



Introduction to React

1. What is React?
2. Working of DOM
3. Problems with JS
4. Working of React
5. JS Vs React
6. Intro to Components



1. What is React



1. JavaScript library to build Dynamic and interactive user interfaces
2. Developed at Facebook in 2011.
3. Currently most widely used JS library for front-end development.
4. Used to create single page application (page does not re-load).

2. Working of DOM



1. Browser takes **HTML** and create **DOM**.
2. JS helps us modify **DOM** based on **user actions** or **events**.
3. In **big applications**, Working with **DOM** becomes **complicated**.

3. Problems with JavaScript

1. React has a simpler mental model
2. JS is cumbersome
3. JS is Error-prone
4. JS is Hard to maintain

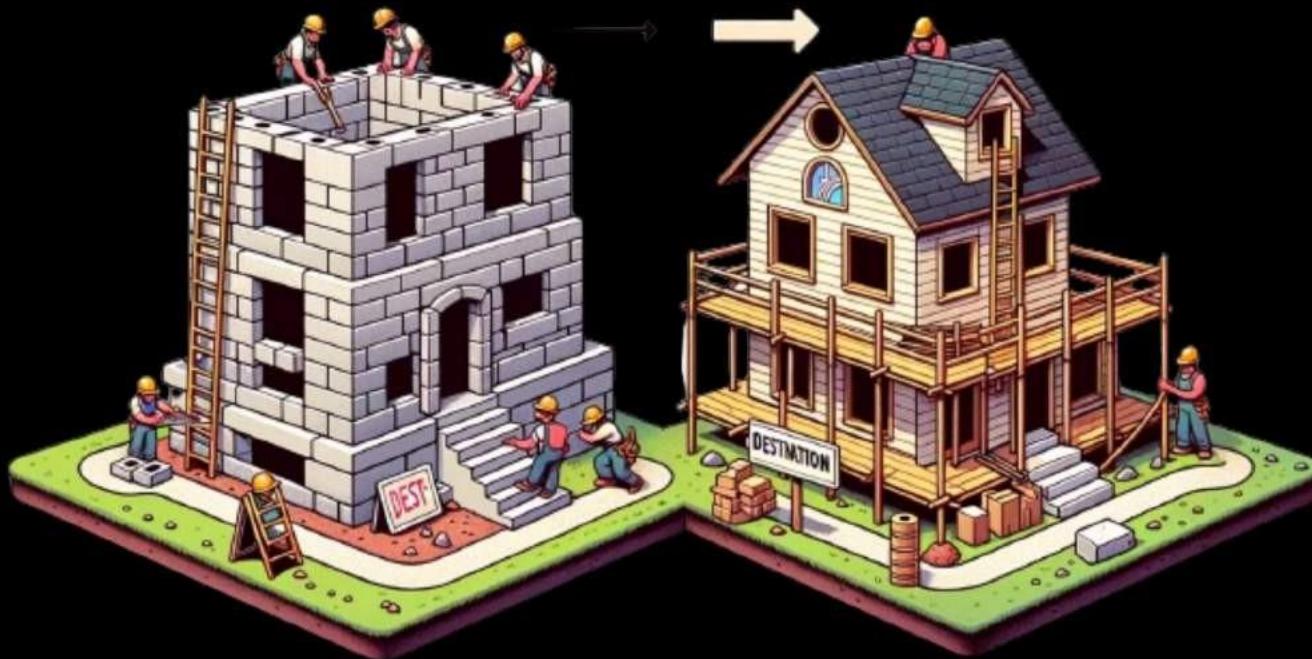


4. Working of React



1. No need to worry about querying and updating DOM elements.
2. React creates a web page with small and reusable components
3. React will take care of creating and updating DOM elements.
4. IT saves a lot of time, cheezein aasan hai, pahele se likhi hui hain

5. JS Vs React



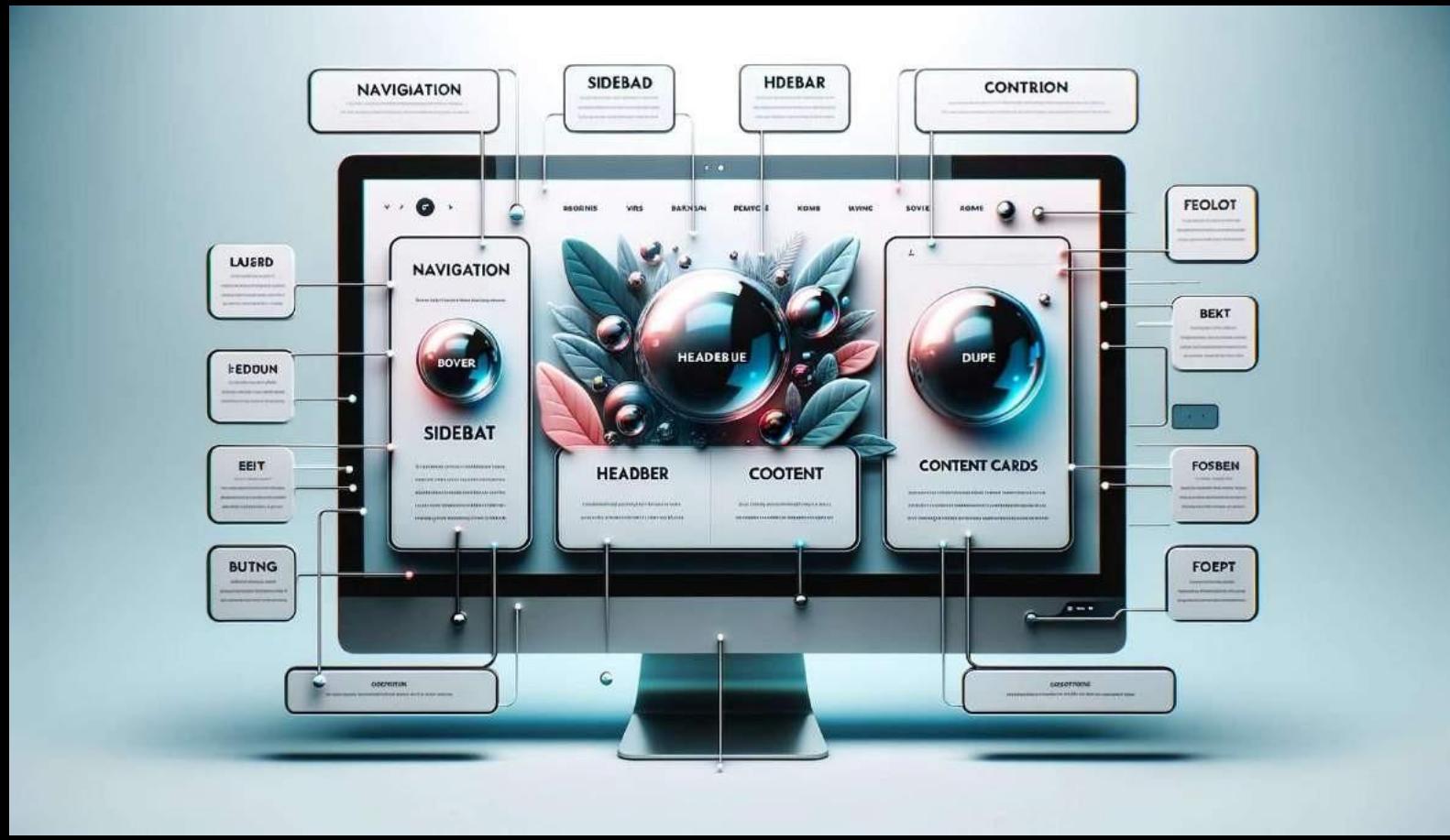
1. JS is imperative: You define **steps** to reach your **desired state**.
2. React is Declarative: You define the **target UI state** and then react figures out how to reach that state.

6. Introduction to Components

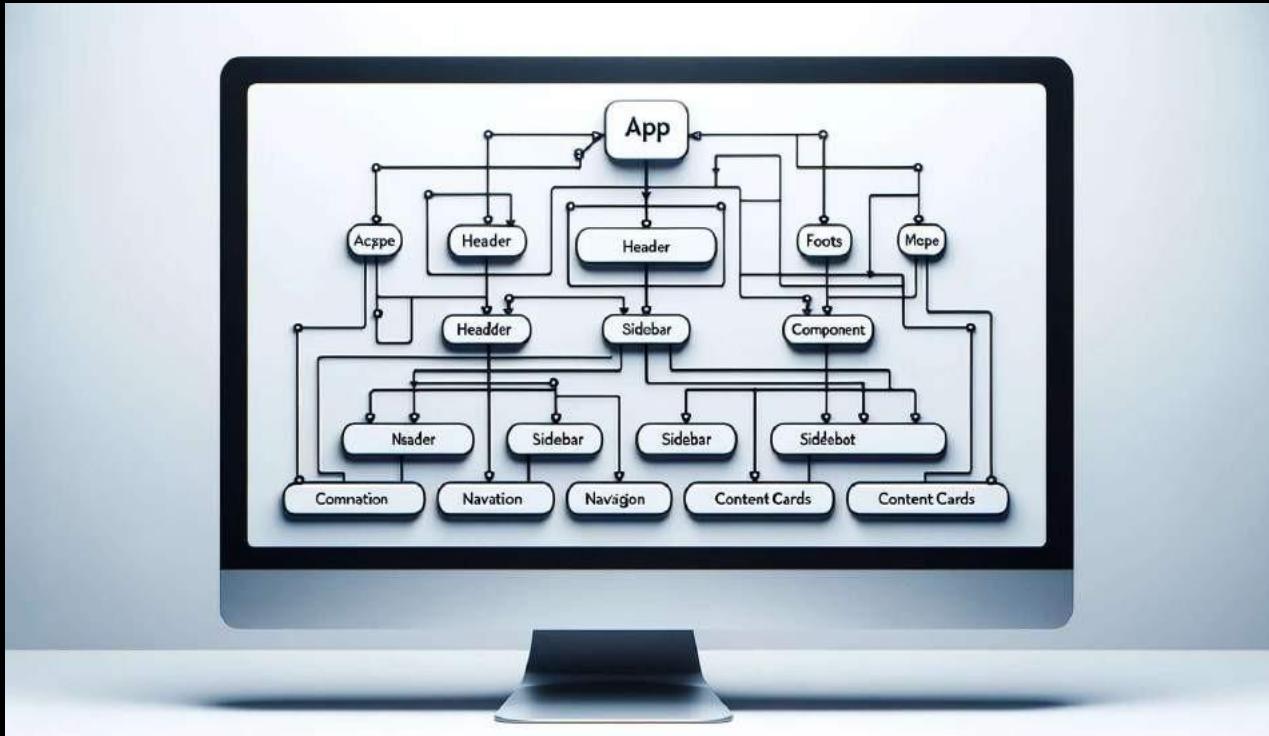


Components help us write reusable, modular and better organized code.

6. Introduction to Components



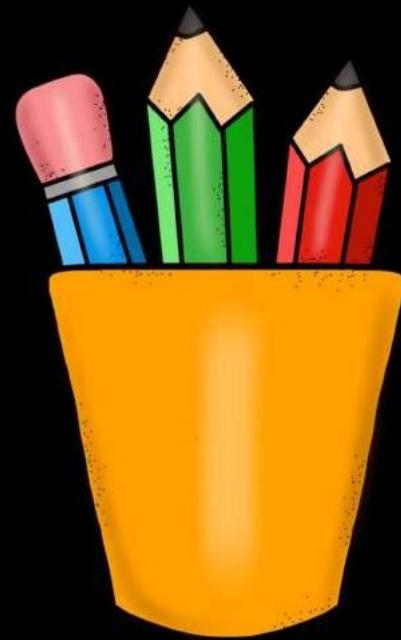
6. Introduction to Components

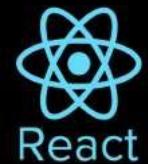


React application is a tree of components with App Component as the root bringing everything together.

Introduction Revision

1. What is React?
2. Working of DOM
3. Problems with JS
4. Working of React
5. JS Vs React
6. Intro to Components





Create a React App

7. Setup IDE
8. Create a React App
9. Project Structure



7. What is IDE

1. IDE stands for **Integrated Development Environment**.
2. Software suite that consolidates basic tools required for **software development**.
3. Central hub for **coding**, finding problems, and testing.
4. Designed to improve **developer efficiency**.



7. Need of IDE

1. Streamlines development.
2. Increases productivity.
3. Simplifies complex tasks.
4. Offers a unified workspace.
5. IDE Features
 1. Code Autocomplete
 2. Syntax Highlighting
 3. Version Control
 4. Error Checking



```
MainActivity.kt
```

```
@Composable
fun MessageCard(msg: Message) {
    Row(modifier = Modifier.padding(all = 8.dp)) {
        Image(
            painter = painterResource(R.drawable.android_studio_logo),
            contentDescription = "Profile Picture",
            modifier = Modifier
                .size(45.dp)
        )
        Spacer(modifier = Modifier.width(8.dp))
        Column (Modifier
            .background(color = Color.White)) {
            Text(text = msg.author, color = Color.Black)
            Spacer(modifier = Modifier.height(1.dp))
            Text(text = msg.body, color = Color.Black)
        }
    }
}
```



7. Install latest Node

1. Search Download NodeJS



7. Installation & Setup

1. Search VS Code
2. Keep Your **Software** up to date

7. VsCode Extensions and Settings

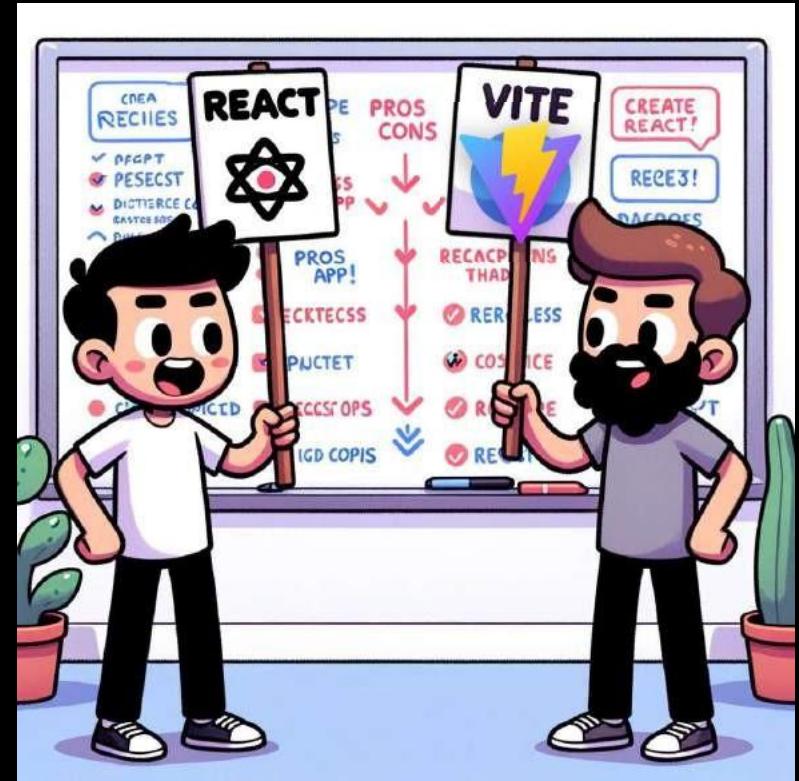
1. Live Server / Live Preview
2. Prettier (Format on Save)
3. Line Wrap
4. Tab Size from 4 to 2





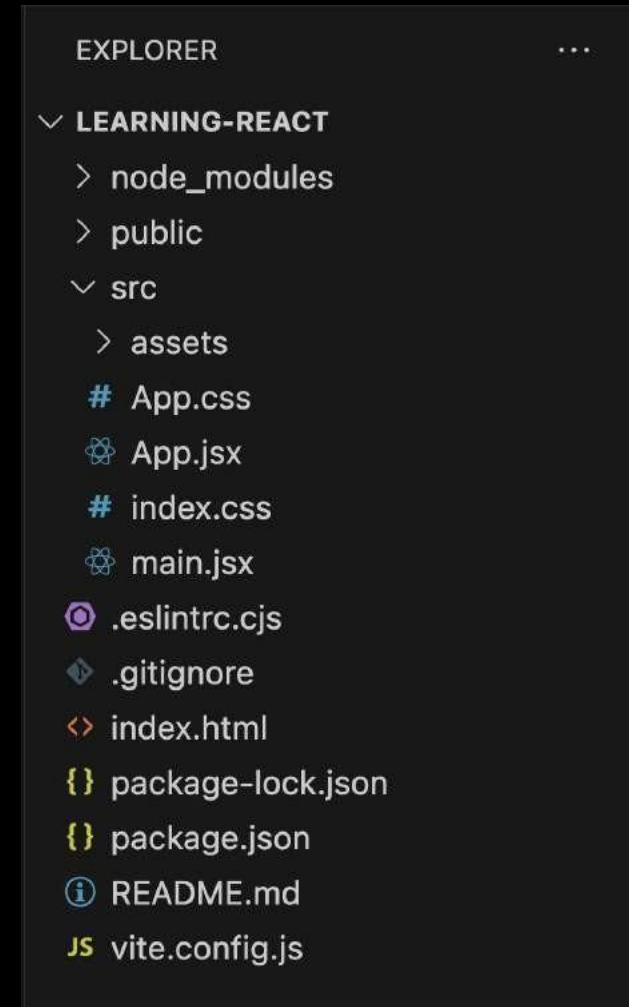
8. Create a React App

1. Official tool is CRA(Create React APP)
2. Vite is a modern tool to create React Project.
3. Vite produces Quick and Small bundle size.
4. Vite: Use `npm run dev` to launch dev server.
5. Use `npm start` for CRA.



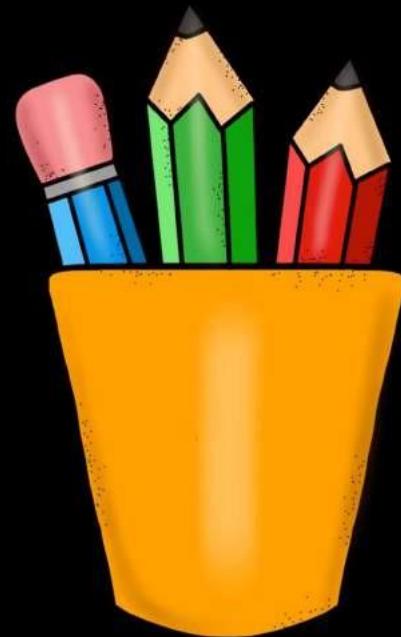
9. Project Structure

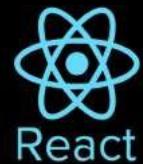
1. `node_modules`/ has all the **installed node packages**
2. `public/` Directory: Contains **static files** that don't change.
3. `src/` Directory: **Main folder** for the React **code**.
 1. `components/`: **Reusable parts** of the UI, like buttons or headers.
 2. `assets/`: **Images, fonts**, and other static files.
 3. `styles/`: **CSS** or **stylesheets**.
4. `package.json` contains information about this project like **name, version, dependencies** on other react packages.
5. `vite.config.js` contains **vite config**.



Create a React App Revision

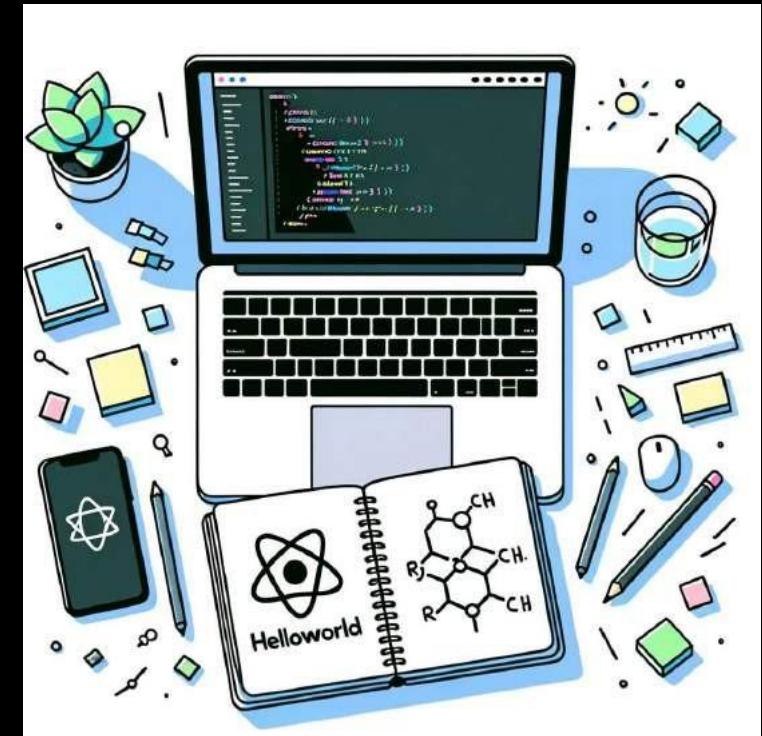
7. Setup IDE
8. Create a React App
9. Project Structure





Creating React Components

10. File Extensions
11. Class vs Function Components
12. What is JSX?
13. Exporting component
14. Other important Points
15. Dynamic Components
16. Reusable Components



10. File Extensions



.JS

- Stands for **JavaScript**
- Contains **regular JavaScript** code
- Used for **general logic** and components

.JSX

- Stands for **JavaScript XML**
- Combines **JavaScript** with **HTML-like tags**
- Makes it easier to design **UI components**



11. Class vs Function Components

Class Components

- **Stateful:** Can manage state.
- **Lifecycle:** Access to lifecycle methods.
- **Verbose:** More boilerplate code.
- Not Preferred anymore.

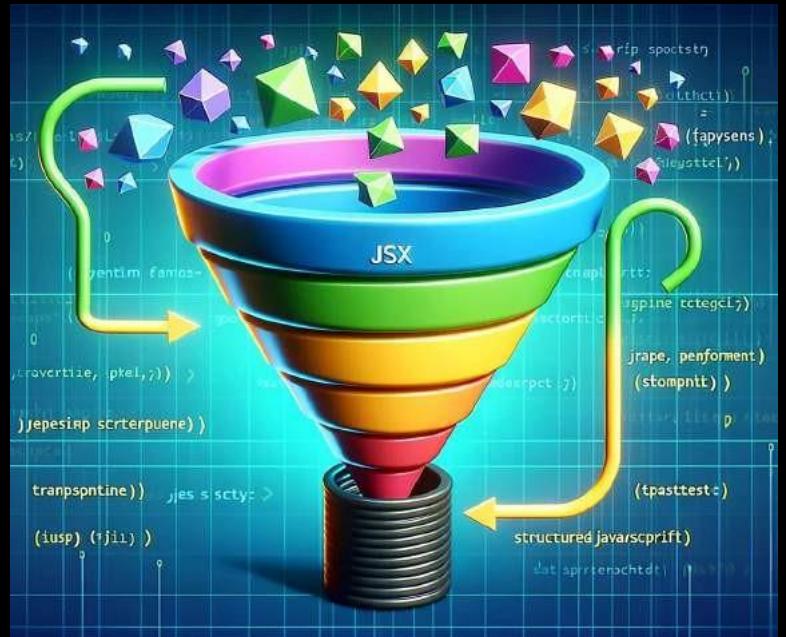
Functional Components

- Initially **stateless**.
- Can use **Hooks** for state and effects.
- **Simpler and more concise.**
- More Popular.



12. What is JSX?

1. **Definition:** JSX determines how the UI will look wherever the component is used.
2. **Not HTML:** Though it **resembles** HTML, you're actually writing JSX, which stands for JavaScript XML.
3. **Conversion:** JSX gets converted to regular JavaScript.
4. **Babeljs.io/repl** is a tool that allows you to see how **JSX** is transformed into **JavaScript**.



13. Exporting components

```
Component.js
[redacted]
[redacted]
export default
function Button() {
  ...
}
```

one default export

```
Components.js
export function Slider() {
  ...
}

export function Checkbox() {
  ...
}
```

multiple named exports

```
MixedComponents.js
export function Avatar() {
  ...
}

export default
function FriendsList() {
  ...
}
```

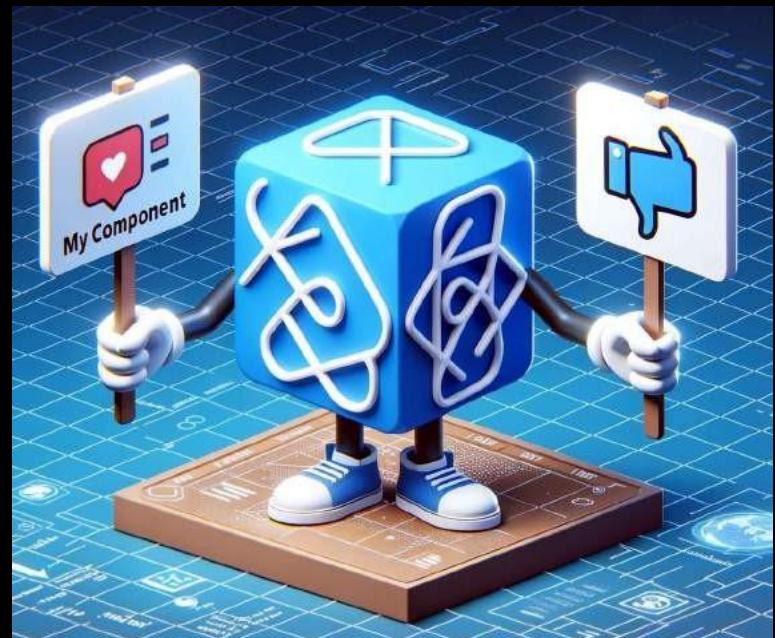
named export(s)
and one default export



1. Enables the use of a component in other parts.
2. Default Export: Allows exporting a single component as the default from a module.
3. Named Export: Allows exporting multiple items from a module.
4. Importing: To use an exported component, you need to import it in the destination file using import syntax.

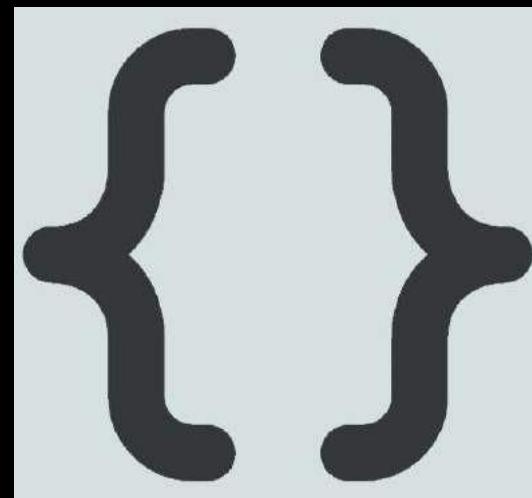
14. Other important Points

1. **Naming:** Must be capitalized; lowercase for default HTML.
2. **HTML:** Unlike vanilla JS where you can't directly write HTML, in React, you can embed HTML-like syntax using JSX.
3. **CSS:** In React, CSS can be directly imported into component files, allowing for modular and component-specific styling.



15. Dynamic Components

1. Dynamic Content: JSX allows the creation of dynamic and interactive UI components.
2. JavaScript Expressions: Using {}, we can embed any JS expression directly within JSX. This includes variables, function calls, and more.



16. Reusable Components

1. **Modularity:** Components are modular, allowing for **easy reuse across different parts** of an application.
2. **Consistency:** Reusing components ensures **UI consistency** and reduces the chance of discrepancies.
3. **Efficiency:** Reduces development time and effort by **avoiding duplication of code**.
4. **Maintainability:** Changes made to a reused component **reflect everywhere** it's used, simplifying updates and bug fixes.



Creating React Components Revision

10. File Extensions

11. Class vs Function Components

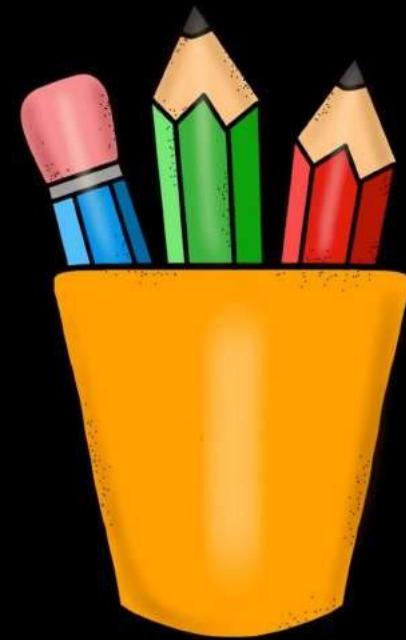
12. What is JSX?

13. Exporting component

14. Other important Points

15. Dynamic Components

16. Reusable Components



17. Including Bootstrap

1. **Responsive:** Mobile-first design for all device sizes.
2. **Components:** Pre-styled elements like buttons and navbars.
3. **Customizable:** Modify default styles as needed.
4. **Cross-Browser:** Consistent look across browsers.
5. **Open-Source:** Free with community support.



1 **Install:**

```
npm i bootstrap@5.3.2
```

2. **import**

```
import "bootstrap/dist/css/bootstrap.min.css";
```

Project: TODO App UI



Project: Clock

Bharat Clock

This is the clock that shows the time in Bharat at all times

This is the current time: 26/10/2023 - 10:38:17 AM

18. Fragments

1. What?

Allows grouping of multiple elements without extra DOM nodes.

2. Why?

- Return multiple elements **without a wrapping parent**.
- **Cleaner DOM** and consistent styling.

3. How? Two syntaxes:

1. `<React.Fragment>...</React.Fragment>`
2. Short: `<>...</>`



19. Map Method



1. Purpose: Render lists from array data.
2. JSX Elements: Transform array items into JSX.

3. Inline Rendering: Directly inside JSX

```
{items.map(item => <li key={item.id}>{item.name}</li>)}
```

4. Key Prop: Assign unique key for optimized re-renders.

```
<div key={item.id}>{item.name}</div>
```

20. Conditional Rendering

Conditional Rendering

- Displaying content based on **certain conditions**.
- Allows for **dynamic** user interfaces.

Methods

- **If-else statements:** Choose **between** two blocks of content.
- **Ternary operators:** Quick way to choose between **two options**.
- **Logical operators:** Useful for rendering content when a condition is **true**.

Benefits

- Enhances user **experience**.
- Reduces unnecessary rendering.
- Makes apps more **interactive** and **responsive**.

```
{ condition && <div>Write something</div> }  
  
{ !condition ? <div>Error do it again</div> :  
            <div>Congratulations</div> }
```

21. Passing Data via Props

Props in React

- Short for **properties**
- Mechanism for **passing data**.
- **Read-only** by default

Usage

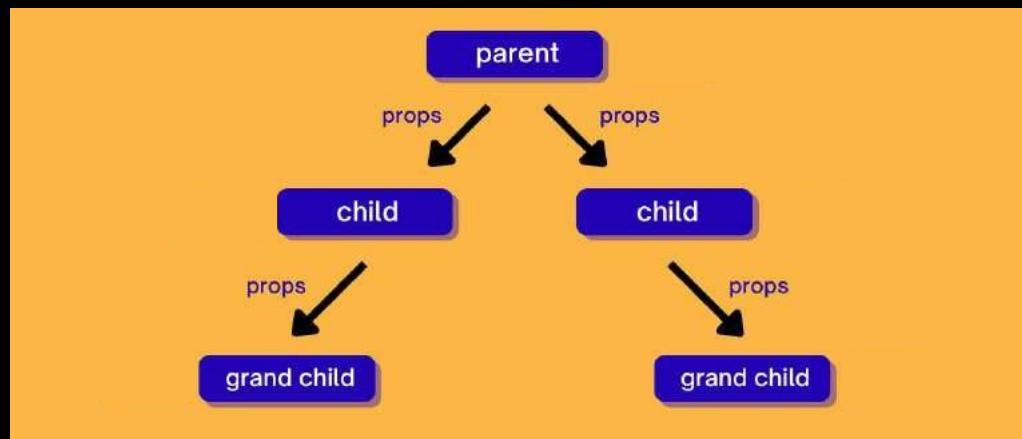
- Pass data from **parent** to **child** component.
- Makes components **reusable**.
- Defined as **attributes** in JSX.

Key Points

- Data flows **one-way** (downwards).
- Props are **immutable**.
- Used for **communication between components**.

Examples

```
<Header title="My App" />
```



22. CSS Modules

Cat.css

```
.meow {  
  color: orange;  
}
```



**CSS Modules
Compiler**

CSS

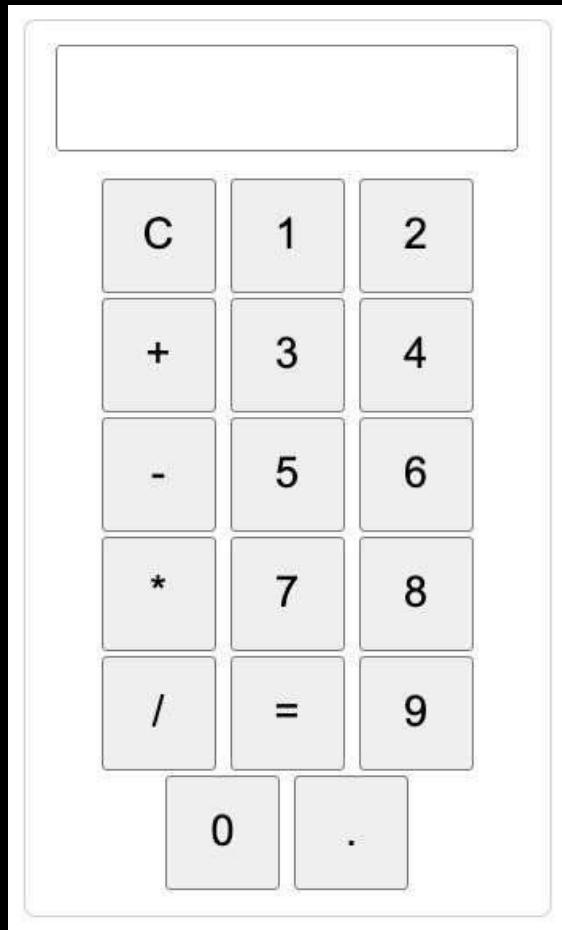
```
.cat_meow_j3xk {  
  color: orange;  
}
```

1. Localized class names to **avoid global conflicts**.
2. Styles are **scoped** to individual components.
3. Helps in creating **component-specific** styles.
4. Automatically generates **unique class names**.
5. Promotes **modular** and **maintainable** CSS.
6. Can use alongside **global CSS** when needed.

Project: TODO App UI



Project Calculator



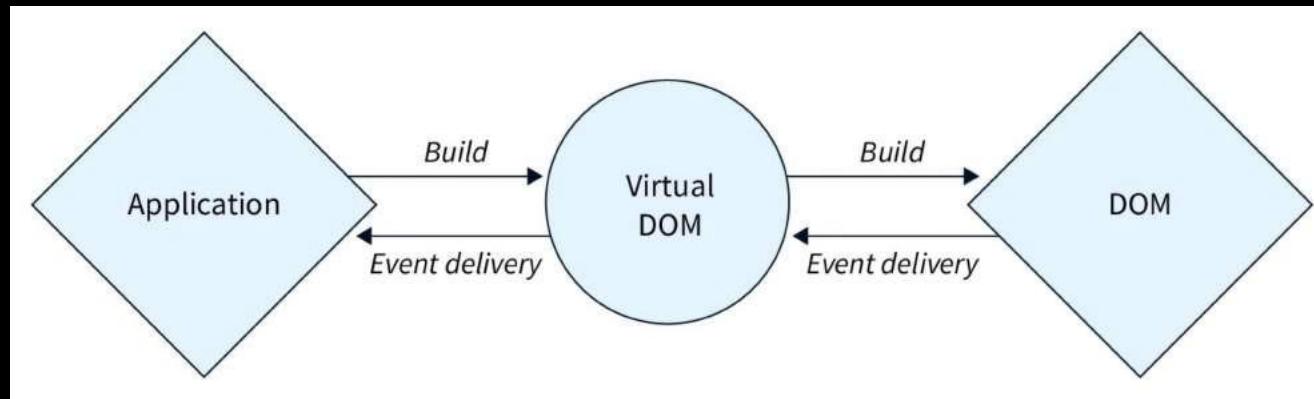
23. Passing Children

```
function Container(props) {  
  return (  
    <div className="container-style">  
      {props.children}  
    </div>  
  );  
}
```

```
<Container>  
  <h1>Welcome to My App</h1>  
  <p>This content is passed as children to the  
    Container component.</p>  
</Container>
```

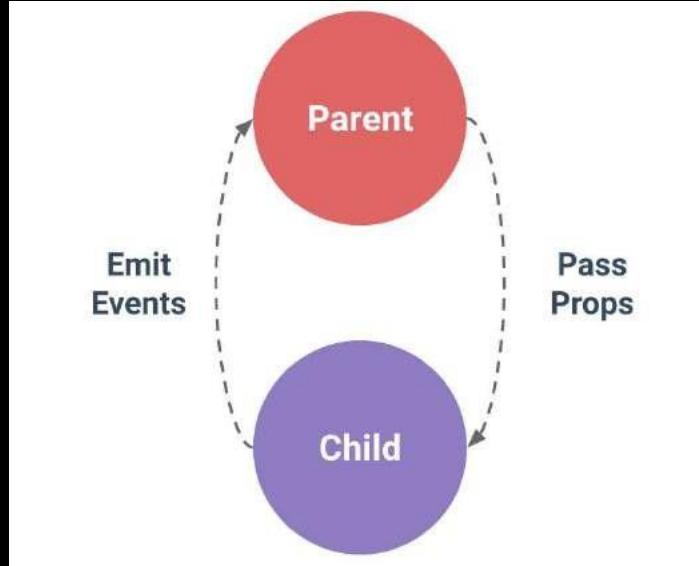
- 1 **children** is a special prop for passing **elements** into components.
2. Used for **flexible** and **reusable** component designs.
3. Common in layout or **container components**.
4. Accessed with **props.children**.
5. Can be any content: strings, numbers, **JSX**, or components.
6. Enhances component **composability** and **reusability**.

24. Handling Events



1. React events use **camelCase**, e.g., `onClick`.
2. Uses **synthetic events**, not direct browser events.
3. Event handlers can be **functions** or **arrow functions**.
4. Use `onChange` for controlled form inputs.
5. Avoid inline arrow functions in **JSX** for performance.

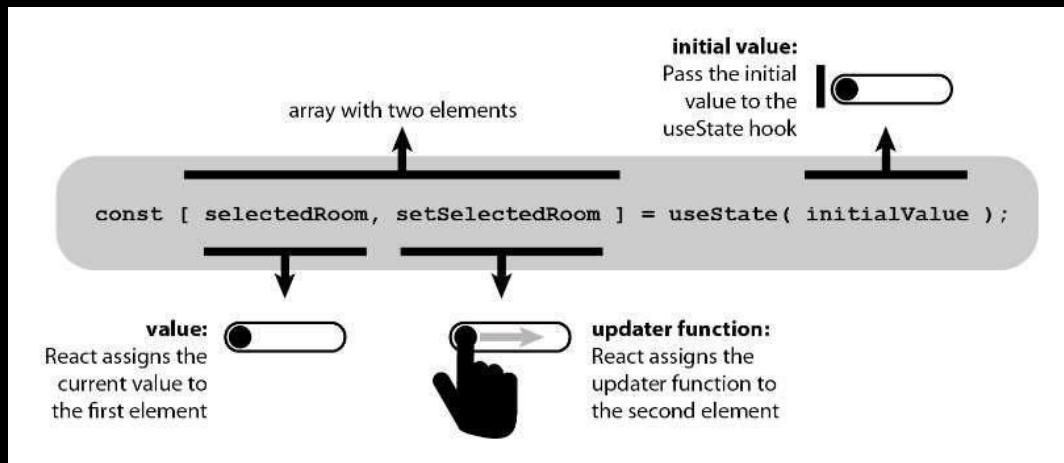
25. Passing Functions via Props



1. Pass dynamic behaviour between components.
2. Enables upward communication from child to parent.
3. Commonly used for event handling.
4. Parent defines a function, child invokes it.
5. Enhances component interactivity.
6. Example:

```
<Button onClick={handleClick} />
```

26. Managing State



- 1 State represents **data that changes over time**.
- 2 State is **local** and **private** to the component.
- 3 State changes cause the component to **re-render**.
- 4 For functional components, use the **useState** hook.
- 5 React Functions that start with word **use** are called **hooks**
- 6 Hooks should **only** be used **inside components**
- 7 Parent components can **pass state down to children via props**.
- 8 Lifting state up: share state between components by moving it to **their closest common ancestor**.

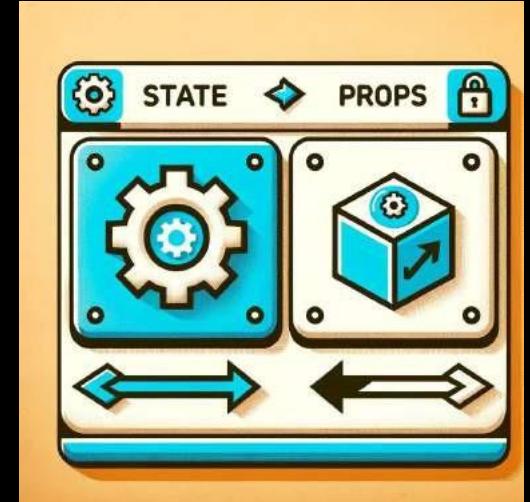
27. State vs Props

State:

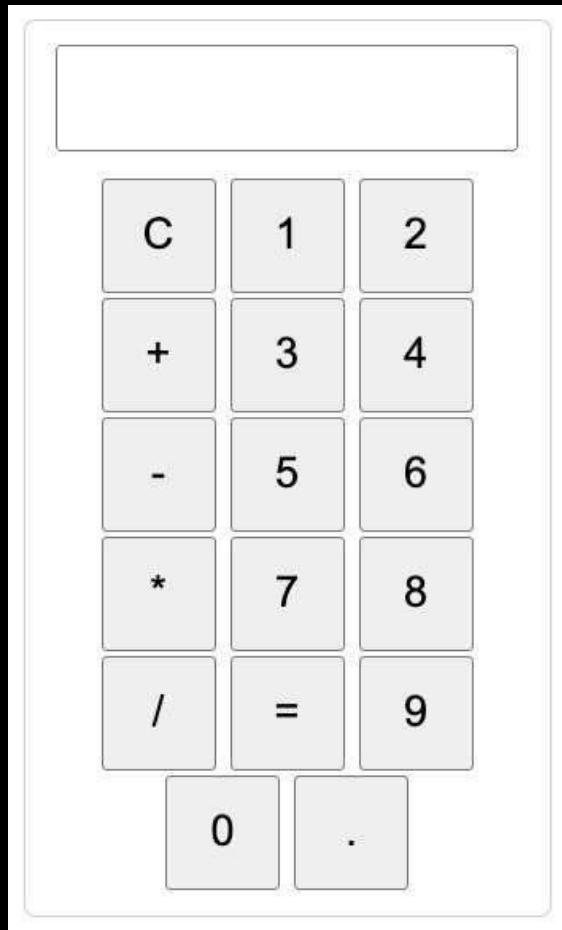
- Local and mutable data within a component.
- Initialized within the component.
- Can change over time.
- Causes re-render when updated.
- Managed using `useState` in functional components.

Props:

- Passed into a component from its parent.
- Read-only (immutable) within the receiving component.
- Allow parent-to-child component communication.
- Changes in props can also cause a re-render.



Project Calculator



Project: TODO App

Todo App

Enter Todo here

dd/mm/yyyy

Add

Todo Item	Due Date	
Buy Milk	4/10/2023	Delete
Go to College	4/10/2023	Delete

28. React-icon Library

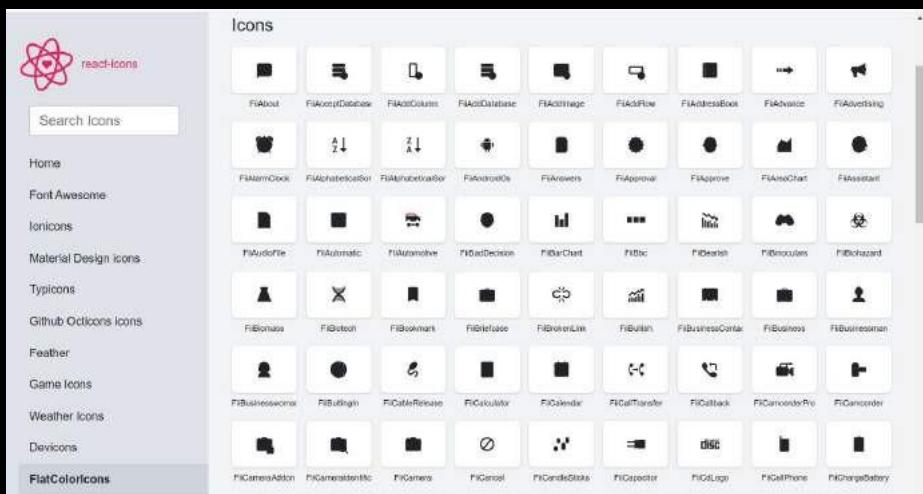
1. You can use a lot of icons without managing them.

2. Install Package

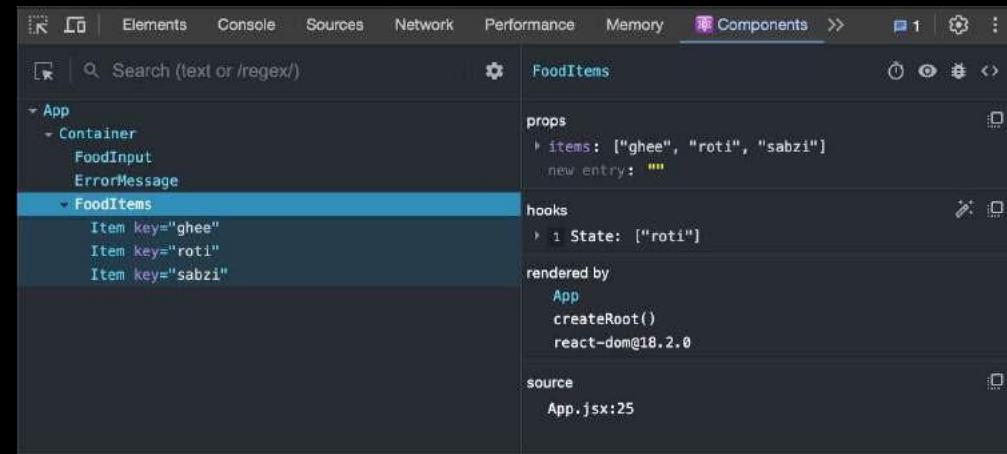
```
npm install react-icons -save
```

3. Use icon:

```
import {IconName} from "react-icons/fc";
```



29. Inspecting with React Dev Tools



1. Inspection: Allows **inspection** of **React component hierarchies**.
2. State & Props: **View and edit** the **current state** and **props** of components.
3. Performance: **Analyze** component **re-renders** and performance bottlenecks.
4. Navigation: Conveniently **navigate** through the **entire component tree**.
5. Filtering: **Filter** components **by name or source** to locate them quickly.
6. Real-time Feedback: See **live changes** as you modify state or props.

30. How React Works

Root Component:

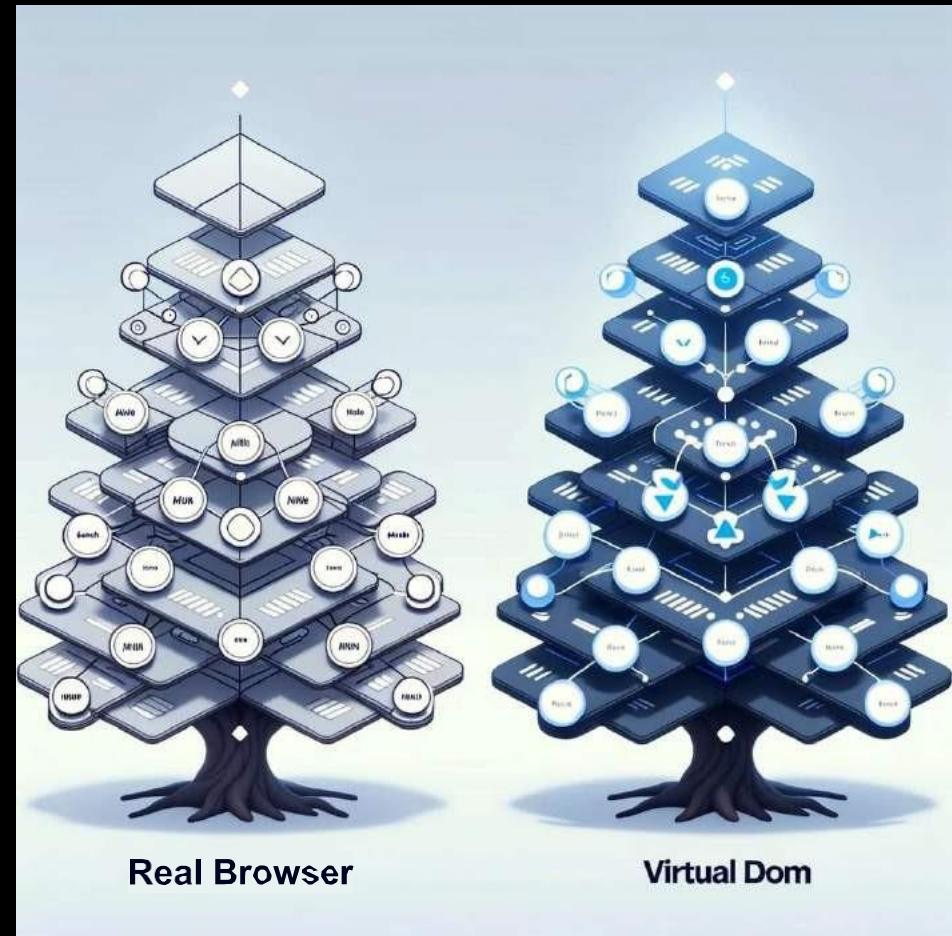
- The **App** is the main or **root component** of a React application.
- It's the **starting point** of your React component tree.

Virtual DOM:

- React creates an **in-memory structure** called the virtual DOM.
- **Different** from the actual browser DOM.
- It's a **lightweight representation** where each node stands for a component and its attributes.

Reconciliation Process:

- When **component data changes**, React **updates** the **virtual DOM's state** to mirror these changes
- React then **compares** the **current** and **previous versions** of the virtual DOM.
- It **identifies** the **specific nodes** that need updating
- **Only** these nodes are **updated** in the real browser DOM, making it efficient.



30. How React Works

React and ReactDOM:

- The **actual updating** of the browser's DOM **isn't done by React** itself.
- It's handled by a **companion library** called **react-dom**.

Root Element:

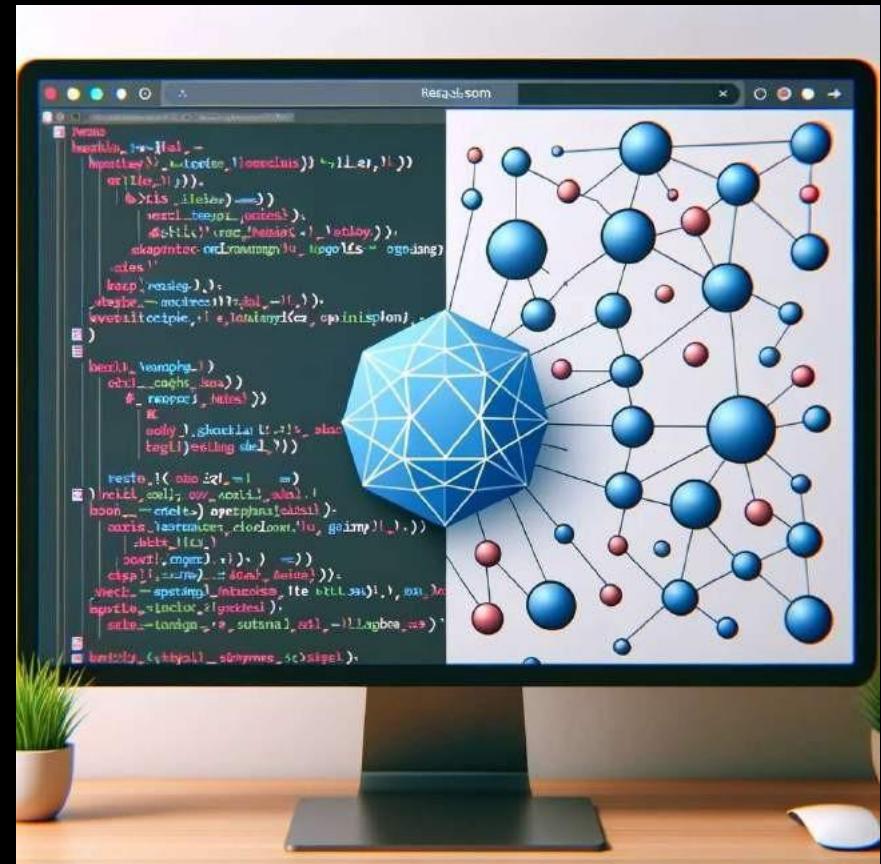
- The **root div** acts as a **container** for the React app.
- The **script tag** is where the React app **starts executing**
- If you check **main.tsx**, the component tree is rendered inside this root element.

Strict Mode Component:

- It's a **special component** in React.
- Doesn't** have a **visual representation**
- Its purpose is to **spot potential issues** in your React app.

Platform Independence:

- React's design allows it to be **platform-agnostic**
- While react-dom helps build web UIs using React, **ReactNative** can be used to craft mobile app UIs.



31. React Vs Angular vs Vue

React, Angular, and Vue:

- React is a **library**, while Angular and Vue.js are **frameworks**.
- React **focuses on UI**; Angular and Vue.js offer comprehensive tools for full app development.

◦ Library vs. Framework:

- A library offers **specific functionality**
- A **framework** provides a **set of tools and guidelines**.
- In simpler terms: React is a tool; Angular and Vue.js are toolsets.

◦ React's Specialty:

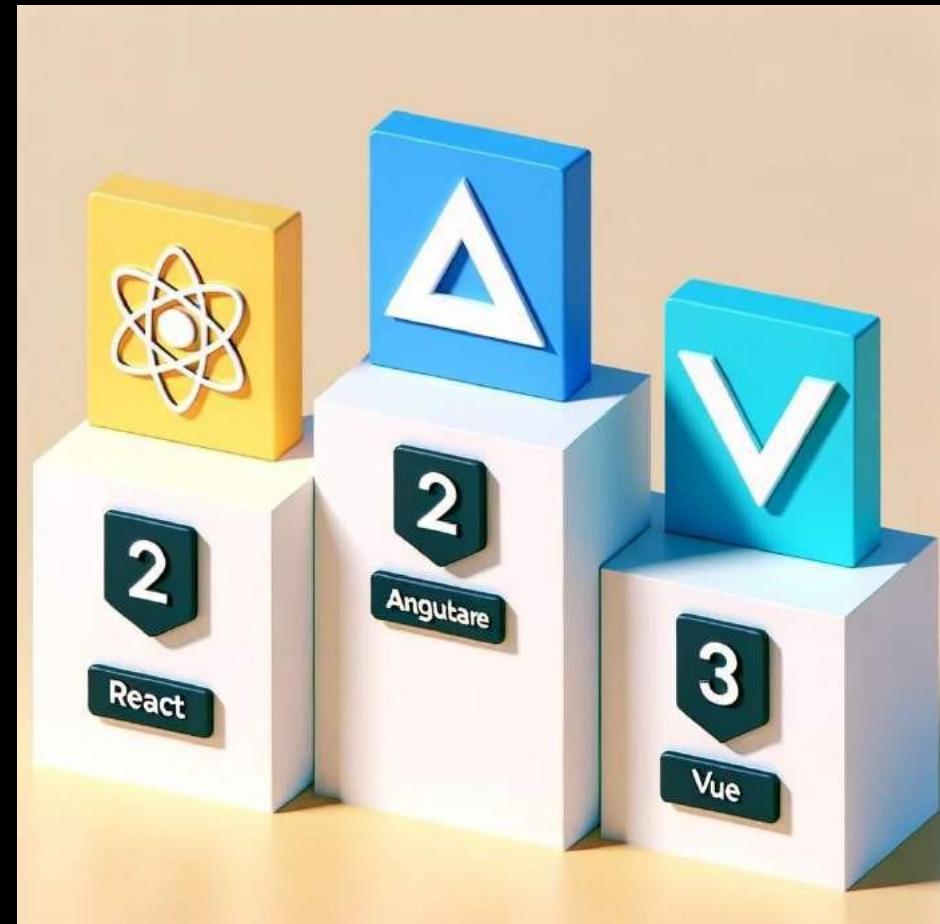
- React's main role is **crafting dynamic, interactive UIs**.
- It **doesn't handle** routing, **HTTP calls**, state management, and more.

◦ React's Flexibility:

- React **doesn't dictate** tool choices for other app aspects
- **Developers pick** what fits their project best.

◦ About Angular and Vue.js:

- Angular, developed by **Google**, provides a robust framework with a steep learning curve.
- Vue.js is known for its **simplicity** and ease of integration, making it beginner-friendly.



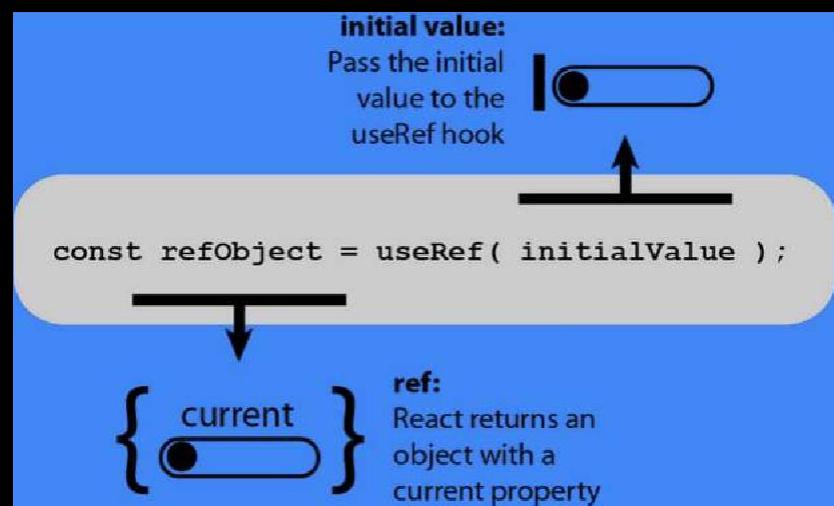
32. Using Forms

1. **State Management:** Each `input`'s state is stored in the `component's state`.
2. **Handling Changes:** Use `onChange` to detect input changes.
3. **Submission:** Utilize `onSubmit` for form submissions and prevent default with `event.preventDefault()`.
4. **Validation:** Implement custom validation or use third-party libraries.



33. Use Ref

1. `useRef` allows access to **DOM** elements and retains **mutable values** without **re-renders**.
2. Used with the `ref` attribute for **direct DOM interactions**.
3. Can hold **previous state** or prop values.
4. Not limited to DOM references; can hold **any value**.
5. **Refs** can be passed as **props** also



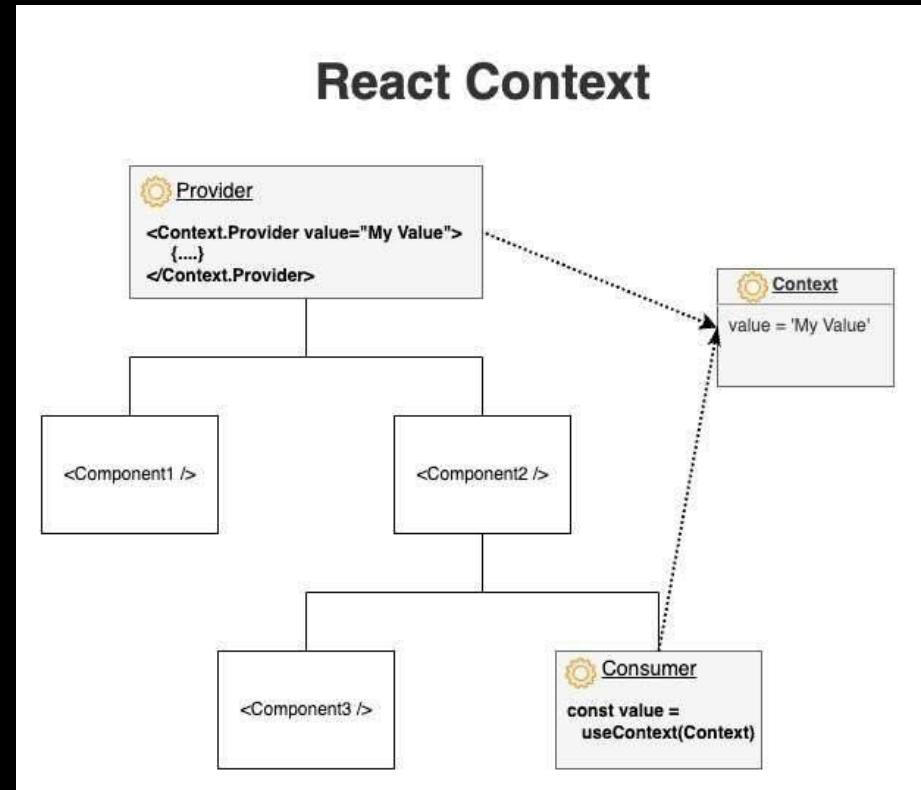
34. Update state from Previous State

- **Spread Operator:** Use to maintain immutability when updating arrays or objects.
- **Functional Updates:** Use `(existingPosts) => [postData, ...existingPosts]` to avoid stale values during asynchronous updates.



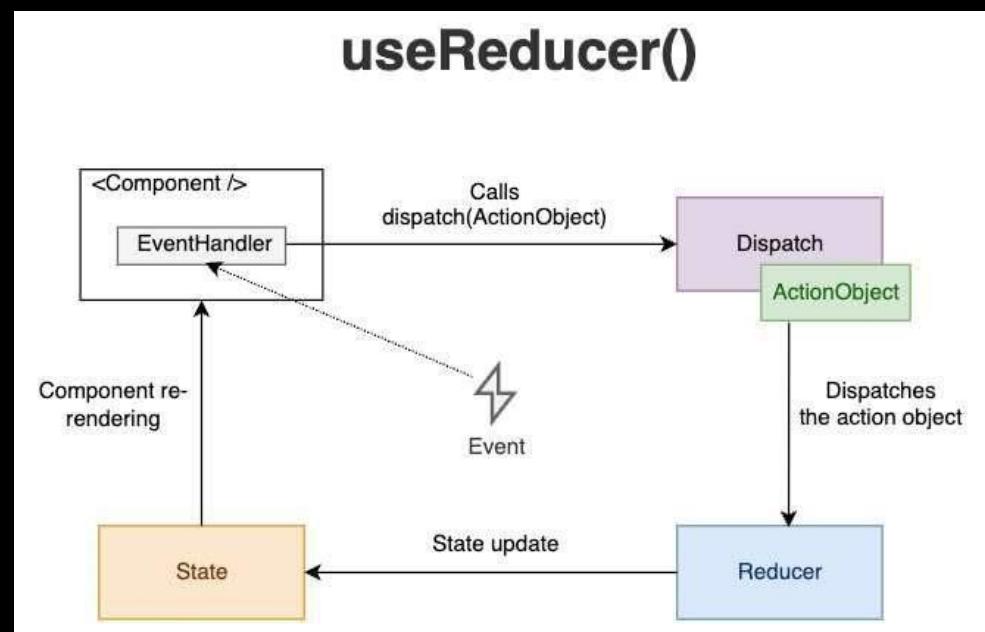
35. Context API

- 1 Prop Drilling: Context API addresses prop drilling; component composition is an alternative.
2. Folder Setup: Use a `store` folder for context files.
3. Initialization: Use `React.createContext` with initial state and export it.
4. Provider: Implement with `contextName.Provider` in components.
5. Access Value: Use the `useContext` hook.
6. Dynamic Data: Combine context value with state.
7. Export Functions: Context can also `export functions` for actions
8. Logic Separation: This helps keep the UI and business logic separate from each other.

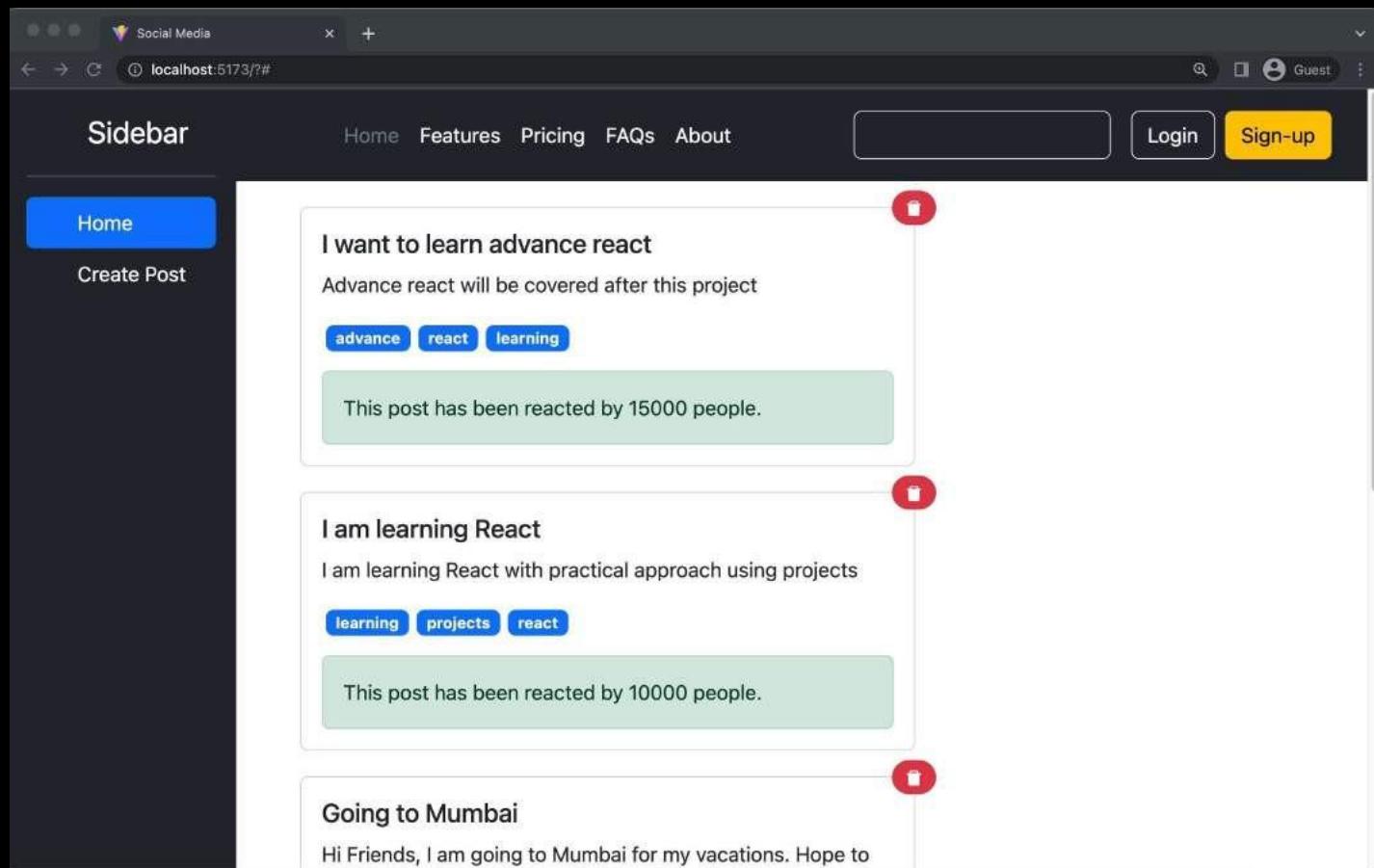


36. Use Reducer

1. `useReducer` is a hook in React that offers **more control** over state operations compared to `useState`, especially **for complex state logic**.
2. **Components:** It involves two main components:
 - **Reducer:** A **pure function** that takes the current state and an action and returns a new state.
 - **Action:** An **object describing what happened**, typically having a type property.
3. **Initialization:** It's invoked as
`const [state, dispatch] = useReducer(reducer, initialState).`
4. **Dispatch:** Actions are dispatched using the `dispatch` function, which **invokes the reducer** with the current state and the given action.
5. **Use Cases:** Particularly useful for **managing state in large components** or when the next state depends on the previous one.
6. **Predictable State Management:** Due to its strict structure, it leads to more **predictable and maintainable state management**.



Project: Social Media



37 Introducing Dummy API

DummyJSON

Get dummy/fake JSON data to use as placeholder in development or in prototype testing.

[View on GitHub](#)[Read Docs](#)

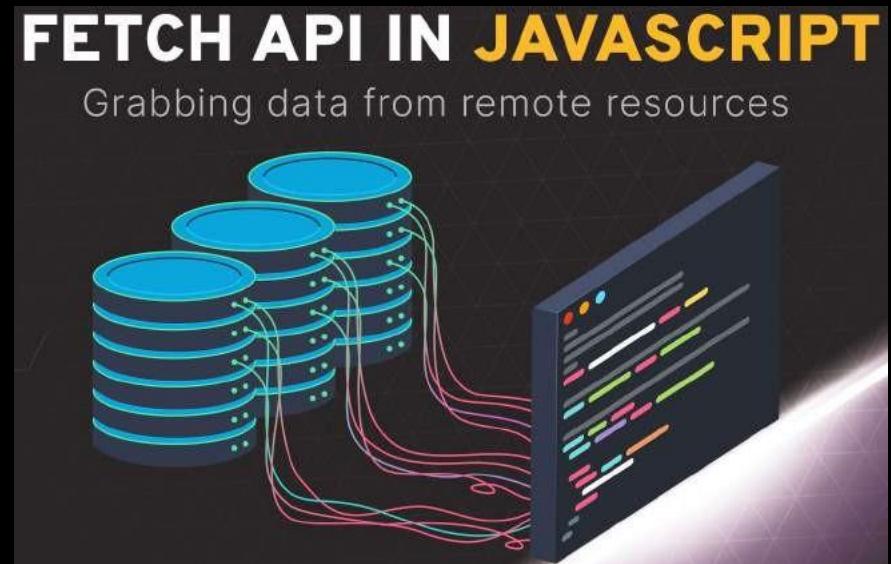
```
{  
  "id": 11,  
  "title": "perfume Oil",  
  "description": "Mega Discount, Impression of A...",  
  "price": 13,  
  "discountPercentage": 8.4,  
  "rating": 4.26,  
  "stock": 65,  
  "brand": "Impression of Acqua Di Gio",  
  "category": "fragrances",  
  "thumbnail": "https://i.dummyjson.com/data/products/11/thumbnail.jpg",  
  "images": [  
    "https://i.dummyjson.com/data/products/11/1.jpg",  
    "https://i.dummyjson.com/data/products/11/2.jpg",  
    "https://i.dummyjson.com/data/products/11/3.jpg",  
    "https://i.dummyjson.com/data/products/11/thumbnail.jpg"  
  ]  
}
```

perfume Oil – fragrances

13\$ — ★ 4.26

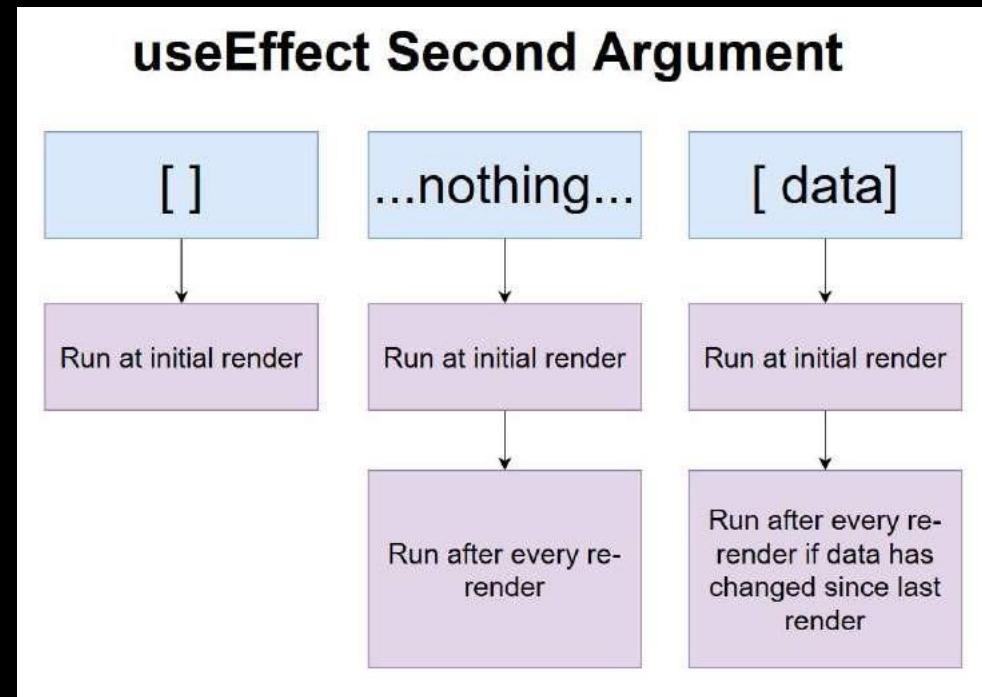
38. Data fetching using Fetch

1. `fetch`: Modern JavaScript API for network requests.
2. **Promise-Based**: Returns a `Promise` with a Response object.
3. **Usage**: Default is `GET`. For POST use `method: 'POST'`
4. **Response**: Use `.then()` and `response.json()` for JSON data.
5. **Errors**: Doesn't reject on HTTP errors. Check `response.ok`.
6. **Headers**: Managed using the `Headers API`.



39. The useEffect Hook

- 1 In function-based components, `useEffect` handles side effects like `data fetching` or `event listeners`.
2. `useEffect` runs automatically `after every render` by default.
3. By providing a `dependency array`, `useEffect` will only run when specified `variables change`. An empty array means the effect runs once.
4. Multiple `useEffect` hooks can be used in a single component for organizing `different side effects separately`.



40. Handling Loading State



41. The `useEffect` Hook Cleanup

```
useEffect(() => {
  const timerID = setInterval(() => {
    // do something
  }, 1000);

  // This is the cleanup function
  return () => {
    clearInterval(timerID);
  };
}, []);
```

Returning a function from `useEffect` allows for **cleanup**, ideal for removing event listeners.

42. Advanced useEffect

Junior



```
useEffect(() => {
  fetch(`/api/users/${id}`)
    .then(res) => res.json()
    .then(data) => {
      setUser(data);
    });
}, [id]);
```



Pro

```
useEffect(() => {
  const controller = new AbortController();
  const signal = controller.signal;

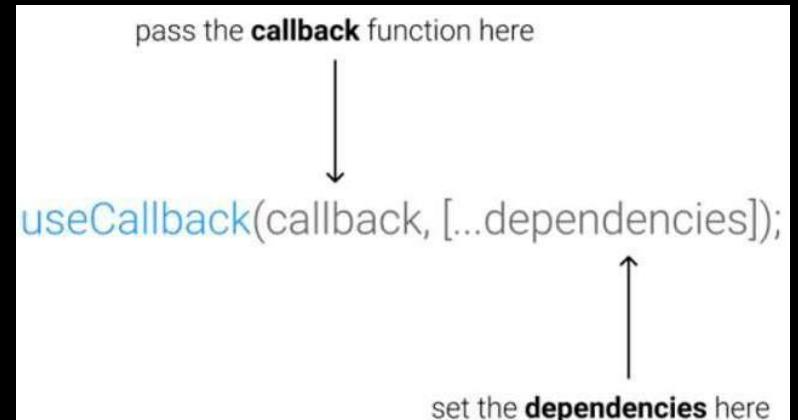
  fetch(`/api/users/${id}`, { signal })
    .then(res) => res.json()
    .then(data) => {
      setUser(data);
    });

  return () => {
    controller.abort();
  };
}, [id]);
```



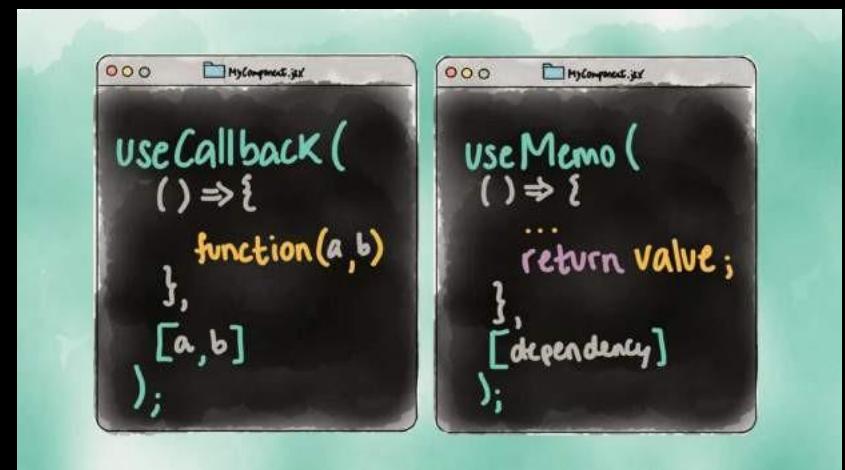
43 The `useCallback` Hook

1. Memoization: Preserves function across renders to prevent unnecessary re-renders.
2. Optimization: Enhances performance in components with frequent updates.
3. Dependency Array: Recreates the function only when specific dependencies change.
4. Event Handlers: Used to keep consistent function references for child components.
5. With `useEffect`: Prevents infinite loops by maintaining function references.



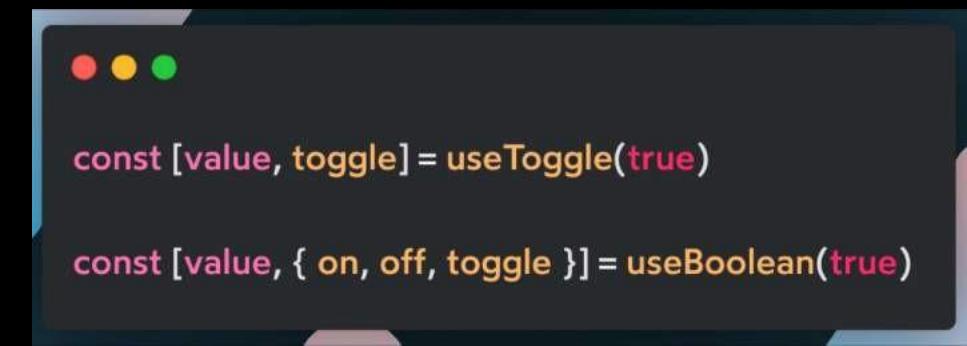
44 The useMemo Hook

- 1 Memoization: useMemo caches the result of expensive calculations to enhance performance.
2. Re-computation: Only re-computes the memoized value when specific dependencies change.
3. Optimization: Helps prevent unnecessary recalculations, improving component rendering efficiency.
4. Dependency Array: Uses an array of dependencies to determine when to recompute the cached value.
5. Comparison with useCallback: While useCallback memoizes functions, useMemo memoizes values or results of functions.
6. Best Use: Ideal for intensive computations or operations that shouldn't run on every render.



45 Custom Hooks

1. **Reusable Logic:** Custom hooks allow you to extract and reuse component logic across multiple components.
2. **Naming Convention:** Typically start with "use" (e.g., `useWindowSize`, `useFetch`).
3. **Combining Hooks:** Custom hooks can combine multiple built-in hooks like `useState`, `useEffect`, and others.
4. **Sharing State:** Enables sharing of stateful logic without changing component hierarchy.
5. **Isolation:** Helps in isolating complex logic, making components cleaner and easier to maintain.
6. **Custom Return Values:** Can return any value (arrays, objects, or any other data type) based on requirements.



46 Submitting data with Fetch

```
1 fetch('http://example.com/users.json', { // http path (Endpoint)
2   headers: { "Content-Type": "application/json; charset=utf-8" }, //Headers
3   method: 'POST', // Method, which is the type of request we want to make
4   body: JSON.stringify({ //Data we want to send to our database
5     username: 'Jorge',
6     email: 'jorge@example.com',
7   })
8 })
9 .then(response => response.json()) //Defines the response type
10 .then(data => console.log(data)); //Gets the response type
11
```

47 React Router

- 1 **Installation:** Use `npm install react-router-dom`.
2. We are going to use the latest version which is 6+
3. **RouterProvider:** Wraps the app for routing capabilities.
4. **createBrowserRouter:** helps creating the mapping for router provider.
5. **Declarative Routing:** Easily define application routes.
6. Routes are React components.



48 Layout Routes

```
export default function Router() {
  return useRoutes([
    {
      path: '/dashboard',
      element: <DashboardLayout />,
      children: [
        { element: <Navigate to="/dashboard/app" replace /> },
        { path: 'app', element: <DashboardApp /> },
        { path: 'user', element: <User /> },
        { path: 'products', element: <Products /> },
        { path: 'blog', element: <Blog /> }
      ]
    },
    {
      path: '/',
      element: <Outlet />
    }
  ])
}
```

- 1 Layout Routes help us to use shared elements
2. Outlet component is used to render the children at the correct places

49 Route Links

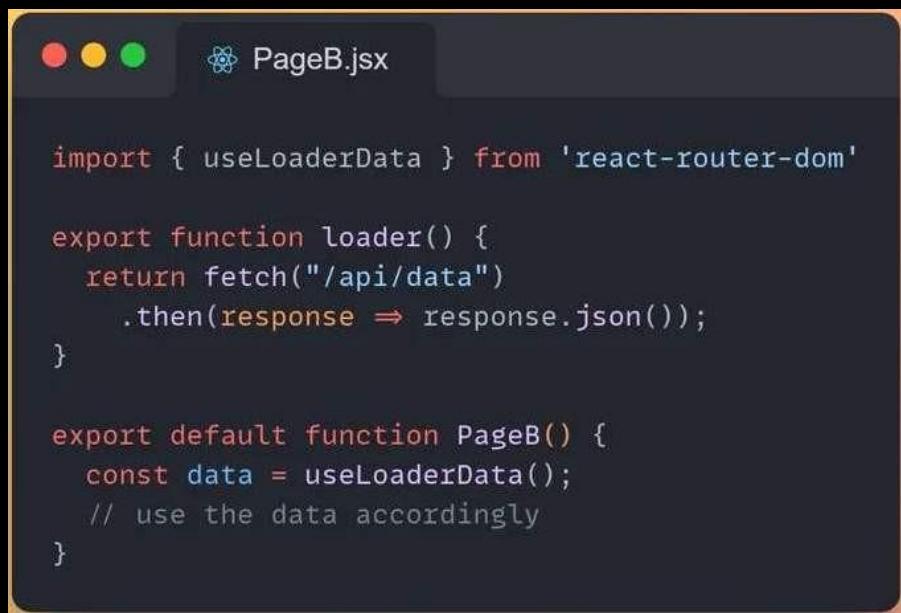
```
import { useNavigate } from "react-router-dom"; // v6

const Component = () => {
  // Triggers re-renders on every path change
  const navigate = useNavigate();
  ...
}
```

- 1 **Link** Component with **to** property can be used to **avoid reloading**
- 2 **useNavigate** hook can be used to do **navigation programmatically**.

50 Data fetching using loader

1. Loader method can be used to load data before a particular route is executed.
2. The loader method must return the data that is loaded or promise.
3. Data is available in component and all the child components.
4. useLoaderData hook can be used to get the fetched data.
5. Loading state can also be used.



```
import { useLoaderData } from 'react-router-dom'

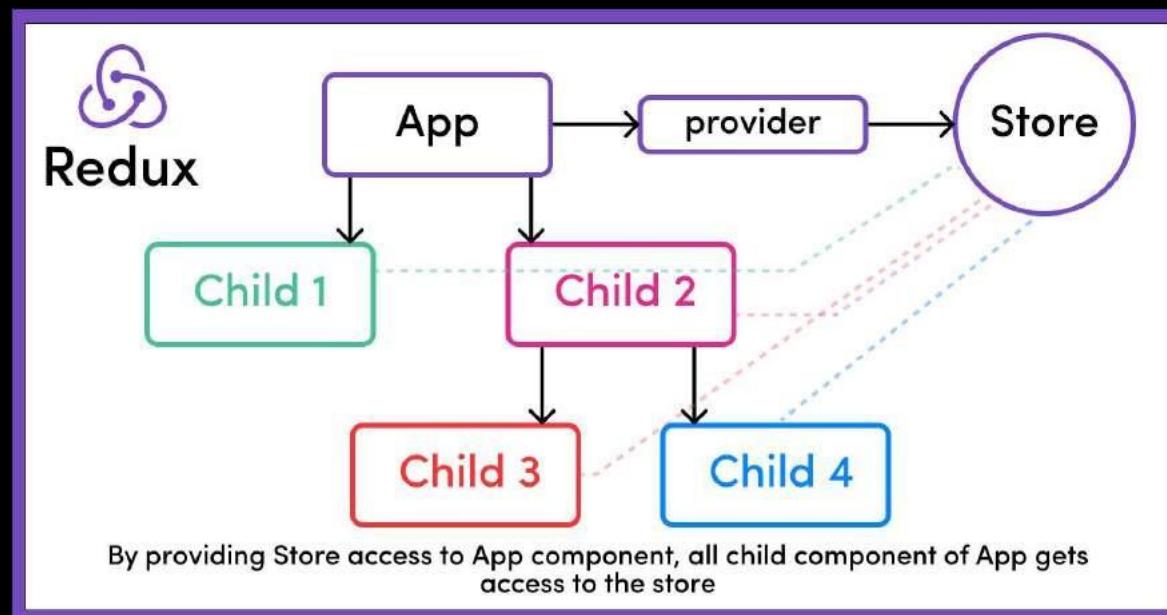
export function loader() {
  return fetch("/api/data")
    .then(response => response.json());
}

export default function PageB() {
  const data = useLoaderData();
  // use the data accordingly
}
```

51 Submitting data using action

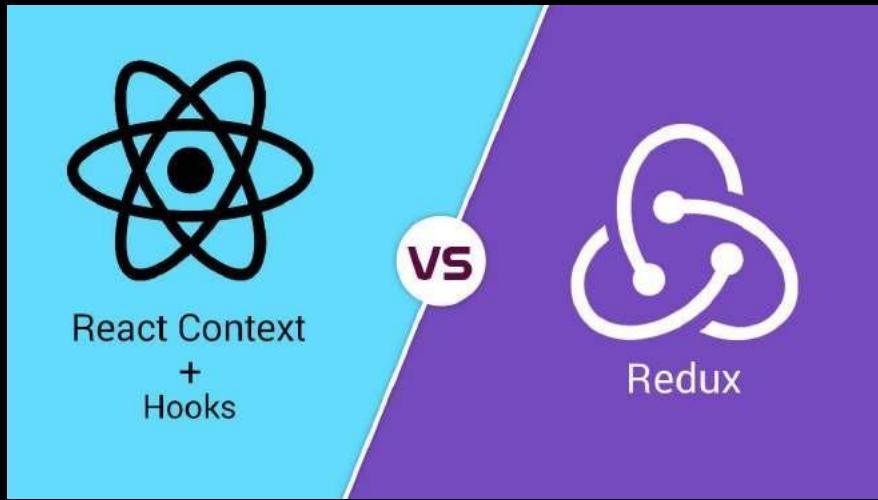
1. Action method can be used to perform an action on submission of Forms.
2. Custom Form component need to be used along with name attribute for all inputs.
3. Action function will get an data object. To generate correct request object method="post" attribute should be used.
4. Data.request.formData() method can be used to get form data Object.
5. Object.fromEntries(formData) can be used to get actual input data.
6. redirect() response can be returned for navigation after submission.

52 What is Redux



- 1 State management for cross component or app-wide state.
2. Redux is a predictable state management library for JavaScript apps.
3. Local State vs Cross-component state vs App-Wide state
4. useState or useReducer vs useState with prop drilling vs useState or useContext or Redux

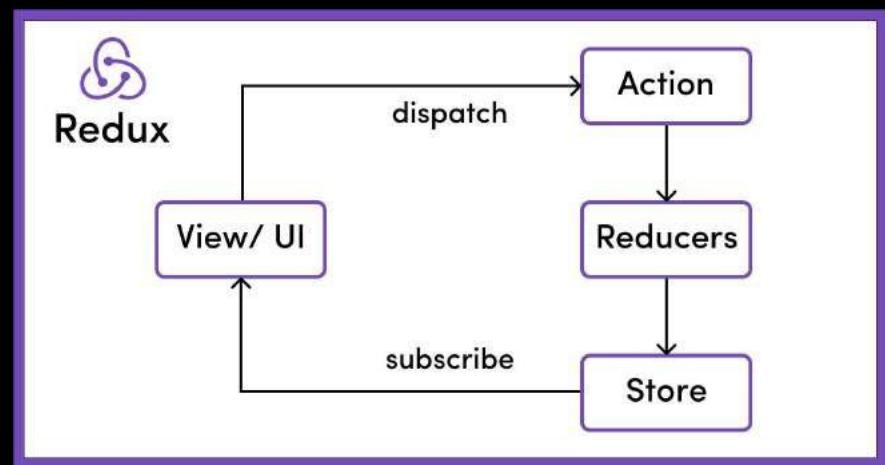
53 React-Context vs Redux



- 1 You can use **both**.
- 2 Setup and Coding is **tough** especially if you have multiple **context providers**.
- 3 Performance is **slow**. Context should only be used for things that **rarely change**. On the other hand **Redux has great performance**.
- 4 If these things **don't matter** to you then you can choose not to use redux and **stay with React-Context**.

54 How Redux Works

- 1 Single Source: Uses a single central store to maintain the entire application's state.
- 2 Actions: Components never directly change the store. Changes to state are made through dispatched actions, which describe events.
- 3 Reducers: Actions are processed by reducers, pure functions that return the new state.
- 4 Immutable: State is immutable; every change results in a new state object.
- 5 This is different from `useReducer` hook.



55 Working with Redux

1. npm init -y
2. npm install redux
3. import in node Const redux = require('redux');
4. We need to setup all 4 basic things:
 1. Reducer
 2. Store
 3. Subscriber
 4. Actions
5. Node redux-demo.js command to run node server

56 React with Redux

- 1 Npm install redux
- 2 Npm install react-redux
- 3 Create store folder with Index.js file
- 4 Creating the store using
Import {createStore} from redux.
- 5 Providing the store with react
 - 1 Provider from react-redux
 2. <Provider store={store}><App /></Provider>
- 6 Using the store
 - 1 useSelector hook gets a slice of the store.
Const counter = useSelector(state => state.counter);
 2. Subscription is already setup and only will re-execute when only your slice is changed. Subscription is automatically cleared also.
- 7 Dispatch Actions using the useDispatch hook.

57 Why Redux Toolkit



1. Action types are difficult to maintain
2. Store becoming too big
3. Mistakenly editing store
4. Reducer becoming too big

58 Working with Redux Toolkit

1. Npm install @reduxjs/toolkit
2. Remove redux from package.json
3. Import {createSlice} from “@reduxjs/toolkit”
4. Slices of the store can be created using the following syntax:

```
Const slice = createSlice({  
  name: "",  
  initialState: {},  
  reducers: {  
    smallReducerMethods: (state, action) => {  
      },  
    }  
  })
```

5. ConfigureStore combines multiple reducers and can be used as:

```
configureStore({  
  reducer: {name: slice.reducer}  
})
```

6. Export actions = slice.actions;
7. Actions can be dispatched like: actions.reducerMethod(payload);

Project Myntra Clone