Mastering React.js: A Comprehensive Crash Course

Fetch Data using React Router

Step 1: Define the API Function to Fetch Data

- Create a function that makes an API call to fetch data. This function will be used as a loader for the route.

Step 2: Assign the Loader Function to the Route

- Pass the getMovies function as the value of the loader property in the route configuration. This ensures that the data is fetched before the route component is rendered.

Step 3: Retrieve Data Using the useLoaderData Hook

- In your route component, use the useLoaderData hook to access the data returned by the loader. This hook provides the data fetched by the loader, making it available for use within the component.

useNavigate() in React Router


```
const navigate = useNavigate()
```

useNavigate is a hook provided by React Router that returns a function you can use to programmatically navigate to a different route.

navigate(-1): Navigates back to the previous page in history. This is context-sensitive and depends on the user's navigation history.

navigate("/"): Directly navigates to the homepage, regardless of the user's current location

Introduction

Introduction to React.js

History of React.js

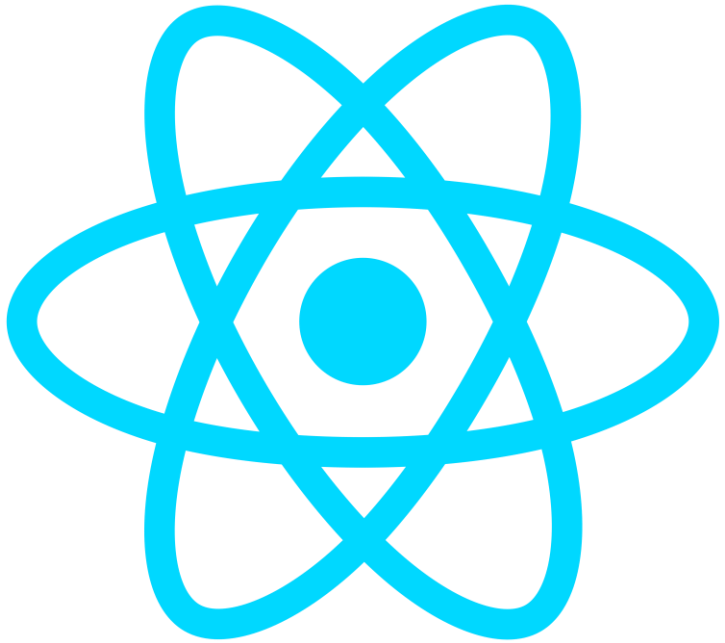
Why React.js over Vanilla JavaScript

Setting up development environment

Creating React.js project

Introduction to React.js

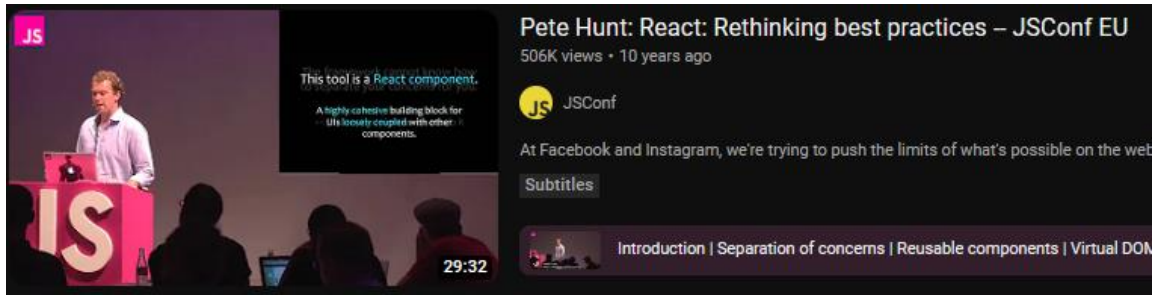
- React.js is a JavaScript library used for building user interfaces (UIs) and single-page applications.
- Created by Jordan Walke at Facebook.
- Most popular JavaScript library for frontend development.



History of React.js

- React.js began as an internal tool for dynamic Facebook components.
- Created in 2011 but remained private initially.
- React.js was open-sourced by Jordan Walke and Tom Occhino at JSConfUS 2013.
- Released with the belief that its success at Facebook could benefit others.
- Initially faced criticism for combining JavaScript and HTML in a single file.
- It was widely hated and criticized because of its combination of JavaScript and HTML in single file.
- It slowly gained traction and blew up in adoption.

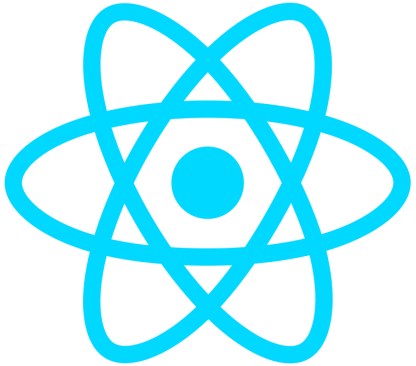
First Keynote given by Tom Occhino and Jordan Walke at JSConfUS 2013.



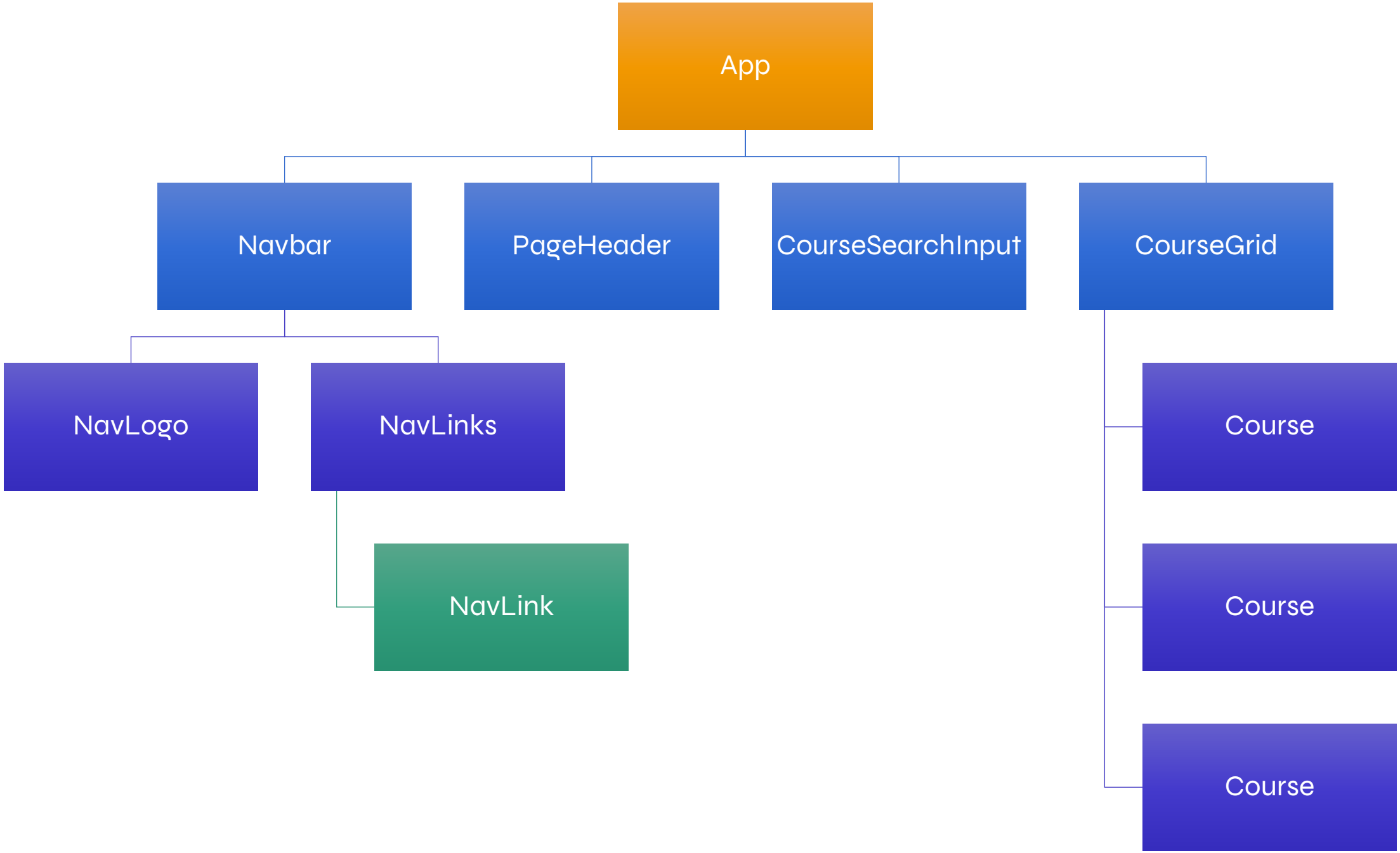
Second Keynote given by Pete Hunt at JSConfEU.

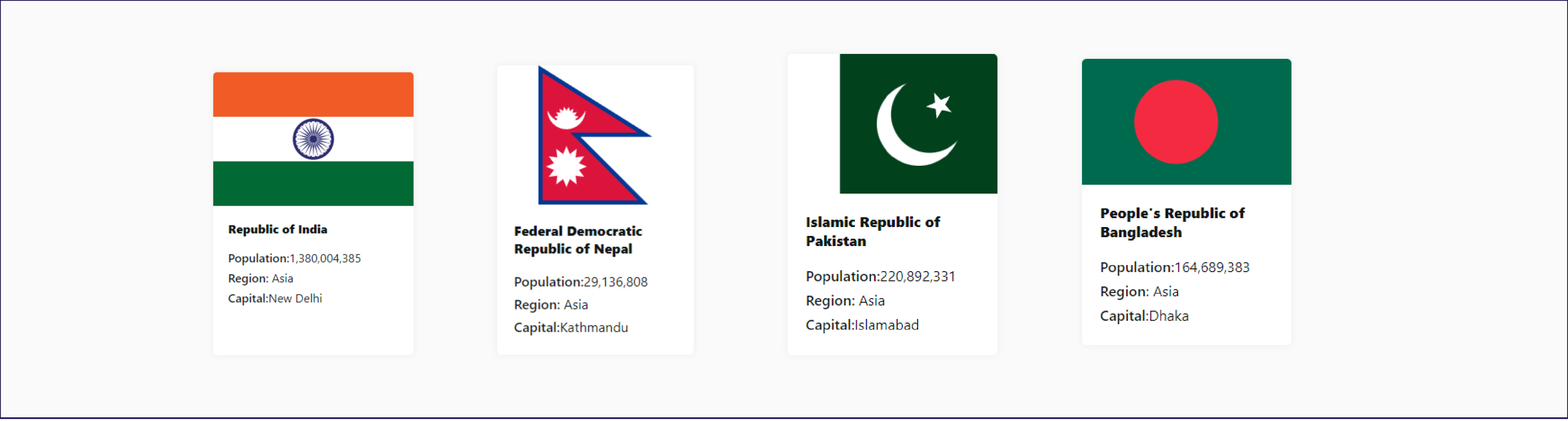
Is React JS Library or Framework?

- React is not a framework. React is a JavaScript library for building user interfaces.
- It is also known as ReactJS and React.js, so don't get confused if you read different notation in different places.
- React knows only one thing that is to create an awesome UI.



React is all about Components


App
Navbar
PageHeader
CourseSearchInput
CourseGrid
NavLogo
NavLinks
NavLink
Course
Course
Course




```html
<a href="/country/Republic%20of%20India">
  <img
    src="path_to_india_flag.svg"
    alt="Indian national flag"
    class="IndiaFlag"
  />
  <div class="countryInfo">
    <p class="countryTitle">Republic of India</p>
    <div>
      <p>
        <span class="infoTopic">Population:</span> 1,380,004,385
      </p>
      <p>
        <span class="infoTopic">Region:</span> Asia
      </p>
      <p>
        <span class="infoTopic">Capital:</span> New Delhi
      </p>
    </div>
  </div>
</a>
```


```html
<a href="/country/Republic%20of%20India">
  <img
    src="path_to_india_flag.svg"
    alt="Indian national flag"
    class="IndiaFlag"
  />
  <div class="countryInfo">
    <p class="countryTitle">Republic of India</p>
    <div>
      <p>
        <span class="infoTopic">Population:</span> 1,380,004,385
      </p>
      <p>
        <span class="infoTopic">Region:</span> Asia
      </p>
      <p>
        <span class="infoTopic">Capital:</span> New Delhi
      </p>
    </div>
  </div>
</a>
```

```html
<a href="/country/Republic%20of%20India">
  <img
    src="path_to_india_flag.svg"
    alt="Indian national flag"
    class="IndiaFlag"
  />
  <div class="countryInfo">
    <p class="countryTitle">Republic of India</p>
    <div>
      <p>
        <span class="infoTopic">Population:</span> 1,380,004,385
      </p>
      <p>
        <span class="infoTopic">Region:</span> Asia
      </p>
      <p>
        <span class="infoTopic">Capital:</span> New Delhi
      </p>
    </div>
  </div>
</a>
```

```html
<a href="/country/Republic%20of%20India">
  <img
    src="path_to_india_flag.svg"
    alt="Indian national flag"
    class="IndiaFlag"
  />
  <div class="countryInfo">
    <p class="countryTitle">Republic of India</p>
    <div>
      <p>
        <span class="infoTopic">Population:</span> 1,380,004,385
      </p>
      <p>
        <span class="infoTopic">Region:</span> Asia
      </p>
      <p>
        <span class="infoTopic">Capital:</span> New Delhi
      </p>
    </div>
  </div>
</a>
```

Republic of India

Population:1,380,004,385
Region: Asia
Capital:New Delhi

Federal Democratic Republic of Nepal

Population:29,136,808
Region: Asia
Capital:Kathmandu

Islamic Republic of Pakistan

Population:220,892,331
Region: Asia
Capital:Islamabad

People's Republic of Bangladesh

Population:164,689,383
Region: Asia
Capital:Dhaka

```jsx
return (
  <div className={`container ${styles.countriesCards}`}>
    {filteredCountries.map((country) => (
      <CountryCard country={country} key={country.name.common} />
    ))}
  </div>
);
```

```jsx
<Link to={`/country/${country.name.official}`}>
  <img
    src={country.flags.svg}
    alt={country.flags.alt}
    className={styles.countryFlag}
  />
  <div className={styles.countryInfo}>
    <p className={styles.countryTitle}>{country.name.official}</p>
    <div>
      <p>
        <span className={styles.infoTopic}>Population:</span>
        {country.population.toLocaleString()}
      </p>
      <p>
        <span className={styles.infoTopic}>Region:</span> {country.region}
      </p>
      <p>
        <span className={styles.infoTopic}>Capital:</span>
        {country.capital[0]}
      </p>
    </div>
  </div>
</Link>
```

React is Declarative

Why React.js ?

- React is declarative because it describes what the UI should look like rather than how to achieve it. This makes the code easier to read and maintain, as it is more focused on the end result rather than the steps involved in getting there.

```
function MyComponent({ name }) {
    return <div>Hello, {name}!</div>;
}
```

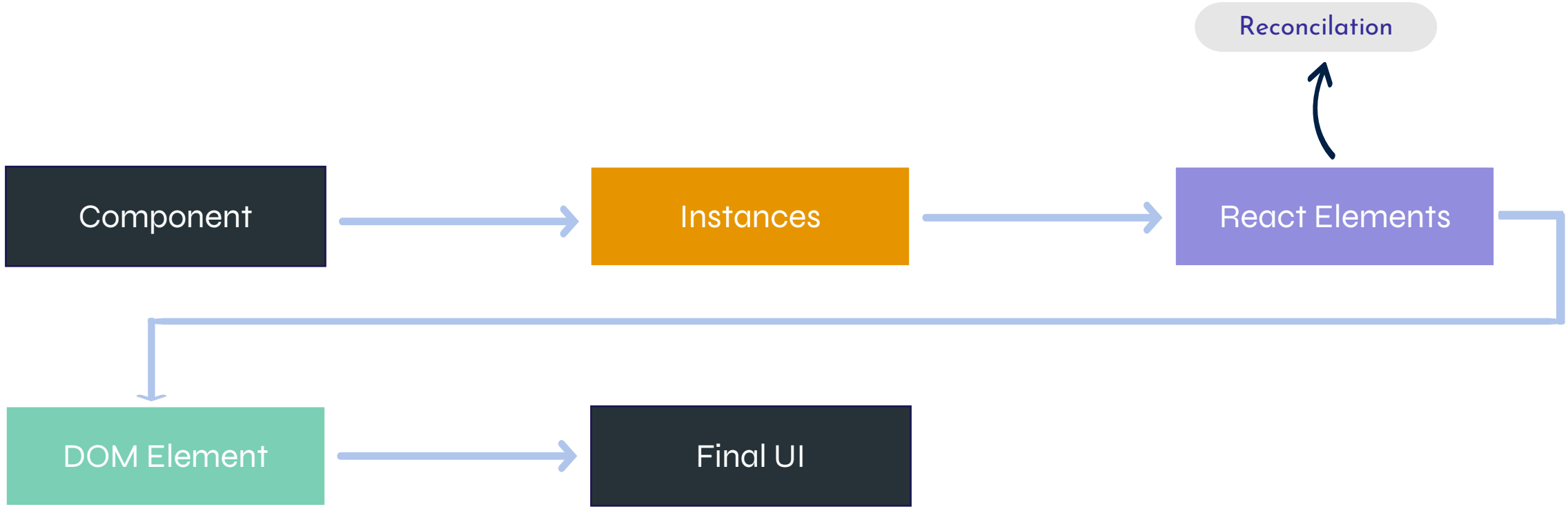
React - Declarative

JS - Imperative

```
function MyComponent(name) {
    const element = document.createElement('div');  // Create a new div element
    element.textContent = `Hello, ${name}!`;        // Set text content manually
    return element;                                 // Return the created element
  }
```

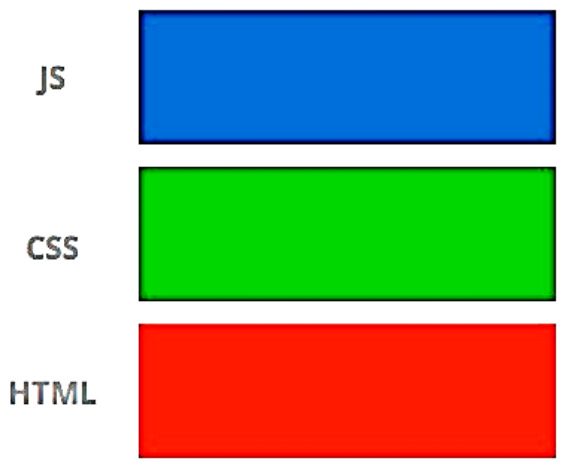
- Here, we are manually creating elements, setting their properties, and appending them to the DOM.

How React works?

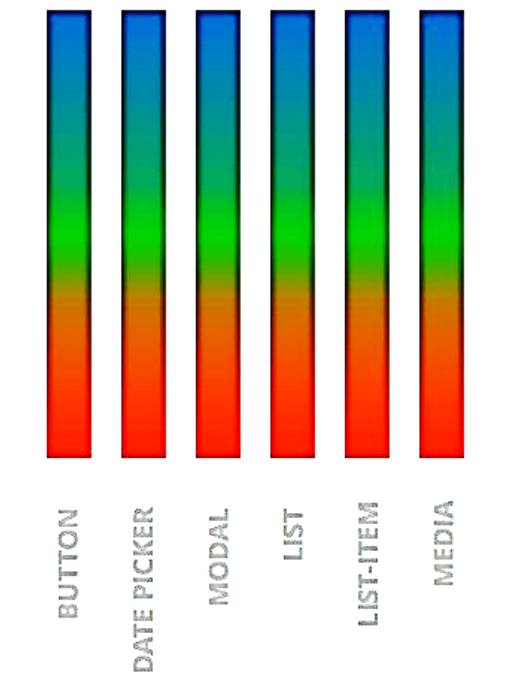


Why React.js over Vanilla JavaScript?

- Oppose to Vanilla JavaScript, React has a concept called **components** which combines all HTML, CSS and JavaScript by features instead of separating HTML, CSS and JavaScript completely.
- React.js manages updating the DOM or Document Object Model with the components written by us.

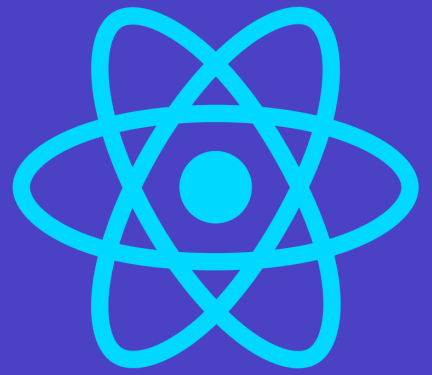


Traditional Way



React.js

Why React.js?



- React is the Most Popular JavaScript library for building user interfaces.
- Component-Based Architecture
- Declarative UI
- Rich Ecosystem – npm packages
- Strong Community Support – Online / Github

Creating React.js Project

Introduction to React.js

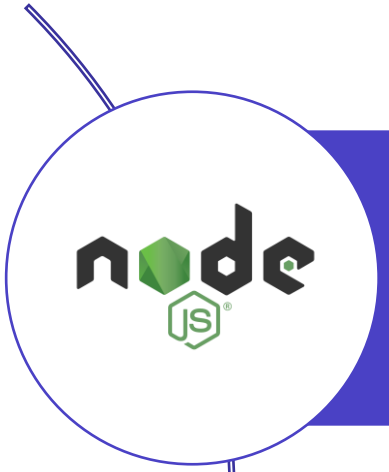
History of React.js

Why React.js over Vanilla JavaScript

Setting up development environment

Creating React.js project

Setting up Development Environment

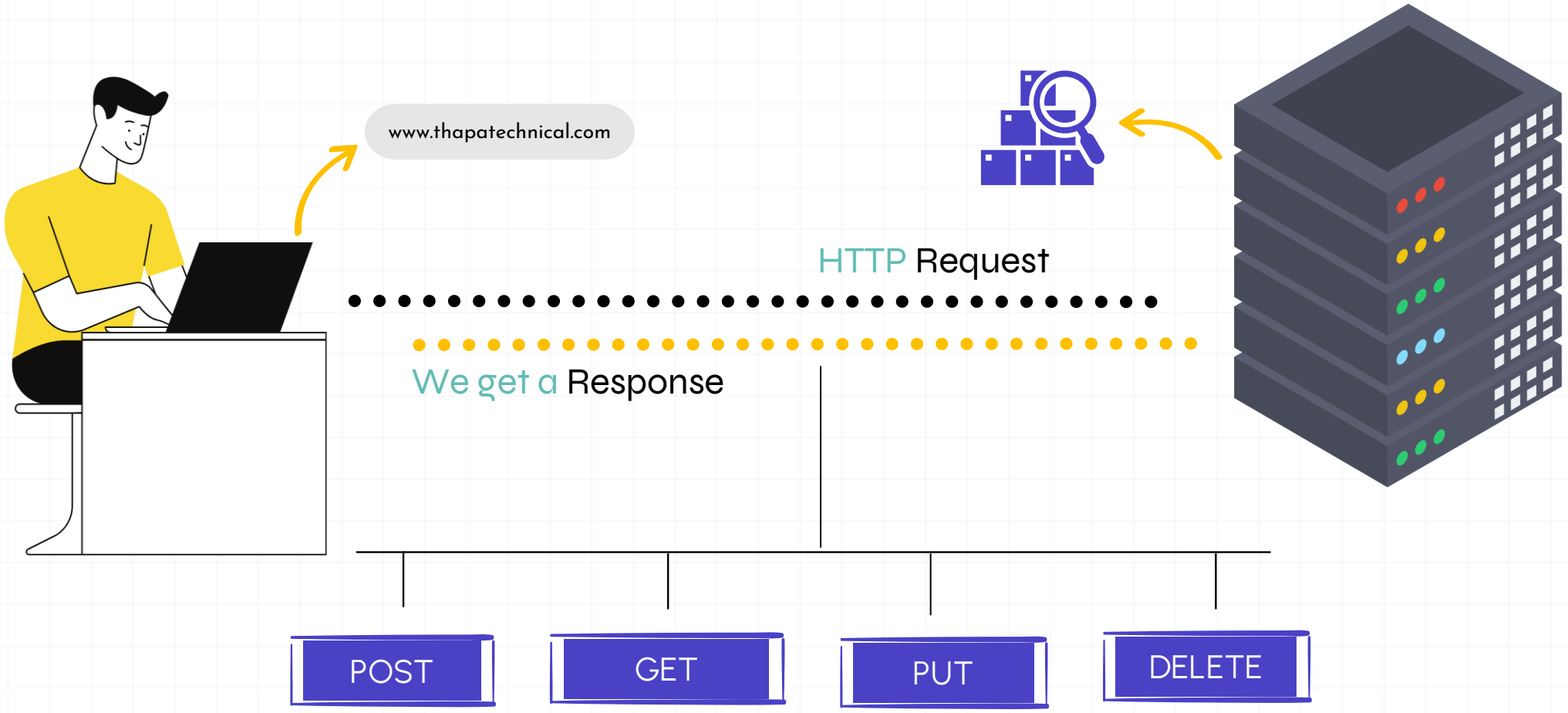


Download Node.js, React & Axios

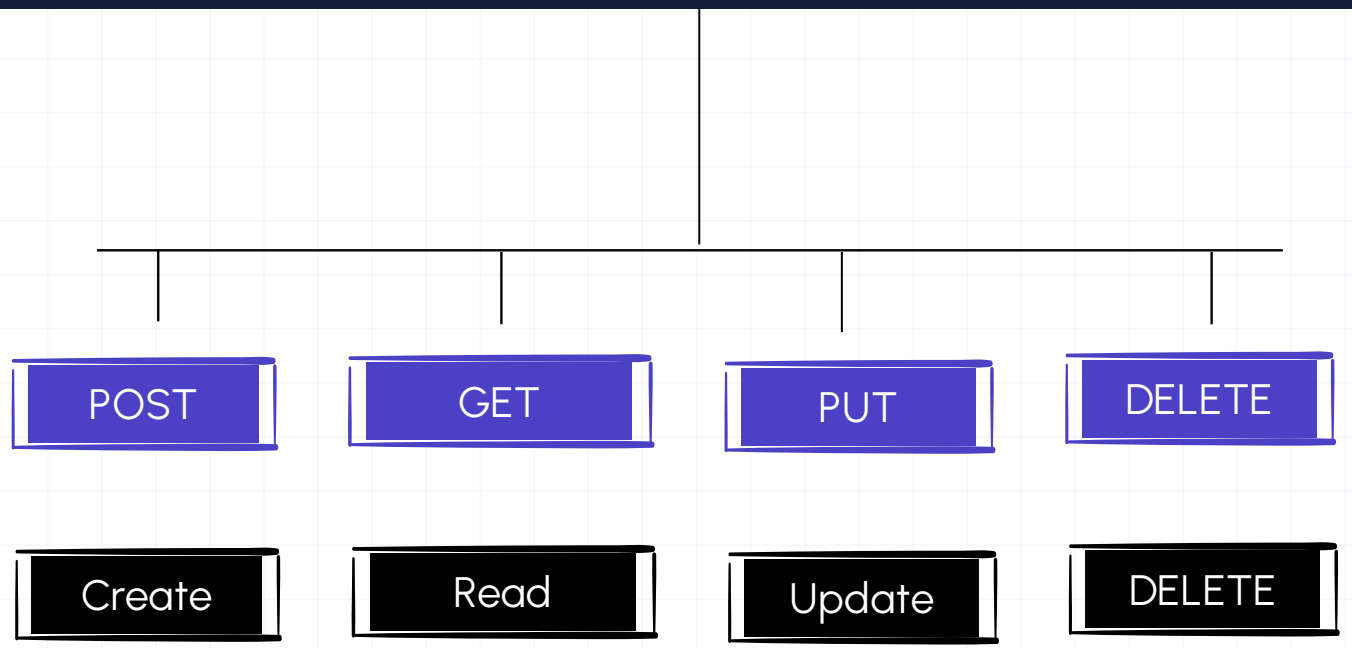
Visual Studio Code

- Prettier
  - Turn on Format on Save
- ESLint

HTTP METHODS



CRUD OPERATIONS WITH AXIOS



Ways of Creating React.js Project

- Custom
  - You can choose transpiler and bundler yourself and choose your own configuration.
  - This was how React.js web apps were created in past, but it's not needed now.
- create-react-app
  - As setting up bundler and transpiler yourself was a tedious task, Facebook created create-react-app as a tool to create React.js projects.
  - But now it's abandoned and no longer recommended. If you have seen this somewhere then avoid using it.
- Vite
  - This is a modern and recommended way of creating React.js. This is what we will use in this course.
  - This is much faster and creates smaller bundle sizes.
- Meta Frameworks (Next.js, Remix, Gatsby, etc.)
  - React.js is slowly moving towards being server focused instead of client focused.
  - That's why you will find these frameworks being suggested if you visit official React.js Documentation website but for learning purposes, starting from meta frameworks can be confusing.
  - To understand the need of Meta Frameworks, you first need to understand how plain React.js projects are created.
  - But you must learn these frameworks after grasping concepts of React.js.

Important Update for Future Viewers:

- As of May 15/16, during the recent conference, React version 19 has advanced from its beta stage to a Release Candidate (RC). If you are watching this after a month from the announcement, it is likely that React v19 has reached its stable release. Therefore, you might not need to append 'RC' at the end of React v19 when installing or upgrading.

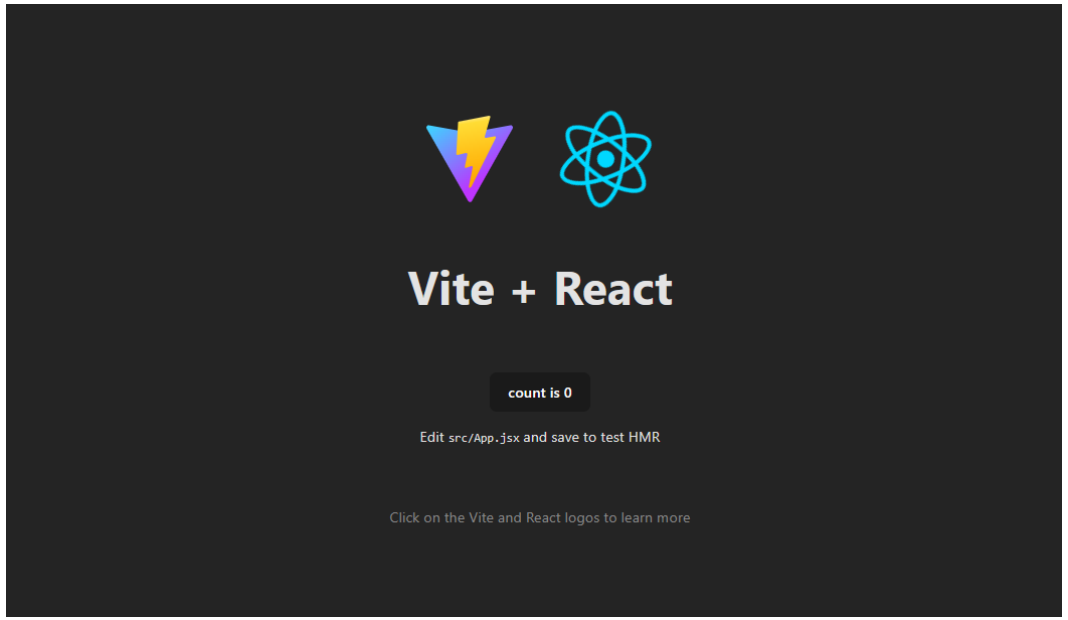
- Please check the description and comments below for the most current updates. I am committed to keeping this information up-to-date to ensure you have the best learning experience. Should there be any significant changes or new best practices, I will promptly provide updates in the course materials.

Creating React.js project

npm create vite@latest

```
❯ npm create vite@latest
✓ Project name: … thapa-react
✓ Select a framework: › React
✓ Select a variant: › JavaScript

Scaffolding project in /home/binamra/thapa-react...

Done. Now run:

  cd thapa-react
  npm install
  npm run dev
```

Our Project is ready



- Go to http://localhost:5173 to visit your website

Prerequisite to learn React.js

Maximize Your Learning: 5 Key Tips for Our React Course

5 Key Tips for Our React Course

THAPA TECHNICAL

1: Practice Regularly (No matter what you have to do this.)

2: Join the Conversation: Post your questions and help others in the comments to build our community together.

3: Explore Additional Resources (Videos, Sites, Blogs etc)

4: Apply Real-World Scenarios

5: Review and Revise

Project Structure

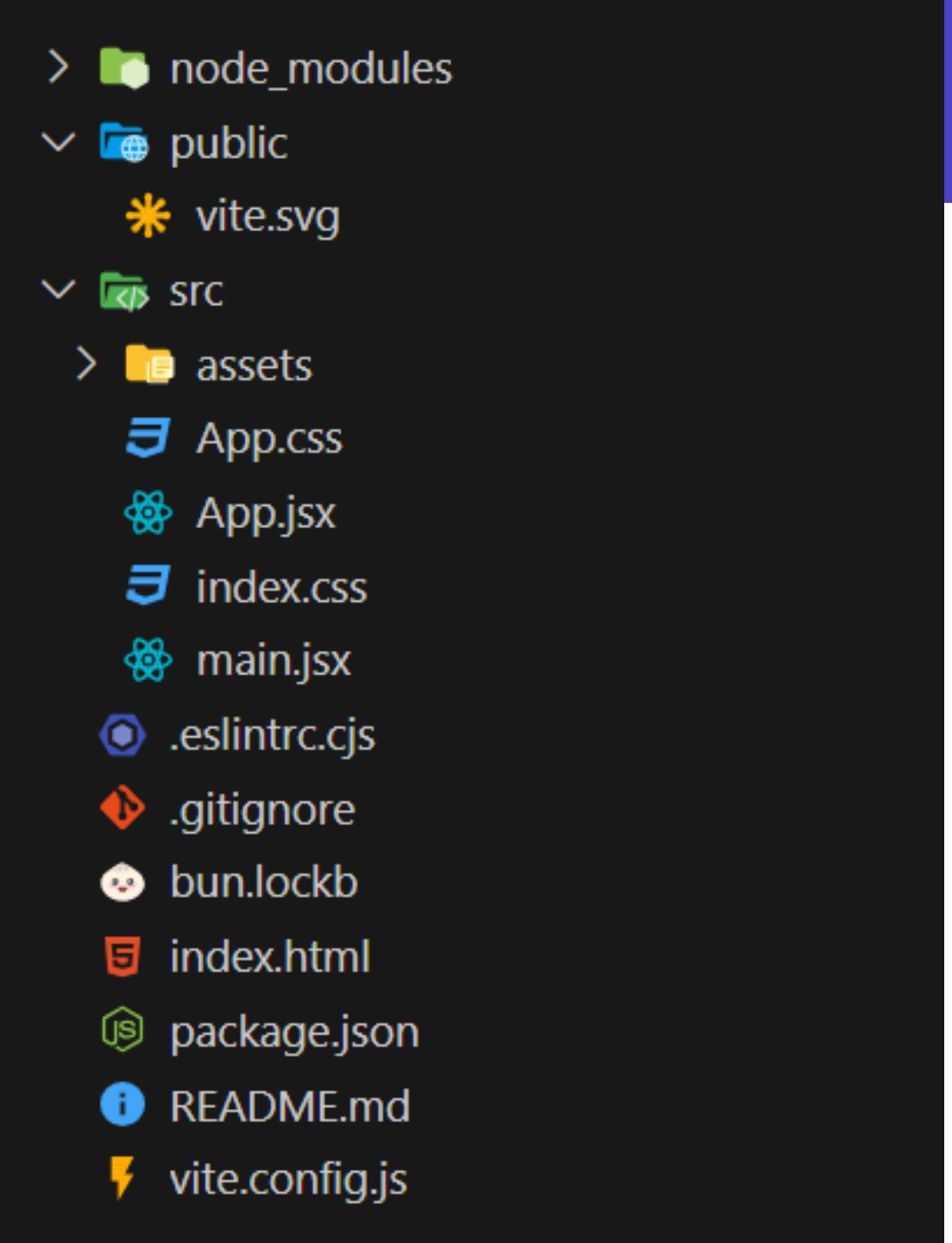
Introduction to React.js

History of React.js

Why React.js over Vanilla JavaScript

Setting up development environment

Creating React.js project



Project Structure

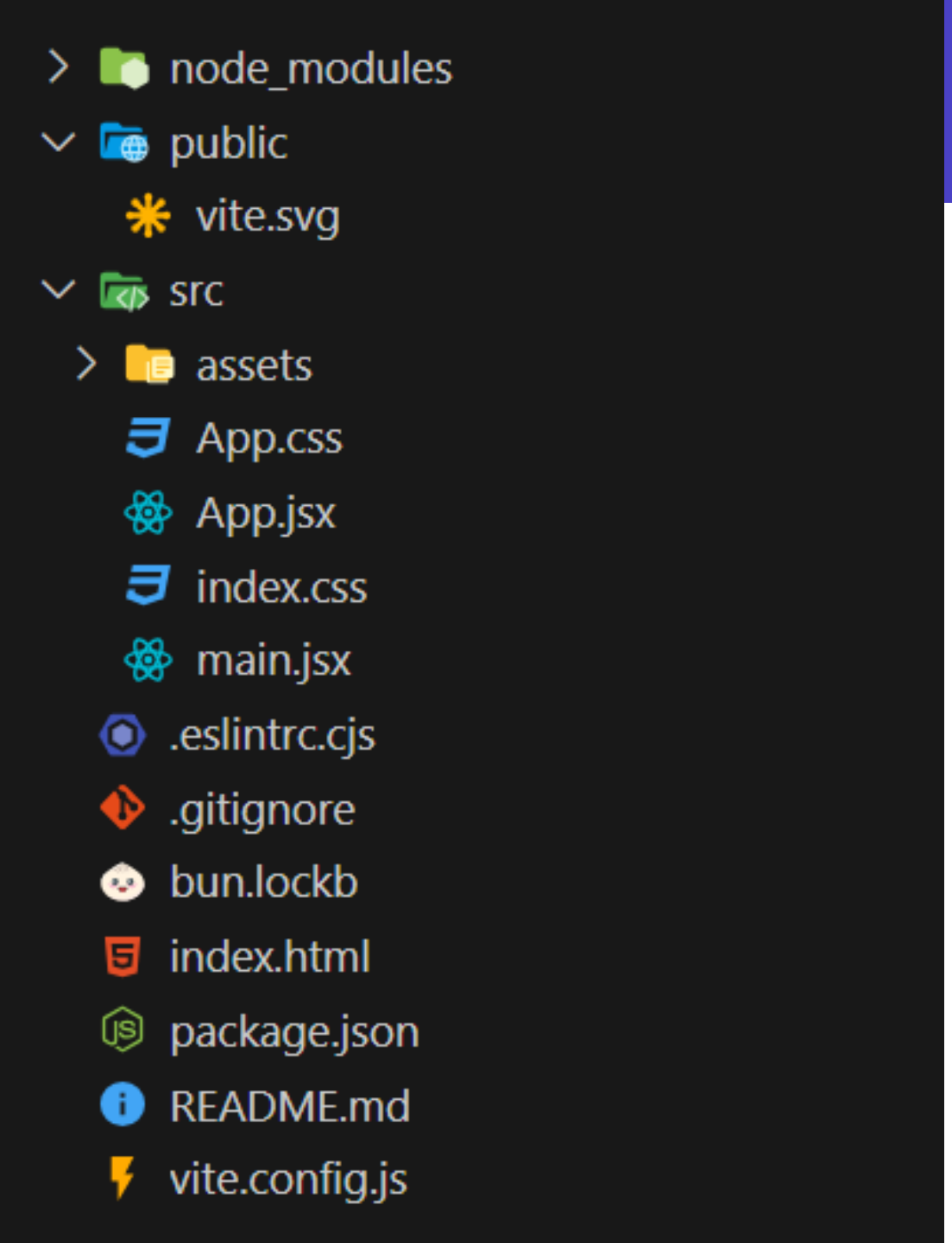
- node_modules
  - This is the folder which contains all the necessary libraries & dependencies by React.js.
  - You can ignore this folder completely.

- public
  - This folder contains all static files like images, videos, fonts, etc.

- src
  - This folder contains all source files (The source directory where your React components, JavaScript files, and CSS are stored.)

  - App.jsx
    - The main React component that acts as the root of your user interface.
    - There is also App.css with it, which contains CSS related to App component.

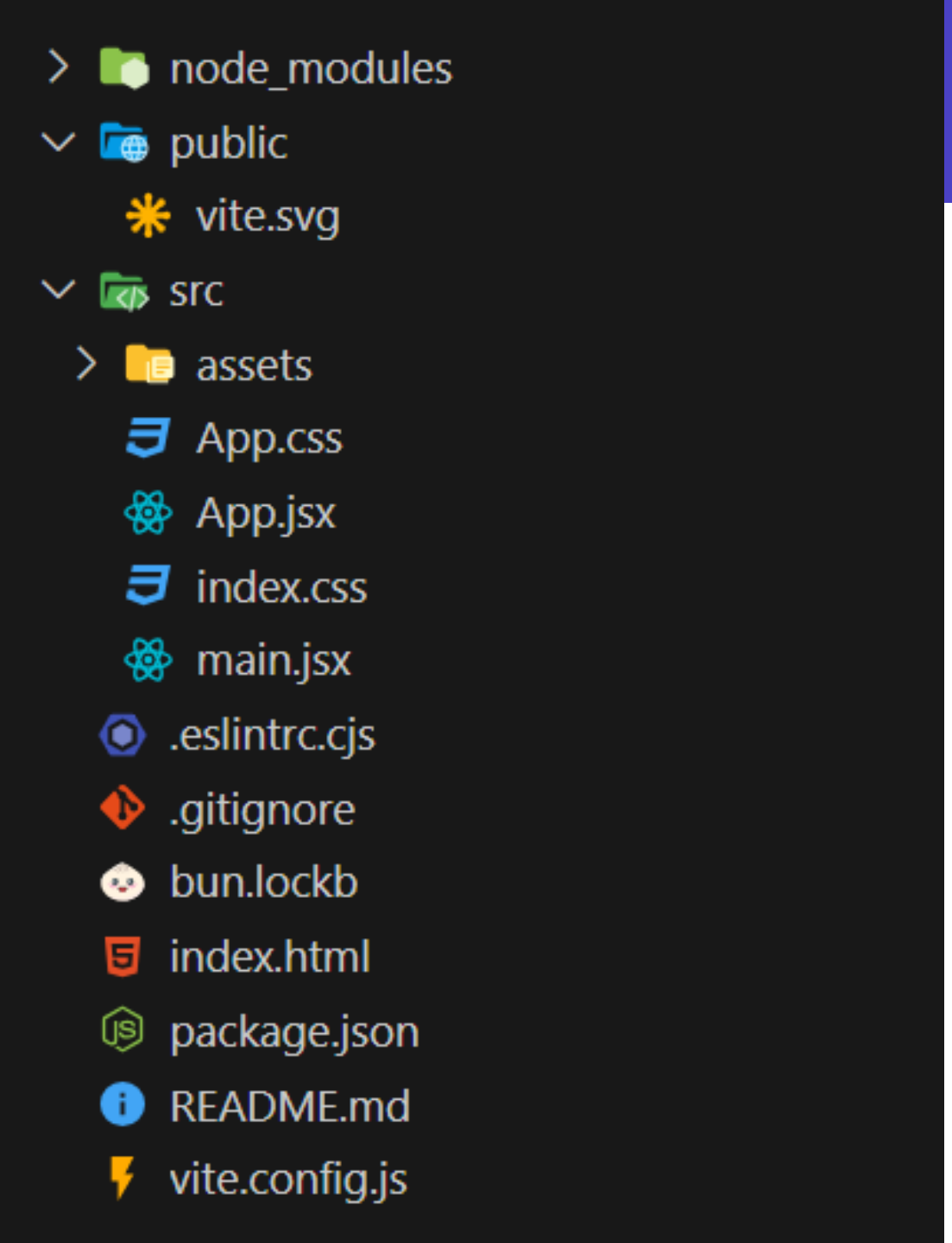
  - main.jsx
    - This is the entry point to our React.js project. where you render the App component.



Project Structure

- .eslintrc.cjs
  - This file includes configuration for ESLint.
  - ESLint is a static code analysis tool for identifying problematic patterns found in JavaScript code.
  - If you don't know about it, you can ignore it.

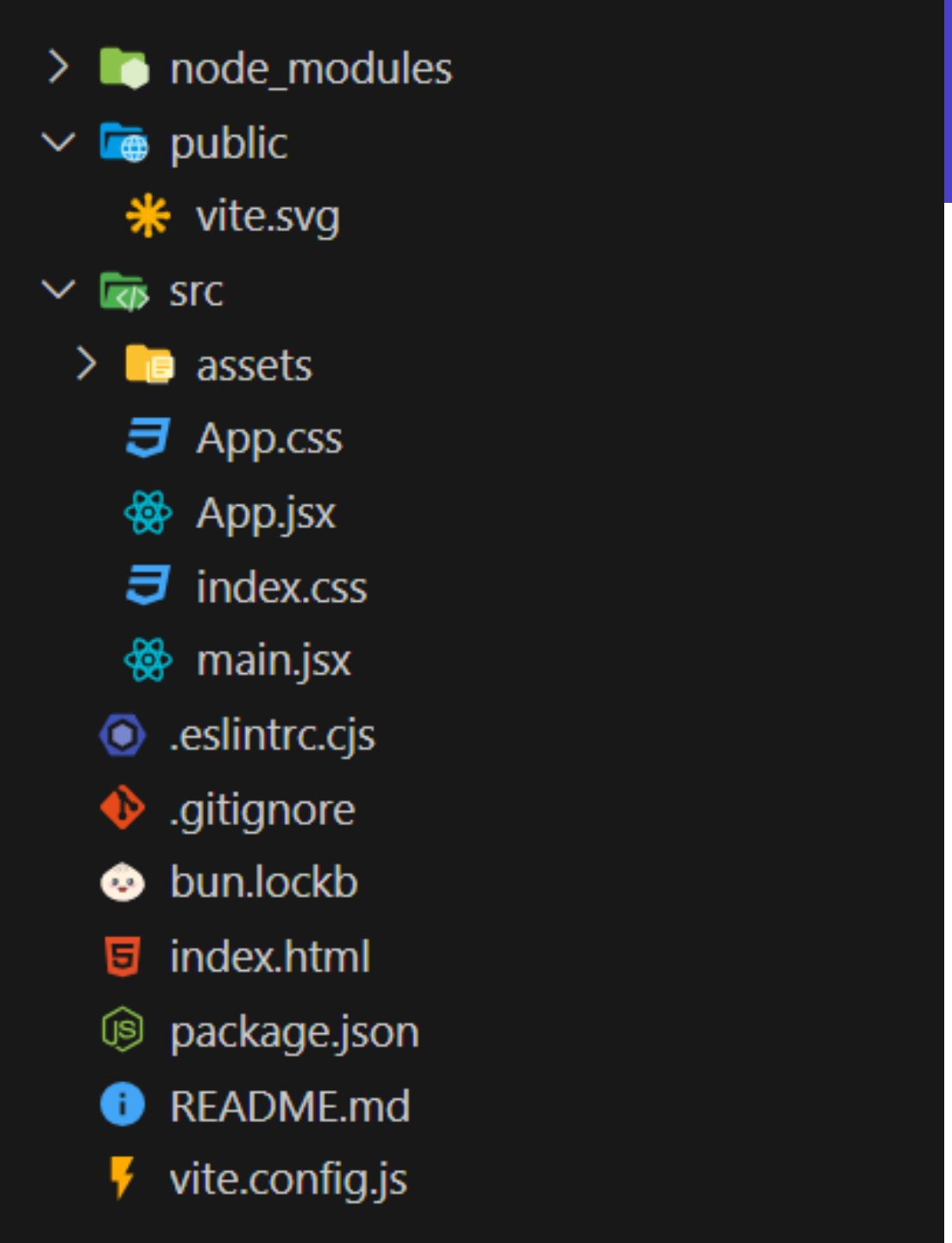
- .gitignore
  - This file includes all the files and folders to be ignored by Git.
  - If you read this file then you will find node_modules folder.
  - As node_modules folder is extremely huge in size, it's not supposed to be pushed to Git. You can easily generate node_modules by using npm install command.



Project Structure

- **index.html**
  - The main HTML file where your React application is loaded, serving as the entry point for the web browser.

- **package.json**
  - This file contains all the details or configuration regarding this project.
  - It includes name, version, description, scripts, dependencies.
  - **“scripts”** here includes all the scripts that you can run via **npm run _____**. You can also create custom scripts.
  - **“dependencies”** and **“devDependencies”** here includes all the packages or dependencies required by our project.



Project Structure

- In dependencies you will see **react** and **react-dom.**
  - React.js is just a library for creating dynamic user interfaces.
  - React DOM is used to render those components on our web page.
  - You might have heard about React Native for creating mobile applications which is also based on React.js.

- **vite.config.js**
  - This file contains config related to Vite
  - You can ignore this file if you don't know much about it.

Naming Convention

Naming Conventions

- camelCase
  - It is used for variables, functions/methods, properties inside objects, file names, etc.
  - Capitalization of each word except the first is done.
- PascalCase
  - It is used for component names, class names, types, etc.
  - Capitalization of each word is done.
- snake_case
  - It is not common in javascript but is used heavily in python.
  - Each word is separated by "_" and is in small letters
- kebab-case
  - It is common for file names, css classes, ids, etc.
  - Each word is separated by hyphen (-)

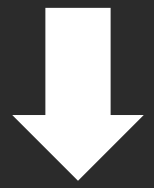
What is JSX?

Writing our First React.js Component

- JSX stands for **JavaScript XML**. It allows you to write HTML elements in JavaScript and place them in the DOM without any **createElement()** and **appendChild()** methods.

```
1 const App = () => {
2   return /*#__PURE__*/React.createElement("h1", {
3     name: "vinod"
4   }, "Hello World ");
5 };
```

```jsx
<h1>Hello World</h1>
```



```jsx
import { jsx as _jsx } from "react/jsx-runtime";
/*#__PURE__*/_jsx("h1", {
  children: "Hello World"
});
```

Go to https://babeljs.io/repl to try it out yourself.

More to JSX

- It is a syntactical sugar introduced by Facebook for writing React.js components easily.
- Because of JSX is a syntax extension for JavaScript that lets you write HTML-like markup inside a JavaScript file.
- At the end, this code is compiled to equivalent JavaScript code.

React Component

Our First React.js Component

- In React, a component is essentially a JavaScript function or class that returns JSX (JavaScript XML), a syntax extension that allows you to write HTML-like code inside JavaScript. Components are the building blocks of any React application, allowing for the creation of complex and interactive user interfaces through the assembly of isolated, reusable pieces.
- You can also use **.js** extension with React.js components but it's recommended to use **.jsx** to differentiate between regular JavaScript Files and React.js components.
- There are **two ways** to create components in React.js:
  - Class Based Components
    - This is how components were created before functional components.
  - Functional Components
    - This is modern and recommended way of writing React.js components.
    - We will follow through functional components in this course as that's what is used in all new projects.
    - If you find class-based components in legacy codebases, you can research online related to it. There are some differences, but main concept is same.

Our First React.js Component

Class Based Components

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

export class Welcome extends Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}
```

Functional Components

```
export function Welcome({ name }) {
  return <h1>Hello, {name}!</h1>;
}
```

Writing our First React.js Component

- After that open **App.jsx** then import our newly created component and use it.
- If you save the file, you will immediately see the changes on the website.
- This is possible due to HMR or Hot Module Replacement which is given by Vite.

```
  VITE v5.0.12  ready in 828 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
  ➜  press h + enter to show help
3:13:14 PM [vite] hmr update /src/App.jsx
3:13:18 PM [vite] hmr update /src/App.jsx (x2)
```

How React.js Works?

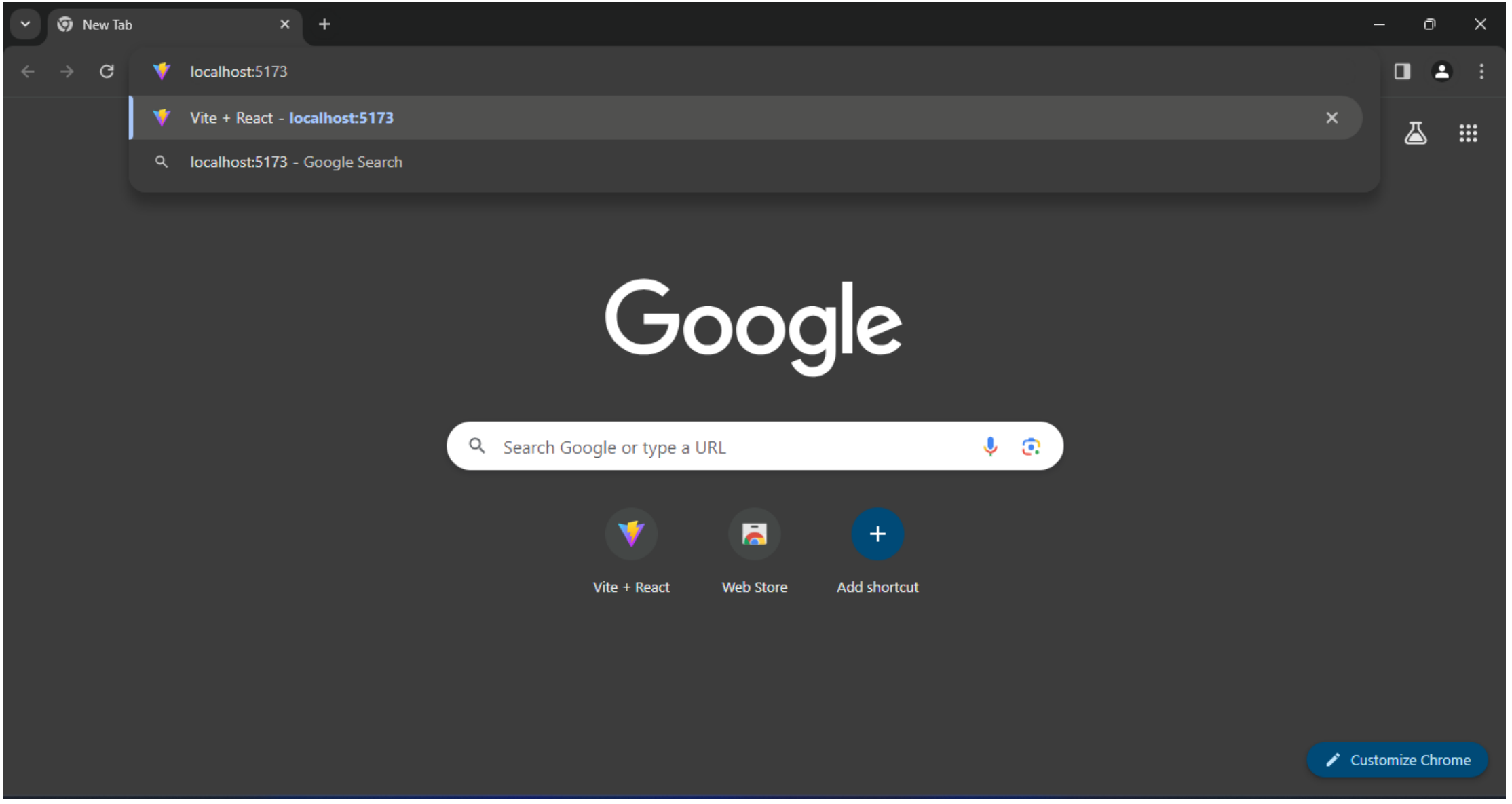
Secret of Client Side React.js

```
<!doctype html>
<html lang="en">
  <head>
    <script type="module">
import RefreshRuntime from "/@react-refresh"
RefreshRuntime.injectIntoGlobalHook(window)
window.$RefreshReg$ = () => {}
window.$RefreshSig$ = () => (type) => type
window.__vite_plugin_react_preamble_installed__ = true
</script>

    <script type="module" src="/@vite/client"></script>

    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

- If you right, click our website and open **View Page Source** then you will realize that **Hello World** text that we wrote is not present here.
- This is because React.js sends dummy HTML file with just a **div with id as “root”**.
- All the UI that is shown on web page is due to JavaScript code that is included with this file.

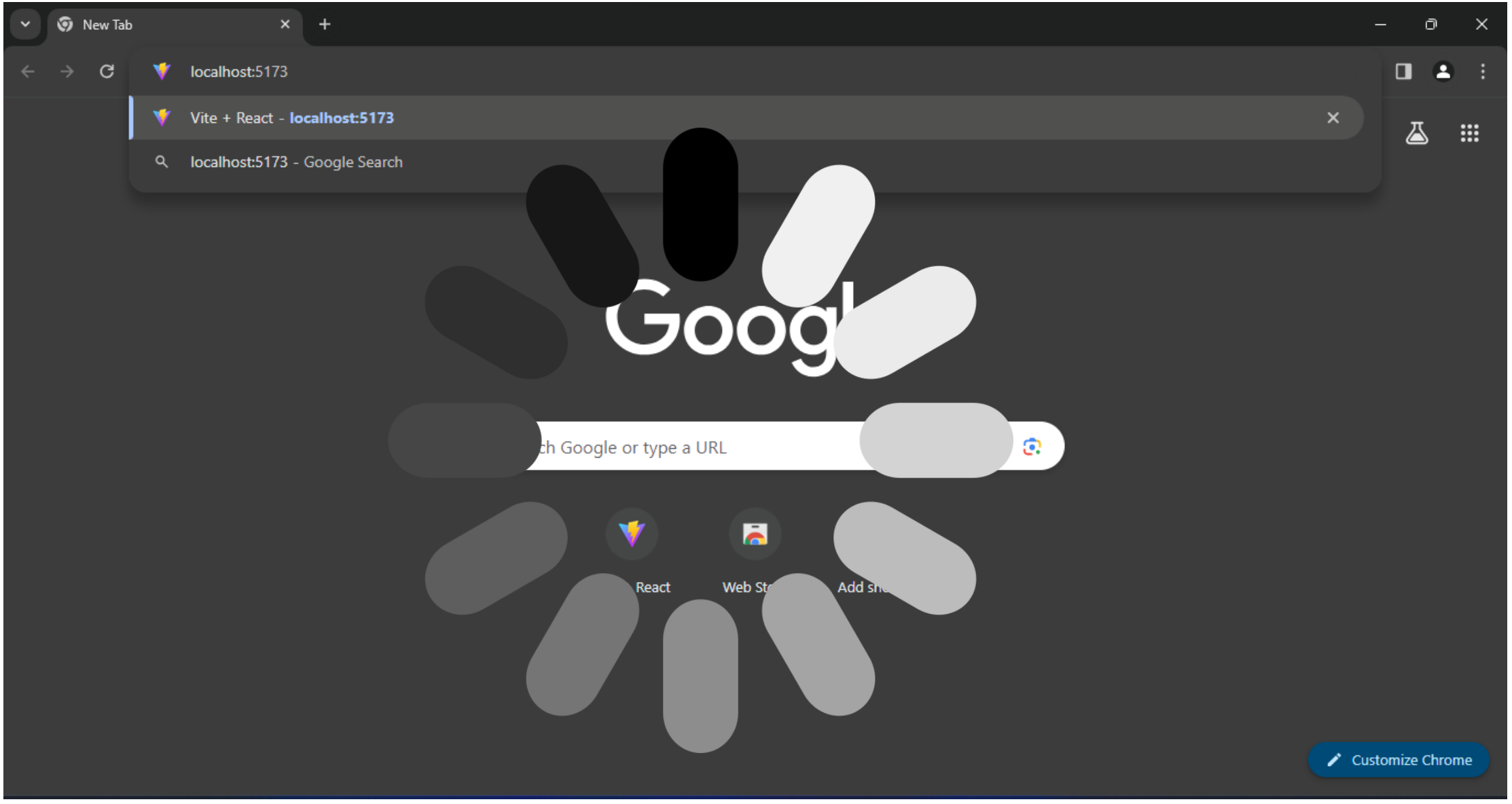
Step 1: Initial Rendering

THAPA TECHNICAL

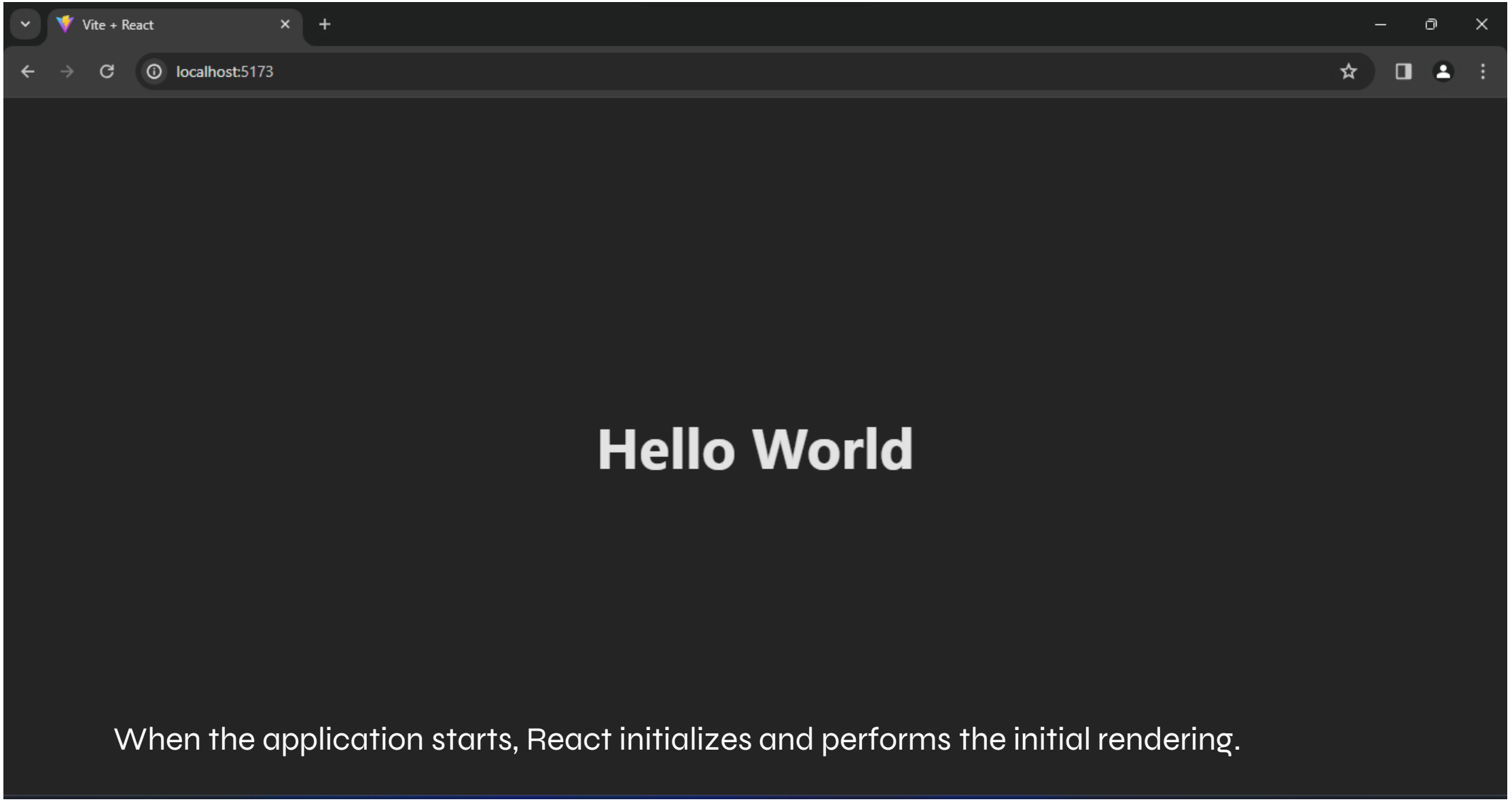


Step 1: Initial Rendering

THAPA TECHNICAL



Step 1: Initial Rendering



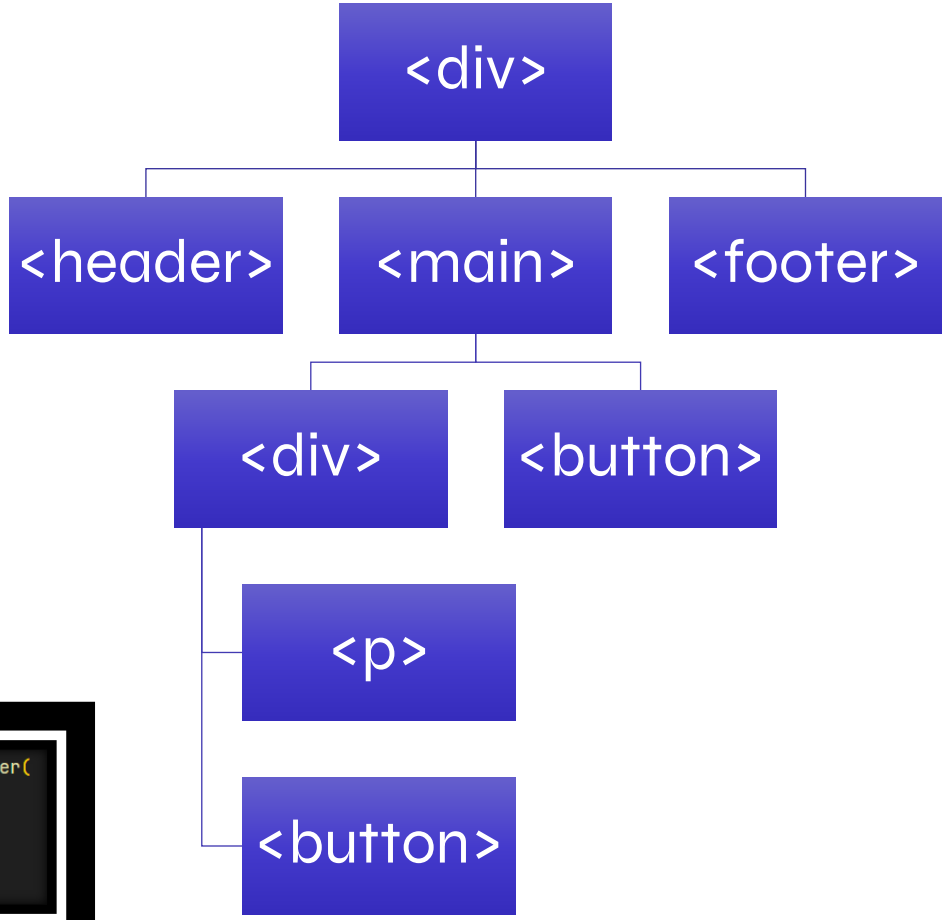
When the application starts, React initializes and performs the initial rendering.

Step 1: Initial Rendering


```
ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

This createRoot function creates a tree of nodes that describes the app is generated and saved in memory.
This is also referred to as **Virtual DOM** by most people.

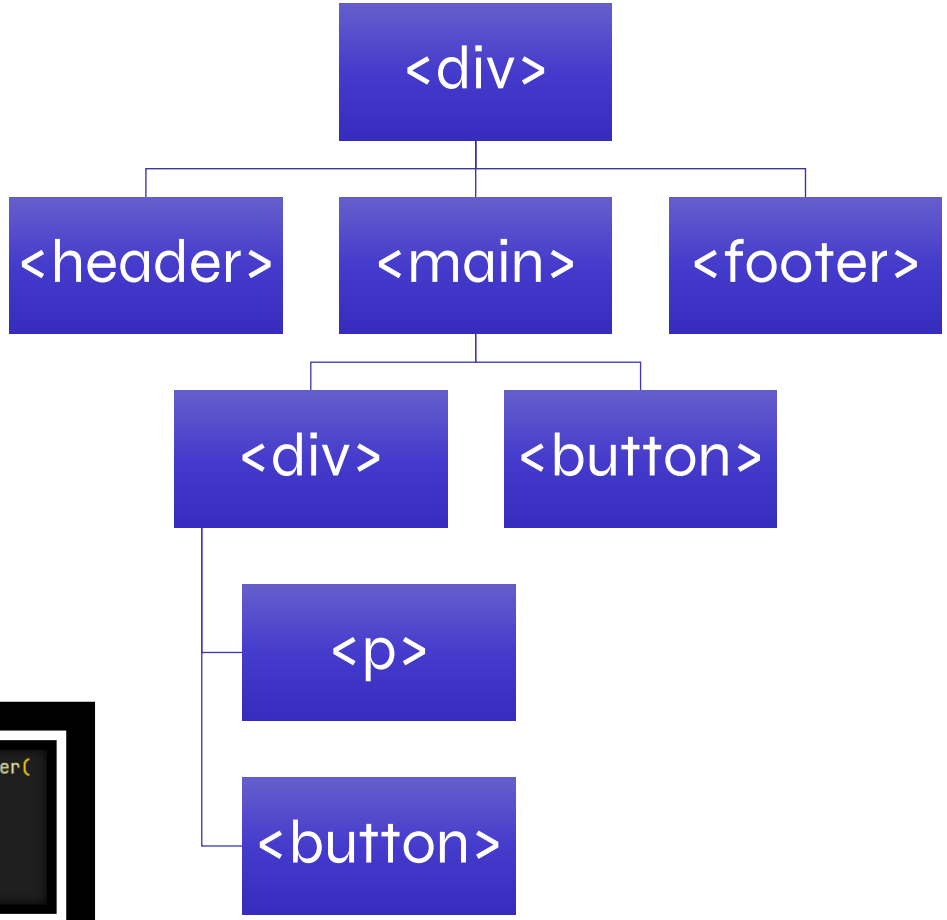
Step 1: Initial Rendering




```
ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

This createRoot function creates a tree of nodes that describes the app is generated and saved in memory.
This is also referred to as **Virtual DOM** by most people.

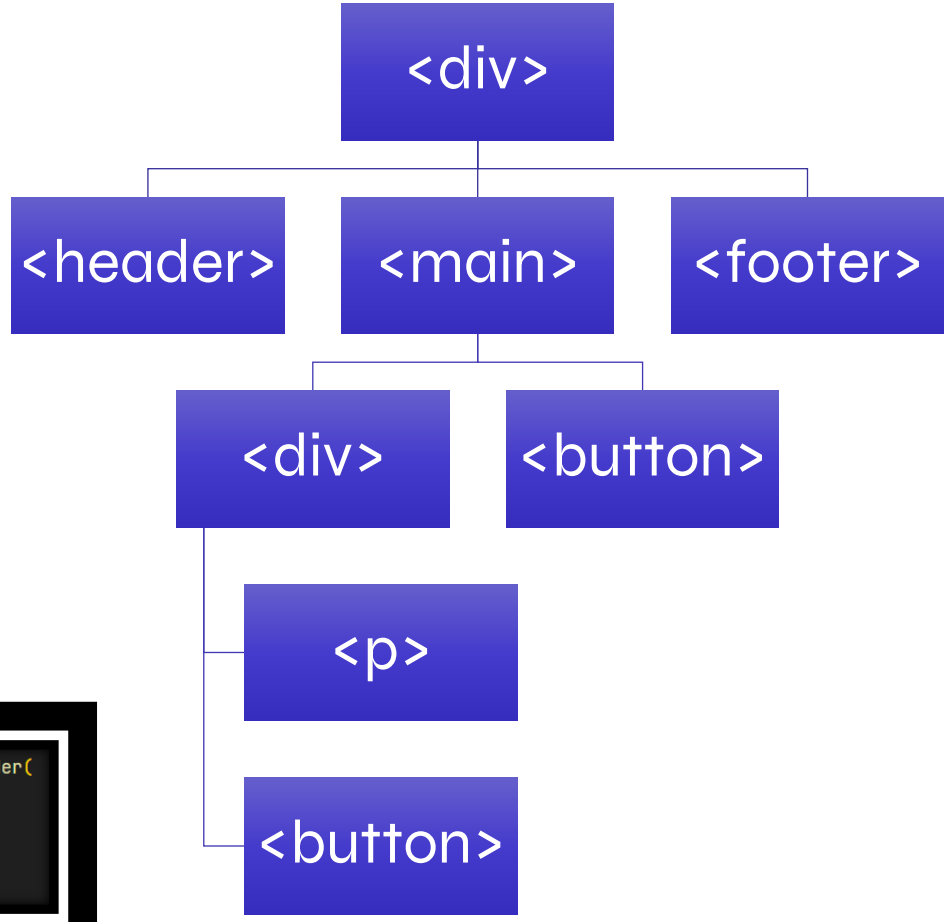
Step 1: Initial Rendering




```
ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

This tree is then flushed to the rendering environment — for example, in the case of a browser application, it's translated to a set of DOM operations.

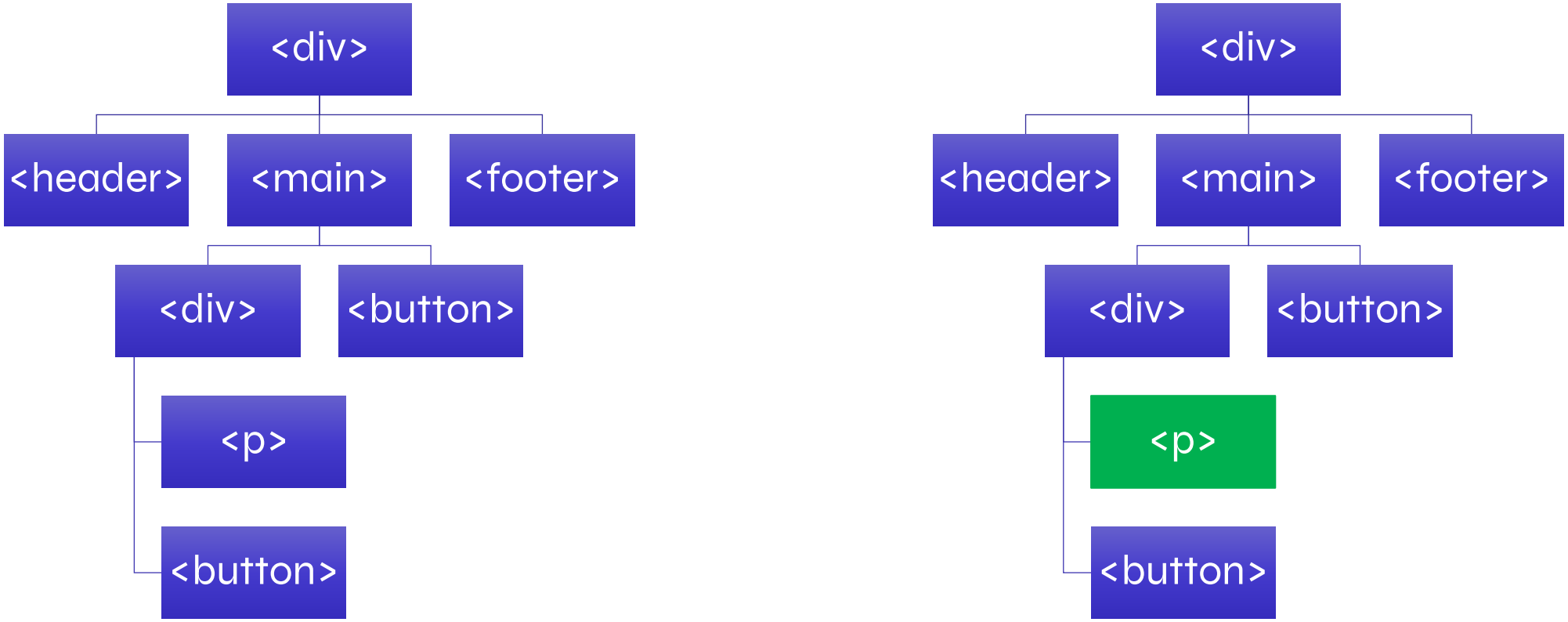
Step 1: Initial Rendering




```
ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

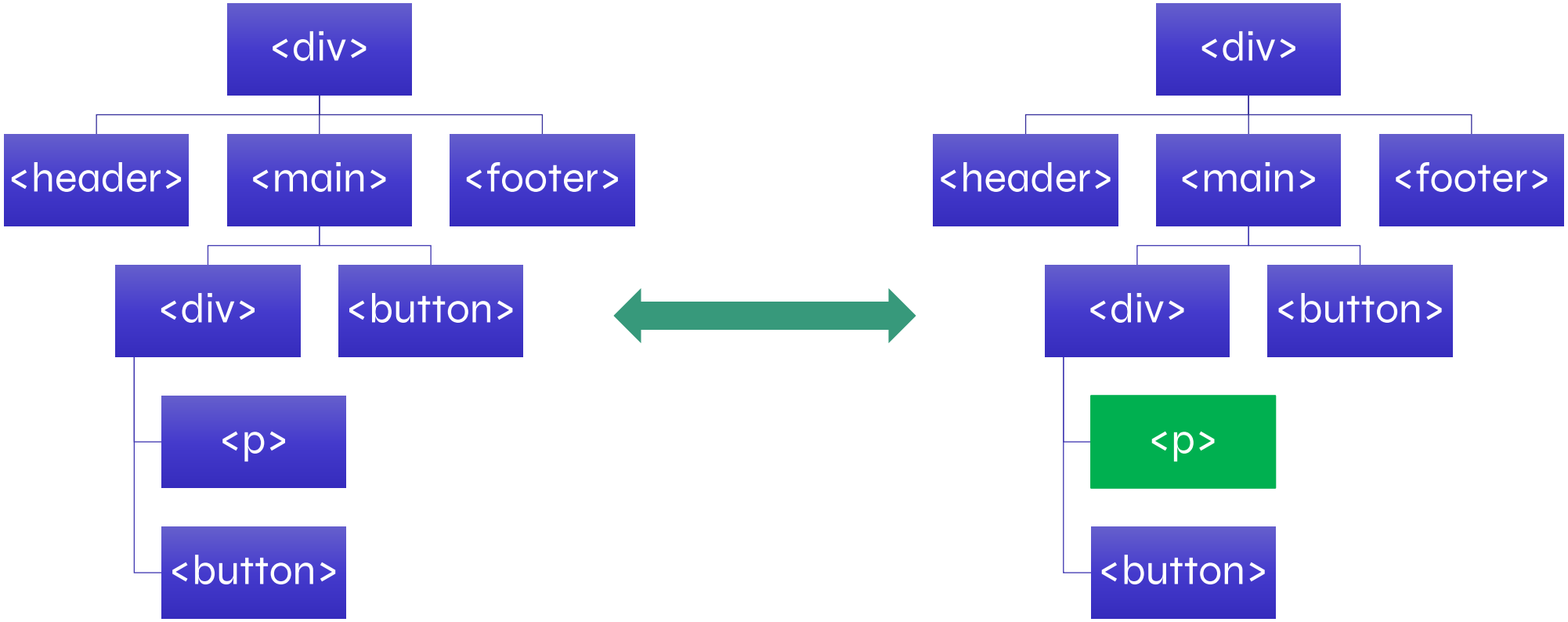
Imagine you click on the button and some kind of update operation is performed and content of <p> is updated.

Step 1: Initial Rendering



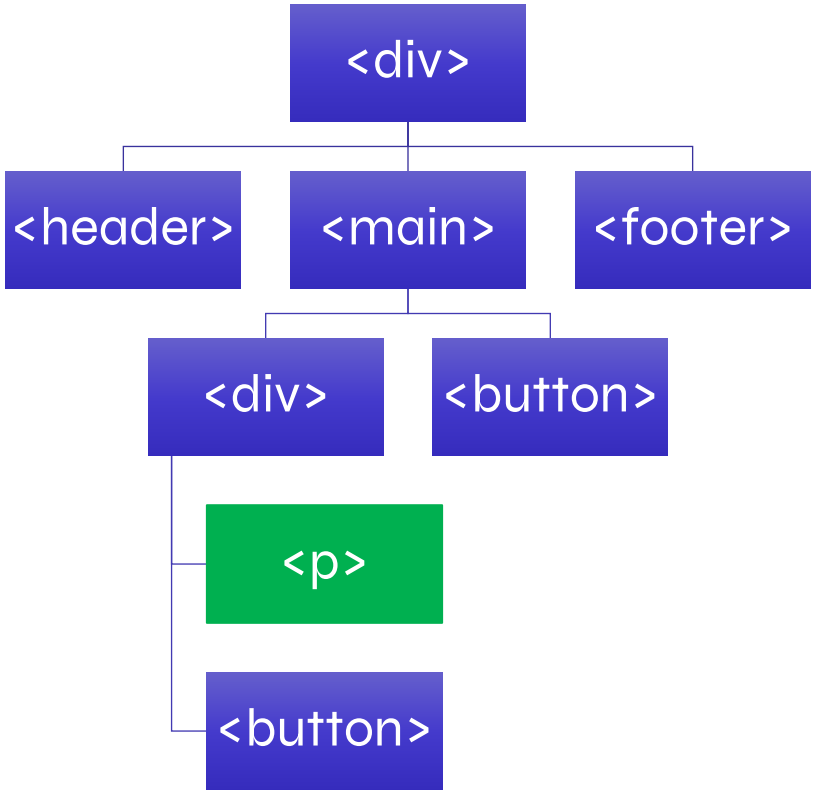
When the app is updated (usually via setState), a new tree is generated.

Step 1: Initial Rendering



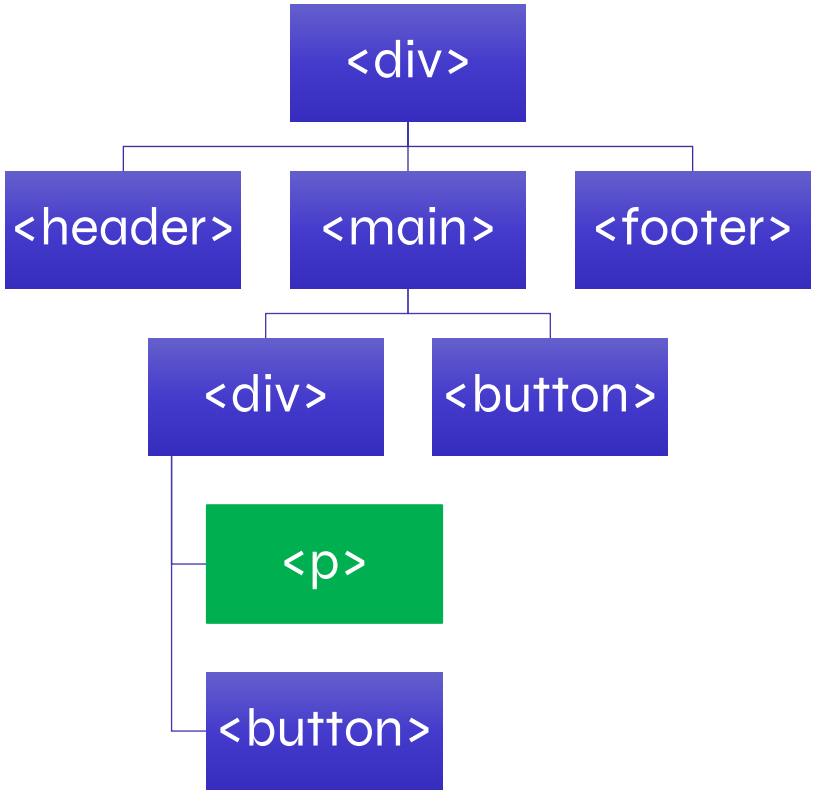
The new tree is diffed with the previous tree to compute which operations are needed to update the rendered app.

Step 1: Initial Rendering



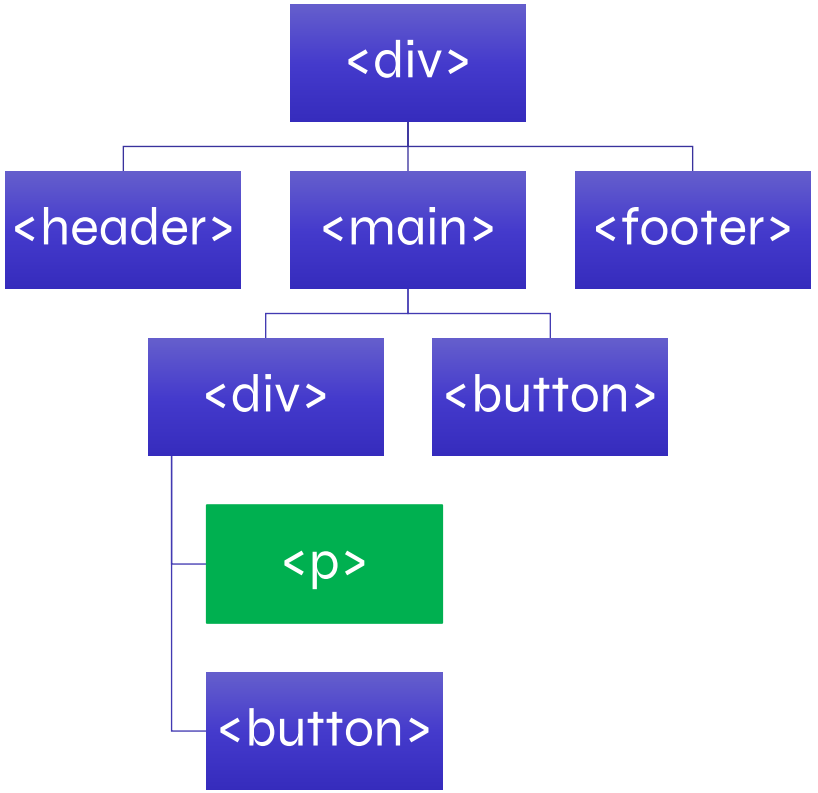
Then React.js will realize that <p> is different, then it will demand ReactDOM to update the DOM. It changes only the parts which are different in new tree.

Step 1: Initial Rendering



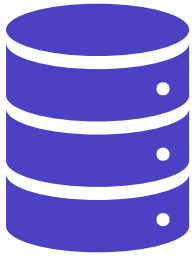
Reconciliation is the algorithm behind what is popularly understood as the "virtual DOM.“. This whole architecture is known as React Fiber Architecture. React Fiber is a reimplementation of React's core algorithm.

Step 1: Initial Rendering



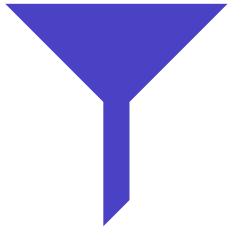
React Fiber was simply introduced to revise the mechanism of React.js to decrease the amount of DOM operations need to be made smartly.

Concepts of React.js Mechanism



reconciliation

The algorithm React uses to diff one tree with another to determine which parts need to be changed.



update

A change in the data used to render a React app. Usually the result of `setState`. Eventually results in a re-render.

Why Virtual DOM term is misleading?

- React can render to various environments, including the DOM, native iOS, and Android views via React Native.
- The term "virtual DOM" is a bit misleading because React's reconciler and renderer are separate phases.
- Reconciliation is the process of computing changes in a tree, while rendering is the actual update of the rendered app.
- React DOM and React Native can have their own renderers but share the same reconciler from React core.
- This separation allows React to efficiently update different rendering targets.

acdlite/react-fiber-architecture: A description of React's new core algorithm, React Fiber (github.com)

Basics of JSX

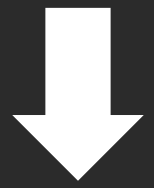
Why Virtual DOM term is misleading?

- React can render to various environments, including the DOM, native iOS, and Android views via React Native.
- The term "virtual DOM" is a bit misleading because React's reconciler and renderer are separate phases.
- Reconciliation is the process of computing changes in a tree, while rendering is the actual update of the rendered app.
- React DOM and React Native can have their own renderers but share the same reconciler from React core.
- This separation allows React to efficiently update different rendering targets.

acdlite/react-fiber-architecture: A description of React's new core algorithm, React Fiber (github.com)

React Fragments

React Fragments

- In React.js, a component can't return multiple elements without a wrapper div.
- You can do this to imitate it: Using an Array of Elements with Keys
- `return [<p>Ram</p>, <strong>20</strong>];`
- But it's tedious, to fix this React 16 introduced Fragments, which allow components to return multiple elements without adding extra nodes to the DOM.
- ```
  import { Fragment } from "react";
  return <Fragment> // </Fragment>
  ```
- **Alternatively, you can also use its shorthand syntax:**
- `<> // </>`
  ```

Why can't we return multiple elements at the same time?


```
<h1>Hello World</h1>
```



```
import { jsx as _jsx } from "react/jsx-runtime";
/*#__PURE__*/_jsx("h1", {
  children: "Hello World"
});
```

- As after JSX is compiled it is converted to normal JavaScript object, you can't return multiple JavaScript object at the same time.
- `return ( jsx(…), jsx(…) );`
- As you already know, above code is not valid JavaScript.

Dynamic Values in JSX

Dynamic Values in JSX

- You can add any variables of your component in JSX using {} syntax.
- ```
  function MyComponent(){
    const value = “Hey”;
    return {value} ;
  }
  ```
- **JSX allows you to write JavaScript expressions inside curly braces. This includes operations, function calls, and other JavaScript expressions that produce a value.**
- `return <div>The result is: {1 + 2 + 3}</div>;`
  ```

Conditionals Values in JSX

Conditionals in JSX

- You can either return a JSX early or you can use ternary operators for conditionals in JSX.

- `return <p>{ age >= 18 ? “Adult” : “Minor” }</p>`

- `return <p>{ age >= 18 && “You can vote” }</p>`

- `return <p>{ username || “Guest” }</p>`

Interview Question Time

Import & Export in React JS

Import Export in Reactjs

- 1: Default Export and Import
- **2: Named Export and Import**
- **3: Mixed Export and Import**

Looping in JSX

Looping in JSX

- We don't have for loops in JSX, so we have to use .map() method of arrays.

- ```
  return (
    <ul>
      {students.map((student) => (
        <li key={student}>{student}</li>
      ))}
    </ul>
  );
  ```
- **React.js will complain if you don't pass key prop to li** DEMO
- **React.js uses keys to differentiate each elements, so that it can remove or edit correct element when a students gets added or removed. Don't worry we will investigate this in further details later.**
- **We can use arrays in JSX too. So, here students.map returns array of JSX which gets rendered on the page.**
  ```

Props in JSX

Props

- React props (properties) facilitate data transfer from parent to child components.
- Data flows unidirectionally, ensuring a clear direction of information in React applications.
- Props are immutable, meaning child components cannot modify the data received from parents.
- Child components access props through their function parameters.
- You can also pass JSX as props to another component.

Destructuring in JSX

Events Handling

Events Handling

Event handling in React.js is the process of capturing and responding to user interactions, such as clicks, keystrokes, or form submissions, within a React application. Event handling in React follows a similar pattern to standard JavaScript event handling but with some key differences, such as using synthetic events for cross-browser compatibility and providing consistent event handling across different elements and browsers.

What is SyntheticBaseEvent in React?

When you handle events in React, like clicking a button or typing in an input box, React wraps the native browser events in something called a SyntheticEvent. This SyntheticEvent is a cross-browser wrapper around the browser's native event, making sure that events behave consistently across all browsers.

Events Handling

Here are some widely used React.js events:

1. **onClick**: Triggered when an element is clicked.
2. **onChange**: Triggered when the value of an input element changes.
3. **onSubmit**: Triggered when a form is submitted.
4. **onMouseEnter**: Triggered when the mouse pointer enters an element.
5. **onMouseLeave**: Triggered when the mouse pointer leaves an element.
6. **onKeyDown**: Triggered when a key is pressed down.
7. **onKeyUp**: Triggered when a key is released.
8. **onFocus**: Triggered when an element receives focus.
9. **onBlur**: Triggered when an element loses focus.
10. **onInput**: Triggered when the value of an input element is changed (similar to onChange).

Events Handling

Event Naming Conventions

CamelCase: Event names in JSX should be written in camelCase.

For example, onClick, onChange, onSubmit.

Prefix with 'on': Event handlers should be prefixed with "on".

For example, onClick, onMouseEnter.

Function Naming Conventions

Prefix with 'handle': It is a common convention to prefix event handler functions with "handle" to clearly indicate their purpose. For example, handleClick, handleChange, handleSubmit.

Descriptive Names: Function names should be descriptive and reflect what the function does. For example, handleFormSubmit instead of just handleSubmit if it's specifically for form submission.

Passing event handlers as props

Passing event handlers as props

Passing event handlers as props in React.js is a common pattern used to allow child components to communicate with parent components. This pattern is useful in various scenarios, including:

Form Handling: Passing event handlers like onChange or onSubmit to form components allows the child components to update the form data and notify the parent component of changes.

User Interaction: Event handlers like onClick can be passed to interactive elements (e.g., buttons, links) to trigger specific actions in the parent component, such as opening a modal or navigating to a different page.

State Management: Event handlers can be used to update state in the parent component, which can then be passed down to child components as props to reflect the updated state.

Callback Functions: Event handlers can be used as callback functions to handle asynchronous operations or to update state based on the result of an operation.

Passing event handlers as props


- ```
  function WelcomeUser({ onClick }){
    return <button onClick={onClick}>Click me</button>
  }
  ```
- `<WelcomeUser onClick={() => alert("Clicked")} />`

- Convention is to start with “on”
- Instead of onClick here, you can also name it onButtonClick, name doesn’t matter.
- Passing functions to another component is not just for events but you can pass any functions you want to other component as props.
  ```

Event Propagation

Event Propagation

- In React.js, event propagation refers to the process by which events propagate or "bubble" up from the target element through its parent elements in the DOM hierarchy. React follows the same event propagation model as regular JavaScript DOM events.

- When an event occurs on an element in a React component, such as a button click, the event is first captured at the target element and then bubbles up through the parent elements, triggering any event handlers that have been defined along the way. This allows you to handle events at different levels of the component hierarchy.

- React provides a way to stop event propagation using the stopPropagation() method, which can be called on the event object within an event handler. This method prevents the event from bubbling up further in the DOM, ensuring that only the event handler on the target element is triggered.

State in React

States in React JS

- In React, state refers to an object that holds data or information about the component. State is managed within the component (just like variables declared in a function). However, unlike regular variables, when state changes, React re-renders the component to reflect these changes, keeping the user interface in sync with the data.

- State is dynamic and mutable, meaning it can change over time, usually in response to user actions, server responses, or other events

State Variable

The current state value.

The initial value of the state..

```
const [count, setCount] = useState(0)
```

Variable (key)

A function that updates the state variable.

Why Do We Need State in React?

1. **Dynamic UI Updates**: State allows your components to update dynamically in response to user input or other events. For example, in a form, the state might hold the current value of the input fields, updating in real-time as the user types.

2. **Interactivity**: State makes your application interactive. By maintaining state, you can create components that respond to user actions, such as clicks, form submissions, or keyboard inputs.

3. **Data Management**: State helps manage data within a component. For instance, you can fetch data from an API and store it in the state, which will then be used to render the UI.

4. **Component Communication**: State can be lifted up to parent components to manage shared state across multiple child components, ensuring consistent data flow and synchronization.

How State works in React

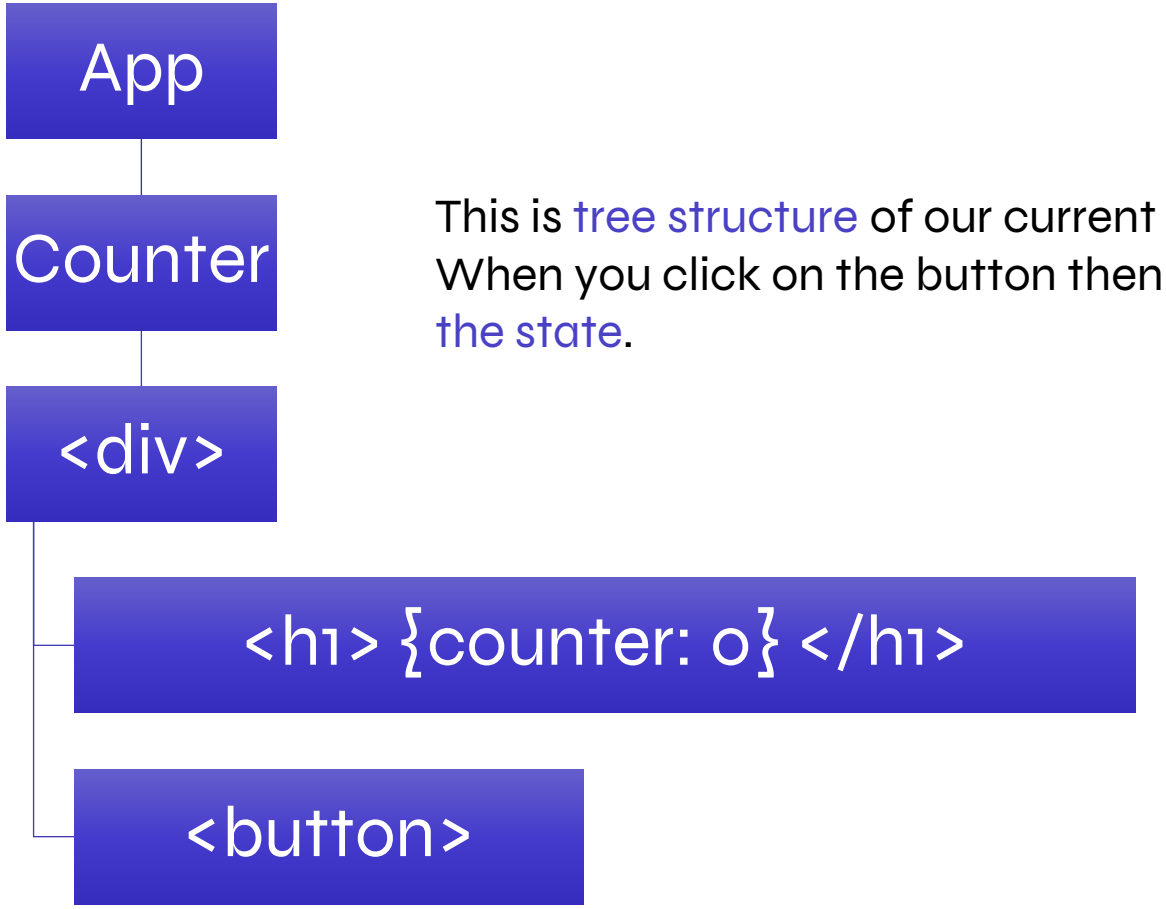
How State Works?

In React, state is a way to store and manage data that can change over the lifetime of a component. When state changes, React re-renders the component to reflect the new state. This ensures that the user interface stays in sync with the underlying data.

Using useState in Function Components

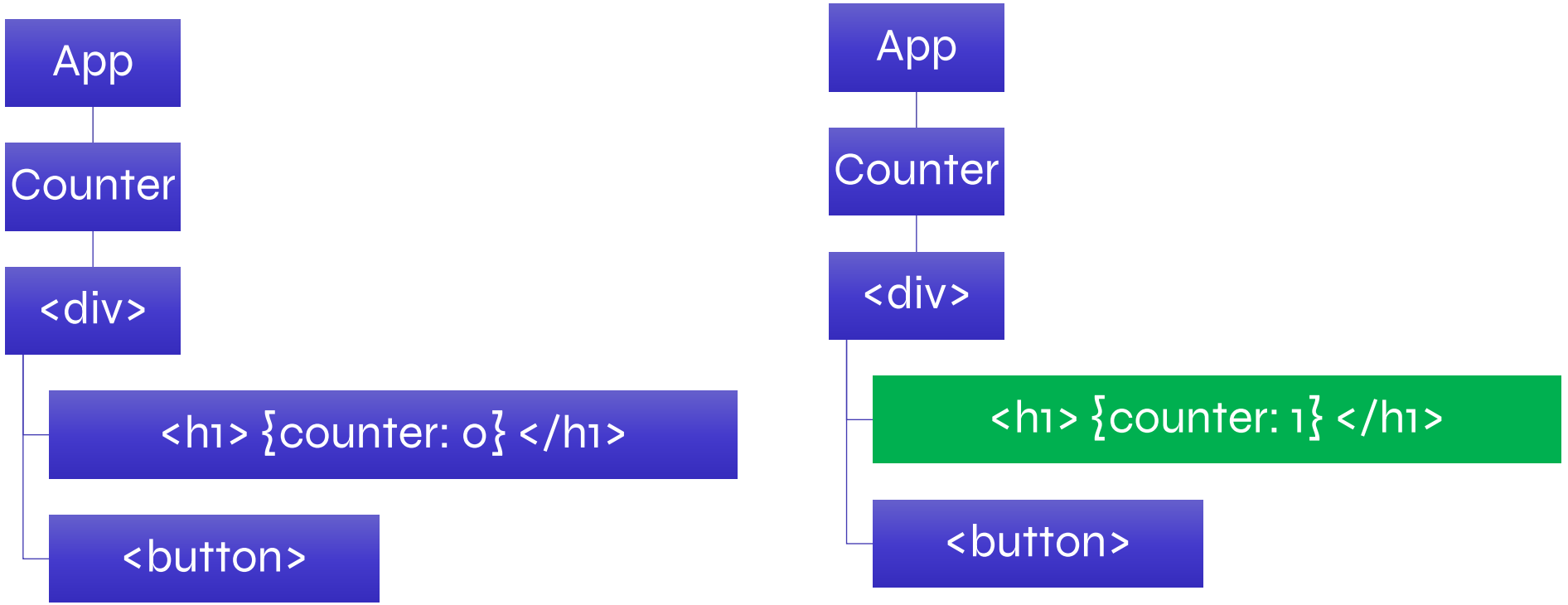
- Initialization: You call useState with the initial state value. It returns an array with two elements: the current state value and a function to update that state.
- Updating State: When you call the updater function, React schedules a re-render of the component with the new state value.
- Preservation of State: React maintains state between renders. When a component re-renders, it doesn't reinitialize the state. Instead, it uses the preserved state from the previous render.

How State Works?



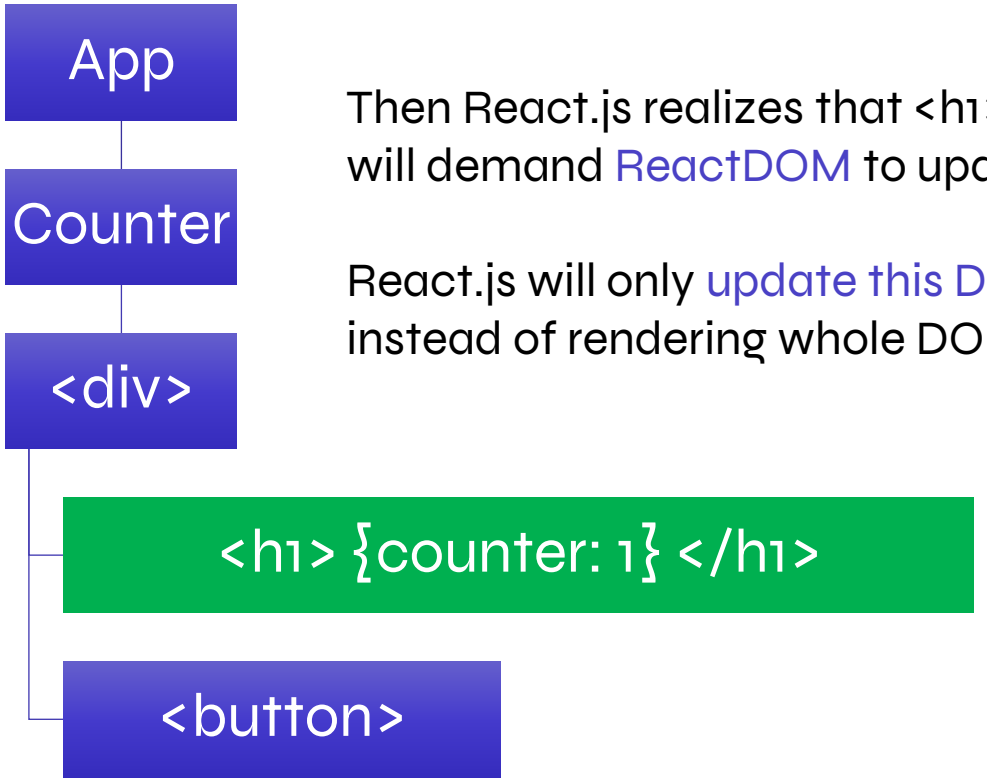
This is tree structure of our current project.
When you click on the button then we are trying to update the state.

How State Works?



The button calls setState() function, which notifies React.js that we are trying to update a state. React.js then creates a new tree with the new state value.

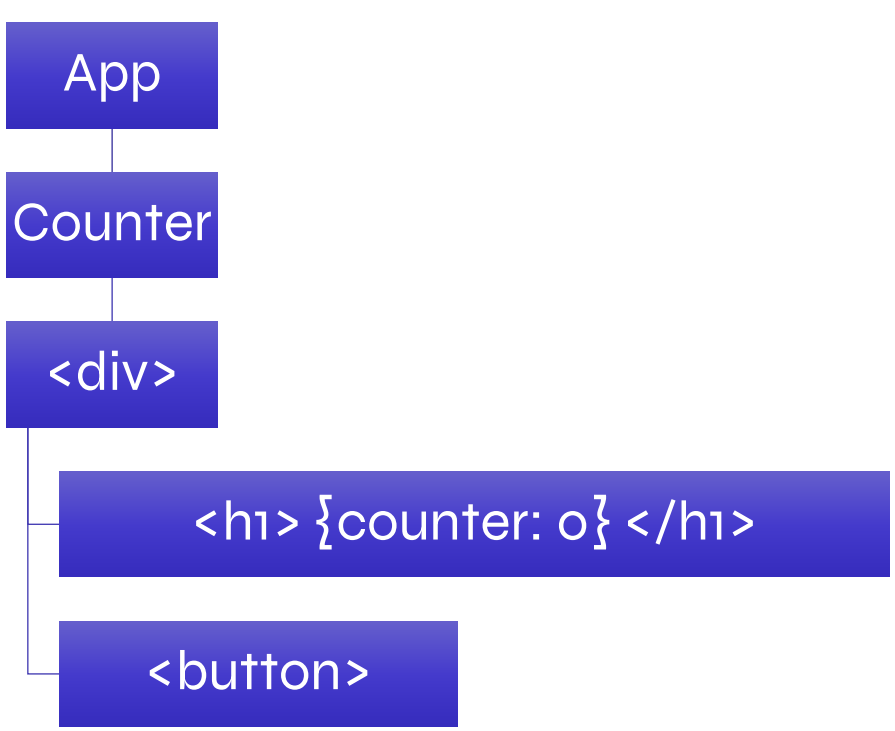
How State Works?

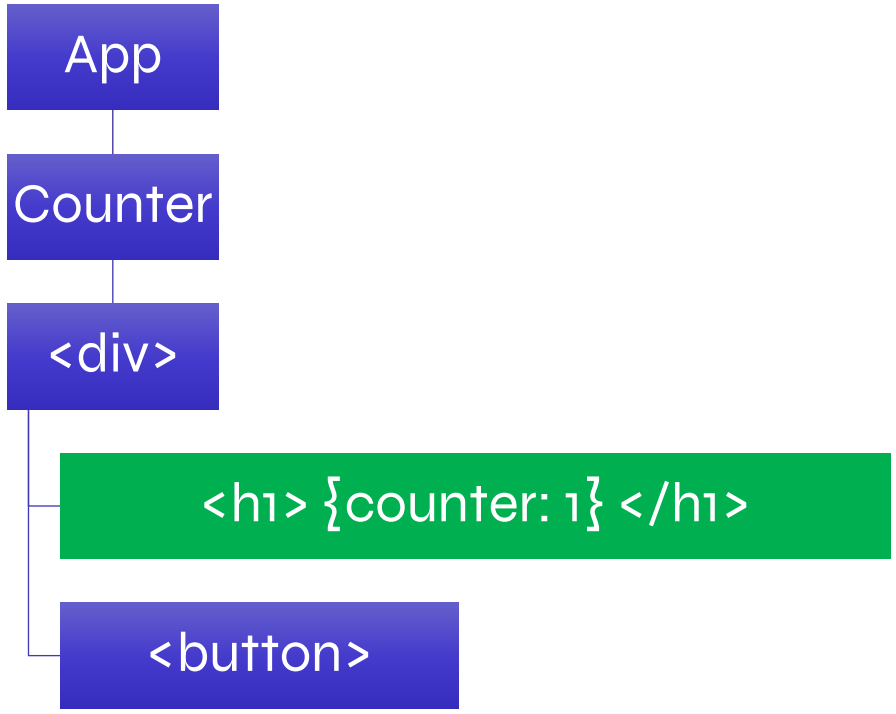


Then React.js realizes that <h1> is different, then React.js will demand ReactDOM to update its value in DOM.

React.js will only update this DOM making it effective instead of rendering whole DOM again.

How State Works?





Note: When React.js creates a new tree, it will re-run or re-render the affected component and all its children. So, in this case It will re-run our <Counter /> component, it won't re-render other components outside. Let's demonstrate it.

How State Works?


```
function App() {
 return (
  <div>
   <ParentComponent>
        <ChildComponent />
        <AnotherChildComponent />
   </ParentComponent>
   <SiblingComponent />
  </div>
 );
}
```

Here's how the re-rendering works:

Initial Render: When the App component first renders, React renders ParentComponent, ChildComponent, AnotherChildComponent, and SiblingComponent.

State Change in ParentComponent: Suppose there is a state change in ParentComponent. React will re-render ParentComponent and all its children (ChildComponent and AnotherChildComponent).

Components Outside: SiblingComponent is not affected by the state change in ParentComponent. Therefore, it will not be re-rendered.

Why the state value does not reset to its initial value on re-render?

First Render: const [count, setCount] = useState(0);

count is initialized to 0.

Button Click: increment function is called.

setCount(count + 1) updates count to 1.

Re-render: React re-renders the component. (when user click button for the 2nd time)

const [count, setCount] = useState(0); sees that count is now 1 and uses 1 as the current state. *The useState hook is smart enough to only use the initial value the very first time the component renders.*

Derived State

What is Derived State in React?

Derived state is any state that can be computed based on other state or props. It is not stored directly in the component's state but is calculated when needed. This approach helps avoid duplication and keeps the state simpler and more manageable.

Ex: `const userCount = users.length;`

Benefits of Derived State

- Avoid Redundancy: By deriving values from existing state, you avoid storing redundant data.
- Consistency: Ensures that derived values are always in sync with the underlying state or props.
- Simplicity: Reduces the complexity of state management by minimizing the number of state variables.

Lift State Up - React

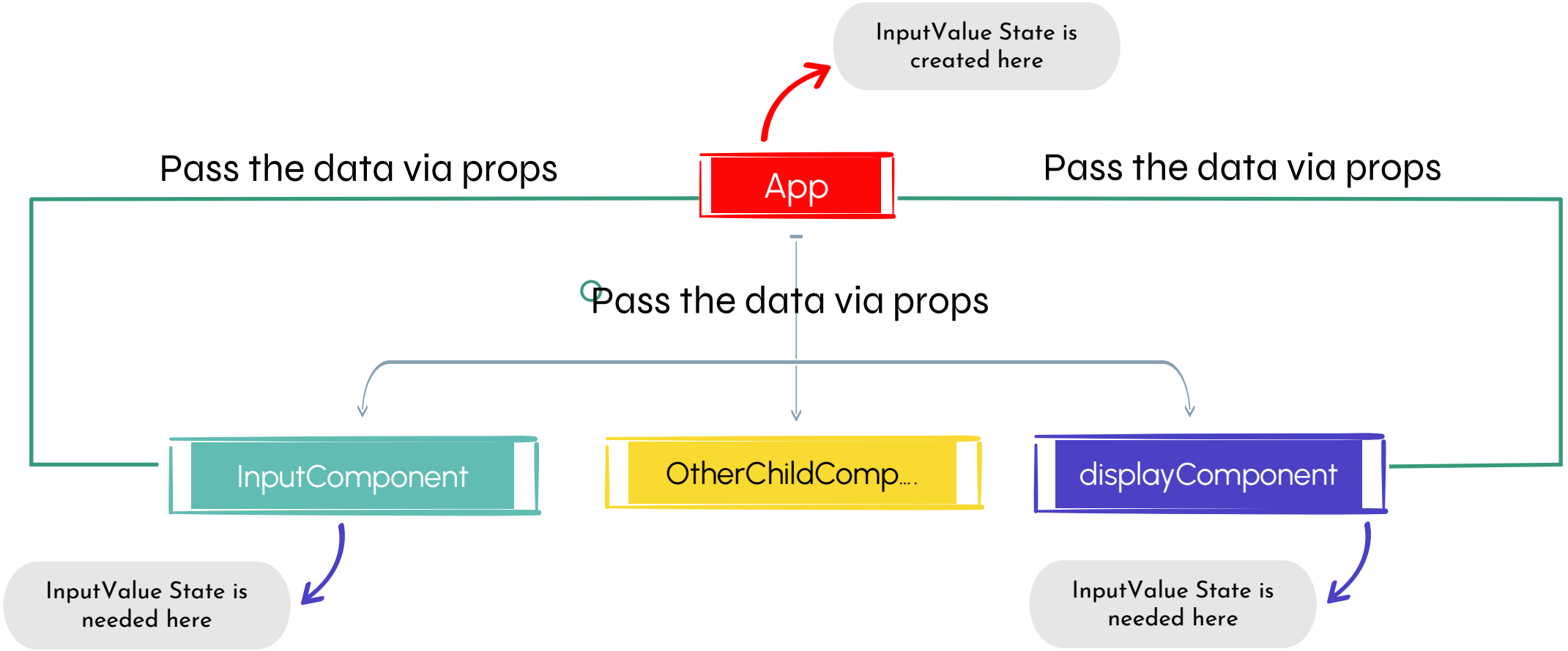
Lifting the State Up in React

Lifting State Up is a pattern in React where you move the state from child components to a common parent component so that multiple child components can share and synchronize this state. This ensures that the state is managed at a higher level in the component hierarchy, allowing data to flow down as props and actions (such as events) to flow up.

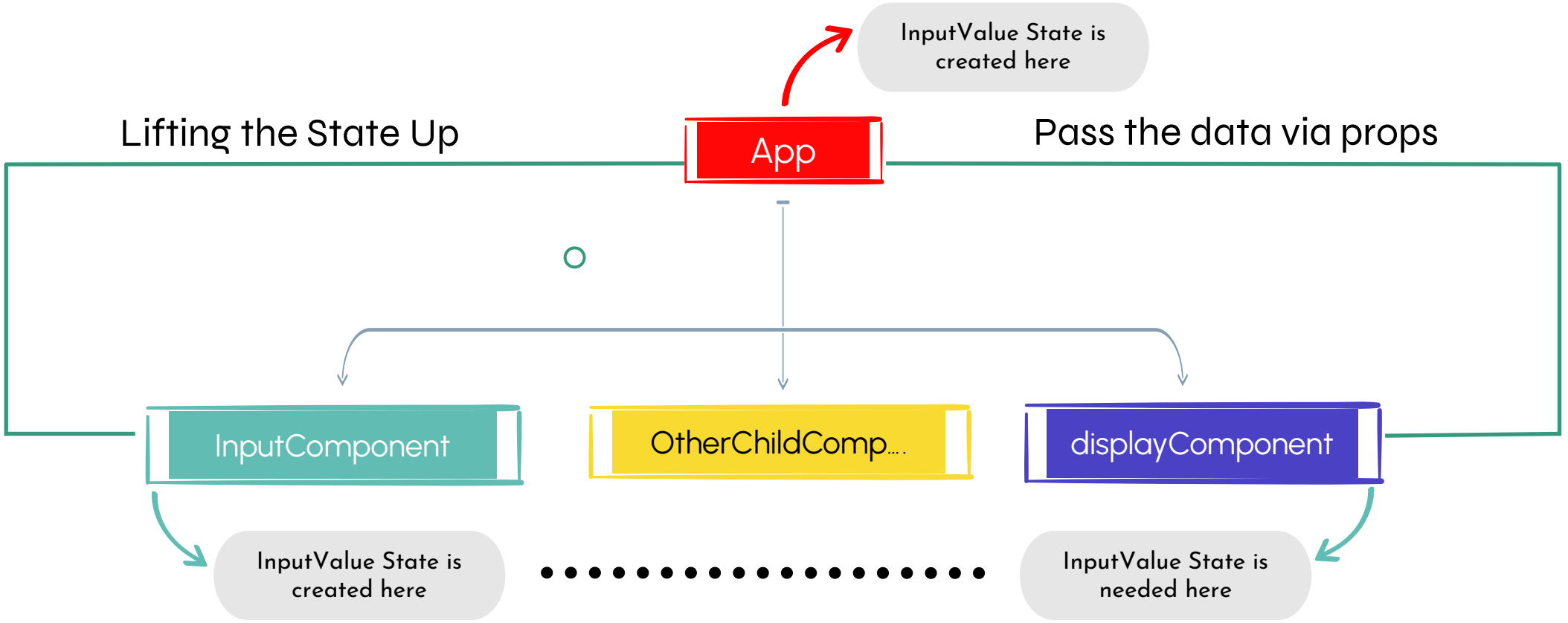
Use Case:

When you have two or more components that need to share the same state, you should lift the state up to their nearest common ancestor. This allows these components to stay in sync and ensures that the state is managed in a single place.

When not to Lifting the State Up in React



Lifting the State Up in React



Updating parent state from child and vice-versa

- For updating state of children from parent, you can use concept called **Lifting the state up**.
- Here, you will lift the state from child to parent then pass that state down using props.

- For updating state of parent from children, you can pass a function which updates the state from parent to children as props, then children can update it.

Lifting the State Up in React

Parent Component: Holds the state inputValue and the state handler setInputValue.

InputComponent: Receives inputValue and setInputValue as props. It updates the state via setInputValue when the input changes.

DisplayComponent: Receives inputValue as a prop and displays the current input value.

Lifting the State Up in React

Parent Component: Holds the state inputValue and the state handler setInputValue.

InputComponent: Receives inputValue and setInputValue as props. It updates the state via setInputValue when the input changes.

DisplayComponent: Receives inputValue as a prop and displays the current input value.

1st Mini Project

Toggle Switch Component

Features

- A switch that toggles between on and off states.
- Change the appearance based on the current state (e.g., different colors for on and off).
- Display the current state (e.g., "On" or "Off").

Use Cases

1. **Feature Toggle:** Use the toggle switch to enable or disable features in your application.
2. **Dark Mode:** Implement a dark mode toggle switch to switch between light and dark themes.
3. **Settings:** Use toggle switches in a settings menu to turn various settings on or off.
4. **Visibility Control:** Toggle the visibility of certain UI elements or sections.
5. **Form Inputs:** Use toggle switches as part of form inputs to capture user preferences.

Spread Operator

Spread Operator (...)

The spread operator (...) in JavaScript (React) allows an iterable (like an array or string) to be expanded into individual elements. It provides a concise way to spread the elements of an iterable object into places where multiple elements (arguments or elements in array literals) are expected.

Syntax:

Arrays: const newArray = [...oldArray];

Objects: const newObject = {...oldObject};

Spread Operator (...)

Use Cases

Use Case 1: Copying Arrays

Use Case 2: Concatenating Arrays

Use Case 3: Adding Elements to an Array

Use Case 4: Spreading Object Properties

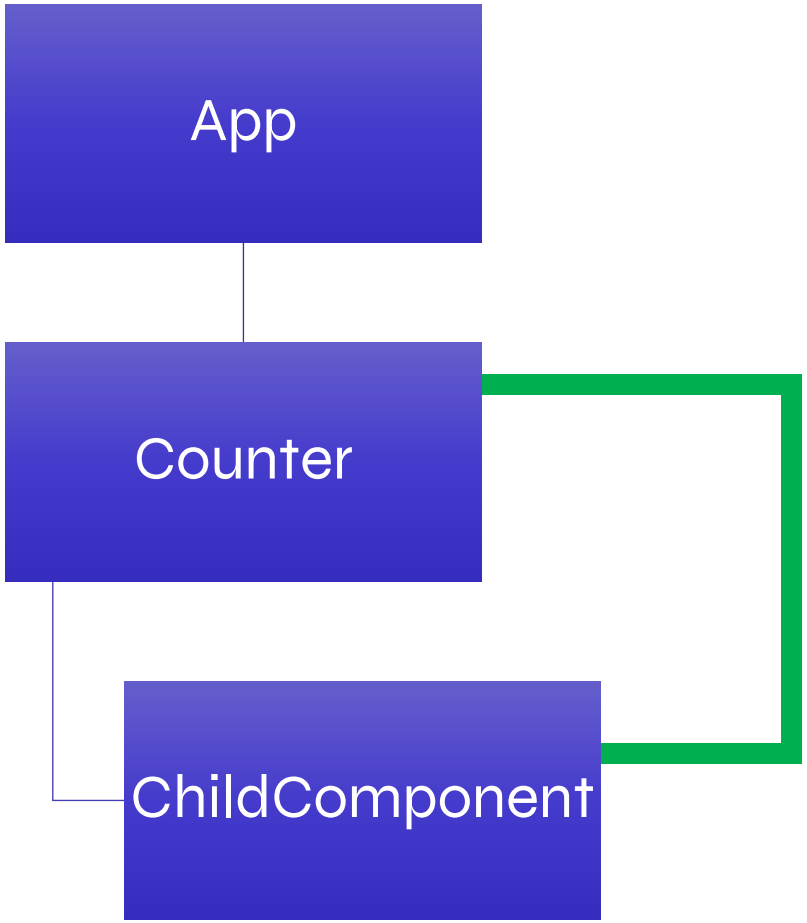
Prop Drilling in React JS

- Props drilling is a pattern in React where you pass data from a parent component to deeply nested child components through multiple layers of components, even if some of the intermediate components don't need the data.
- As your component tree deepens, prop drilling can make the code more complex and harder to maintain.

Problem with Prop Drilling

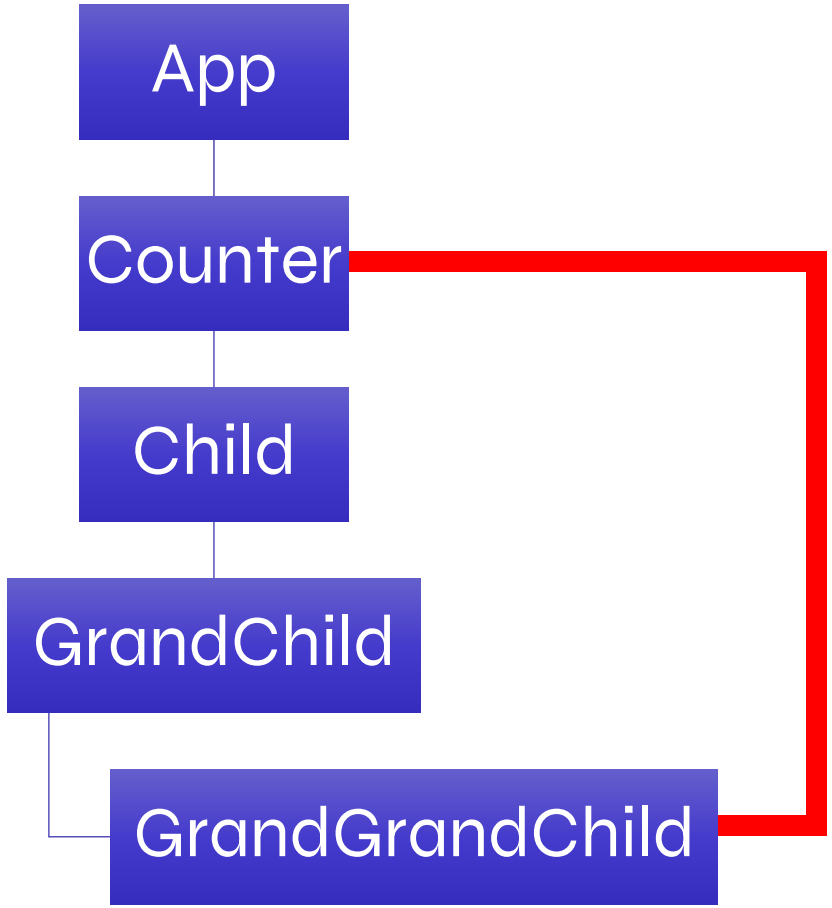
This is how our component tree looks. ChildComponent is taking props from Counter.

This is fine here.



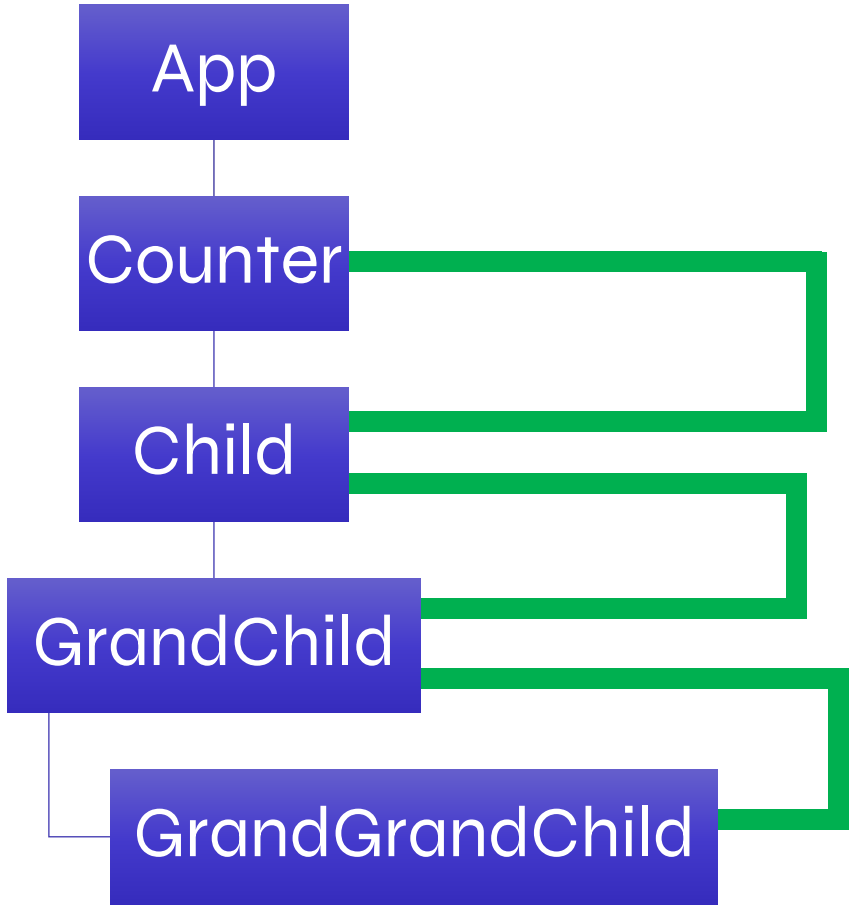
Problem with Prop Drilling

Now, we need counter value in GrandGrandChild, just passing from Counter to GrandGrandChild is not possible.



Problem with Prop Drilling

To fix this, we need to pass from Counter > Child > GrandChild > GrandGrandChild. Here, Child and GrandChild doesn't need counter state, but we still need to pass to it just so that GrandGrandChild can access it.



Concluding Prop Drilling

- Passing upto 1 or 2 levels is probably fine but might be harder to maintain more than that.
- When a prop needs to be passed through many levels, making changes to the component hierarchy or adding/removing props can become cumbersome.
- Prop drilling may result in boilerplate code as each intermediate component needs to accept and pass down the props even if it doesn't use them.
- We will learn about fixing it in Next Video using ContextAPI.

ContextAPI

Context API: A way to pass data through the component tree without having to pass props down manually at every level.

- createContext: Creates a Context object.
- Provider: A component that provides the context value to its children.
- useContext (Consumer): A hook that allows you to consume a context.

ContextAPI

Initial Value: We don't pass an initial value directly to the context.

Context Creation: createContext returns a Context component, not a variable. The first letter of the Context component's name must be uppercase.

Provider Component: The Provider is a property of the Context component. We pass the value to the Provider, which makes it accessible to child components. The value should be passed inside double curly braces {{ }} if it’s more then one.

Consuming Context Data: To access the context data, we use the useContext hook. As a parameter, we pass the entire context to useContext to access all values provided by the Provider.

ContextAPI - Notes

In the Context API, the data provided by a context can only be accessed by the components that are its children within the component tree. This means that any component that needs access to the context data must be a descendant of the provider component that supplies the context value.

```
export function App() {
  return (
    <>
      <AuthProvider>
        <Home />
      </AuthProvider>
      <Test />
    </>
  );
}
```

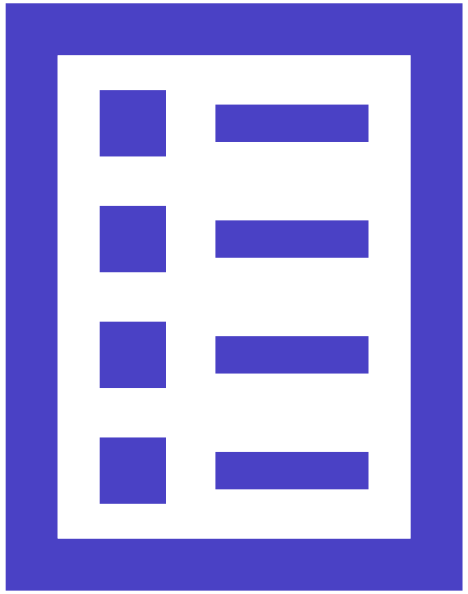
```
▶ Uncaught                                                    Text.jsx:5
TypeError: Cannot destructure property 'myName' of 'useContext(...)' as it is
undefined.
    at Test (Text.jsx:5:11)
    at react-stack-bottom-frame (react-dom_client.js?v=04140e54:16581:20)
    at renderWithHooks (react-dom_client.js?v=04140e54:4711:24)
    at updateFunctionComponent (react-dom_client.js?v=04140e54:6301:21)
    at beginWork (react-dom_client.js?v=04140e54:7358:20)
    at runWithFiberInDEV (react-dom_client.js?v=04140e54:1009:18)
    at performUnitOfWork (react-dom_client.js?v=04140e54:11208:98)
    at workLoopSync (react-dom_client.js?v=04140e54:11090:11)
    at renderRootSync (react-dom_client.js?v=04140e54:11067:15)
    at recoverFromConcurrentError (react-dom_client.js?v=04140e54:10806:27)
```

Rules for Custom Hooks

Creating custom hooks in React is a powerful way to encapsulate logic and make your components cleaner and more maintainable.

1. Prefix with use: Custom hooks must start with the word use. This convention ensures that hooks are easily identifiable and adhere to the hook rules.
2. Use Built-in Hooks: Custom hooks should utilize React's built-in hooks (e.g., useState, useEffect, useContext) to leverage React's state and lifecycle features.
3. Avoid Side Effects Outside Hooks: Side effects (e.g., data fetching, subscriptions) should be managed within hooks using useEffect or other appropriate hooks.
4. Keep Hooks Pure: Hooks should be free from side effects and return values or functions that the component can use.

Project Time: Light Dark Mode Website



Problems with our initial version

- We aren't splitting our code into components to make it more manageable.
- We aren't following proper folder structure.
- Since, we are using todo value as keys, we will get error if we try to add same todo.
- Even when using the index as a key, deleting items with the same value can result in issues.
- We need to prevent the insertion of duplicate todos to ensure the integrity of our data.
- Storing our data in LocalStorage will ensure that todos persist even after a page refresh.

Keys in React.js

- A string or a number that uniquely identifies it among other items in that array.
- Keys tell React which array item each component corresponds to, so that it can match them up later. This becomes important if your array items can move (e.g. due to sorting), get inserted, or get deleted.
- Rules
    - Keys must be unique among siblings
    - Keys must not change

When you don’t use a key attribute, React.js will automatically use index.

```
const array = [“Thapa”, “Technical” ]


return (
  <ul>
    {array.map((a) =>
      <li>{a}</li>
    )}
  </ul>
)
```

This code will lead to a structure something like this.

```
const array = [“Thapa”, “Technical” ]

return (
  <ul>
    {array.map((a) =>
      <li>{a}</li>
    )}
  </ul>
)
```

```
return (
  <ul>
    <li key={0}>Thapa</li>
    <li key={1}>Technical</li>
  </ul>
)
```

Imagine we pushed “YouTube” at the end, then react will generate new tree of structure.

```
return (
  <ul>
    <li key={0}>Thapa</li>
    <li key={1}>Technical</li>
  </ul>
)
```

```
return (
  <ul>
    <li key={0}>Thapa</li>
    <li key={1}>Technical</li>
    <li key={2}>YouTube</li>
  </ul>
)
```

Since, we are using indexes, new item will get key value as 2.

```
return (
  <ul>
    <li key={0}>Thapa</li>
    <li key={1}>Technical</li>
  </ul>
)
```

```
return (
  <ul>
    <li key={0}>Thapa</li>
    <li key={1}>Technical</li>
    <li key={2}>YouTube</li>
  </ul>
)
```

```
return (
  <ul>
    <li key={0}>Thapa</li>
    <li key={1}>Technical</li>
  </ul>
)
```

```
return (
  <ul>
    <li key={0}>Thapa</li>
    <li key={1}>Technical</li>
    <li key={2}>YouTube</li>
  </ul>
)
```

Now, React.js will perform reconciliation to differentiate which item is added and which needs to be changed.

React.js sees that text node of item with key 0 and 1 is unchanged.

But we have a new <li> node at the end based on the fact that new key is added.

```
return (
  <ul>
    <li key={0}>Thapa</li>
    <li key={1}>Technical</li>
  </ul>
)
```

```
return (
  <ul>
    <li key={0}>Thapa</li>
    <li key={1}>Technical</li>
    <li key={2}>YouTube</li>
  </ul>
)
```

Then React.js will inform ReactDOM add a new <li> at the end of <ul> children.

Sounds fine? Let's see a scenario when we add item to the beginning of our array.

```
return (
  <ul>
    <li key={0}>Thapa</li>
    <li key={1}>Technical</li>
  </ul>
)
```

```
return (
  <ul>
    <li key={0}>Thapa</li>
    <li key={1}>Technical</li>
    <li key={2}>YouTube</li>
  </ul>
)
```

If you add “Subscribe” to the beginning of our array, then new tree is generated by React.js.
Now, React.js starts reconciliation process to find out what’s changed.

```
return (
  <ul>
    <li key={0}>Thapa</li>
    <li key={1}>Technical</li>
  </ul>
)
```

```
return (
  <ul>
    <li key={0}>Subscribe</li>
    <li key={1}>Thapa</li>
    <li key={2}>Technical</li>
  </ul>
)
```

Since we are using index as keys, keys of all items will change.

React.js will compare items based on keys and it sees that item with key (0) in previous tree has different text node than key (0) in new tree.

```
return (
  <ul>
    <li key={0}>Thapa</li>
    <li key={1}>Technical</li>
  </ul>
)
```

```
return (
  <ul>
    <li key={0}>Subscribe</li>
    <li key={1}>Thapa</li>
    <li key={2}>Technical</li>
  </ul>
)
```

It then also realizes that element with key (1) in old tree and element with key (1) in new tree has different text node.

```
return (
  <ul>
    <li key={0}>Thapa</li>
    <li key={1}>Technical</li>
  </ul>
)
```

```
return (
  <ul>
    <li key={0}>Subscribe</li>
    <li key={1}>Thapa</li>
    <li key={2}>Technical</li>
  </ul>
)
```

React.js then realizes that item with key (2) is new.

At the end, React.js will update text node of all <li> at the beginning and it will append <li> at the end.

But shouldn't <li> gets added at the top?

```
return (
  <ul>
    <li key={0}>Thapa</li>
    <li key={1}>Technical</li>
  </ul>
)
```

```
return (
  <ul>
    <li key={0}>Subscribe</li>
    <li key={1}>Thapa</li>
    <li key={2}>Technical</li>
  </ul>
)
```

React.js is updating text node of all <li> elements when it shouldn't have. This is happening because of indexes being used as key.

That's why, your key should never ever change for an element. This can cause lots of problems.

```
return (
  <ul>
    <li key={0}>Thapa</li>
    <li key={1}>Technical</li>
  </ul>
)
```

```
return (
  <ul>
    <li key={0}>Subscribe</li>
    <li key={1}>Thapa</li>
    <li key={2}>Technical</li>
  </ul>
)
```

Problems with using index as keys

- If you add items at the beginning or inside your lists, then React.js will unnecessarily update text node of all elements.
- React.js won't be able to find out where an element should be added while doing sorting, adding, removing, etc.
- If you have an <input /> tag inside your lists then you will get weird bug while adding or removing items.
- Do not generate keys on the fly, e.g. with key={Math.random()}. This will cause keys to never match up between renders, leading to all your components and DOM being recreated every time. Not only is this slow, but it will also lose any user input inside the list items. Instead, use a stable ID based on the data.

Styling React.js Components

Vanilla CSS

- You can create files named as _____.css anywhere in your project and import like this:
  - import “./styles.css”
- It’s recommended to separate CSS files for each components to make it more manageable.
- Use className attribute to apply class names to your JSX element. Instead of class, you have to use className attribute.
- For global styles, you can import the CSS file in the index.js or App.js file to apply styles globally across the application.

Conditional Styling in React

Conditional Styling

- You can use ternary operators to add conditional stylings to your CSS.
- `<li className={`todoItem ${todo.completed ? “completed” : “” }`}>`
- You can also separate it outside or use if statements in case of complex conditions.
- You can use states to determine the styles of an element conditionally.

Utility Helpers; classnames, clsx

Classnames

A simple JavaScript utility for conditionally joining classNames together.

NPM V2.5.1 LICENSE MIT SPONSORED BY THINKMILL

Install from the npm registry with your package manager:

```
npm install classnames
```

Use with Node.js, Browserify, or webpack:

```
const classNames = require('classnames');
classNames('foo', 'bar'); // => 'foo bar'
```

Alternatively, you can simply include `index.js` on your page with a standalone `<script>` tag and it will export a global `classNames` method, or define the module if you are using

Install

```
> npm i classnames
```

Repository
github.com/JedWatson/classnames

Homepage
github.com/JedWatson/classnames#rea...

Weekly Downloads
13,388,141

Version
2.5.1

License
MIT

Unpacked Size
23.6 kB

Total Files
10

clsx

2.1.0 • Public • Published a month ago

Readme | Code (Beta) | 0 Dependencies | 7,345 Dependents | 14 Versions

clsx

CI passing coverage 100%

A tiny (239B) utility for constructing `className` strings conditionally.
Also serves as a faster & smaller drop-in replacement for the `classnames` module.

This module is available in three formats:

- ES Module: `dist/clsx.mjs`
- CommonJS: `dist/clsx.js`
- UMD: `dist/clsx.min.js`

Install

```
$ npm install --save clsx
```

Install

```
> npm i clsx
```

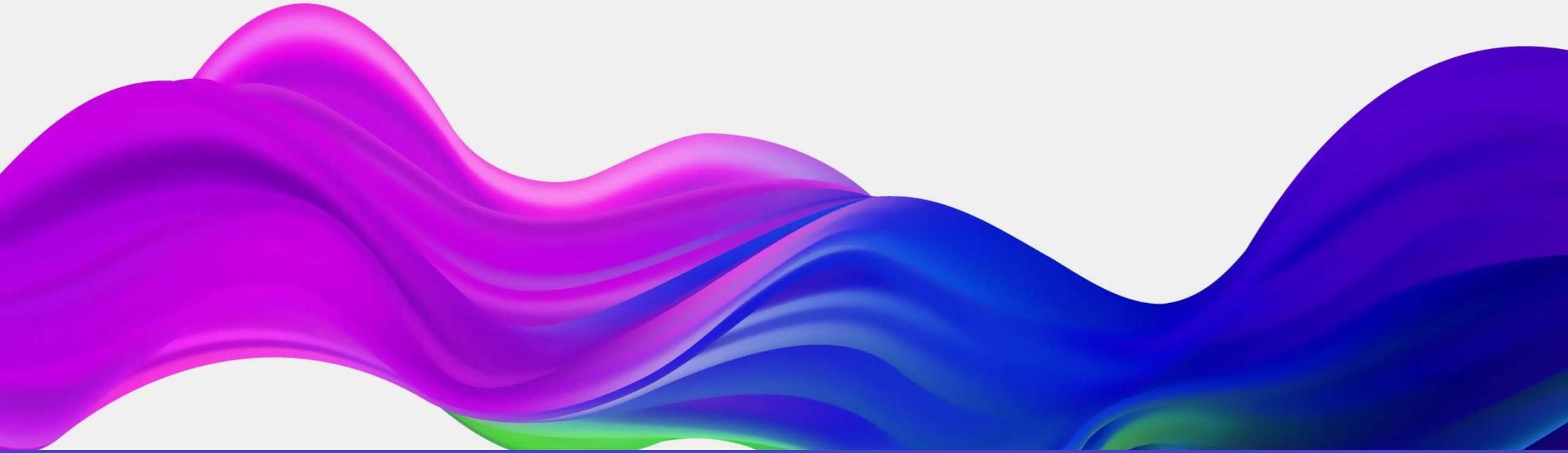
Repository
github.com/lukeed/clsx

Homepage
github.com/lukeed/clsx#readme

Weekly Downloads
15,992,716

Version
2.1.0

License
MIT



Implementing our own utility helper

CSS Modules in React

CSS Modules

- Every time you import a CSS file in your react component, they are regarded as global CSS.
- CSS Modules help you keep styles specific to the component they're used in.
- Each module encapsulates its styles, preventing unintended style conflicts with other modules.
- The class names in CSS modules are often automatically generated, reducing the likelihood of naming clashes.

CSS Modules – Rules to Keep in Mind

- Name your CSS module files with the .module.css or .module.scss extension.
- Import the CSS module file in your React component. Access class names as properties of the imported styles object.
- Access class names as properties of the imported styles object.
- Combine multiple class names using template literals or the classnames library.
- Dynamically apply class names based on component state or props.

CSS Modules - Important

- When using CSS Modules, class names with hyphens can sometimes cause issues if not accessed correctly. In CSS Modules, you need to use bracket notation for class names with hyphens.

- <div className={styles['card-content']}>
- Explanation:
- Using Bracket Notation: In JavaScript, when accessing object properties with special characters (like hyphens), you need to use bracket notation instead of dot notation.
- Dot Notation: styles.card-content will not work because . expects a valid JavaScript identifier, which cannot contain hyphens.
- Bracket Notation: styles['card-content'] correctly accesses the class name from the imported styles object.

CSS in JS

- CSS-in-JS is one of the approach in React.js to style your web page by including your CSS next to your React.js components.
- This helps to maintain scoped styles like CSS modules.
- All HTML, CSS and JS code are combined into single React.js components.
- Easier to apply conditional stylings.
- Examples: styled-components, emotion, jss, etc.
- Meta has also recently introduced their own version of CSS in JS called stylex. You can use this vite plugin for that: HorusGoul/vite-plugin-stylex: Vite Plugin for StyleX (github.com)

Inline Styles

- You can also use inline styles similar to without React.js
- Every element accepts a prop called style.
- `style={{ backgroundColor: “red” }}`
- Unlike normal inline styles, you pass an object to inline styles in JSX.
- All CSS properties are in camel case.
- All CSS values needs to be a string.
- It’s not recommended to use it as it’s not manageable.

UI Libraries

- There are lots of UI libraries in React.js which can be used to make your work easier.
- Examples:
  - Material UI
  - Tailwind
  - Bootstrap
  - Chakra UI
  - Mantine
  - Shadcn UI

Styled Component in React JS

Using Template literals


```
const Button = styled.button`
  color: grey;
`;
```

1: This is a styled component, which is a React component created using styled-components. It is both a React component and a styled component. As a React component, it can be used in JSX. As a styled component, it has styles directly attached to it.

2: styled is an object provided by the library, and button is a method on that object.

Using Style objects


```
const Button = styled.button({
  color: grey,
});
```

1: This is a styled component, which is a React component created using styled-components. It is both a React component and a styled component. As a React component, it can be used in JSX. As a styled component, it has styles directly attached to it.

2: styled is an object provided by the library, and button is a method on that object.

Icons

- There are lots of libraries for adding icons in react.js projects.
- Examples:
  - react-icons
  - Tabler Icons
  - Lucide Icons

Hooks in React.js

Short Circuit Evaluation in React.js

Short circuit evaluation is a technique used in JavaScript (and many other programming languages) where expressions are evaluated from left to right. In logical operations, evaluation stops as soon as the result is determined. This is often used in React to conditionally render components or execute code based on the truthiness of certain conditions.

Rules of Short Circuit Evaluation

Short circuit evaluation involves logical operators (&&, ||, ??) and how they evaluate expressions based on the truthiness of their operands.

Short Circuit Evaluation in React.js

Logical OR (||)

Syntax: expression1 || expression2

Rule: If **expression1** is truthy, return **expression1**. Otherwise, return expression2.

```
const result = false || 'Hello';  // 'Hello'
const result2 = true || 'World';  // true
```

Short Circuit Evaluation in React.js

Logical AND (&&)

Syntax: expression1 && expression2

Rule: If expression1 is falsy, return expression1. Otherwise, return expression2.

```
const result = true && 'Hello';  // 'Hello'
const result2 = false && 'World';  // false
```

Short Circuit Evaluation in React.js

Nullish Coalescing (??)

Syntax: expression1 ?? expression2

Rule: If expression1 is not null or undefined, return expression1. Otherwise, return expression2.

```
const result = null ?? 'Hello';  // 'Hello'

const result2 = undefined ?? 'World';  // 'World'

const result3 = '' ?? 'Fallback';  // ''
```

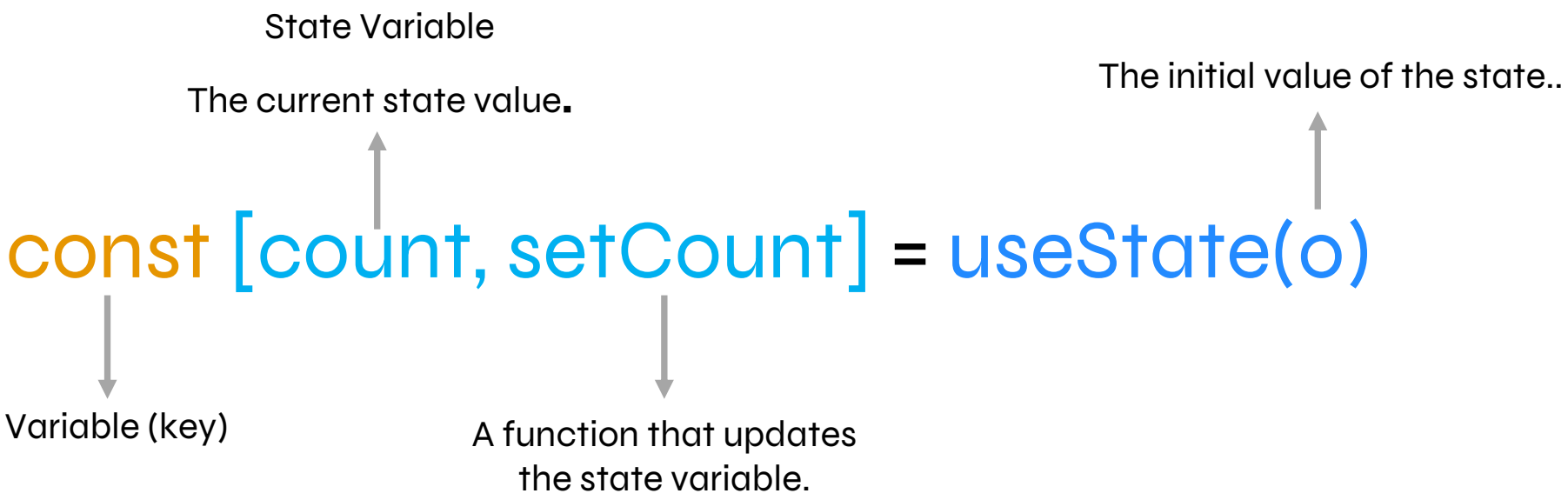
React Hooks Rules

- Hooks let you use different React features from your components.
- It starts with use_____. (useState, useEffects, useReducer)
- Hooks can only be used at the top level of your component.
- Do not call Hooks inside loops, conditions, or nested functions. ✅
- Call Hooks only from React function components or custom Hook. Do not call Hooks from regular JavaScript functions or in Class components.
- You can also create your own custom hooks by creating functions starting with use_____. This ensures that React can identify it as a Hook. (ex. useFetch, useForm)
- Ensure Hooks are Called in the Same Order. This allows React to properly preserve the state of Hooks between re-renders.

useState Hook

useState() hook

- The useState hook is a fundamental hook in React that allows you to add state management to your functional components. It lets you store and update values that change over time, enabling dynamic and interactive user interfaces.



useState() hook

- Functions passed to useState() and setState() runs twice during development mode to prevent developers from running side effects.
- Side effects in programming refer to changes or interactions that occur outside the scope of a function or block of code.
- React.js compares previous state and new state using Object.is(), if it finds both are same then it will ignore re-rendering.
- React.js batches all setStates() inside a event handlers and run them at the same time. So, if you try to access the value immediately after setting a state, then you will get old value as it's asynchronous.

Controlled and Uncontrolled Components

- A component is “controlled” when the important information in it is driven by props rather than its own local state.
- Meanwhile uncontrolled components are components in which its important information is driven by its own local state.
- In practice, “controlled” and “uncontrolled” aren’t strict technical terms.
- They can be used in multiple cases.

Controlled vs Uncontrolled Components

Controlled Components

Controlled components in React are those where form data is handled by React component state.

This means:

- State Management: The value of the input field is controlled by React state (useState or this.state in class components).
- Event Handlers: Changes to the input field (like typing into a text field) are handled by React event handlers (onChange, onBlur, etc.).
- State Updates: When the user interacts with the form input, React state is updated through event handlers, and the input value reflects the current state value.

Controlled Components

Uncontrolled components in React are those where form data is handled by the DOM itself.

This means:

- Direct DOM Manipulation: The value of the input field is controlled by the DOM (document.getElementById, etc.). It's not declarative way right.
- Event Handling: Changes are directly handled by DOM events (onchange, onblur, etc.).
- Accessing Form Data: Form data is accessed through refs or DOM traversal methods, not through React state.

Controlled Components

Pros:

- React has complete control over the input values, making it straightforward to implement features like validation and conditional formatting based on state.
- Easier to handle form submission and integration with React's lifecycle methods.
- Enables clear data flow and centralized state management within React components.

Cons:

- Requires more code compared to uncontrolled components due to state management.
- Can lead to more re-renders if not optimized, although React handles this efficiently in most cases.

useEffect Hook

useEffect Hook

The useEffect hook in React is used for handling side effects in functional components.

```
useEffect(() => {
  // Your side effect code here

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

1. Initial Render: When the component mounts, useEffect can run its effect function to perform operations like data fetching.
2. Dependencies: The second argument, an array of dependencies, which determines when the effect should re-run. If any value in this array changes, the effect will re-run.
3. Cleanup: useEffect can return a cleanup function to clean up after the effect, such as unsubscribing from an event or clearing a timer.

useEffect Hook

A side effect is any operation that affects something outside the scope of a function (Pure function). In React, side effects are managed using hooks like useEffect to ensure they are handled in a controlled and predictable manner. This includes tasks like (In next Slide)

```
let count = 0;

function increment() {
  count += 1; // Modifies an external variable
}

increment();
console.log(count); // 1
```

useEffect Hook

Fetching Data as a Side Effect

When you fetch data in a React component, you're performing a side effect because:

External Interaction: You're interacting with an external data source, such as an API or a server.

State Updates: The fetched data will usually update the component's state, causing a re-render.

Others ….

- Subscribing to or unsubscribing from a service.
- Updating the browser's DOM.
- Logging data to the console

useRef()

- useRef is a React Hook that lets you reference a value that’s not needed for rendering.
- Unlike states, it’s directly mutable.
- You can access value of it’s using yourRef.current;

Uncontrolled Components:

- Uncontrolled components manage their own state internally and are typically used with refs.

forwardRef()

React forwardRef allows parent components to move down (or “forward”) refs to their children. It gives a child component a reference to DOM entity created by its parent component in React. This helps the child to read and modify the element from any location where it is used.

It takes a function with props and ref arguments.

```
React.forwardRef(
(props, ref) =>
{})
```

useId()

useId is a React Hook for generating unique IDs that can be passed to accessibility attributes. It helps to ensure that each instance of a component gets a unique ID, which is useful for associating labels with input fields and other elements

Syntax = `const id = useId()`

Parameters: useId does not take any parameters.

Returns: useId returns a unique ID string associated with this particular useId call in this particular component.

Note: useId should not be used to generate keys in a list.

useReducer()


```
const [state, dispatch] = useReducer(reducer, initialState);
```

It returns an array containing the current state and a dispatch function.

Dispatch function is used to send actions to the reducer, which in turn updates the state based on the action's type and any associated data (payload).

Reducer Function: A function that takes the current state and an action as arguments, and returns a new state.

Initial State: The initial state value. (The initial state can be a simple value, an object, or even derived from a function if the initialization is complex.)

useReducer() hook

State Variable

The current state value.

The initial value of the state..

```
const [count, dispatch] = useReducer(reducer,o)
```

Variable (key)

To send actions to the reducer function

Reducer Function: A function that takes the current state and an action as arguments, and returns a new state.

React.memo()

- The React.memo function is used for memoization of functional components.
- If the props of a memoized component have not changed, React skips the rendering for that component, using the cached result instead.
- Do memoizations only when necessary.

OR

React.memo() is a higher-order component that we can use to wrap components that we do not want to re-render unless props within them change

useMemo Hook


```
const memoizedValue = useMemo( () => {   // Your computation logic here
  return computedValue;
}, [dependencies]);
```

- useMemo is a React Hook used for memoization.
- Memoization is a technique to optimize performance by caching the results of expensive function calls.
- Use it when you want to prevent unnecessary re-execution of a function on every render.
- Useful for optimizing performance in situations where calculations or operations are computationally expensive.
- Overusing useMemo might lead to unnecessary complexity and impact readability.

useCallback()

- useCallback is a React hook used to memoize functions, preventing unnecessary re-creation of functions on each render.
- Useful for optimizing performance in scenarios where a function is passed as a prop to child components, preventing unnecessary re-renders.
- Helps in avoiding re-renders of child components when the parent component re-renders but the function reference remains the same.

Controlled form inputs vs uncontrolled form inputs

- Controlled input
  - In a controlled input, the form data is controlled by the React component state.
  - The input field's value is bound to the state, and any changes to the input trigger a state update.
  - Typically, you use the onChange event to handle input changes and update the state accordingly.
- Uncontrolled input
  - In an uncontrolled input, the form data is not directly managed by React state.
  - The DOM handles the input's value, and React doesn't necessarily know or control it explicitly.
  - Refs are often used to interact with the DOM element directly.

useCallback vs useMemo vs memo()

useCallback:

- When passing functions as props to child components.
- Memoizes a function, preventing it from being recreated on each render. Useful to optimize performance, especially when functions are used as dependencies in child components.

memo:

- When you have pure functional components that receive the same props and produce the same result.
- Memoizes the entire component, preventing re-renders when props haven't changed. Useful for optimizing functional components that rely only on props and don't have internal state or side effects.

useMemo:

- When you want to memoize the result of an expensive computation or function.
- Memoizes the result of a function or computation and only recalculates it if the dependencies change. Useful for optimizing performance when a component's render depends on a costly operation.

Axios in React

Axios

Axios is a promise-based HTTP library that helps you easily communicate with servers or APIs over the internet. It allows your website or app to send and receive data from a server, like fetching information, submitting forms, or updating content without reloading the entire page.

Axios uses promises to handle HTTP requests and responses.

Why Choose Axios over Fetch?

- Easier syntax and cleaner code.
- Automatic JSON transformation without extra code.
- Better built-in error handling.
- Support for older browser