Q1: Explain the difference between state and props in React.

State and props are both used for managing data in React components, but they serve different purposes:

- State: Internal and mutable, controlled by the component itself
- Props: External and immutable, controlled by the parent component

```
const Counter = ({ initialCount }: { initialCount: number }) => {
 const [count, setCount] = useState(initialCount); // State with initial count

 return (
<button onClick={() => setCount(count + 1)}>
 Clicked {count} times
</button>
);
};

const Parent = () => {
 const initialCount = 5;
 return <Counter initialCount={initialCount} />; // Props
};
```

Q2: Describe the Context API in React and its use cases.

The Context API in React is a feature that allows state to be shared across the entire app or a part of it. It's particularly useful for passing data that can be considered global, such as user authentication status or theme preferences.

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

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

const Toolbar = () => {
 const theme = useContext(ThemeContext);
 return <div>Current theme: {theme}</div>;
};
```

Q3: How does reconciliation work in React?

Reconciliation is the process by which React updates the DOM. When a component's state changes, React creates a new virtual DOM tree and compares it with the previous one.

React then calculates the most efficient way to update the browser's DOM to match the new tree, minimizing the number of actual DOM operations required. This process is also known as diffing.

Q4: Explain the purpose and usage of React Fragments.

React Fragments allow you to group multiple elements without adding an extra node to the DOM. They're useful when you need to return multiple elements from a component without wrapping them in a div or other container element.

```
const ListItems = () => {
 return (
<>
<li>Item 1</li>
<li>Item 2</li>
</>
);
};
```

Q5: Describe React hooks and their advantages.

React hooks allow you to use state and other React features in functional components. They solve problems like reusing stateful logic between components, managing side-effects, and handling complex lifecycle events. Hooks also enable you to create custom hooks, which encapsulate reusable logic.

Hooks offer several advantages: they make code more reusable and composable, reduce the complexity of components, and eliminate the confusion around 'this' keyword in class components.

Q6: List and explain common React hooks.

React provides several built-in hooks for different purposes:

- useState: Manages state in functional components
- useEffect: Handles side effects and implements lifecycle-like behavior
- useContext: Consumes a React context
- useRef: Creates a mutable reference that persists across re-renders
- useMemo: Memoizes a computed value to optimize performance
- useCallback: Memoizes a function to prevent unnecessary re-creation on every render

```
const Counter = () => {
 const [count, setCount] = useState(0);
 const prevCountRef = useRef<number>();

 useEffect(() => {
 prevCountRef.current = count;
}, [count]);

 const increment = useCallback(() => setCount((c) => c + 1), []);

 const expensiveComputation = useMemo(() => {
 return count * 2;
}, [count]);

 return (
<div>
```

```
 Now: {count}, before: {prevCountRef.current}
<br />
 Expensive Computation Result: {expensiveComputation}
<button onClick={increment}>Increment</button>
</div>
);
};
```

Q7: Explain the concept of pure components in React.

Pure components in React are components that render the same output for the same state and props. They implement a shallow comparison on their props and state to determine if they should re-render.

In class components, you can extend React.PureComponent to create a pure component. For functional components, you can use React.memo to achieve similar behavior.

Q8: Describe the process of creating a custom hook in React.

Creating a custom hook in React involves defining a function that starts with 'use' and can call other hooks. Custom hooks allow you to extract component logic into reusable functions.

```
const useCounter = (initialValue = 0) => {
 const [count, setCount] = useState(initialValue);
 const increment = () => setCount(count + 1);
 const decrement = () => setCount(count - 1);
 return { count, increment, decrement };
};

const Counter = () => {
 const { count, increment, decrement } = useCounter();
 return (
<div>
 Count: {count}
<button onClick={increment}>+</button>
<button onClick={decrement}>-</button>
</div>
);
};
```

Q9: Explain Higher Order Components (HOCs) in React.

A Higher Order Component (HOC) is a function that takes a component as an argument and returns a new component. HOCs are used to share common functionality between components without repeating code.

```
const withLogger = (WrappedComponent) => {
 return (props) => {
 useEffect(() => {
 console.log('Component rendered');
}, []);
```

```
 return <WrappedComponent {...props} />;
};
};

const MyComponent = ({ name }) => <div>Hello, {name}!</div>;
const EnhancedComponent = withLogger(MyComponent);
```

Q10: Why are keys important in React lists?

Keys help React identify which items in a list have changed, been added, or been removed. They should be unique strings among siblings, but don't need to be globally unique.

```
const TodoList = ({ todos }) => {
 return (
<ul>
{todos.map(todo => (
<li key={todo.id}>{todo.text}</li>
))}
</ul>
);
};
```

Using keys properly can significantly improve the performance of your React application when dealing with dynamic lists.

Q11: Demonstrate how to destructure props in React components.

Destructuring props in React components allows you to extract specific properties from the props object, making your code cleaner and more readable.

```
interface PersonProps {
 person: {
 name: string;
 age: number;
 skill: string;
};
}

const PersonInfo = ({ person: { name, age, skill } }: PersonProps) => {
 return (
<div>
<p>Name: {name}</p>
<p>Age: {age}</p>
<p>Skill: {skill}</p>
</div>
);
};
```

Q12: Explain the Virtual DOM and its differences from the real DOM.

The Virtual DOM is a lightweight copy of the real DOM kept in memory. React uses it to optimize updates to the actual DOM, improving performance.

When state changes, React creates a new Virtual DOM tree and compares it with the previous one through a process called reconciliation. It then updates only the parts of the real DOM that have changed, minimizing expensive DOM operations.

Q13: Define controlled components in React and provide an example.

A controlled component in React is a component where form data is handled by the component's state. The component controls the value of the input element and how it changes in response to user input.

```
const ControlledInput = () => {
 const [value, setValue] = useState('');

 const handleChange = (event: React.ChangeEvent<HTMLInputElement>) => {
 setValue(event.target.value);
};

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

Q14: How can a child component update the state in a parent component?

A child component can update the state in a parent component through callback functions passed as props. The parent component defines a function to update its state and passes this function to the child as a prop.

```
const Parent = () => {
 const [count, setCount] = useState(0);

 const incrementCount = () => setCount(count + 1);

 return <Child onIncrement={incrementCount} />;
};

const Child = ({ onIncrement }: { onIncrement: () => void }) => {
 return <button onClick={onIncrement}>Increment</button>;
};
```

Q15: Explain the purpose of the dependency array in useEffect.

The dependency array in useEffect is used to specify which values the effect depends on. When any of these dependencies change, the effect will re-run.

```
const Counter = () => {
 const [count, setCount] = useState(0);

 useEffect(() => {
 const updateTitle = () => {
 document.title = 'Count: ' + count;
};
 updateTitle();
}, [count]); // Only re-run the effect if count changes

 return (
<button onClick={() => setCount(count + 1)}>
 Increment
</button>
);
};
```

If the dependency array is empty, the effect runs only once after the initial render. If it's omitted, the effect runs after every render.

Q16: Describe the purpose of the return function in useEffect.

The return function in useEffect is used for cleanup. It runs before the component unmounts and before the effect runs again (if it does).

```
useEffect(() => {
 const subscription = subscribeToData();
 return () => {
 subscription.unsubscribe();
};
}, []);
```

This is useful for cleaning up side effects like subscriptions or timers to prevent memory leaks.

Q17: Provide an overview of React and its main features.

React is a JavaScript library for building user interfaces, particularly single-page applications. It allows developers to create reusable UI components and manage the state of these components efficiently.

Key features of React include:

- Virtual DOM for efficient updates
- JSX syntax for describing UI
- Component-based architecture
- Unidirectional data flow
- Declarative programming model

Q18: Explain JSX and its role in React development.

JSX is a syntax extension for JavaScript that allows you to write HTML-like code in your JavaScript files. It's not valid JavaScript on its own and needs to be transpiled before it can run in a browser.

```
const element = <h1>Hello, {name}</h1>;
```

JSX makes it easier to write and understand the structure of React components. It's transformed into regular JavaScript function calls and objects before being rendered.

Q19: List the major advantages of using React for web development.

React offers several advantages for web development:

- Efficient updates and rendering through Virtual DOM
- Component-based architecture for better code reuse and maintenance
- Strong community support and extensive ecosystem
- Declarative programming model for easier debugging
- Seamless integration with other libraries and frameworks

These features make React a powerful tool for building complex, dynamic user interfaces with high performance and maintainability.

Q20: Demonstrate how to update state in a functional React component.

In a functional React component, state is updated using the setState function provided by the useState hook. This function can take either a new state value or a function that returns the new state based on the previous state.

```
const Counter = () => {
 const [count, setCount] = useState(0);

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

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

Q21: Describe the phases in a React component's lifecycle.

React components go through three main phases in their lifecycle: Mounting, Updating, and Unmounting. Each phase has its own set of lifecycle methods in class components, while functional components use hooks to manage these phases.

In functional components, useEffect can be used to replicate lifecycle behavior. For example, an empty dependency array mimics componentDidMount, while a cleanup function mimics componentWillUnmount.

Q22: Demonstrate how to create a form in React.

Creating a form in React involves using form elements in JSX and managing their state. Here's an example of a simple form using hooks:

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

 const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
 const { name, value } = e.target;
 setFormData(prevData => ({
...prevData,
[name]: value
}));
};

 const handleSubmit = (e: React.FormEvent<HTMLFormElement>) => {
 e.preventDefault();
 console.log('Form submitted:', formData);
};

 return (
<form onSubmit={handleSubmit}>
<input
 type="text"
 name="name"
 value={formData.name}
 onChange={handleChange}
 placeholder="Name"
/>
<input
 type="email"
 name="email"
 value={formData.email}
 onChange={handleChange}
 placeholder="Email"
/>
<button type="submit">Submit</button>
</form>
);
};
```

Q23: Explain how React Router works.

React Router is a standard library for routing in React applications. It enables navigation among views in a React application, allowing you to create a single-page application with navigation without page refreshes.

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

const App = () => (
<Router>
<nav>
<Link to="/">Home</Link>
<Link to="/about">About</Link>
</nav>

<Routes>
<Route path="/" element={<Home />} />
<Route path="/about" element={<About />} />
</Routes>
</Router>
);
```

React Router uses components to define routing rules. When a URL matches a route's path, the corresponding component is rendered.

Q24: List the major advantages of using Next.js.

Next.js is a popular React framework that offers several advantages:

- Server-side rendering for better SEO and initial load performance
- Automatic static optimization for static pages
- File-system based routing for easier navigation setup
- API routes for building API endpoints with serverless functions
- Built-in CSS and Sass support
- Automatic code splitting for faster page loads

These features make Next.js an excellent choice for building production-ready React applications with improved performance and SEO capabilities.

Q25: Compare class and functional components in React.

Class and functional components in React have some key differences:

- Syntax: Class components use ES6 class syntax, while functional components are just JavaScript functions
- State management: Class components use this.state and this.setState(), functional components use the useState hook
- Lifecycle methods: Class components have lifecycle methods, functional components use the useEffect hook
- Performance: Functional components can have slightly better performance and are easier to optimize

With the introduction of hooks, functional components can now handle all use cases that previously required class components, making them the preferred choice in modern React development.

Q26: Describe prop drilling and its potential solutions.

Prop drilling occurs when you pass props through multiple levels of components that don't need those props themselves, but only pass them down to lower levels. This can make your code harder to maintain and understand.

Solutions to prop drilling include:

- Using Context API for global state management
- Implementing state management libraries like Redux
- Composition: restructuring your components to avoid deep nesting
- Render props pattern
- Hooks like useContext for easier context consumption

Q27: Explain the use cases for useMemo() and useCallback() hooks.

useMemo() and useCallback() are hooks used for performance optimization in React. They help prevent unnecessary re-renders by memoizing values and functions.

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);

const memoizedCallback = useCallback(
() => {
 doSomething(a, b);
},
[a, b],
);
```

useMemo() is used to memoize expensive computations, while useCallback() is used to memoize callback functions. They're particularly useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders.

Q28: Describe event handling in React and the purpose of e.target.

In React, events are handled through special functions called event handlers. These are typically defined as methods on your component or as inline arrow functions.

```
const HandleClick = () => {
 const handleClick = (e: React.MouseEvent<HTMLButtonElement>) => {
 console.log('Button text:', e.target.textContent);
};

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

e.target refers to the DOM element that triggered the event. It's useful for accessing properties of the element that was interacted with, such as its value or textContent.

Q29: Explain what triggers a component re-render in React.

A React component re-renders when there are changes to its state or props. This includes:

- State updates using setState or a state setter from useState
- Prop changes passed from a parent component
- Context value changes if the component consumes a context
- Force update (though this is rarely needed and generally discouraged)
- Re-render of a parent component, which causes all child components to re-render by default

Understanding what triggers re-renders is crucial for optimizing React applications and avoiding unnecessary render cycles.

Q30: Describe React's synthetic events and their purpose.

React's synthetic events are a cross-browser wrapper around the browser's native event system. They provide a consistent interface across different browsers, ensuring that events work identically across all supported browsers.

```
const HandleEvent = () => {
 const handleClick = (e: React.MouseEvent<HTMLButtonElement>) => {
 e.preventDefault();
 console.log('Event type:', e.type);
};

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

Synthetic events pool for performance, meaning that the event object is reused and all properties are nullified after the event callback has been invoked. This means you can't access the event in an asynchronous way.

Q31: Compare controlled and uncontrolled components in React.

Controlled components have their state controlled by React. The component receives its current value via props and updates it via callbacks. Uncontrolled components maintain their own internal state.

```
// Controlled component
const ControlledInput = () => {
 const [value, setValue] = useState('');
 return <input value={value} onChange={e => setValue(e.target.value)} />;
};

// Uncontrolled component
const UncontrolledInput = () => {
 const inputRef = useRef<HTMLInputElement>(null);
 const handleSubmit = () => {
 console.log('Input value:', inputRef.current?.value);
};
 return <input ref={inputRef} />;
};
```

Controlled components offer more control and are preferred for form elements, while uncontrolled components can be simpler for integrating React with non-React code.

Q32: Explain the purpose and core concepts of Redux.

Redux is a predictable state container for JavaScript apps, commonly used with React. It helps manage the state of an application in a single store, making state mutations predictable.

Core concepts of Redux include:

- Store: Holds the state of the application
- Actions: Plain objects describing what happened

- Reducers: Pure functions that take the previous state and an action, and return the next state
- Dispatch: Method to send actions to the store
- Subscribe: Method to listen for state changes

Redux follows a unidirectional data flow, which makes it easier to track state changes and debug an application.

Q33: Describe the role of a reducer in Redux.

A reducer in Redux is a pure function that takes the current state and an action as arguments, and returns a new state. Reducers specify how the application's state changes in response to actions sent to the store.

```
const counterReducer = (state = 0, action) => {
 switch (action.type) {
 case 'INCREMENT':
 return state + 1;
 case 'DECREMENT':
 return state - 1;
 default:
 return state;
}
};
```

Reducers must be pure functions, meaning they should not modify the existing state, but instead return a new state object. This ensures predictability and enables features like time-travel debugging.

Q34: Outline the data flow in a Redux application.

Data in Redux flows in a unidirectional pattern:

- User interacts with the view
- Action is dispatched
- Reducer processes the action
- Store is updated with new state
- View re-renders based on new state

This cycle ensures that all state updates are centralized and occur in a predictable manner. It makes the application easier to understand and debug, as every state change is traceable to a specific action.

Q35: Explain methods for handling asynchronous actions in Redux.

Handling asynchronous actions in Redux typically involves using middleware. Two popular options are:

- Redux Thunk: Allows action creators to return functions instead of actions
- Redux Saga: Uses ES6 generators to manage side effects

```
// Using Redux Thunk
const fetchUserData = (userId) => {
 return async (dispatch) => {
 dispatch({ type: 'FETCH_USER_REQUEST' });
 try {
 const response = await fetch('/api/users/' + userId);
 const data = await response.json();
 dispatch({ type: 'FETCH_USER_SUCCESS', payload: data });
```

```
} catch (error) {
 dispatch({ type: 'FETCH_USER_FAILURE', error });
}
};
};
```

Q36: Explain the purpose and usage of refs in React.

Refs in React provide a way to access DOM nodes or React elements created in the render method. They are useful for managing focus, text selection, or integrating with third-party DOM libraries.

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

While refs can be useful, they should be used sparingly. Overuse of refs can make your code harder to understand and maintain.

Q37: What is a React portal and when would you use it?

A React Portal allows you to render children into a DOM node that exists outside the DOM hierarchy of the parent component. This is especially useful for elements like modals, pop-ups, and tooltips that need to visually break out of their parent container.

```
const Modal = ({ isOpen, onClose }) => {
 if (!isOpen) return null;

 return ReactDOM.createPortal(
<div className="modal">
<button onClick={onClose}>Close</button>
<div>Modal Content</div>
</div>,
 document.getElementById('modal-root')
);
};
```

Portals are also useful when dealing with z-index issues, as they allow you to place components in different parts of the DOM tree while keeping them visually in the right place.

Q38: How do you optimize performance in a React application?

To optimize performance in React, consider the following strategies:

- Use React.memo to prevent unnecessary re-renders
- Implement shouldComponentUpdate or React.PureComponent in class components
- Lazy load components using React.lazy and React.Suspense
- Use useMemo and useCallback to memoize expensive computations and functions
- Optimize list rendering by using unique keys and avoiding index as a key
- Split code with dynamic imports and bundle analyzers

```
const ExpensiveComponent = React.memo(({ data }) => {
 return <div>{data}</div>;
});

const ParentComponent = ({ items }) => {
 const memoizedValue = useMemo(() => calculateExpensiveValue(items), [items]);
 return <ExpensiveComponent data={memoizedValue} />;
};
```

By implementing these techniques, you can significantly enhance the performance and responsiveness of your React applications, especially as they grow in complexity.

Q39: What is the purpose of PropTypes in React?

PropTypes are used in React to enforce type checking on the props passed to a component. They help catch bugs by ensuring that components receive the correct types of props, making your code more predictable and easier to debug.

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

const Greeting = ({ name, age }) => {
 return (
<div>
<p>Hello, {name}!</p>
<p>You are {age} years old.</p>
</div>
);
};

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

While TypeScript is often preferred for static type checking, PropTypes remain a useful tool in plain JavaScript projects to validate the types of props at runtime.

Q40: What are React Suspense and React.lazy, and how do they work together?

React Suspense and React.lazy are used together to enable code-splitting and lazy loading in React applications. React.lazy allows you to dynamically import components, while Suspense lets you display a fallback UI while the component is being loaded.

```
```

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

const App = () => {
 return (
<div>
<Suspense fallback={<div>Loading...</div>}>
<LazyComponent />
</Suspense>
</div>
);
};
```

By using React.lazy and Suspense, you can improve the initial load time of your application by splitting your code into smaller chunks and only loading what is necessary.

Q41: Explain the significance of the useEffect cleanup function and provide examples of when it's needed.

The useEffect cleanup function is used to perform cleanup tasks, such as unsubscribing from events, canceling network requests, or clearing timers when a component unmounts or before the effect runs again.

```
useEffect(() => {
 const timer = setInterval(() => {
 console.log('Timer running');
}, 1000);

 return () => clearInterval(timer); // Cleanup interval on unmount
}, []);
```

Without cleanup functions, you may experience memory leaks or unwanted behavior due to leftover processes running in the background.

Q42: What are React Server Components, and how do they differ from traditional components?

React Server Components (RSC) allow components to be rendered on the server and sent to the client as HTML. They differ from traditional components as they can fetch data and render the UI server-side, reducing the amount of JavaScript sent to the client and improving performance.

RSCs allow developers to write server-side logic directly in components without managing API endpoints separately, offering better integration between server and client code.

Q43: Describe how the useReducer hook works and when you would use it over useState.

useReducer is a React hook that manages complex state logic by using a reducer function. It's an alternative to useState for cases where state transitions involve complex logic or multiple actions.

```
const reducer = (state, action) => {
 switch (action.type) {
 case 'increment':
 return { count: state.count + 1 };
 case 'decrement':
 return { count: state.count - 1 };
```

```
 default:
 throw new Error();
}
};

const Counter = () => {
 const [state, dispatch] = useReducer(reducer, { count: 0 });

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

Q44: How does React handle reconciliation and diffing during updates, and what optimizations are applied?

React uses a process called reconciliation to update the DOM efficiently. When the state or props of a component change, React creates a new Virtual DOM tree and compares it with the previous one using a diffing algorithm.

Optimizations include updating only the parts of the DOM that have changed, skipping unchanged subtrees, and batching updates to minimize re-renders and improve performance.

Q45: Describe how to handle conditional rendering in React and provide different techniques to achieve it.

Conditional rendering in React can be handled using JavaScript conditional statements or operators inside JSX.

- Using if-else statements outside JSX
- Using ternary operators inside JSX
- Using logical && operator for short-circuit evaluation
- Using conditional rendering components like ConditionalWrapper

```
const Greeting = ({ isLoggedIn }: { isLoggedIn: boolean }) => {
 return (
<div>
{isLoggedIn ? <h1>Welcome back!</h1> : <h1>Please sign in.</h1>}
</div>
);
};
```

Q46: What is the significance of React’s Strict Mode, and how does it help developers?

React's Strict Mode is a tool that highlights potential problems in an application. It helps identify unsafe lifecycles, legacy API usage, unexpected side effects, and other potential issues.

Strict Mode does not render any visible UI and does not affect the production build, making it safe to use for detecting problems during development.

Q47: How do you manage state in a large-scale React application, and what are the best practices?

In large-scale React applications, state management can become complex. Best practices include using state management libraries like Redux, MobX, or Context API for global state, and keeping local state within components whenever possible.

Other best practices include normalizing state shape, using selectors to access state, avoiding deeply nested state objects, and keeping components pure by separating stateful and stateless logic.

Q48: What are the best practices for structuring a React project, and why are they important?

Structuring a React project properly helps maintain code readability, scalability, and reusability. Best practices include using a modular folder structure, grouping related components, and separating business logic from UI components.

- Use a flat folder structure to avoid deeply nested folders
- Organize components by feature or route
- Separate hooks, services, and utilities into their own folders
- Use index files to manage imports and exports

These practices make it easier to manage, test, and scale your application as it grows.