React Notes

Webpack

Webpack is a popular open-source module bundler for JavaScript applications. we write modular code, meaning we break down our codebase into smaller files (called modules). However, browsers don't understand this modular structure directly; they expect a single JavaScript file or a few files. Webpack solves this by bundling these modules together into a single file (or a few files) that the browser can understand.

Module Bundling: Webpack bundles JavaScript files (and other assets like CSS, images, etc.) into a single file or a few files, reducing the number of HTTP requests made by a web application.

Code Splitting:

- Webpack allows for **code splitting**, breaking up your JavaScript code into smaller "chunks" that can be loaded on demand. Instead of loading the entire app at once, code can be split by routes or components. This reduces the initial load time and improves app performance.
- In React, libraries like `React.lazy()` and `dynamic imports` use Webpack's code splitting to load components asynchronously.

Tree Shaking:

- Webpack supports **tree shaking**, a process that eliminates dead or unused code from the final bundle. By analyzing your imports and exports, Webpack removes any code that is not being used, reducing the bundle size and making your React app more efficient.

Minification:

- Webpack uses plugins like **Terser** to minify the JavaScript code, reducing file sizes by removing unnecessary characters such as white spaces, comments, and shortening variable names. This reduces the amount of code that needs to be downloaded by the browser, improving performance.

HMR (Hot Module Replacement) is a powerful feature of Webpack that allows modules (like JavaScript, CSS, etc.) to be **updated in the browser without a full page reload**.

Caching:

- Webpack generates **hashed filenames** for output files, ensuring that browsers can cache content effectively. When the code changes, Webpack updates the hash, prompting the browser to fetch new assets while keeping the unchanged files cached. This reduces loading time for subsequent visits.

Bundle Splitting (Optimization Plugins):

- Webpack can optimize bundles by using plugins like `SplitChunksPlugin`, which splits vendor code (such as libraries like React and Lodash) into separate bundles. This prevents bundling everything together and allows frequently used libraries to be cached separately, reducing the amount of data loaded when only your app code changes.

Feature	Webpack	Parcel	Vite
Configuration	Complex, highly customizable	Zero configuration, beginner-friendly	Minimal configuration, simple setup
Speed (Development)	Slower, especially for large projects	Fast initial build times	Lightning fast, near-instant startup
Speed (Production)	Fast with optimizations, but slower setup	Fast with automatic optimizations	Fast production build using Rollup
HMR Support	Yes, robust HMR	Yes, built-in HMR	Yes, extremely fast HMR
Code Splitting	Advanced, fully customizable	Automatic code splitting	Automatic code splitting via Rollup
Tree Shaking	Yes, fully configurable	Yes, built-in and automatic	Yes, via Rollup for production builds
Minification	Customizable with plugins (Terser)	Built-in minification	Automatic in production using Rollup
Caching	Customizable with hashed filenames	Automatic caching with zero config	Supports hashed filenames for caching
Plugins	Extensive and mature	Smaller, but covers	Growing, uses Rollup plugins

Tree Shaking	Yes, fully configurable	Yes, built-in and automatic	Yes, via Rollup for production builds
Minification	Customizable with plugins (Terser)	Built-in minification	Automatic in production using Rollup
Caching	Customizable with hashed filenames	Automatic caching with zero config	Supports hashed filenames for caching
Plugins Ecosystem	Extensive and mature	Smaller, but covers most use cases	Growing, uses Rollup plugins
Use Case	Large, complex projects needing full control	Quick setups, small to medium projects	Ideal for fast development, especially in large projects
Learning Curve	Steep, requires learning configuration	Very easy to start	Easy, especially for modern frameworks

Choosing the best bundler between Webpack, Parcel, and Vite depends on the specific needs of your project. Webpack is ideal for large-scale, complex applications where full control and extensive configuration are required. It has a steep learning curve but offers powerful customization and a vast plugin ecosystem. Parcel, on the other hand, is perfect for developers looking for a zero-configuration setup. It's easy to use, provides automatic performance optimizations, and works well for small to medium projects. Vite stands out for its blazing-fast development experience, especially in modern frameworks like React and Vue, thanks to its native ES module-based development and rapid Hot Module Replacement (HMR).

Babel

Babel is a JavaScript compiler and transpiler that converts modern JavaScript (ES6+ or JSX) into older versions (like ES5) to ensure compatibility across a wide range of browsers, especially older ones that don't support the latest JavaScript features

How Babel Optimizes Performance:

1. Code Transformation:
    - Transpiling Modern JavaScript

2. Tree Shaking:

3. Minification

4. Selective Transformations

Key Advantages of Using React / Why is React Fast?

1. **Component-Based Architecture**:
   React promotes building UI components that can be reused across different parts of an application, improving code maintainability and scalability.
2. **Virtual DOM**:
   React uses a virtual DOM to optimize updates to the actual DOM, making applications faster by minimizing direct manipulation of the real DOM.
3. **Single Page Application (SPA) Support**:
   React is ideal for building SPAs, where content dynamically updates without refreshing the entire page. This leads to a faster and smoother user experience, using libraries like **React Router** to manage navigation between views.
4. **One-Way Data Binding**:
   React uses one-way data flow, meaning data flows from parent to child components, making it easier to track data changes and debug.
5. **Rich Ecosystem**:
   React has a large ecosystem with libraries and tools (like **Redux**, **React**

Router, etc.) for managing state, routing, and more, simplifying the development of complex applications.

6. **JSX Syntax**:
   React uses **JSX**, a syntax extension that allows developers to write HTML-like code within JavaScript, making the code more readable and easier to write.
7. **Fast Rendering**:
   React's virtual DOM and optimized re-rendering processes result in high-performance applications with fast user interactions.
8. **SEO-Friendly**:
   React allows **Server-Side Rendering (SSR)**, which improves the SEO of applications, making them more easily discoverable by search engines.
9. **Declarative UI**:
   React's declarative nature means developers can describe **what the UI should look like**, and React will efficiently manage the updates to the DOM. This approach simplifies the development process and makes the code more predictable.
10. **Strong Community and Support**:
    React is backed by **Meta** (formerly Facebook) and has a vast community of developers, offering abundant resources, tutorials, and tools for learning and troubleshooting.
11. **Cross-Platform Development**:
    React Native, a derivative of React, enables developers to build mobile applications using the same architecture, increasing code reuse across web and mobile.
12. **Easy to Learn**:
    React is relatively easier to learn compared to other frontend frameworks due to its simple and flexible nature, making it a popular choice for new developers.

How React is different from other frameworks?

React is a **library**, not a full-fledged framework, meaning it only handles the UI layer, allowing developers to choose their own tools for routing, state management, and other needs. This provides more flexibility but requires more setup for large projects. React also uses **JSX**, which combines JavaScript and HTML in one file, promoting component reusability, and relies on a **unidirectional data flow**, making state management predictable. It optimizes performance with its **Virtual DOM** and **Reconciliation algorithm**, efficiently updating only the parts of the DOM that change. In contrast, **Angular** is a **complete framework** that comes with built-in tools for forms, routing, and state management, but has a steeper learning curve due to its more opinionated structure and use of TypeScript. **Vue**, like React, is flexible and easy to learn, offering a middle ground between React’s flexibility and Angular’s

all-in-one approach, but with **built-in two-way data binding** and simpler templating. React's modularity and focus on efficiency through the Virtual DOM set it apart.

Virtual DOM Reconciliation

React increases performance primarily through its **Virtual DOM** and **Reconciliation algorithm**. The Virtual DOM is a lightweight, in-memory representation of the actual DOM. When the state or props change, React first updates the Virtual DOM rather than directly interacting with the real DOM, which is slow. React then compares the new Virtual DOM with the previous version in a process called **reconciliation**. This **diffing** process allows React to detect only the parts of the UI that have changed. It then efficiently updates just those parts in the real DOM, minimizing costly DOM manipulations. By batching changes and applying minimal updates, React avoids unnecessary reflows and repaints in the browser, significantly boosting performance and ensuring faster, smoother UI updates.

React Life Cycle Methods

1. Mounting Phase (When the component is first created):

- `constructor()`: Initializes state and binds event handlers.
- `static getDerivedStateFromProps(props, state)`: Syncs state with props before the initial render. Rarely used.
- `render()`: The only required method, returns JSX to display.
- `componentDidMount()`: Runs after the first render. Ideal for API calls, subscriptions, or DOM manipulation.

2. Updating Phase (When props or state change):

- `static getDerivedStateFromProps(props, state)`: Also called during updates, syncing props to state before every render.
- `shouldComponentUpdate(nextProps, nextState)`: Optimizes performance by controlling re-renders. Returns `true` (default) or `false`.
- `render()`: Re-renders the component with updated state/props.
- `getSnapshotBeforeUpdate(prevProps, prevState)`: Captures information (like scroll position) before the DOM is updated.
- `componentDidUpdate(prevProps, prevState, snapshot)`: Called after the update. Useful for performing side effects based on prop/state changes.

3. Unmounting Phase (When the component is removed from the DOM):

- `componentWillUnmount()`: Cleans up resources like timers, event listeners, or subscriptions to prevent memory leaks.

4. Error Handling Phase (Error Boundaries):

- `static getDerivedStateFromError(error)`: Catches errors and updates the state to display a fallback UI.
- `componentDidCatch(error, info)`: Logs error details and provides info about the error source. Useful for debugging.

Diff between State and Props

Definition: State is a mutable object that represents a component's local data, while props are an immutable object used to pass data from a parent component to a child component.
Mutability: State can be changed within the component using `setState`, whereas props cannot be modified by the child component that receives them.
Purpose: State is used to manage data that can change over time within a component, while props are used to pass data and event handlers from parent components to child components.
Ownership: State is owned and managed by the component itself, whereas props are owned and controlled by the parent component.
Access: State is accessed directly via `this.state`, while props are accessed via `this.props`.
Updates: Updating state triggers a re-render of the component, while props updates are controlled by the parent component, requiring a re-render of the parent to pass new values down.

Higher Order Component

A Higher-Order Component (HOC) in React is a function that takes a component and returns a new component, allowing for code reuse and enhanced functionality

Why to use HOC

HOCs are used to enhance component reusability by encapsulating shared logic, making it easier to maintain and manage complex behaviors. They also promote separation of concerns by allowing components to focus on their primary purpose while the HOC handles additional functionality, like data fetching or authentication.

Use Cases -

Code Sharing , State Management , Memoization/Caching , Styling - Applying Themes ,Conditional Rendering

Real Life examples - React.memo , withRouter used in React Router , connect() in redux

Limitations of HOC -

1. **Props Collision**: HOCs can lead to conflicts if both the HOC and the wrapped component use the same prop names, potentially causing unexpected behavior.
2. **Wrapper Hell**: Overusing HOCs can create a deeply nested component tree, making the code harder to read and debug.
3. **Static Methods**: HOCs do not automatically copy static methods from the wrapped component, which can lead to loss of functionality unless explicitly handled.
4. **Performance Overhead**: Each HOC introduces an additional layer, which can impact performance if not managed properly, especially in large applications.

Use of Key Prop in react

In React, the `key` **prop** is essential for efficiently updating the Virtual DOM during the reconciliation process. When rendering lists of elements, React uses keys to uniquely identify each component, enabling it to quickly determine which items have changed, been added, or removed. When the state or props of a component change, React first updates the Virtual DOM. During this process, it compares the new Virtual DOM with the previous version using the keys as references. This comparison allows React to perform a more efficient diffing algorithm, focusing only on the components with different keys. As a result, React can avoid unnecessary re-renders by updating only the specific components that require changes, rather than re-rendering the entire list. By providing stable identities through keys, React enhances performance, maintains correct component state, and ensures a smoother user experience.

Memoization in React

Memoization is an optimization technique used to speed up function calls by caching the results of expensive function calls and returning the cached result when the same inputs occur again. This is particularly useful for functions that are called frequently with the same arguments. By storing the results, memoization can significantly reduce the amount of time spent recomputing values, thus improving performance.

React.memo is a higher-order component in React that uses memoization to optimize functional components. It prevents a component from re-rendering unless

its props have changed. When you wrap a functional component with `React.memo`, React will do a shallow comparison of the component's props.

Pure Components are a type of React component that only re-renders when there is a change in the component's props or state. They provide a way to optimize performance in React applications by implementing a shallow comparison of props and state.

Characteristics of Pure Components:

- **Shallow Comparison**: Pure components automatically perform a shallow comparison of their props and state. If no changes are detected, the component does not re-render.
- **Prevent Unnecessary Re-renders**: They help avoid unnecessary re-renders, which can enhance performance, especially in applications with complex UI or large component trees

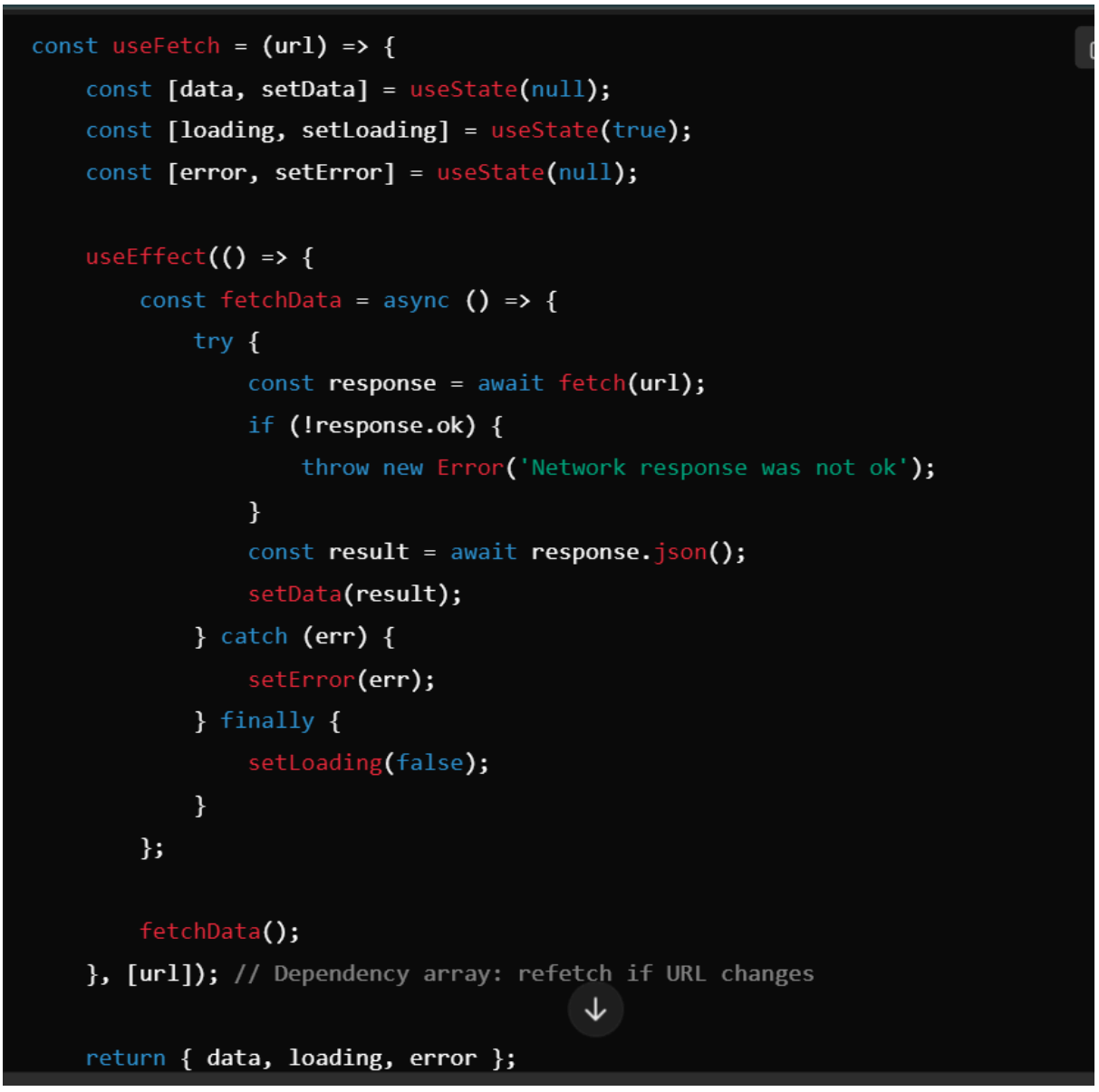
Pure Components can be used in class-based components. In React, you can create a pure component by extending `React.PureComponent` instead of `React.Component`.

```
import React from 'react';

class MyPureComponent extends React.PureComponent {
  render() {
    return (
      <div>
        <h1>{this.props.title}</h1>
        <p>{this.props.content}</p>
      </div>
    );
  }
}
```

Custom Hooks in React

Custom hooks in React are a powerful way to reuse stateful logic across different components.



```
const useFetch = (url) => {
    const [data, setData] = useState(null);
    const [loading, setLoading] = useState(true);
    const [error, setError] = useState(null);

    useEffect(() => {
        const fetchData = async () => {
            try {
                const response = await fetch(url);
                if (!response.ok) {
                    throw new Error('Network response was not ok');
                }
                const result = await response.json();
                setData(result);
            } catch (err) {
                setError(err);
            } finally {
                setLoading(false);
            }
        };

        fetchData();
    }, [url]); // Dependency array: refetch if URL changes

    return { data, loading, error };
```

```jsx
import React from 'react';
import useFetch from './useFetch';

const DataDisplay = () => {
    const { data, loading, error } = useFetch('https://api.example.com/data');

    if (loading) return <p>Loading...</p>;
    if (error) return <p>Error: {error.message}</p>;

    return (
        <div>
            <h1>Fetched Data:</h1>
            <pre>{JSON.stringify(data, null, 2)}</pre>
        </div>
    );
};

export default DataDisplay;
```


Benefits of Custom Hooks

- Reusability: You can use the same logic across different components without duplicating code.
- Separation of Concerns: Custom hooks help keep your components cleaner by abstracting complex logic into separate functions.
- Easier Testing: Logic contained in hooks can be tested independently from the components that use them.

Fragments in React

Fragment is a component that allows you to group multiple elements without adding extra nodes to the DOM. It's particularly useful when you need to return multiple elements from a component's render method without wrapping them in an additional HTML element like a `<div>`.

Why Use Fragments?

Avoiding Unnecessary Markup: Using fragments helps keep the HTML structure cleaner by preventing extra wrapper elements that can affect styling or layout.
Grouping Elements: If you want to group a list of children or a set of elements without introducing extra nodes in the DOM, fragments are the way to go.

Performance: By avoiding unnecessary DOM nodes, fragments can improve performance slightly, especially in large applications.

State Lifting in React

State lifting in React is a concept where you move the state of a component up to its parent component to allow shared access between sibling components

Why Lift State?

1. **Shared State**: When two or more sibling components need to access the same piece of state, lifting state to their common ancestor allows them to synchronize their behavior based on that state.
2. **Simplified State Management**: It simplifies state management and avoids duplication of state across multiple components.
3. **Better Component Structure**: Lifting state can help keep components more focused and reusable, as they can rely on props for their data rather than managing their own state.

Synthetic Events in React/ Why were they introduced?

Synthetic events in React are a cross-browser wrapper around the browser's native events, designed to provide a consistent and normalized interface for handling events in a React application. They were introduced to address issues related to inconsistent event handling across different browsers, ensuring that events behave the same way regardless of the environment. By using Synthetic events, React can enhance performance through event delegation, where a single event listener is attached to a parent element instead of multiple listeners on child elements. This approach reduces memory usage and improves efficiency. Additionally, Synthetic events are pooled, meaning that they are reused for performance gains, which also leads to a slight difference in how event properties are accessed (as they may be nullified after the event callback is executed). Overall, Synthetic events help streamline event management in React, making it easier to write and maintain event-driven code.

Pure Components

In React, **Pure Components** are components that only re-render when their props or state change. They implement a shallow comparison of the props and state to determine if a re-render is necessary, which can lead to performance improvements, especially in complex applications.

Key Features of Pure Components:

1. **Shallow Comparison**: Pure components use a shallow comparison to check if the values of props and state have changed. If they haven't changed, the component does not re-render.
2. **Performance Optimization**: By preventing unnecessary re-renders, pure components can improve the performance of your application, particularly for components that receive the same props repeatedly.
3. **Implementation**: Pure components can be created using `React.PureComponent`, which is a base class that extends `React.Component` with the additional functionality of shallow prop and state comparison.

```
import React from 'react';

class MyPureComponent extends React.PureComponent {
  render() {
    return (
      <div>
        <h1>{this.props.title}</h1>
        <p>{this.props.content}</p>
      </div>
    );
  }
}
```

How to validate Props in React?

1. Using PropTypes

React provides a built-in package called `prop-types` for validating props. Here's how you can use it:

Step 1: Install `prop-types`

If you haven't already, you need to install the `prop-types` package:

```
npm install prop-types
```

Step 2: Define PropTypes in Your Component

You can define the expected prop types for your component by using the `propTypes` property:

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

const MyComponent = ({ name, age }) => {
    return (
        <div>
            <h1>Hello, {name}!</h1>
            <p>Your age is {age}.</p>
        </div>
    );
};

MyComponent.propTypes = {
    name: PropTypes.string.isRequired, // name should be a string and is required
    age: PropTypes.number,             // age should be a number but is optional
};
```

Step 3: Check for Prop Validation

When the component is used, React will log a warning in the console if the props do not match the specified types:

```jsx
<MyComponent name="John" age={30} />    // Correct
<MyComponent name={123} age="thirty" /> // Warnings in the console
```

2. Using TypeScript

If you prefer static type checking, you can use TypeScript to validate props. TypeScript provides strong typing capabilities, allowing you to define interfaces or types for your component props.

Example of TypeScript Props Validation

```tsx
import React from 'react';

interface MyComponentProps {
    name: string;
    age?: number; // age is optional
}

const MyComponent: React.FC<MyComponentProps> = ({ name, age }) => {
    return (
        <div>
            <h1>Hello, {name}!</h1>
            <p>Your age is {age}.</p>
        </div>
    );
};

export default MyComponent;
```

3. Custom Prop Validation

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

const customPropType = (props, propName, componentName) => {
    if (!/^[A-Z]/.test(props[propName])) {
        return new Error(`Invalid prop ${propName} supplied to ${componentName}. It shou
    }
};

const MyComponent = ({ title }) => {
    return <h1>{title}</h1>;
};

MyComponent.propTypes = {
    title: customPropType.isRequired, // Custom validation
};

export default MyComponent;
```

Clean Up Function in UseEffect -

```
useEffect(() => {
  // Side effect logic here

  return () => {
    // Cleanup logic here
  };
}, [dependencies]);
```

```
useEffect(() => {
  const intervalId = setInterval(() => {
    console.log('This will run every second');
  }, 1000);

  // Cleanup function
  return () => {
    clearInterval(intervalId); // This will clear the
  };
}, []);
```

Prop Drilling

Prop drilling is a term used in React to describe the process of passing data from a parent component down to a deeply nested child component through multiple layers of intermediary components.

Solutions to Prop Drilling

To avoid prop drilling, developers can use several approaches:

1. **Context API**
2. **State Management Libraries** - Redux

Controlled vs UnControlled Components

Feature	Controlled Components	Uncontrolled Components
Definition	Component's state is managed by React.	Component's state is managed by the DOM.
State Management	Uses `state` and `setState` to manage input values.	Uses `refs` to access input values.
Data Flow	One-way data flow; parent component controls input.	Two-way data flow; input value can change independently.
Form Submission	Form data is retrieved from the component's state.	Form data is accessed directly from the DOM when needed.
Validation	Validation can be performed in React's state.	Validation requires accessing the DOM directly.
Reactivity	Automatically re-renders when state changes.	Requires manual updates or event handling to re-render.
Ease of Testing	Easier to test due to clear state management.	Harder to test as state is not directly managed by React.
Performance	May have performance overhead due to frequent re-renders.	Generally more performant for simple forms, but can lead to inconsistencies.

Stateful vs Stateless Components

Feature	Stateful Components	Stateless Components
Definition	Components that manage their own state.	Components that do not manage state.
State Management	Uses `state` and `setState` to manage internal state.	Receives data and renders UI based on props.
Lifecycle Methods	Can use lifecycle methods (e.g., `componentDidMount`, `componentDidUpdate`).	Do not have lifecycle methods.
Reactivity	Can trigger re-renders based on state changes.	Renders based on props without reactivity to internal state.
Performance	Can have performance overhead due to state management and re-renders.	Generally more performant, as they are simpler and pure.
Use Case	Suitable for components that require interaction or dynamic behavior.	Suitable for presentational components that display data.
Example	Class components or functional components using hooks (e.g., `useState`, `useEffect`).	Functional components that only receive props and render UI.

Advantages of react/ React is better than other frameworks

- Component-Based Architecture (Reusable, modular components)
- Virtual DOM (Efficient updates and fast rendering)
- Declarative UI (Simplified and predictable UI development)
- Large Community & Ecosystem (Many resources and libraries)
- JSX Syntax (Combines HTML-like code within JavaScript)
- One-Way Data Binding (Predictable data flow, easier debugging)
- Performance Optimizations (Lazy loading, code splitting)
- React Native (Cross-platform mobile app development)
- Integration Flexibility (Works well with other libraries/frameworks)
- Strong Backward Compatibility (Easy upgrades)

JSX

JSX stands for JavaScript XML and it is an XML-like syntax extension to ECMAScript. Basically it just provides the syntactic sugar for the React.createElement(type, props, ...children) function, giving us expressiveness of JavaScript along with HTML like template syntax.

React Performance and Tools

To improve front-end performance, focus on asset optimization through minification, compression, and lazy loading of images and scripts. Implement code splitting and tree-shaking to reduce the initial JavaScript payload. Use browser caching, service workers, and a CDN for faster content delivery. Analyze performance with tools like Google Lighthouse and Chrome DevTools, and optimize Core Web Vitals (LCP, FID, CLS) for better user experience. Employ efficient DOM manipulation using frameworks like React or Vue, minimize JavaScript execution, and use modern techniques like CSS Grid and Flexbox for better rendering performance.

Client Side Rendering vs Server Side Rendering

1. Client-Side Rendering (CSR)

- **Where Rendering Happens**: In CSR, the browser (client) is responsible for rendering the content. The server sends a bare-bones HTML page with JavaScript, and the client executes the JavaScript to generate and display the content.
- **Performance**: Initial load is slower because the browser must download and execute the JavaScript before showing the content. Subsequent interactions, however, are faster, as the page doesn't need to be reloaded.
- **SEO**: CSR can be less SEO-friendly because search engine bots may not fully render the page's content, which relies on JavaScript.
- **User Experience**: It can show a blank or loading screen initially while JavaScript is fetching data and rendering the content.

2. Server-Side Rendering (SSR)

- **Where Rendering Happens**: In SSR, the content is generated on the server. The server sends a fully rendered HTML page to the client, which can be displayed immediately.
- **Performance**: Initial load is faster since the browser gets fully rendered content, but subsequent navigation might require additional page reloads, depending on the framework used.
- **SEO**: SSR is more SEO-friendly because the content is present in the HTML when it is sent to the client, making it easier for search engines to crawl.
- **User Experience**: Users see a fully rendered page quickly, but subsequent interactions may be slower unless hybrid approaches like **hydration** (e.g., in React) are used.