1. E-Commerce Cart Management

Scenario:  
In an e-commerce website, users can add items to their cart, update quantities, or remove items. The cart data should persist even after a page refresh.

Answer:  
To handle cart management, I’d use React’s useState or useReducer for local state management. For global state, I’d use Redux or Context API. Data persistence can be achieved by saving the cart in localStorage.

Sub-questions:

How would you structure the cart state?

The cart can be an array of objects, with each object representing an item:

jsx

Copy code

const [cart, setCart] = useState([

{ id: 1, name: 'Product A', price: 100, quantity: 2 },

{ id: 2, name: 'Product B', price: 200, quantity: 1 },

]);

How do you handle adding items to the cart?

When adding items, I’d check if the item already exists in the cart and update its quantity if it does:

jsx

Copy code

const addToCart = (item) => {

setCart((prevCart) => {

const existingItem = prevCart.find((i) => i.id === item.id);

if (existingItem) {

return prevCart.map((i) =>

i.id === item.id ? { ...i, quantity: i.quantity + 1 } : i

);

}

return [...prevCart, { ...item, quantity: 1 }];

});

};

Real-world example:  
Think of Amazon’s cart. If you add an item, it reflects in the cart badge, and refreshing the page doesn't clear the cart because it uses local storage or a backend service to persist data.

2. Dynamic Form Rendering

Scenario:  
An admin dashboard allows users to create surveys with different field types like text, dropdowns, or checkboxes. The form fields should render dynamically based on the survey configuration.

Answer:  
Dynamic forms can be created by mapping over a fields array.  
Example fields array:

jsx

Copy code

const fields = [

{ type: 'text', label: 'Name', value: '' },

{ type: 'select', label: 'Gender', options: ['Male', 'Female'], value: '' },

];

Sub-questions:

How do you handle form validation?

I’d use libraries like Formik or React Hook Form for validation, which provide hooks to validate fields and manage errors.

jsx

Copy code

const { register, handleSubmit, errors } = useForm();

How do you conditionally render fields based on user input?

Use state to control the visibility of fields. For example, show a "State" dropdown only if the selected country is "USA":

jsx

Copy code

{country === 'USA' && <Dropdown options={states} />}

Real-world example:  
Google Forms allows dynamic rendering of questions based on user input, e.g., showing additional questions if a user selects "Yes" for a specific option.

3. Handling API Errors

Scenario:  
You build a dashboard fetching data from an external API. Sometimes the API fails, and you need to show a friendly error message and retry the request.

Answer:  
API calls can be handled using axios or fetch with error handling in a try...catch block. I’d also use states for loading, error, and data.

Sub-questions:

How do you implement a retry mechanism?

A simple approach is to use a counter to limit retries:

jsx

Copy code

const fetchData = async (retryCount = 3) => {

try {

const response = await fetch('/api/data');

setData(await response.json());

} catch (error) {

if (retryCount > 0) {

fetchData(retryCount - 1);

} else {

setError('Failed after retries');

}

}

};

How do you display error messages to users?

Use a toast notification or error banner:

jsx

Copy code

{error && <div className="error-banner">{error}</div>}

Real-world example:  
Netflix displays a "Retry" button if a video fails to load due to network issues, allowing users to attempt loading again.

4. Real-time Chat Application

Scenario:  
You are tasked with building a real-time chat application where users can instantly see messages sent by others.

Answer:  
I’d use WebSockets (e.g., Socket.IO) to establish a persistent connection between the server and clients. Messages would be pushed to the clients in real-time and stored in a state.

Sub-questions:

How do you handle message storage and rendering?

Store messages in a state array:

jsx

Copy code

const [messages, setMessages] = useState([]);

Render messages using .map():

jsx

Copy code

messages.map((msg, index) => <p key={index}>{msg}</p>);

How do you handle typing indicators?

Emit a "userTyping" event when a user types and listen for it on the client to display a "User is typing..." message.

Real-world example:  
WhatsApp shows real-time messages and typing indicators, ensuring users know when someone is actively responding.

5. User Authentication

Scenario:  
A platform requires user login, and certain pages should be accessible only after authentication.

Answer:  
Authentication can be implemented using JWT tokens. Secure routes using React Router’s PrivateRoute pattern.

Sub-questions:

How do you store tokens securely?

Use httpOnly cookies to store tokens securely, reducing XSS vulnerabilities.

How do you implement logout functionality?

Clear the token from storage and redirect the user:

jsx

Copy code

localStorage.removeItem('token');

navigate('/login');

Real-world example:  
Facebook logs you out automatically after inactivity to secure your account and requires re-authentication.

6. Performance Optimization in a Large Table

Scenario:  
You are building a data dashboard that displays a large table with thousands of rows. Rendering all rows at once causes significant lag and poor performance.

Answer:  
To optimize performance, I’d implement virtualized rendering using libraries like react-window or react-virtualized. These libraries render only the visible rows, improving performance.

Sub-questions:

How does virtualization work in React?

Virtualization calculates the visible rows and renders only those, while maintaining the scrollable layout.

jsx

Copy code

import { FixedSizeList as List } from 'react-window';

<List

height={500}

itemCount={data.length}

itemSize={35}

>

{({ index, style }) => (

<div style={style}>{data[index]}</div>

)}

</List>

How would you handle sorting and filtering in such a table?

For sorting: Use the Array.sort() method on the dataset before passing it to the table.

For filtering: Use Array.filter() to return only the rows that match the filter criteria.

Real-world example:  
Google Sheets uses virtualized rendering to handle massive datasets without compromising performance, allowing smooth scrolling and filtering.

7. Debouncing User Input in Search

Scenario:  
You are implementing a search bar where typing a query sends an API request. Rapid typing can lead to multiple unnecessary API calls, affecting performance.

Answer:  
I’d use a debounce function to delay the API call until the user stops typing for a specified time. This can be achieved using lodash.debounce or a custom debounce function.

Sub-questions:

What is the difference between debounce and throttle?

Debounce: Delays the execution of a function until after a specified period of inactivity.

Throttle: Ensures a function is executed at most once in a given time interval.

How do you implement a debounce in React?

jsx

Copy code

import { useState, useEffect } from 'react';

import debounce from 'lodash.debounce';

const SearchBar = () => {

const [query, setQuery] = useState('');

const debouncedSearch = debounce((query) => {

// API call

console.log('Searching:', query);

}, 500);

useEffect(() => {

debouncedSearch(query);

return debouncedSearch.cancel;

}, [query]);

return <input onChange={(e) => setQuery(e.target.value)} />;

};

Real-world example:  
E-commerce websites like Flipkart or Amazon implement debounced search to provide real-time suggestions without overloading their servers.

8. Lazy Loading Components

Scenario:  
You’re working on a website with multiple routes. Some routes, like a dashboard, are used frequently, while others, like an admin panel, are rarely accessed. You want to optimize initial load time by loading less critical components only when required.

Answer:  
Lazy loading can be implemented using React’s React.lazy and Suspense.

Sub-questions:

How do you implement lazy loading?

jsx

Copy code

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

return (

<Suspense fallback={<div>Loading...</div>}>

<AdminPanel />

</Suspense>

);

How would you handle errors during lazy loading?

Use ErrorBoundary to catch errors and show fallback UI:

jsx

Copy code

class ErrorBoundary extends React.Component {

state = { hasError: false };

static getDerivedStateFromError() {

return { hasError: true };

}

render() {

if (this.state.hasError) {

return <h1>Something went wrong.</h1>;

}

return this.props.children;

}

}

Real-world example:  
YouTube lazy-loads the comments section, improving the initial page load time while comments load when the user scrolls.

9. Implementing Dark Mode

Scenario:  
You want to add a dark mode toggle to your application and persist the user’s preference across sessions.

Answer:  
Dark mode can be implemented using CSS variables and React’s useState. To persist the preference, use localStorage.

Sub-questions:

How do you toggle between light and dark themes?

jsx

Copy code

const [theme, setTheme] = useState('light');

useEffect(() => {

document.body.className = theme;

}, [theme]);

const toggleTheme = () => {

const newTheme = theme === 'light' ? 'dark' : 'light';

setTheme(newTheme);

localStorage.setItem('theme', newTheme);

};

How do you load the saved theme on page load?

jsx

Copy code

useEffect(() => {

const savedTheme = localStorage.getItem('theme') || 'light';

setTheme(savedTheme);

}, []);

Real-world example:  
Twitter offers a dark mode toggle, and the preference persists even after closing and reopening the app.

10. Infinite Scrolling

Scenario:  
You’re building a social media feed that loads more posts as the user scrolls to the bottom of the page.

Answer:  
Infinite scrolling can be implemented using an intersection observer to detect when the user reaches the bottom of the page and fetch more data.

Sub-questions:

How do you implement an intersection observer?

jsx

Copy code

const observer = new IntersectionObserver(

([entry]) => {

if (entry.isIntersecting) {

loadMorePosts();

}

},

{ threshold: 1.0 }

);

useEffect(() => {

const target = document.querySelector('#end-of-list');

if (target) observer.observe(target);

return () => observer.disconnect();

}, []);

How do you handle loading indicators?

Use a loading spinner:

jsx

Copy code

{isLoading && <div>Loading...</div>}

Real-world example:  
Instagram’s feed continuously loads more posts as users scroll, keeping them engaged without the need for pagination.

11. Handling Form Validation

Scenario:  
You are building a registration form that requires users to input their name, email, and password. The form should validate input fields and provide error messages for invalid entries.

Answer:  
Form validation can be handled using controlled components and validation logic. Libraries like Formik or React Hook Form can simplify the process.

Sub-questions:

How do you implement basic form validation?

jsx

Copy code

const [formData, setFormData] = useState({ name: '', email: '', password: '' });

const [errors, setErrors] = useState({});

const validate = () => {

const errors = {};

if (!formData.name) errors.name = 'Name is required';

if (!/\S+@\S+\.\S+/.test(formData.email)) errors.email = 'Email is invalid';

if (formData.password.length < 6) errors.password = 'Password must be at least 6 characters';

setErrors(errors);

return Object.keys(errors).length === 0;

};

const handleSubmit = (e) => {

e.preventDefault();

if (validate()) {

console.log('Form submitted:', formData);

}

};

How do you handle real-time validation?

Add validation logic in the onChange event:

jsx

Copy code

const handleChange = (e) => {

setFormData({ ...formData, [e.target.name]: e.target.value });

validate();

};

Real-world example:  
Websites like LinkedIn validate user details in real-time during registration, ensuring correct input before submission.

12. State Management with Redux

Scenario:  
You are developing an e-commerce website where the cart data needs to be shared across multiple components, such as the product listing, cart summary, and checkout page.

Answer:  
Use Redux to centralize the cart data and manage the global state.

Sub-questions:

How do you set up a Redux store?

jsx

Copy code

import { configureStore } from '@reduxjs/toolkit';

import cartReducer from './cartSlice';

const store = configureStore({

reducer: {

cart: cartReducer,

},

});

export default store;

How do you connect a component to Redux?

jsx

Copy code

import { useSelector, useDispatch } from 'react-redux';

import { addItem } from './cartSlice';

const Product = ({ product }) => {

const dispatch = useDispatch();

const handleAddToCart = () => {

dispatch(addItem(product));

};

return <button onClick={handleAddToCart}>Add to Cart</button>;

};

Real-world example:  
Amazon uses a centralized state to handle the cart, allowing users to view and modify it across different pages.

13. Implementing Authentication with JWT

Scenario:  
You need to secure your application by implementing user authentication. After a user logs in, their credentials should remain secure, and certain pages should be restricted to authenticated users only.

Answer:  
Use JWT (JSON Web Token) for authentication and protect routes using React Router.

Sub-questions:

How do you store and use JWTs?

Store the token in localStorage or sessionStorage:

jsx

Copy code

localStorage.setItem('token', jwtToken);

How do you protect routes?

jsx

Copy code

const PrivateRoute = ({ children }) => {

const token = localStorage.getItem('token');

return token ? children : <Navigate to="/login" />;

};

Real-world example:  
Gmail restricts access to emails by verifying authentication tokens, ensuring secure user sessions.

14. Context API for Theme Switching

Scenario:  
You’re building a blog platform where the user can switch between light and dark themes. This preference should be applied across all pages without passing props manually.

Answer:  
Use the Context API to manage the theme globally.

Sub-questions:

How do you create and use a context?

jsx

Copy code

const ThemeContext = React.createContext();

const ThemeProvider = ({ children }) => {

const [theme, setTheme] = useState('light');

const toggleTheme = () => setTheme(theme === 'light' ? 'dark' : 'light');

return (

<ThemeContext.Provider value={{ theme, toggleTheme }}>

{children}

</ThemeContext.Provider>

);

};

How do you access context in a component?

jsx

Copy code

const { theme, toggleTheme } = useContext(ThemeContext);

return <button onClick={toggleTheme}>Switch to {theme === 'light' ? 'dark' : 'light'} mode</button>;

Real-world example:  
Medium offers light and dark theme options, applied seamlessly across the platform using similar logic.

15. Implementing Error Boundaries

Scenario:  
You want to ensure that errors in one part of your React app don’t break the entire application. For instance, an error in a dashboard widget should not crash the entire page.

Answer:  
Use Error Boundaries to catch and handle errors in React.

Sub-questions:

How do you create an Error Boundary?

jsx

Copy code

class ErrorBoundary extends React.Component {

constructor(props) {

super(props);

this.state = { hasError: false };

}

static getDerivedStateFromError(error) {

return { hasError: true };

}

componentDidCatch(error, errorInfo) {

console.log('Error:', error, errorInfo);

}

render() {

if (this.state.hasError) {

return <h1>Something went wrong.</h1>;

}

return this.props.children;

}

}

When should you use Error Boundaries?

Use them for critical parts of the UI, such as widgets, forms, and dashboards.

Real-world example:  
Facebook uses Error Boundaries to ensure that an issue in one post or comment doesn't crash the entire news feed.