications.
- **Benefits for Flipkart:**
    - **Improved Initial Load Performance (Perceived Performance):** Users see content much faster because the HTML is delivered directly, leading to a better user experience, especially on slower networks or devices. This is critical for e-commerce, as every second counts in conversion.
    - **Better SEO (Search Engine Optimization):** Search engine crawlers can more easily index the content of server-rendered pages because the full HTML is available immediately, which is vital for product discovery on a platform like Flipkart.
    - **First Contentful Paint (FCP) and Largest Contentful Paint (LCP):** SSR significantly improves these core web vitals, which are important ranking factors for Google.
    - **Accessibility:** Content is available even before JavaScript is fully loaded.
    - **Shared Code:** Allows sharing code between client and server (e.g., data fetching logic).
- **Trade-offs:**
    - **Increased Server Load:** Rendering on the server consumes server resources.
    - **More Complex Setup:** While Next.js simplifies it, the overall architecture is more complex than pure client-side rendering.
    - **Time to First Byte (TTFB):** Can be slightly higher than CSR for very simple pages if the server rendering process is extensive.

Error Boundaries: Handling UI Errors Gracefully

Error Boundaries are React components that **catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI** instead of crashing the entire application. They prevent a single UI error from breaking the whole user experience.

- **How they work:**
    - Error Boundaries are class components that implement either static getDerivedStateFromError() (to render a fallback UI after an error) or componentDidCatch() (to log error information).
    - They only catch errors in the render phase, lifecycle methods, and constructors of components *below* them in the tree. They do not catch errors in event handlers, asynchronous code (e.g., setTimeout), or in the error boundary itself.
- **Example:**

JavaScript

```
// src/components/ErrorBoundary.js

import React from 'react';
```

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

 // This static method is called if an error is thrown in a child component
 static getDerivedStateFromError(error) {
  // Update state so the next render will show the fallback UI.
  return { hasError: true };
 }

 // This method is called after an error has been caught
 componentDidCatch(error, errorInfo) {
  // You can also log the error to an error reporting service
  console.error("Caught an error in ErrorBoundary:", error, errorInfo);
  this.setState({
   error: error,
   errorInfo: errorInfo,
  });
 }

 render() {
  if (this.state.hasError) {
   // You can render any custom fallback UI
   return (
    <div style={{ border: '1px solid red', padding: '20px', backgroundColor: '#ffe6e6' }}>
     <h2>Something went wrong.</h2>
     <p>We are working to fix this. Please try again later.</p>
     {/* Optional: display error details in development */}
     {process.env.NODE_ENV === 'development' && (
```

```
          <details style={{ whiteSpace: 'pre-wrap' }}>
            {this.state.error && this.state.error.toString()}
            <br />
            {this.state.errorInfo && this.state.errorInfo.componentStack}
          </details>
        )}
      </div>
    );
  }

  return this.props.children;
 }
}

export default ErrorBoundary;

// src/components/RiskyProductDisplay.js (A component that might throw an error)
import React from 'react';

function RiskyProductDisplay({ product }) {
 if (!product || !product.name) {
  // Simulate an error if product data is malformed
  throw new Error("Invalid product data received!");
 }
 return (
  <div>
   <h4>{product.name}</h4>
   <p>Price: ₹{product.price}</p>
  </div>
 );
}
export default RiskyProductDisplay;
```

```
// In your App or a parent component:
/*
import ErrorBoundary from './components/ErrorBoundary';
import RiskyProductDisplay from './components/RiskyProductDisplay';

function App() {
  const goodProduct = { id: 'p1', name: 'Good Phone', price: 15000 };
  const badProduct = { id: 'p2', price: 20000 }; // Missing name, will cause error

  return (
    <div>
      <h1>Flipkart Product Page</h1>

      <ErrorBoundary>
        <h3>Section 1 (Good Product)</h3>
        <RiskyProductDisplay product={goodProduct} />
      </ErrorBoundary>

      <ErrorBoundary>
        <h3>Section 2 (Bad Product - will cause fallback)</h3>
        <RiskyProductDisplay product={badProduct} />
      </ErrorBoundary>

      <h3>Other content on the page continues to work...</h3>
      <p>Even if one section crashes, the rest of the UI remains functional.</p>
    </div>
  );
}
*/
```

- **Benefits for Flipkart:**

- **Improved User Experience:** Prevents the entire application from crashing due to an isolated error, showing a user-friendly message instead.
- **Graceful Degradation:** The rest of the UI remains interactive and usable.
- **Better Debugging:** Centralized error logging makes it easier to track and fix bugs in production.

Type Checking with PropTypes or TypeScript: For Robust Code

Ensuring that components receive and use data of the correct type is crucial for building robust and maintainable applications.

PropTypes

PropTypes is a library (originally built into React, now a separate package) for runtime type checking of props passed to React components.

- **How it works:** You define the expected types for each prop using ComponentName.propTypes. If an invalid prop type is passed in development mode, a warning will be logged to the console.
- **Example:**

JavaScript

```
// src/components/ProductInfo.js

import React from 'react';

import PropTypes from 'prop-types'; // Import PropTypes


function ProductInfo({ name, price, imageUrl, isInStock }) {

 return (

  <div style={{ border: '1px solid #ddd', padding: '15px', margin: '10px' }}>

   <img src={imageUrl} alt={name} width="150" />

   <h3>{name}</h3>

   <p>Price: ₹{price.toFixed(2)}</p>

   <p style={{ color: isInStock ? 'green' : 'red' }}>

    {isInStock ? 'In Stock' : 'Out of Stock'}

   </p>

  </div>

 );

}


// Define prop types

ProductInfo.propTypes = {

 name: PropTypes.string.isRequired, // 'name' must be a string and is required
```

```
  price: PropTypes.number.isRequired, // 'price' must be a number and is required
  imageUrl: PropTypes.string,        // 'imageUrl' is an optional string
  isInStock: PropTypes.bool,         // 'isInStock' is an optional boolean
};

// Default props in case they are not provided
ProductInfo.defaultProps = {
  imageUrl: 'https://via.placeholder.com/150',
  isInStock: true,
};

export default ProductInfo;

// Usage:
/*
function App() {
  return (
    <div>
      <ProductInfo name="Smartwatch" price={2500} />
      <ProductInfo name="Wired Headphones" price={500} isInStock={false} />
      {/* This would cause a warning in console because price is a string, not number }
      {/* <ProductInfo name="Bad Product" price="expensive" /> }
    </div>
  );
}
*/
```

TypeScript

TypeScript is a superset of JavaScript that adds static type definitions. It allows you to define types for variables, functions, and props *during development* (compile-time), catching errors much earlier than runtime.

- **How it works:** You write code in TypeScript (.ts or .tsx files), and it's transpiled down to plain JavaScript. Type checks happen during this compilation process.
- **Benefits over PropTypes:**

- **Compile-time Checks:** Catches type errors before the code even runs, preventing bugs from reaching production.
- **Better Autocompletion and Refactoring:** IDEs can provide much smarter suggestions and refactoring capabilities.
- **Improved Code Readability and Maintainability:** Clear type definitions act as documentation for your code.
- **Scalability:** Essential for large codebases with many developers, ensuring consistency and reducing errors.

- **Example:**

TypeScript

```
// src/types.ts
export interface Product {
  id: string;
  name: string;
  price: number;
  imageUrl?: string; // Optional property
  isInStock?: boolean;
}

// src/components/ProductInfo.tsx
import React from 'react';
import { Product } from '../types'; // Import the interface

// Define the type for the component's props
interface ProductInfoProps {
  product: Product; // Expects a 'product' prop of type Product
}

function ProductInfo({ product }: ProductInfoProps) {
  const { name, price, imageUrl, isInStock = true } = product; // Destructure and provide default for isInStock

  return (
    <div style={{ border: '1px solid #ddd', padding: '15px', margin: '10px' }}>
      <img src={imageUrl || 'https://via.placeholder.com/150'} alt={name} width="150" />
```

```
      <h3>{name}</h3>
      <p>Price: ₹{price.toFixed(2)}</p>
      <p style={{ color: isInStock ? 'green' : 'red' }}>
       {isInStock ? 'In Stock' : 'Out of Stock'}
      </p>
    </div>
  );
}

export default ProductInfo;

// Usage:
/*
import ProductInfo from './components/ProductInfo';
import { Product } from './types';

function App() {
  const goodProduct: Product = { id: 'sw1', name: 'Smartwatch', price: 2500 };
  const outOfStockProduct: Product = { id: 'hp1', name: 'Wired Headphones', price: 500, isInStock: false };
  // This would cause a TypeScript compilation error because 'price' is missing
  // const badProduct: Product = { id: 'bp1', name: 'Bad Product' };

  return (
    <div>
      <ProductInfo product={goodProduct} />
      <ProductInfo product={outOfStockProduct} />
    </div>
  );
}
*/
```