React js - TypeScript

Disclaimer : Original course presented in English by Mosh Hamedani on his youtube channel, this is a personal version with more notes and details.

Prerequisites :

1. Html
2. CSS
3. JavaScript

We will use typescript because it has **static typing** and it saves us time **catching errors while coding** our projects.

What is React ?

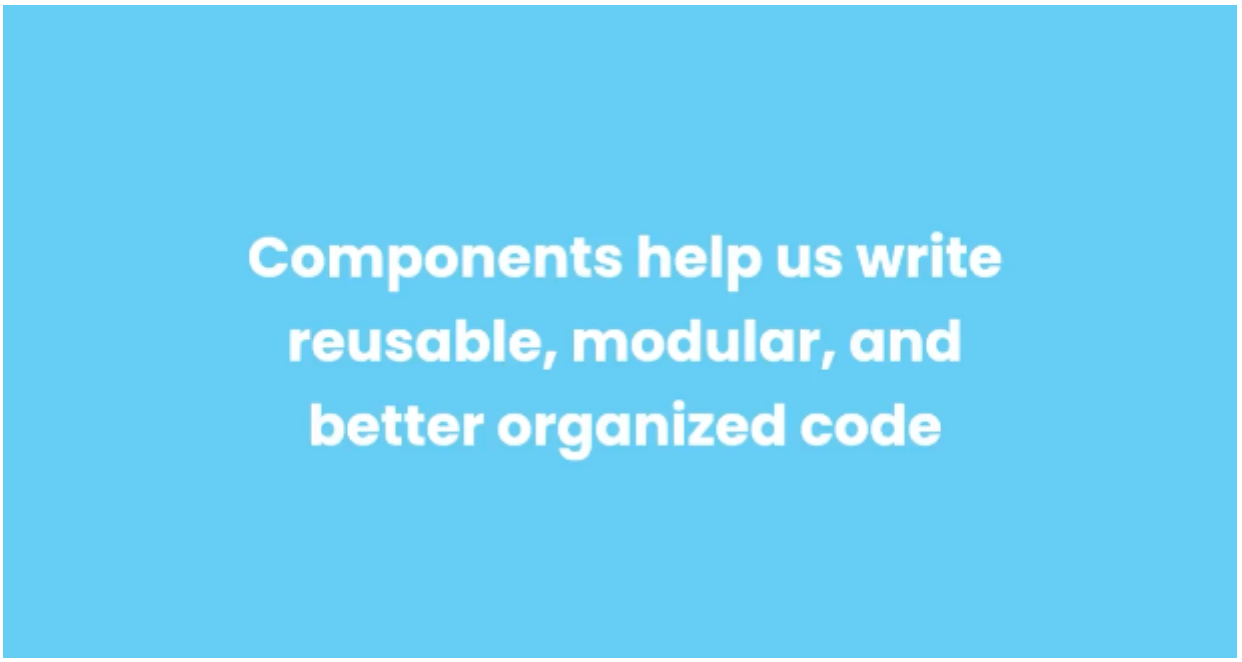
It's a javascript library to create dynamic and user interfaces created by Google, it's the most js library used at the moment.

When an html file is loaded to the browser, the browser builds what we call the DOM tree; Document Object Model, basically for using js to react to user actions on the website, like hiding a div element if a button is clicked :

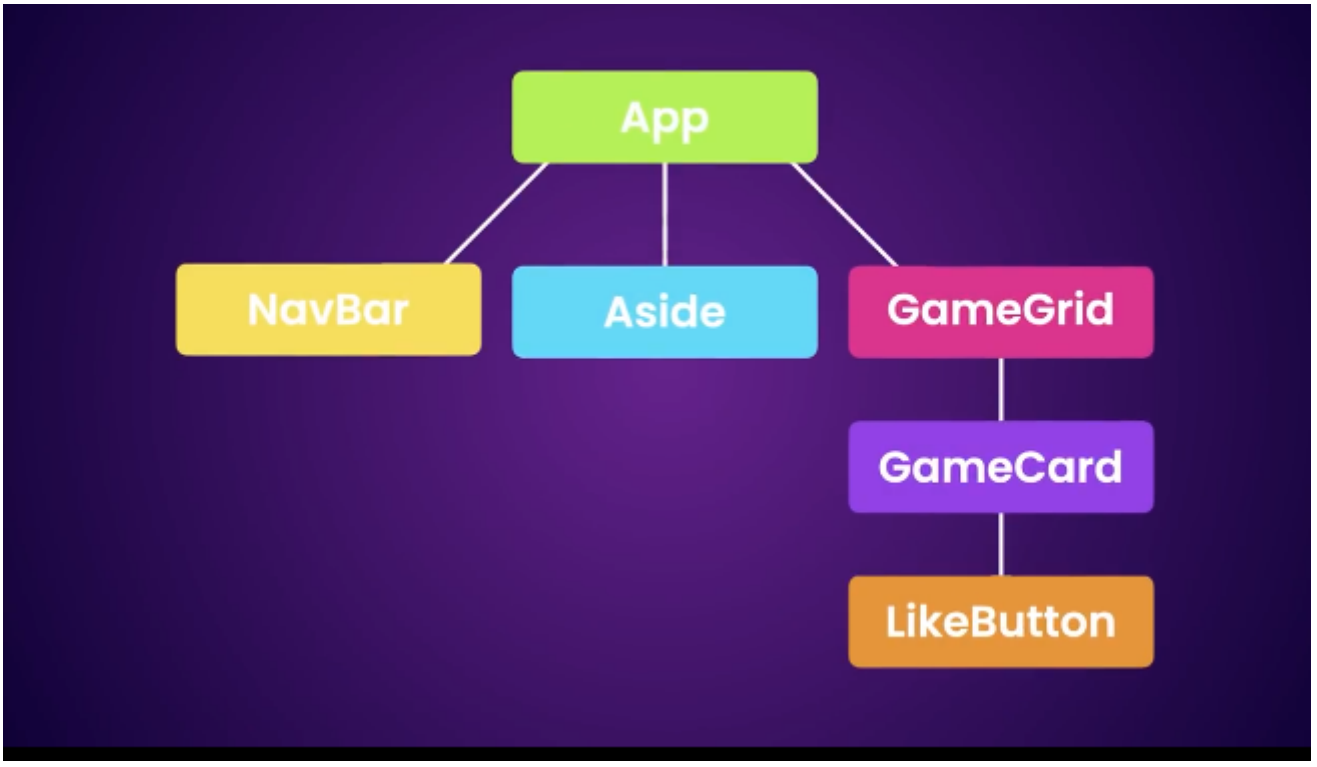
This, above, is called vanilla js (no third party tools)

But the more you build and the bigger and complex your project get, the DOM gets harder to manage, that's where React comes in!

With it, we won't be worrying about querying elements and updating them, instead, we will be using react small and reusable components, React will take care of updating the DOM.



We can build each component of a website individually and then combine them a page :



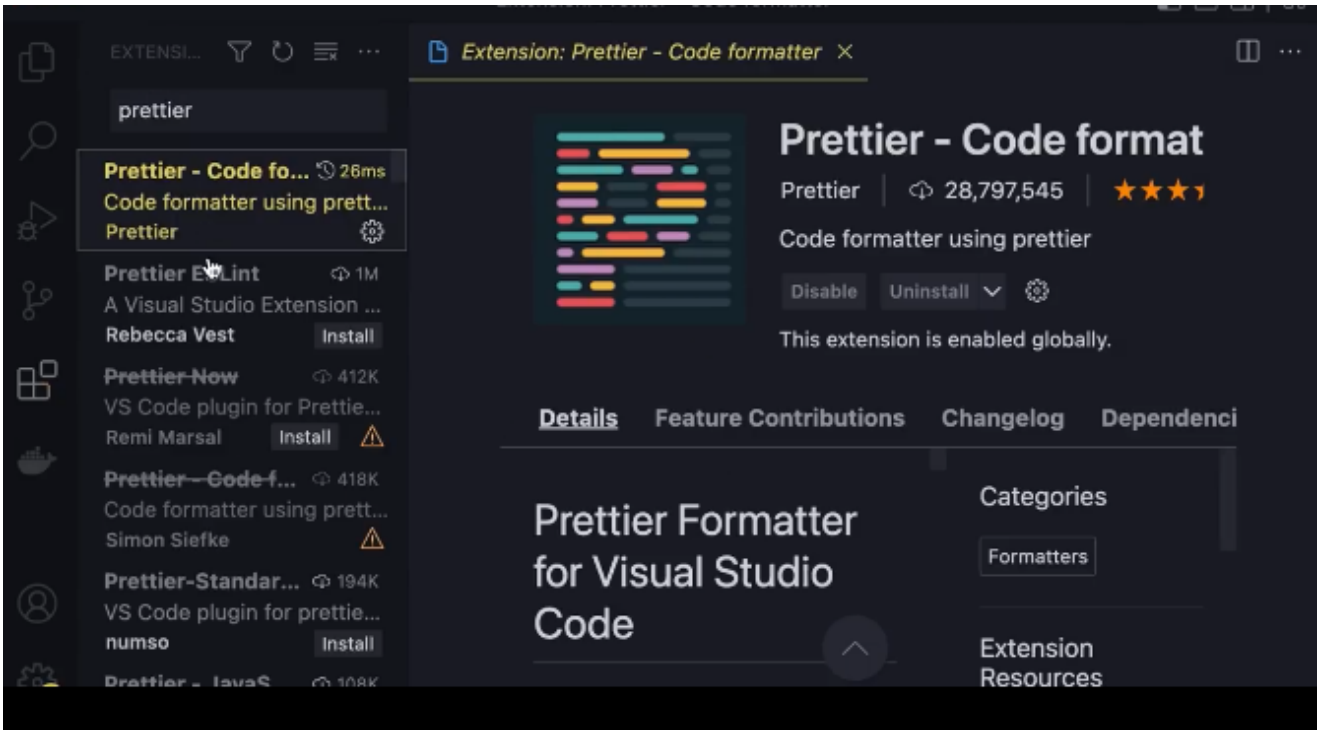
Each child of the `App` is a component, so you'll have a tree of components.

Setting up the dev environment

You need node v16 or higher :

```
node -v #to check it's version
```

We will be using vs code too, get **Prettier** extension on it :



After that go to settings and search for *format on save* to enable it, this will let this extension do each work each time you save the file, you can try it now with any source file that it supports!

Creating a React App

The are two ways we can create a React app; one is using the default CRA (create react app) provided by react team, or use vite (faster).

What's vite?

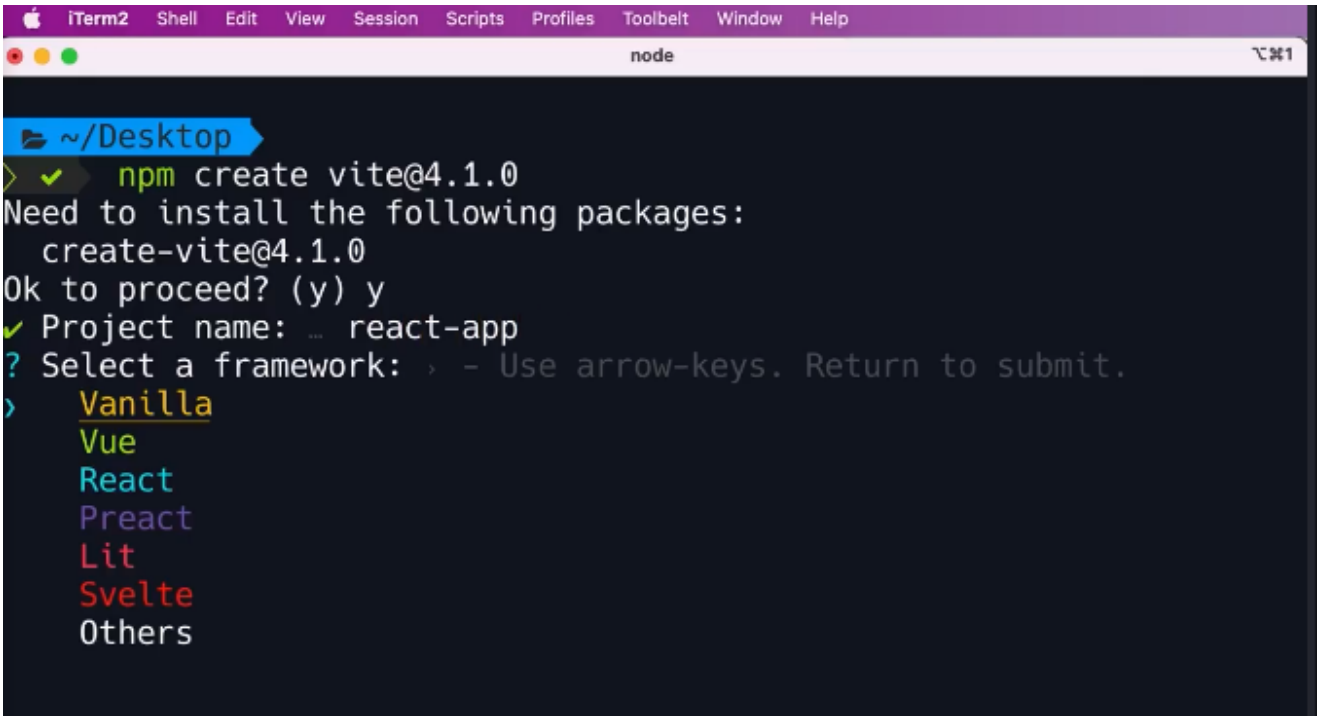
To create an app using vite :

```
npm create vite@latest #for latest version
```

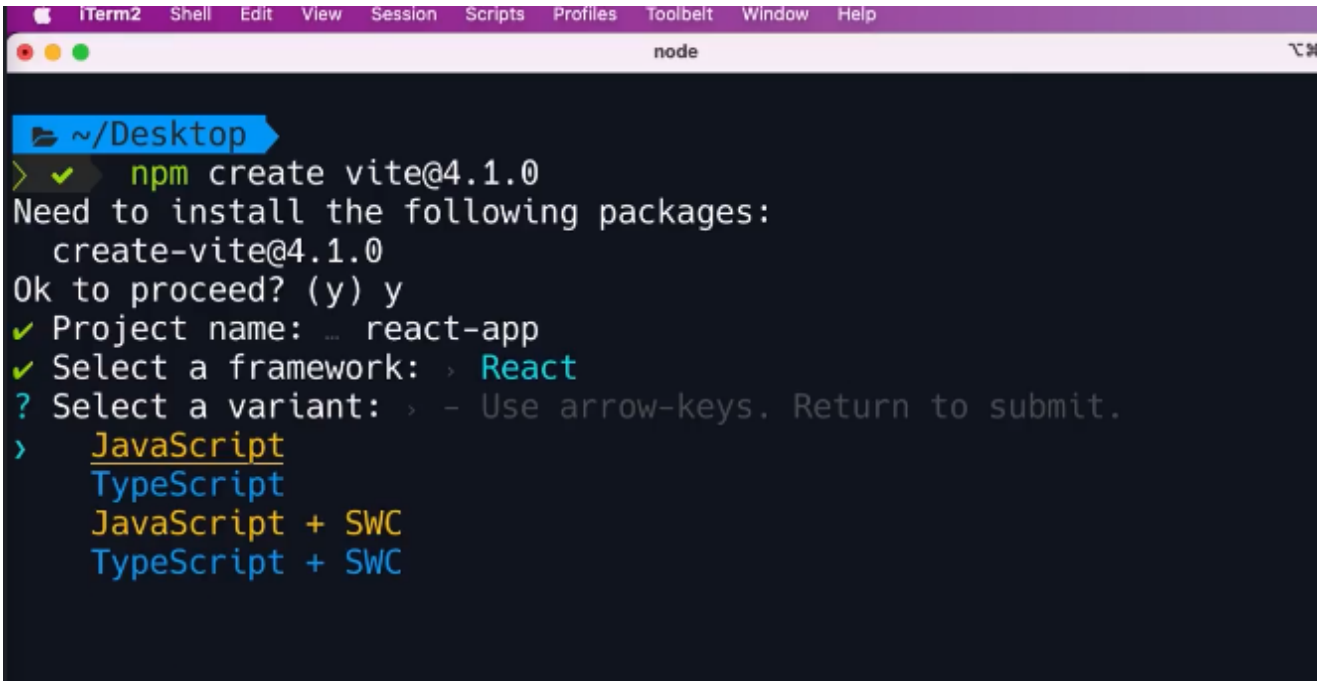
Or,

```
npm create vite@4.1.0 #to be consistent with the course
```

Along the prep, you'll have a select a framework :



See, you can use vite to create any javascript app, after that you'll need to select a language (js or ts) :

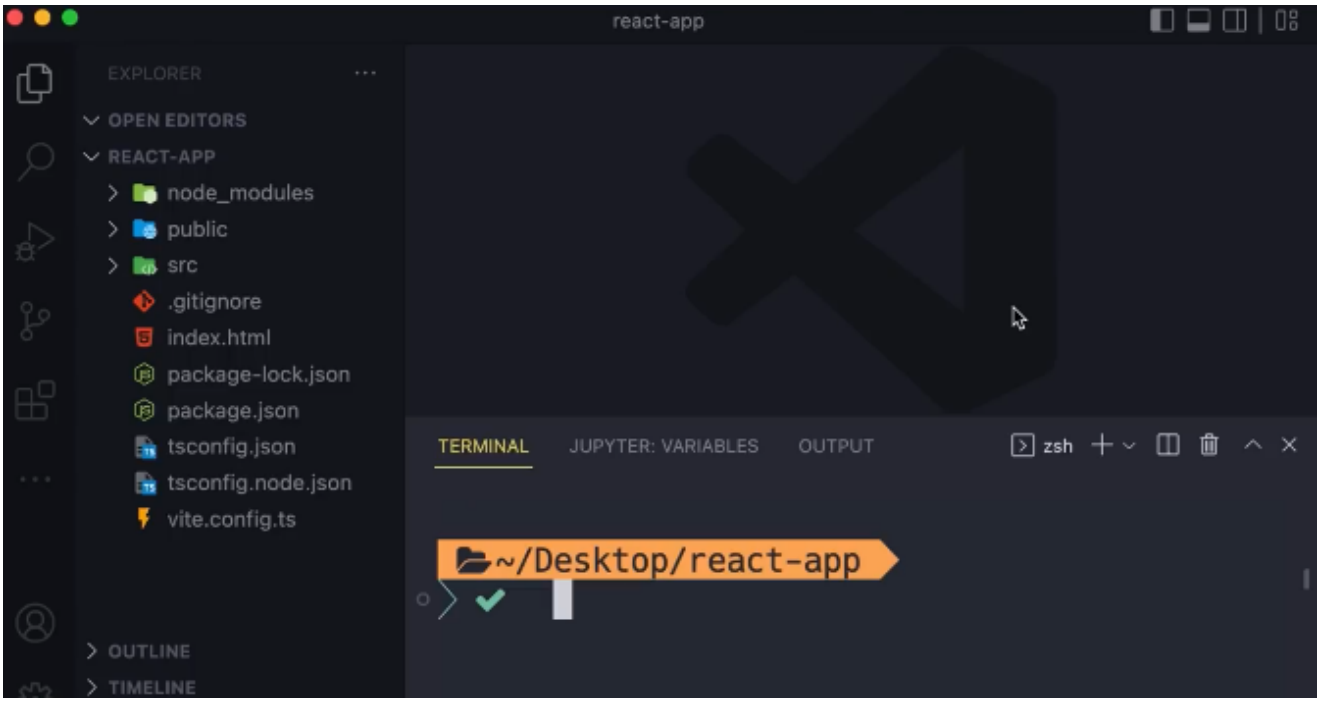


Now, run these commands :

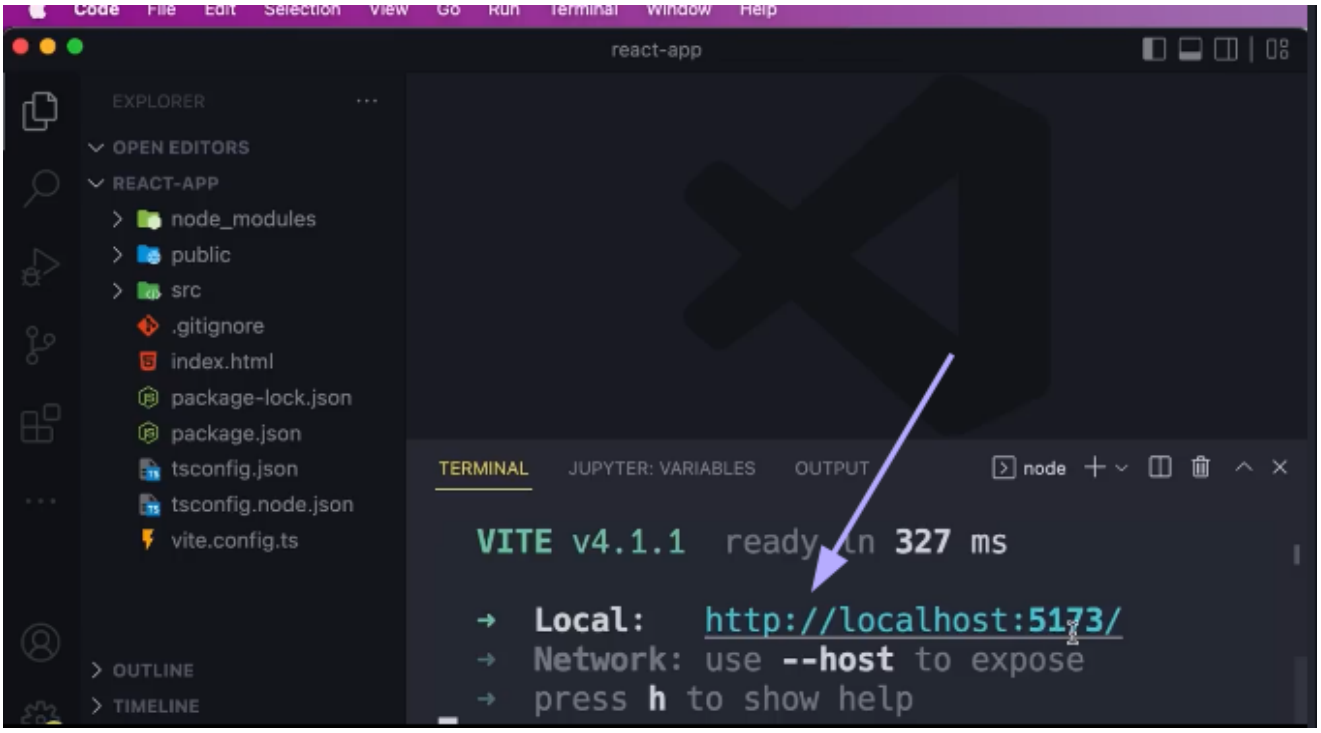
This is for installing the dependencies and running the web server.

what does that mean?

When open the app folder on your vs code editor it'll look like this :



Run the last command in the vs code terminal and get this :



And open it in your browser, welcome to the default react app using vite.

Project Structure

1. **node_modules** : all the third party libraries are installed here, never have to touch it.
2. **public** : public assets for our website like images and other media.

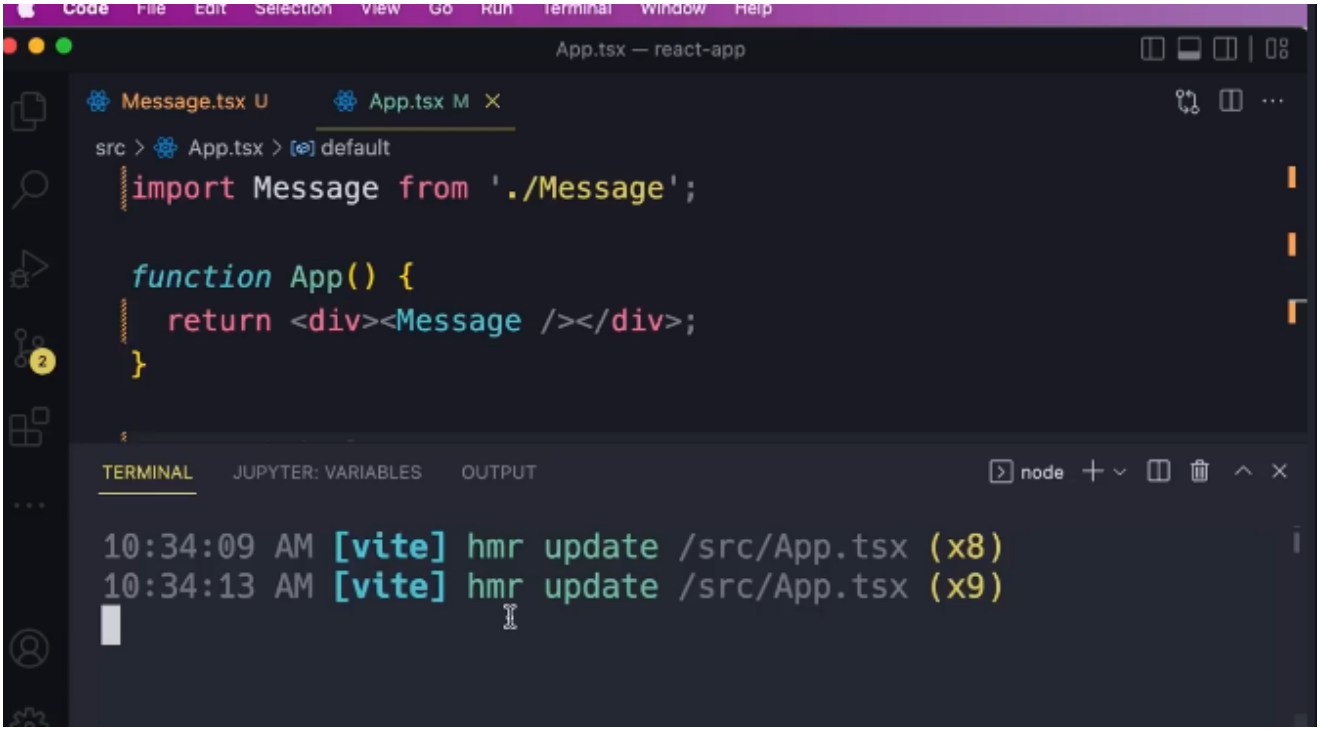
3. **src** : source code for our app, currently and by default, you'll have the app component `App.tsx`
4. `index.html` : a simple html file with a template for the page you saw on your browser if you opened the link provided to you.
5. `package.json` : information about the app, you can get in it and explore it.
6. `tsconfig.json` : settings on how to ts compiler should compile our code to javascript, generally, you won't need to touch this unless you wanna make advanced changes.
7. `vite.config.ts` : vite configuration file.

Creating a React component

Create a `.tsx` or `.ts` file inside the src folder.

Now, there are two ways to create a react component; js *class* or *function*, function based component are more popular, because they're more concise and easier to write.

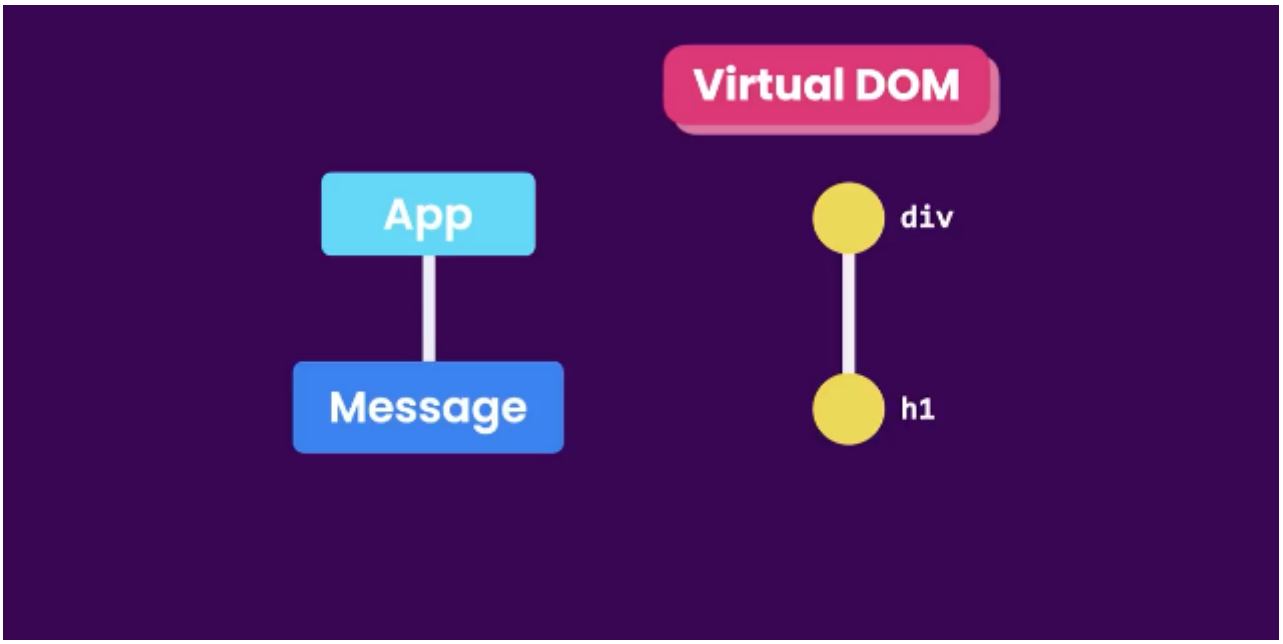
While coding, vite monitor our real time changes and does an **HMR** (hot module replacement on the web server of course) for auto refreshing :



Of course, as should we have on a UI, components have behaviors, and that's easily created using JSX.

How React works ?

In our project we have component tree, when the app runs, react takes it and build a js data structure called the **virtual DOM** :



what's a virtual DOM ?

The **Virtual DOM (VDOM)** in React is a lightweight copy of the actual **DOM (Document Object Model)** that helps optimize rendering performance. Instead of directly updating the real DOM, React first updates this virtual representation, determines the differences (diffing), and then efficiently updates only the changed parts of the real DOM.

How It Works:

1. **Render:** React creates a virtual DOM representation of the UI.
2. **Diffing:** When a state or prop changes, React creates a new virtual DOM and compares it with the previous one.
3. **Reconciliation:** React calculates the minimal set of changes needed and updates only the affected parts of the real DOM.
4. **Efficient Updates:** This avoids unnecessary re-rendering and boosts performance.

what is state management ?

What is State in React?

In React, **state** is like a storage box where you keep data that can change over time. For example, if you're building a to-do list app, you would keep the list of to-dos in the state. If someone adds a new to-do, that list changes, so the state needs to update.

Where is State Used?

Each component in React can have its own state. Components are like pieces of your app (for example, a button, a form, or a list). The state keeps track of things that change inside that component.

Example of Using State in a Component

Let’s look at a simple example:

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

function Counter() {
  // Declare a state variable called 'count' with an initial value of 0
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      {/* When the button is clicked, increase the count */}
      <button onClick={() => setCount(count + 1)}>Increase Count</button>
    </div>
  );
}

export default Counter;
```

How it works:

1. **useState(0):** This hook is used to create a state variable (`count`) with an initial value of `0`. It also gives you a way to change that value (with `setCount`).
2. **setCount:** This is a special function React gives you to update the state. In the example, when the button is clicked, `setCount` is called to increase the count by 1.
3. **Rendering the State:** Inside the `return` part, `{count}` displays the current state value in the paragraph tag. Every time you update the state, the component re-renders to reflect the new state.

What Happens When the State Changes?

Whenever the state changes (e.g., when you click the button), React will automatically re-render the component to show the updated state. You don't have to manually update the screen. React handles it for you!

Passing State Between Components

Sometimes, you want to pass state from one component to another. For example, the parent component may hold the state, and the child component can use it.

Here's a simple example:

```jsx
import React, { useState } from 'react';

function Parent() {
  const [message, setMessage] = useState("Hello from Parent!");

  return (
    <div>
      {/* Passing the state as a prop to the Child component */}
      <Child message={message} />
    </div>
  );
}

function Child({ message }) {
  return <h1>{message}</h1>;
}

export default Parent;
```

How it works:

1. **Parent Component:** It has its own state `message` and passes it as a **prop** to the `Child` component.
2. **Child Component:** It receives the `message` prop and displays it inside an `<h1>` tag.

Why Props and State are Important?

- **State** is for data that can change within a component (like a button click or input field).
- **Props** are for passing data from one component to another, usually from a parent to a child.

When Do You Need More Complex State Management?

When your app gets bigger, managing state across multiple components can become tricky. For simple cases like the ones we've seen, using `useState` and props is enough. But when your app becomes large, you might need tools to manage global state that needs to be shared across many components. These tools can make things easier, but for beginners, you can start with local state and props.

Recap:

- **State**: Keeps track of data that can change in a component (like a counter).

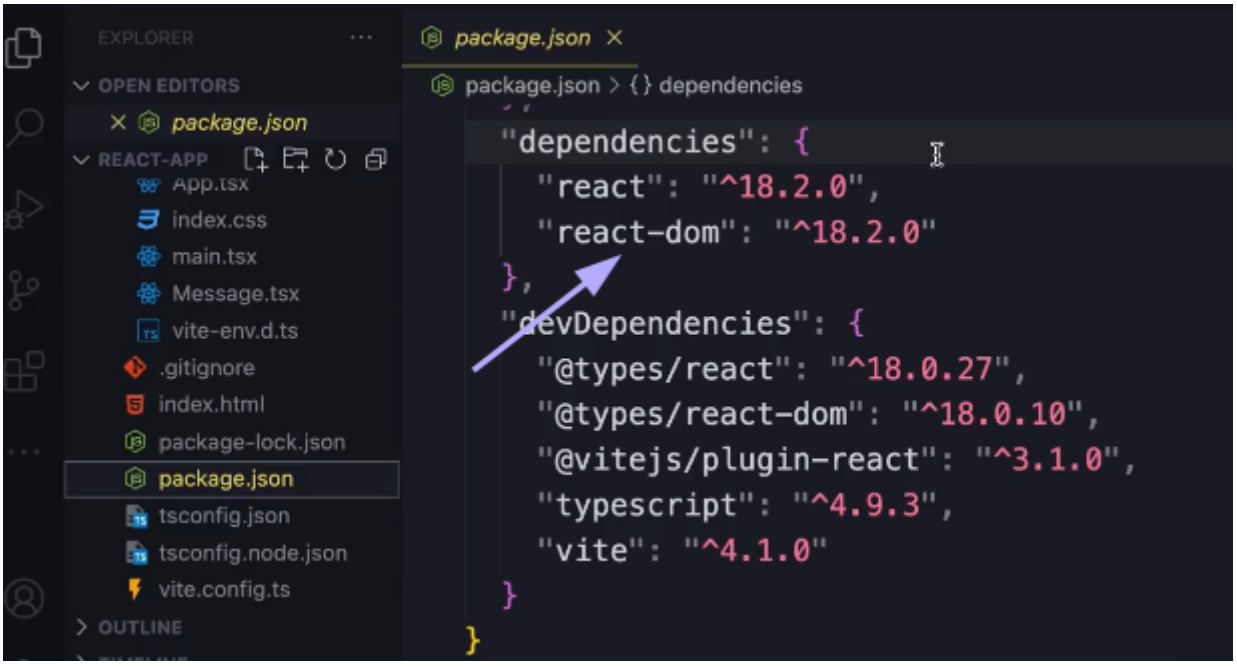
- **Props**: Used to pass data from one component to another.
- `useState` **Hook**: A way to add state to a functional component.
- **Re-rendering**: React automatically updates the screen when the state changes.

By using state and props, React lets you build interactive UIs where components can update and pass information to each other in a simple way!

Why Use Virtual DOM?

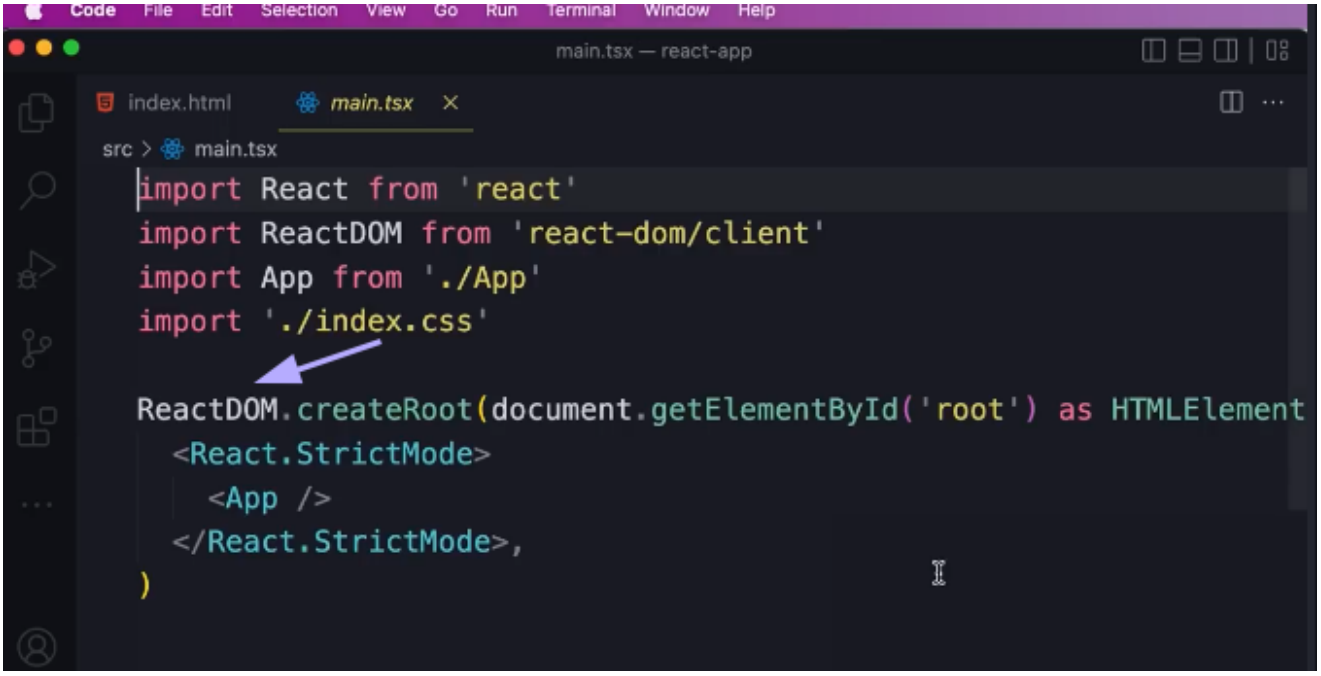
- **Performance Optimization:** Updating the real DOM directly is slow; VDOM minimizes reflows and repaints.
- **Efficient UI Updates:** React ensures only necessary components re-render.
- **Better Developer Experience:** React manages updates efficiently without requiring manual DOM manipulation.

Updating the DOM is done through a companion library called react-dom:



So our app is *dependent* on those two libraries. To understand how they work together, take a look at the `index.html` file in your project :

the `root` div is where our app will be rendered, and if you take a look inside the src or the script in the body :



You'll see that ReactDOM is used to render(display) the component tree into that `root` div element.

→ the `StrictMode` react component is a built in component in react.

what's its purpose ?

The `StrictMode` component in React is a **developer tool** that helps identify potential problems in your React application. It does not affect the UI but enables additional checks and warnings in development mode.

Key Roles of `StrictMode`

1. **Identifies Unsafe Lifecycles**
   - Warns about deprecated lifecycle methods in class components (e.g., `componentWillMount`).
2. **Highlights Side Effects in `useEffect`**
   - React **intentionally runs effects twice** in development mode to help catch unintended side effects.
3. **Detects Legacy String Refs**
   - Encourages the use of `useRef` instead of legacy string refs.
4. **Warns About Unsafe Legacy API Usage**
   - Identifies usage of outdated APIs like `findDOMNode`.
5. **Checks for Unexpected Side Effects**
   - Helps detect issues that might arise due to impure components.

How to Use `StrictMode`

Wrap your app (or part of it) inside `<React.StrictMode>`:

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";

const root = ReactDOM.createRoot(document.getElementById("root"));

root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

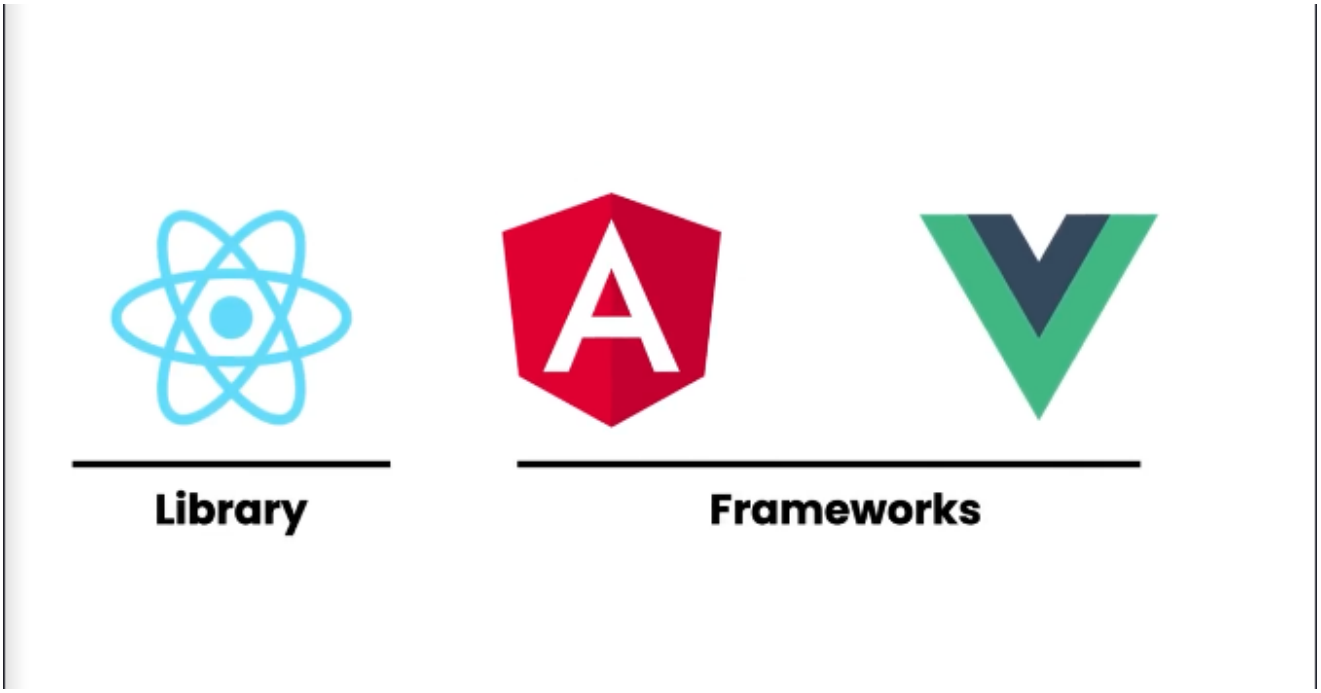
Important Notes

- It **only runs in development mode**—it has no effect in production.
- It helps in **early detection of issues** before they cause bigger problems.

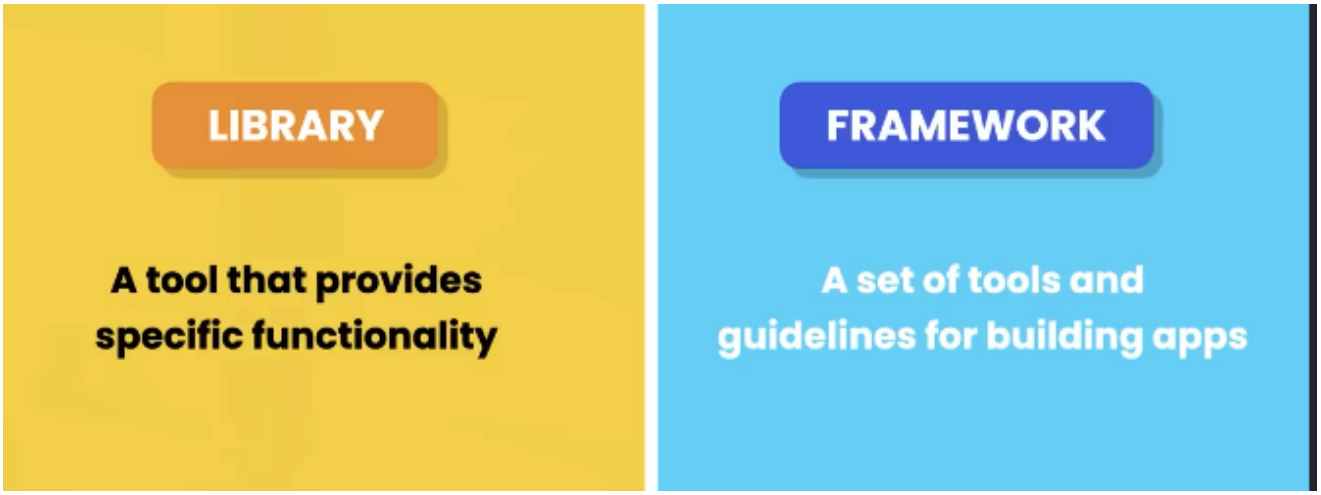
→ We can also render that tree using reactNative, basically because react is not platform dependent.

React Ecosystem

In contrast to react, we have these guys here :

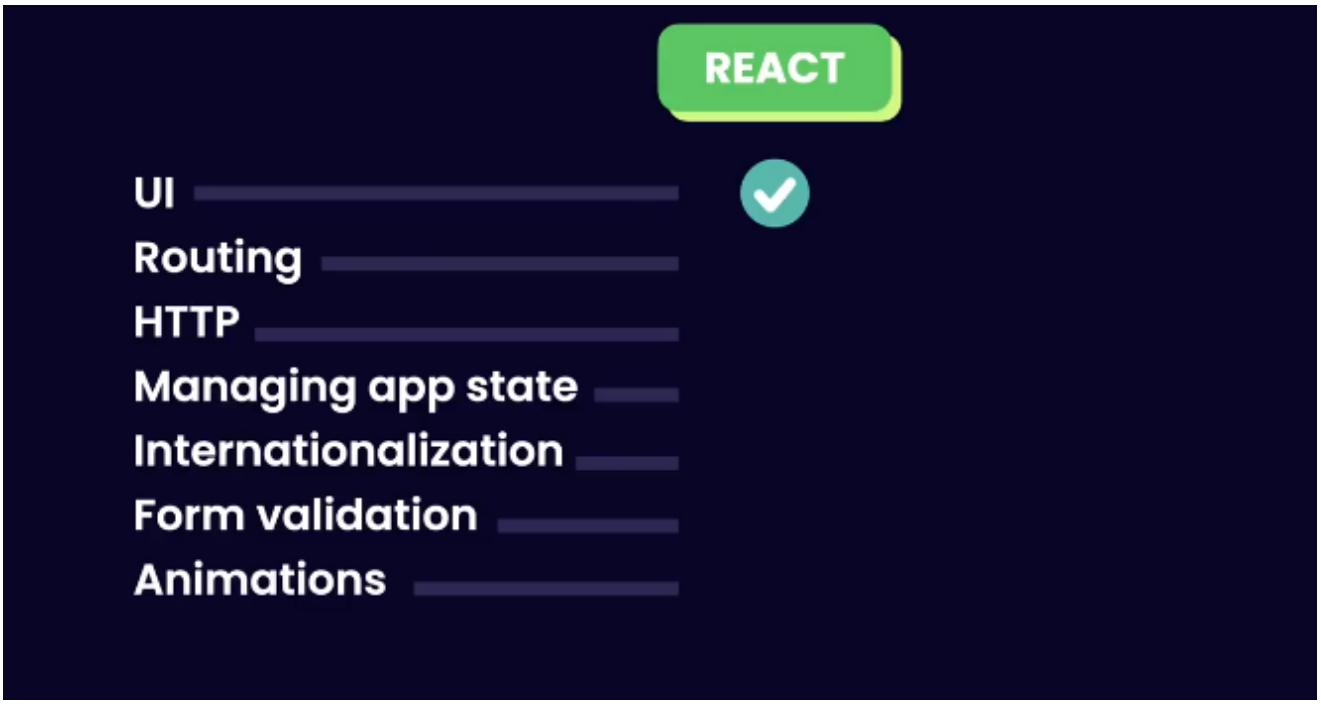


The difference :



It's like a tool and a tool set.

Some times we need more tools to do more just one thing like creating the UI, like these list of functionalities :



The great thing about react, it does not oblige you to use certain set of tools for the rest of the programming you do, but it's your responsibility to choose the right tools!

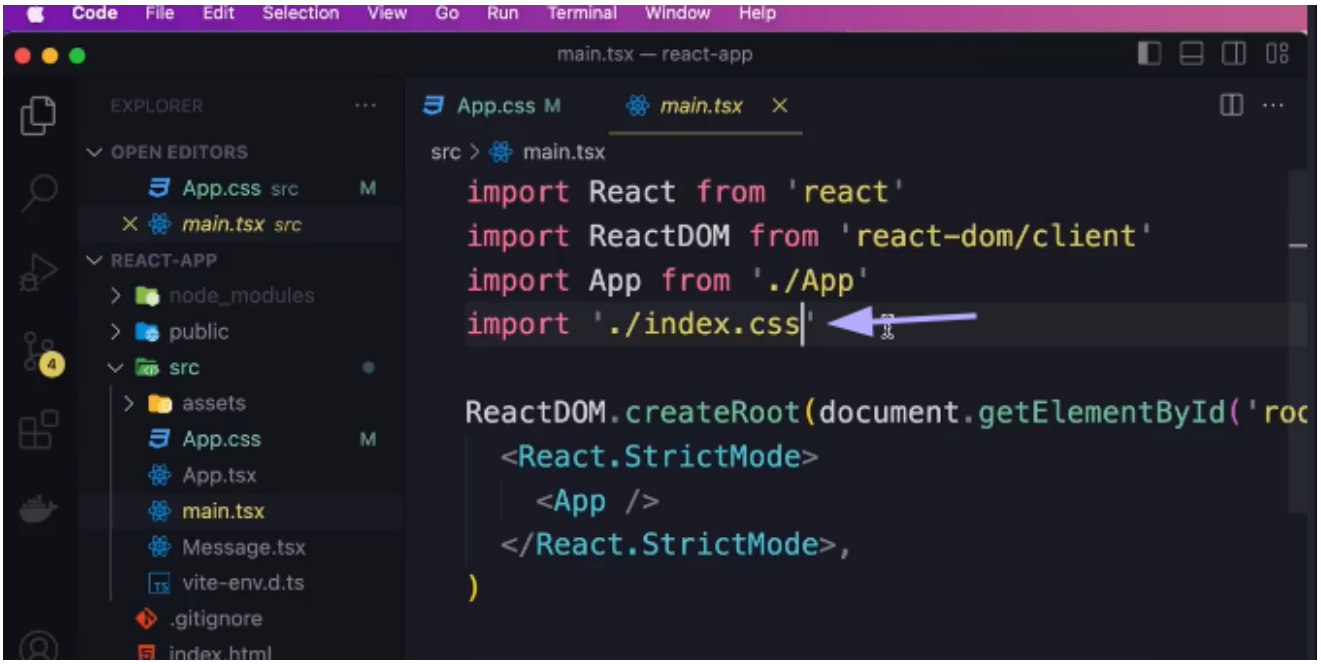
Creating a ListGroup component

Let's first install Bootstrap (CSS lib) to give our app a modern look :

```
npm i bootstrap@latest
```

and we need to import it in one of our CSS files, the `App.css` contains all the styles for our app components, we don't need it so delete its content, and for `index.css` it has global styles for our app, delete it all, just the content of the files, not the actual files.

Replace this in the `main.tsx` :



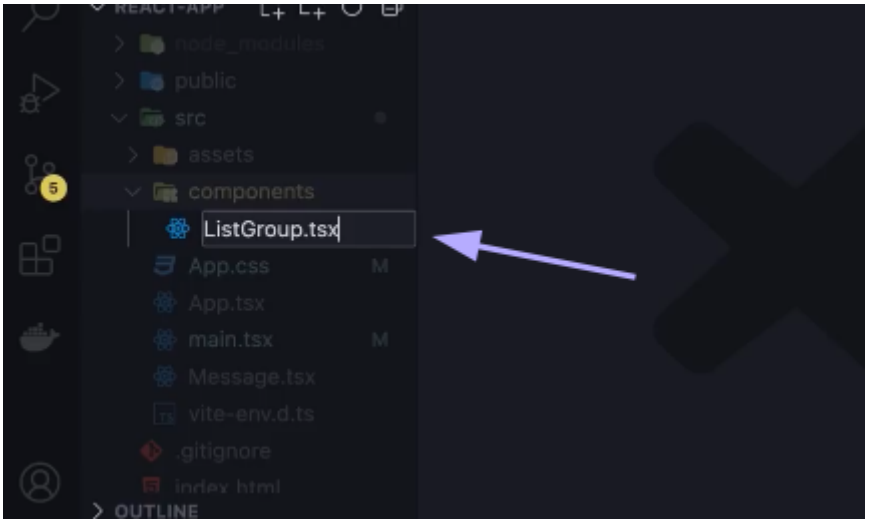
with this :

```
import 'bootstrap/dist/css/bootsrap.css'
```

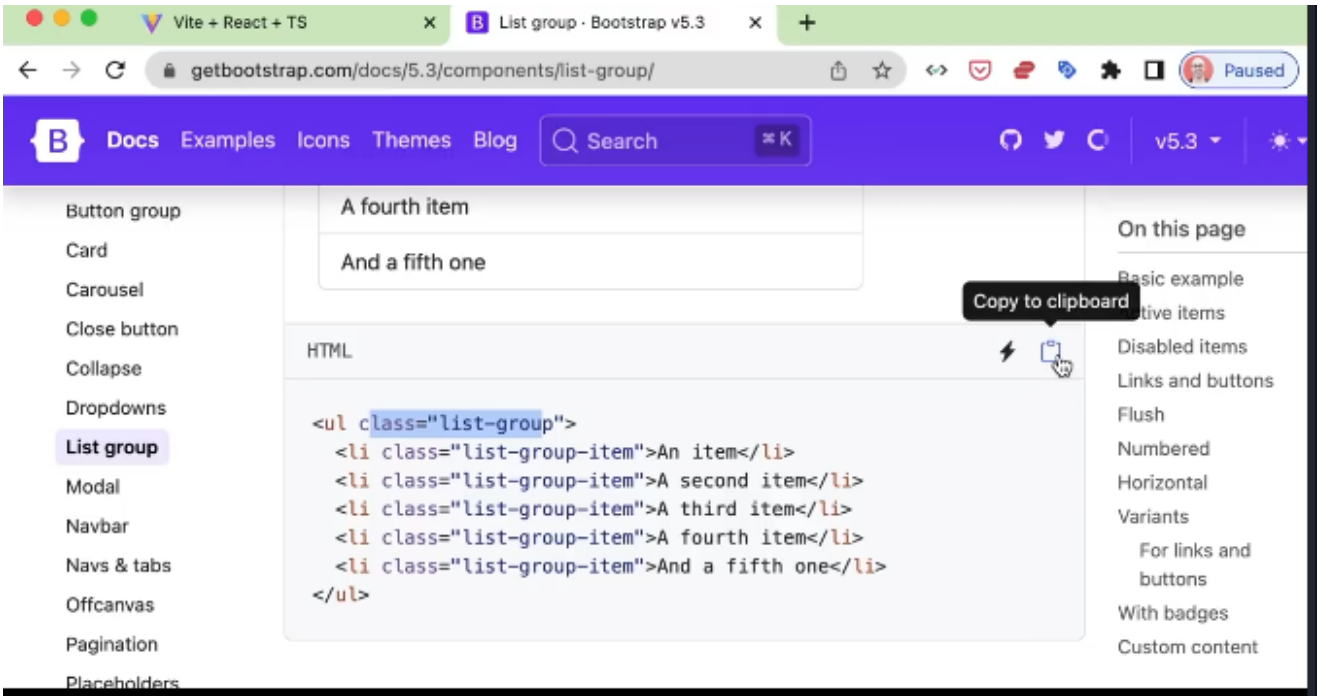
now save and check the changes in your browser!

In the src folder create a components folder to contain our components that we create, it's conventional, and since it's for a better structure and clarity, we want it!

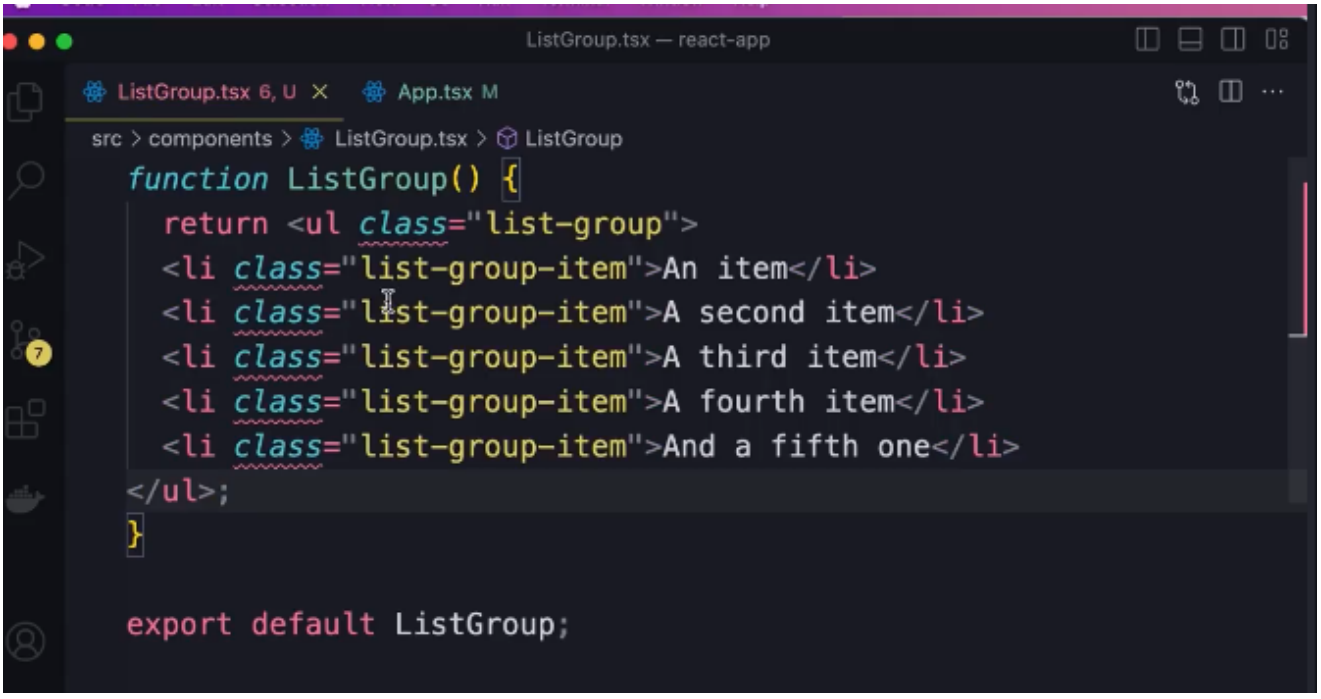
Now we create this file and name it using PascalCasing convention as usual :



Go to bootstrap docs to check for the proper way to use its styling; how to create the html elements on your page with the classes and ids corresponding to the predefined styles, for a ListGroup :



Return it in the `ListGroup.tsx` file to have the proper results of course :



You can get an error of having the keyword **class** in use because its reserved to js and ts, so we have to change it from class to **className**, `ctrl + d` for multi cursor editing and `esc` to disable it.

If you have many extensions as Prettier, you'd like to make it your default tool for formatting your document through command palette and choosing `format document` :

Fragments

In React, components **can** return multiple elements, but they must be wrapped inside a single parent element. This is because React's **render method** expects a single JSX element to return a valid UI tree.

Why Can't Components Return Multiple Root Elements Directly?

React's rendering logic is based on **a single root node per component** because:

1. **JSX Syntax Limitation** – JSX must return a **single expression**.
2. **Reconciliation Process** – React needs a clear hierarchy for efficiently updating the Virtual DOM.
3. **DOM Structure Consistency** – The browser expects a properly nested DOM tree.

How to Return Multiple Elements?

✅ Using a Wrapper Element (div, section, etc.)

A common approach is to wrap elements in a `div` or another container:

```
function MyComponent() {
  return (
    <div>
      <h1>Hello</h1>
      <p>Welcome to React!</p>
    </div>
```

```
  );
}
```

🔴 **Downside**: Adds unnecessary extra nodes to the DOM.

✅ Using React Fragments (`<>...</>` or `<React.Fragment>`)

A better way is **React Fragments**, which let you group elements without adding extra DOM nodes:

```
function MyComponent() {
  return (
    <>
      <h1>Hello</h1>
      <p>Welcome to React!</p>
    </>
  );
}
```

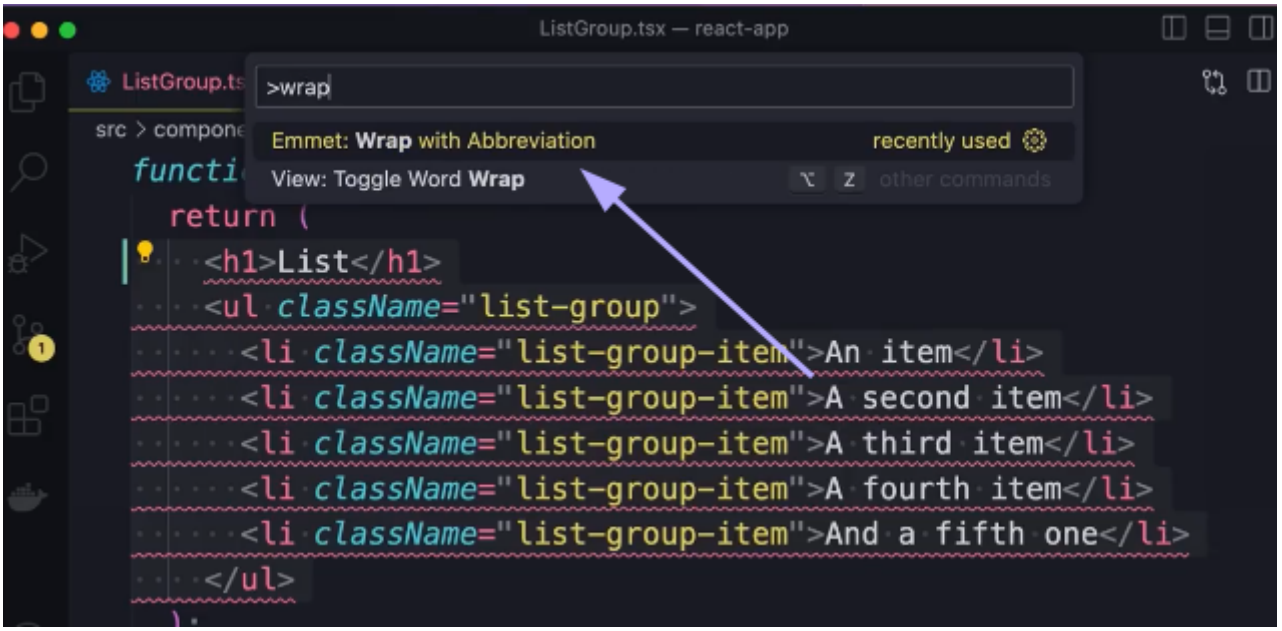
OR explicitly:

```
function MyComponent() {
  return (
    <React.Fragment>
      <h1>Hello</h1>
      <p>Welcome to React!</p>
    </React.Fragment>
  );
}
```

Why use Fragments?
✔ No unnecessary `<div>` wrappers.
✔ Better performance (fewer DOM nodes).
✔ Cleaner, semantic structure.

A simple solution is to wrap the elements into a div or another element.

→ As a great shortcut, select all the code you wanna wrap and open the command palette and select wrap with abbreviation :

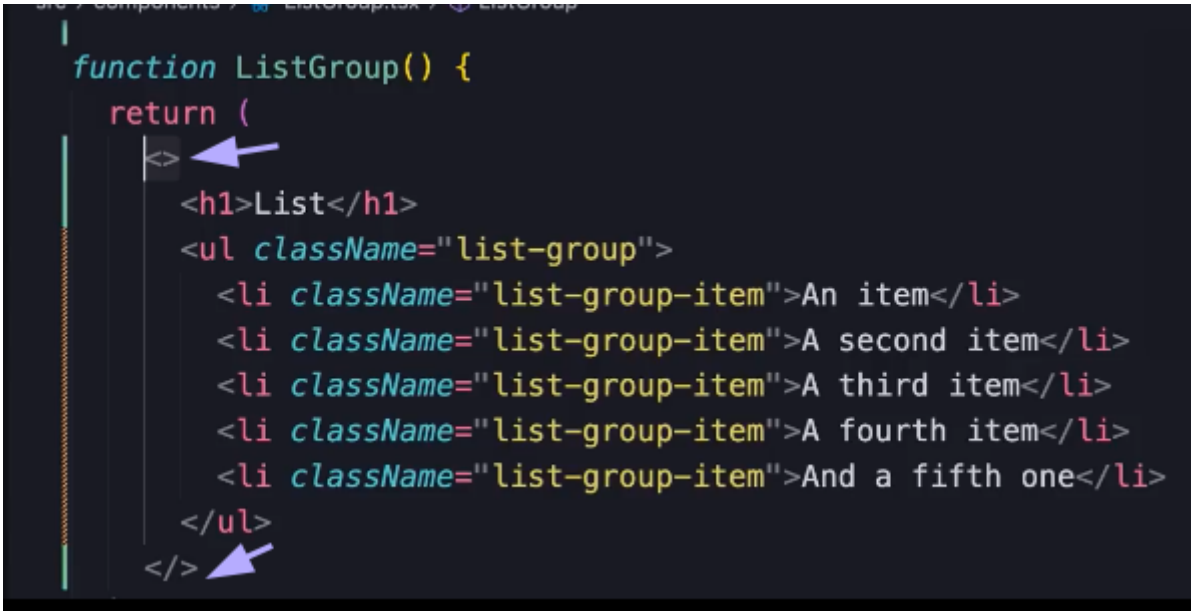


And enter the type of element you want.

A better way, of course, is to use **fragments**, not just adding more code to hide the mess under it, even it has a tempting to use shortcut like this one!

Like this, when we render the DOM, the fragment won't be an additional node in it.

You know what's better, doing this instead of importing fragment :



This is telling react you wanna have a fragment there.

Rendering Lists

why is it important for a developer to know how rendering is done ?

React's rendering process plays a crucial role in performance, UI updates, and application efficiency. Knowing **how rendering works** helps developers build faster, more optimized, and bug-free applications.

1. Prevents Unnecessary Re-renders 🔄

- React **re-renders** a component when:
  - Its **state** changes.
  - Its **props** change.
  - Its **parent re-renders** (even if its own props haven't changed).
- **Optimization Techniques:**
  - `React.memo`: Prevents re-renders if props haven't changed.
  - `useCallback` / `useMemo`: Optimizes function and data reference stability.

```
const MemoizedComponent = React.memo(MyComponent);
```

---

2. Improves Performance 🚀

Frequent unnecessary renders can slow down an app.
Optimizations include:

- **Lazy Loading (`React.lazy`)**: Load components only when needed.
- **Code Splitting (`React.Suspense`)**: Reduce initial bundle size.
- **Virtualization (`react-window`)**: Efficiently render long lists.

---

3. Helps Debugging UI Issues 🔍

Understanding rendering helps in fixing:

- **Flickering UI** due to frequent updates.
- **Stale state issues** when state updates aren't batched properly.
- **Components re-rendering unexpectedly** (causing lag).

🛠 Tools:

- **React Developer Tools** → Check renders in Components tab.
- **Profiler API** → Measure render times and bottlenecks.

---

4. Enables Efficient State Management 🏗

- **Global State (Redux, Context API) vs. Local State (`useState`)**:
  - Poor state design can trigger unnecessary renders across the entire app.
  - Example: Storing UI-specific data (like modals) in **global state** instead of component-level state can slow things down.

---

5. Helps in Server-Side Rendering (SSR) & Hydration 🌍

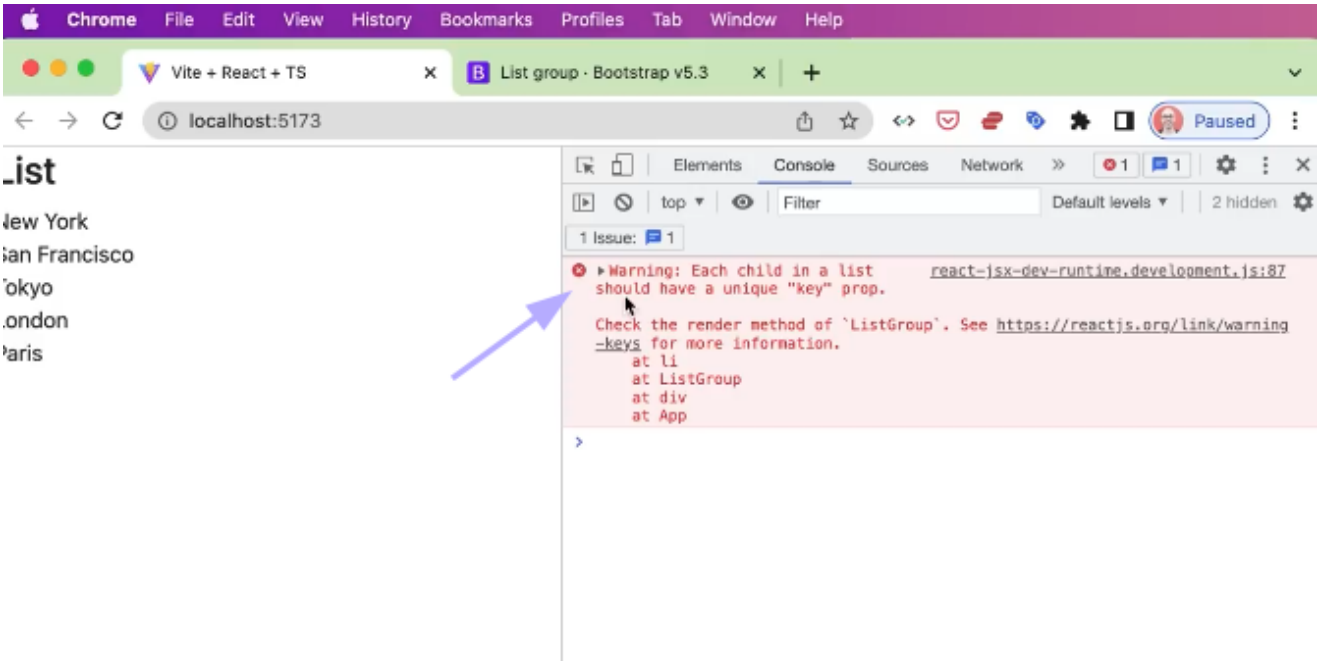
- **Next.js & SSR**: Understanding how React hydrates components helps in optimizing page loads and SEO.
- **Hydration Mismatches**: Debugging why SSR-rendered content differs from client-side rendering.

Conclusion:

Understanding rendering: ✅ **Boosts performance** (avoiding unnecessary updates).

✅ **Prevents unexpected UI behavior**.
✅ **Optimizes state management**.
✅ **Improves debugging skills**.

if we display the list the way we did, we will get a warning that says this :



We use `key` property to help react keep track of what it should change dynamically, since the key is unique, if the items in a list are unique in themselves you can use them

as keys, but bad practice all over that, there must be a better way of course!

→ In a real world app, we will have the key marked with an id like this :

```
<li key={item.id}></li>
```

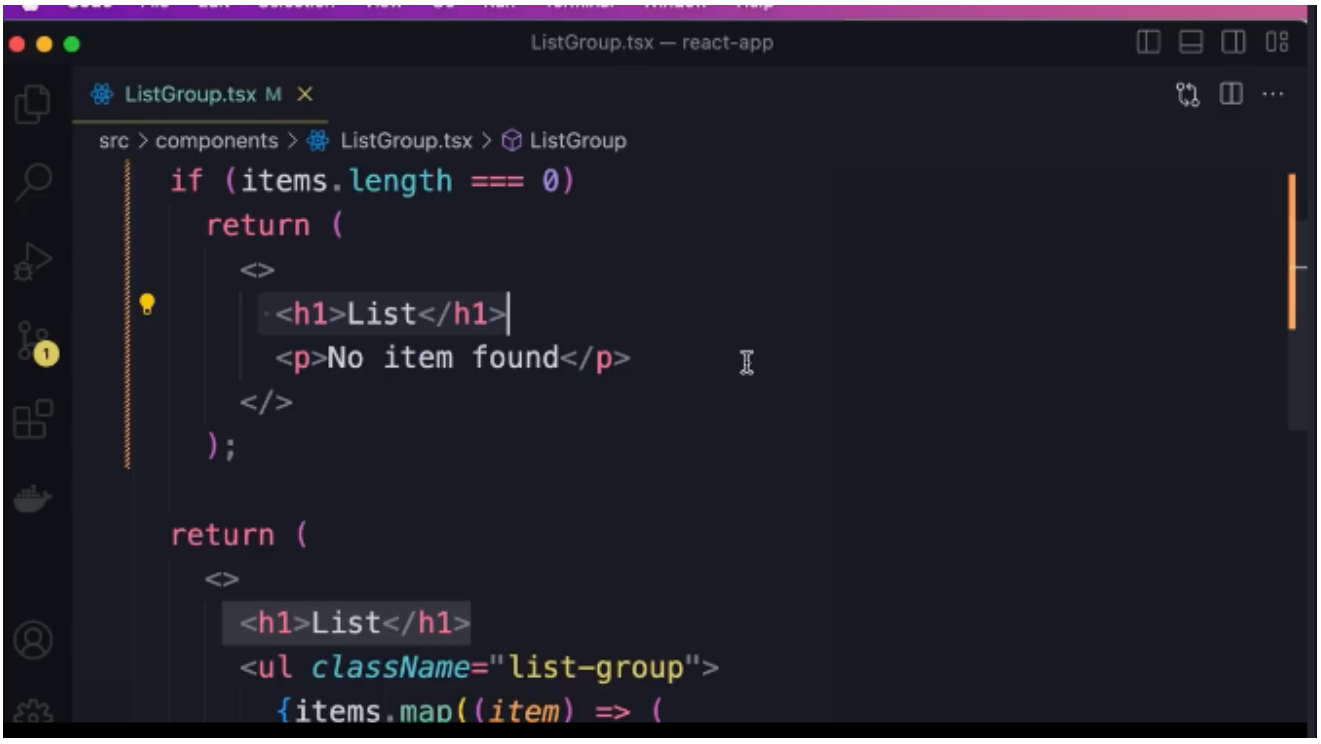
for API fetching or something like that.

Rendering Conditionals

Sometimes you wanna render some elements if some conditions are true.

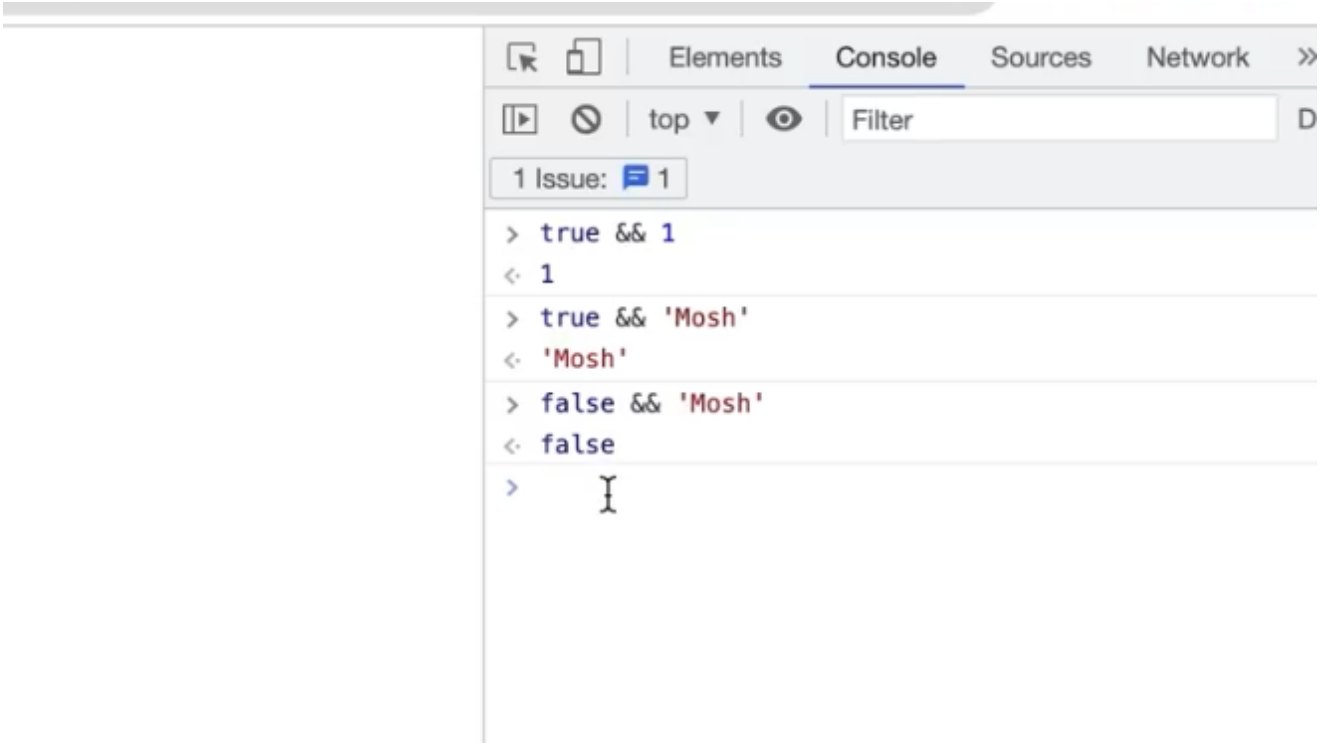
Sometimes, when you implement that, you'll have duplication, and you know that's a bad thing in programming, so we need to fix it.

So instead of this :



We can have this (conditional inside the jsx) :

In both cases, the code works just fine, but the second one is cleaner and better because in js, when we perform an AND logical op with a true value we get the other value (get this console on your developer tools in your browser) :



Handling Events

what's an event ?

An **event** is an action or occurrence that happens in a system, typically triggered by **user interactions** (like clicks, typing, scrolling) or **system changes** (like loading, resizing). Events allow applications to respond dynamically to user inputs.

Events in Web Development 🌍

In web development, an event represents an interaction in the browser, such as:

- Clicking a button (**click event**)
- Typing in an input field (**keydown event**)
- Moving the mouse (**mousemove event**)
- Submitting a form (**submit event**)

Example in plain JavaScript:

```
document.getElementById("btn").addEventListener("click", function () {
  alert("Button clicked!");
});
```

Events in React ⚛️

React uses a **synthetic event system**, which wraps around native browser events for better performance and cross-browser compatibility.

✅ React Event Example (onClick)

```
function MyComponent() {
  const handleClick = () => {
    alert("Button Clicked!");
  };

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

📌 **Note:** React event handlers use camelCase (`onClick`, `onChange`) instead of lowercase (`onclick`, `onchange`).

Common React Events 🖱️

Event Name	Description
`onClick`	Fires when an element is clicked
`onChange`	Detects input changes (text fields, dropdowns)
`onSubmit`	Triggers when a form is submitted

Event Name	Description
`onMouseEnter`	Fires when mouse hovers over an element
`onKeyDown`	Detects key presses on a keyboard
`onScroll`	Fires when scrolling occurs

Example for input change:

```
function InputComponent() {
  const handleChange = (event) => {
    console.log("Value:", event.target.value);
  };

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

Why Are Events Important? 🚀

- They make applications **interactive**.
- Enable **real-time user feedback**.
- Control **form submissions, animations, and UI changes**.
- Improve **user experience (UX)**.

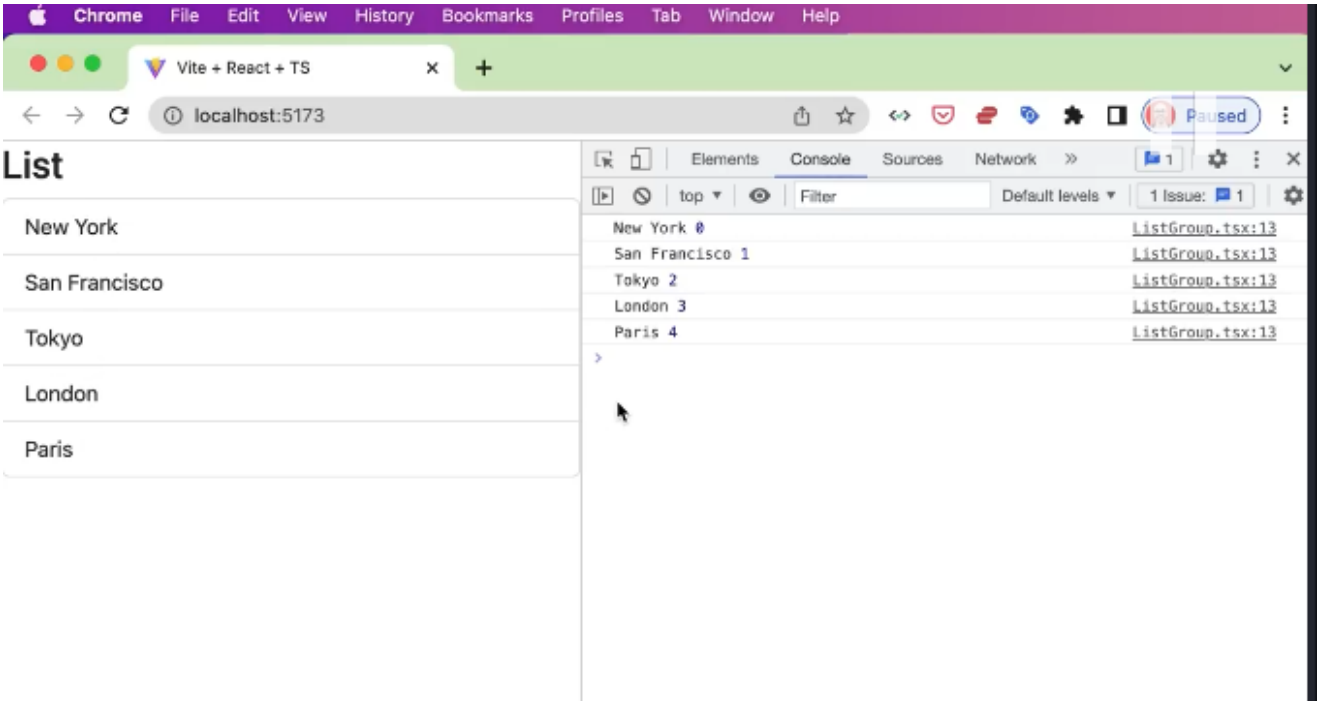
Let's see how to handle a **click** event in a component.

In react, each element has a prop called `onClick`, it will let us determine what will happen if a specific element is clicked by the user.

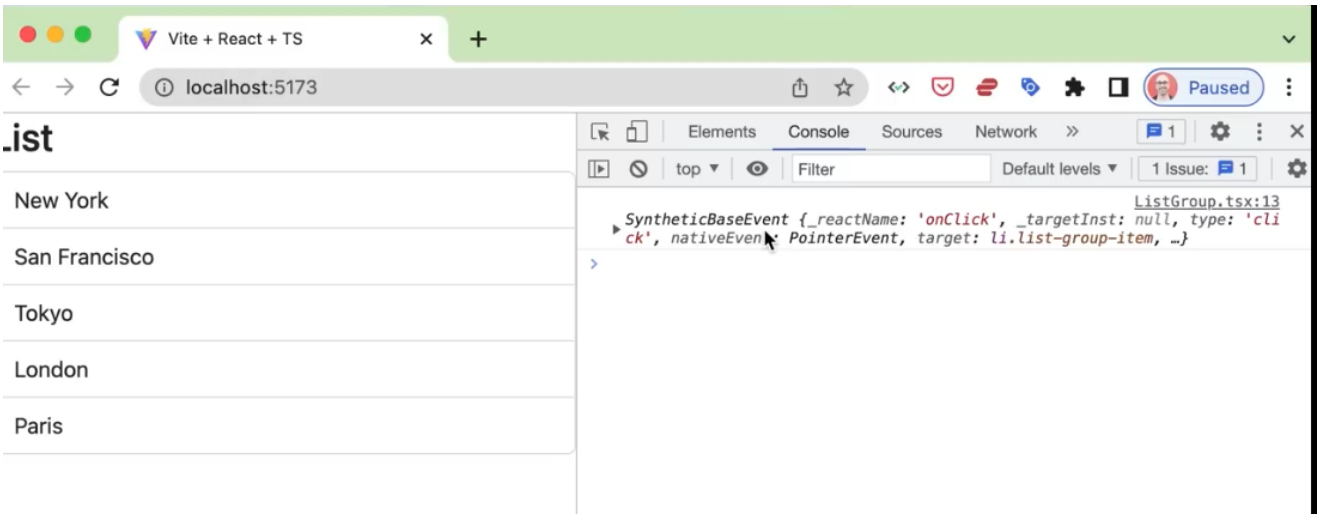
we can have this simple exmaple here :

→ Notice that it take a function as a value.

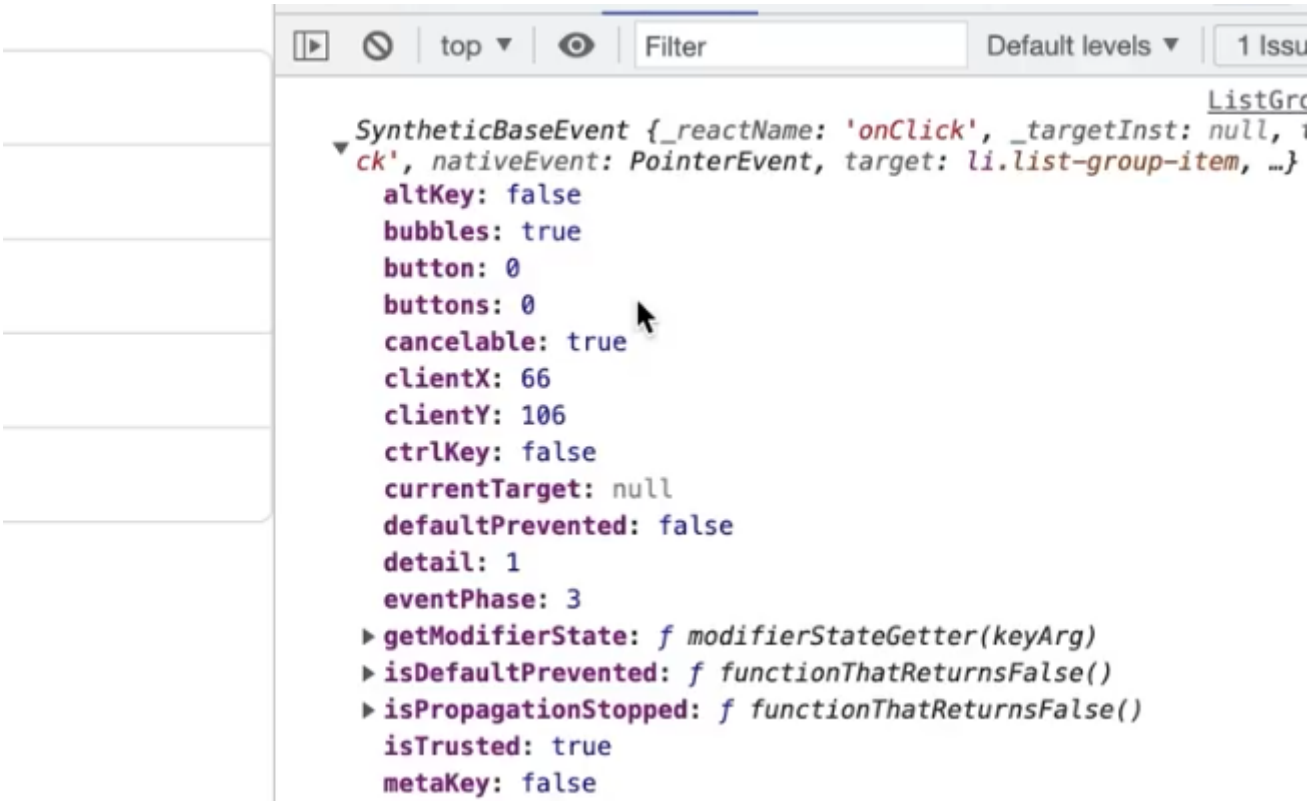
We have this result, note that we are displaying the index of each item too :



The onClick arrow function has a a parameter that refers to the event that happened for it to be called, you can try and display it when the event happens, it's an object of course :



Among the cool properties of this object is the clientX and clientY to tell you where did the clicking happen on the page :



Among other props, feel free to search for their roles if you wish.

Again for a better design done by a better programmer, meaning you, any functions shouldn't be nested by definition, it should be called by its name.

what is type annotation ?

Type annotation is a way to explicitly define the data type of a variable, function parameter, or return value in programming languages that support static typing (like TypeScript, Python, and Java).

In dynamically typed languages (like JavaScript or Python without annotations), types are inferred at runtime. Type annotations help catch errors early by enforcing type

correctness at compile or development time.

Type Annotation in Different Languages

1️⃣ TypeScript (JavaScript with Static Typing)

TypeScript adds type annotations to JavaScript for better safety.

```
let age: number = 25;  // 'age' must always be a number

function greet(name: string): string {
  return `Hello, ${name}!`;
}
```

✔ Prevents accidental type mismatches.
✔ Helps with auto-completion and better debugging.

2️⃣ Python (Optional Type Hinting)

Since Python is dynamically typed, type annotations are **optional** but help improve code clarity.

```
def add(x: int, y: int) -> int:
    return x + y

name: str = "Alice"
```

✔ Helps tools like **mypy** check type consistency.
✔ Improves code readability.

3️⃣ Java (Statically Typed Language)

Java requires explicit type declarations.

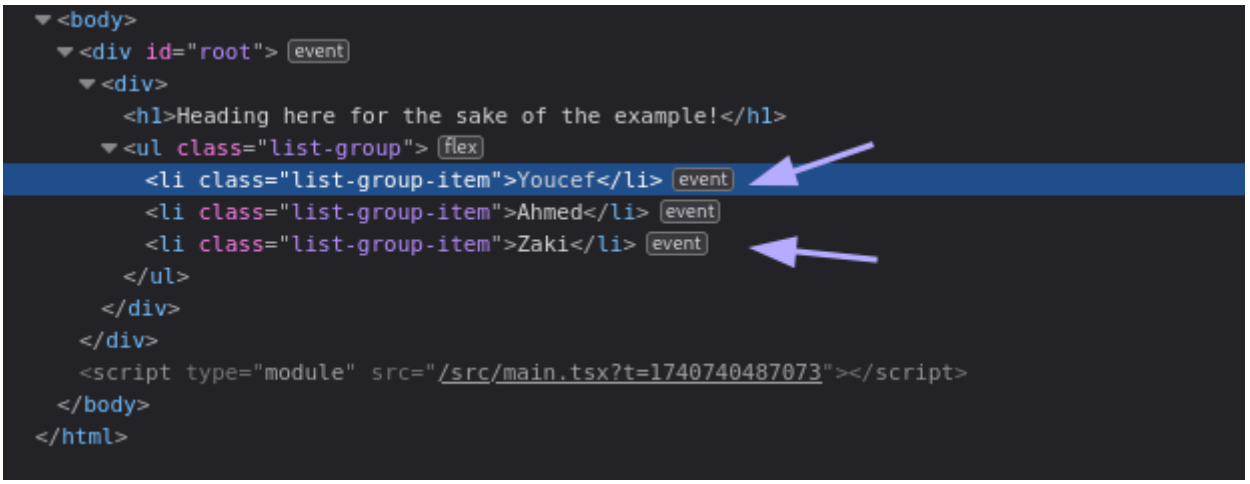
```
int count = 10;  // 'count' must always be an integer

public String sayHello(String name) {
    return "Hello, " + name;
}
```

✔ Prevents runtime errors by catching type mismatches during compilation.

Why Use Type Annotations? 🚀

✅ **Prevents Bugs** – Avoids type-related errors at runtime.
✅ **Improves Readability** – Code is more self-explanatory.
✅ **Enhances Auto-completion** – IDEs provide better suggestions.
✅ **Boosts Performance** – Some languages optimize based on known types.

BTW, in your dev tools, you can see that in the inspector tab, the elements with events attached to them are marked :



what is flex that's mentioned in the photo ?

In the image, the word **"flex"** appears next to the `<ul class="list-group">` element in the **Chrome DevTools Elements panel**. This indicates that the **ul** element has `display: flex` applied to it, either via CSS or inline styles.

What Does `flex` Mean?

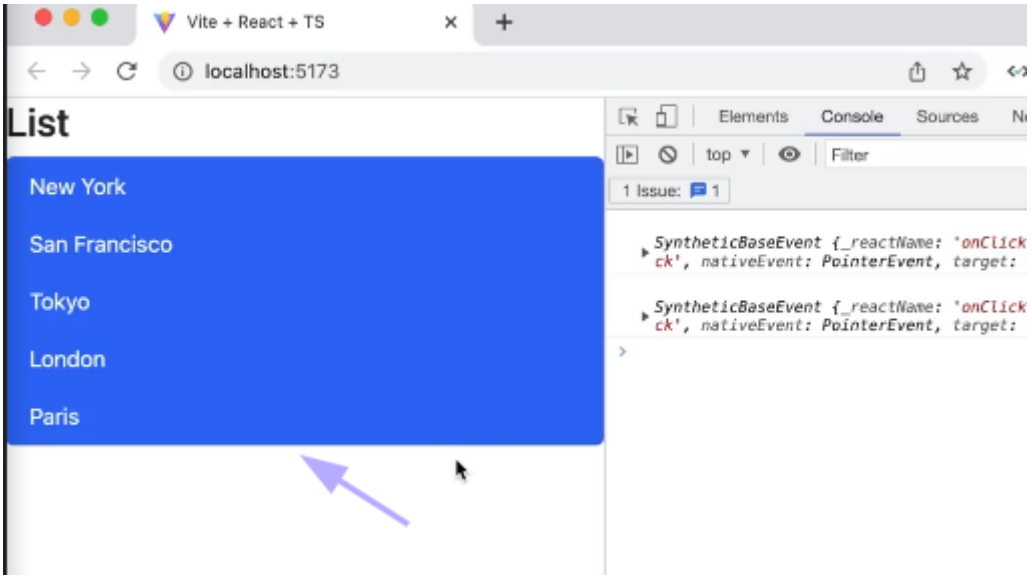
The `flex` label in DevTools signifies that the element is a **Flexbox container**. In CSS, `display: flex;` makes child elements (the `<li>` items in this case) behave as **flex items**, allowing for more control over alignment, spacing, and layout.

Why is This Useful?

- It ensures that the **list items (`<li>`)** inside the `<ul>` are laid out in a flexible way.
- The **items might be arranged in a row or column** depending on the `flex-direction` property.
- It enables **better responsiveness** for different screen sizes.

Managing State

In bootstrap we have the `active` class, with it we can make highlighted elements :



But we need to highlight one item at a time when clicked, we do that by keeping track of the clicked element's index in the list.

what is a hook ?

A **Hook** in React is a special function that allows **functional components** to use **state** and **lifecycle features** without needing class components. Hooks were introduced in **React 16.8** to simplify state management and side effects in functional components.

Why Are Hooks Important? 🤔

✅ Allow **stateful logic** in functional components.
✅ Simplify **component logic** and reusability.
✅ Remove the need for class components.
✅ Improve **code readability** and **performance**.

Common React Hooks 🎣

Hook	Purpose
`useState`	Manages state in a functional component.
`useEffect`	Handles side effects (e.g., fetching data, subscriptions).
`useContext`	Accesses values from React's Context API.
`useRef`	Creates a reference to a DOM element or persists values without re-rendering.
`useMemo`	Optimizes performance by memoizing values.
`useCallback`	Memoizes functions to prevent unnecessary re-renders.

Basic Example: `useState` Hook (Managing State)

```
import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0); // State variable

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

- `useState(0)` initializes state (`count`) with `0`.
- `setCount(count + 1)` updates state when the button is clicked.

Example: `useEffect` Hook (Side Effects)

```
import { useState, useEffect } from "react";

function Timer() {
  const [time, setTime] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => setTime((t) => t + 1), 1000);
    return () => clearInterval(interval); // Cleanup function
  }, []); // Empty dependency array runs effect only once

  return <p>Timer: {time} seconds</p>;
}
```

- `useEffect` runs when the component mounts.
- The **cleanup function** prevents memory leaks.

Conclusion: Hooks Make React Easier 🚀

Hooks **simplify state management, side effects, and performance optimizations** without class components.

what does a functional component mean ?

A **functional component** is a simple JavaScript function that **returns JSX** to render UI. Unlike class components, functional components:

✅ Are **simpler** and **easier to read**.
✅ Do **not require a** `this` **keyword**.
✅ Use **React Hooks** (`useState`, `useEffect`, etc.) to manage state and side effects.
✅ Are **more performant** than class components in most cases.

Basic Functional Component Example

```
function Greeting() {
  return <h1>Hello, React!</h1>;
}

export default Greeting;
```

- This component **returns JSX** (`<h1>Hello, React!</h1>`).
- It **does not** use a class or `this`.

Functional Component with `useState` Hook

```
import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0); // State in a functional
component

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

export default Counter;
```

- 🔹 `useState(0)` initializes **state** inside the functional component.
- 🔹 `setCount(count + 1)` updates the state when the button is clicked.

Class Component vs Functional Component

Before React Hooks (`React 16.8`), **class components** were used for stateful logic. Example of a **class component** (now less common):

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

class Counter extends Component {
  state = { count: 0 };

  increment = () => {
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}

export default Counter;
```

🔴 Downsides of Class Components:

- More **boilerplate code** (`this.state`, `this.setState`).
- Harder to **reuse logic** across components.
- More complex for **performance optimizations** (e.g., memoization ; The term “Memoization” comes from the Latin word “memorandum” (to remember), which is commonly shortened to “memo” in American English, and which means “**to transform the results of a function into something to remember**”.).

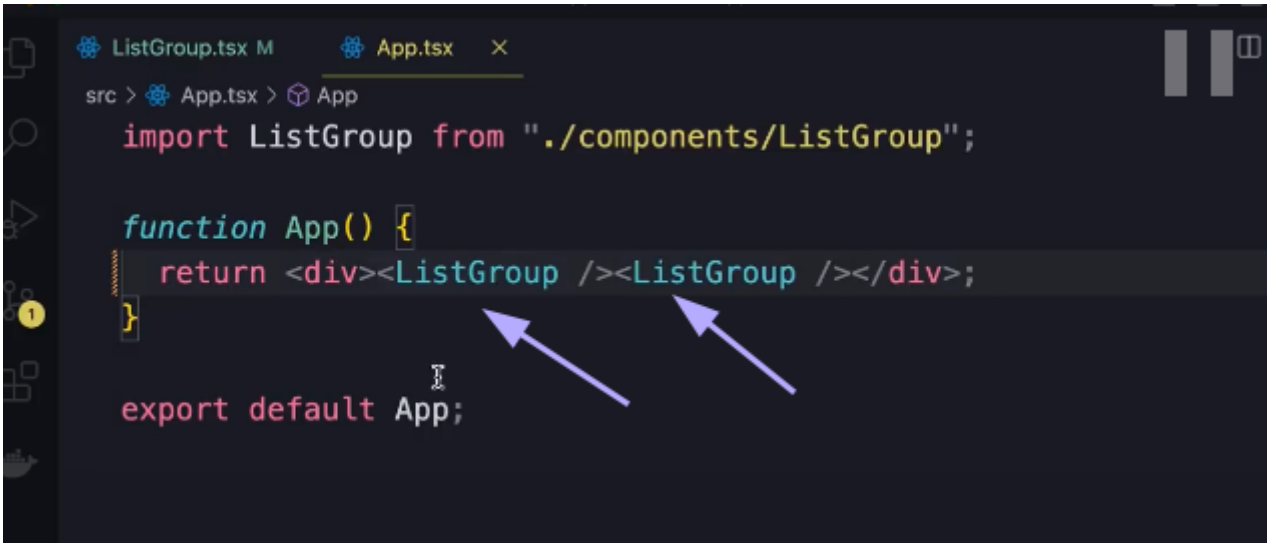
✅ Why Use Functional Components?

- **Easier to read & write** 📝
- **Better performance** 🚀
- **Can use hooks** (e.g., `useState`, `useEffect`) 🎣

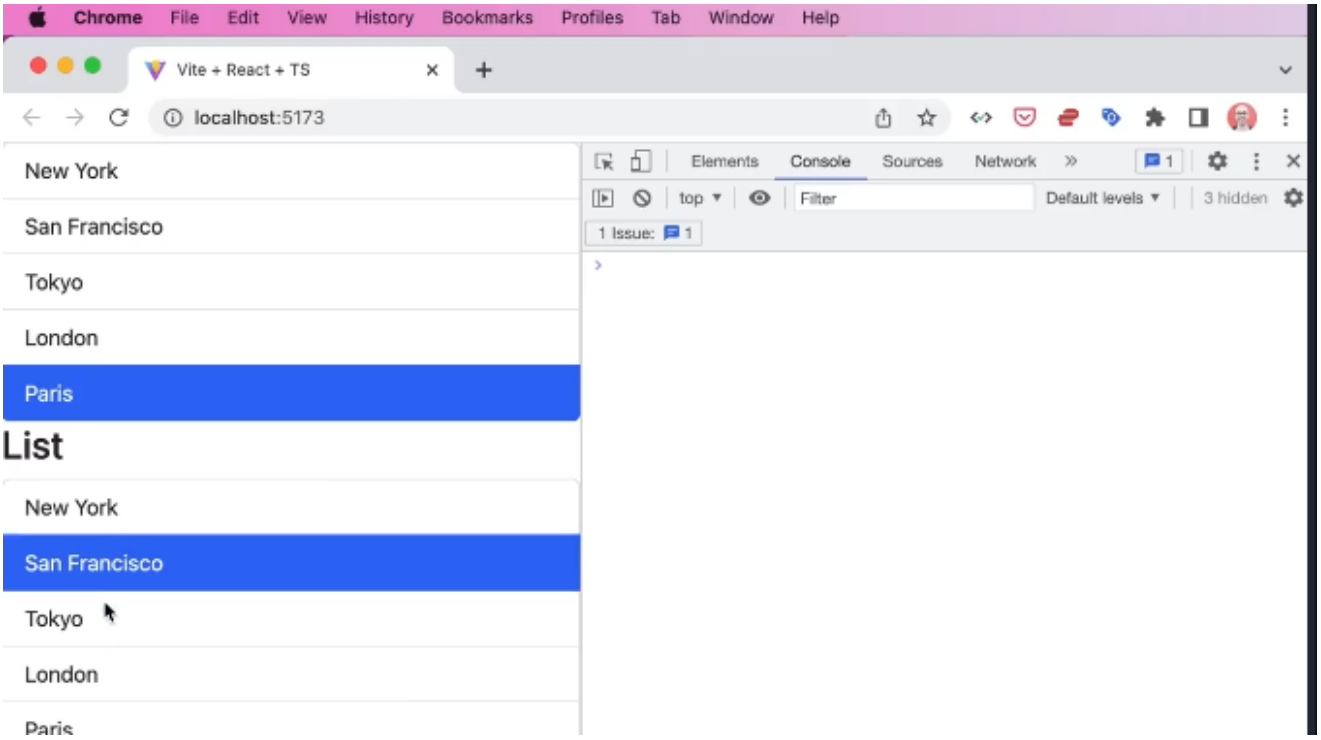
- **Encouraged by modern React** (Class components are still supported but not recommended).

Conclusion: Functional Components = Simpler & More Powerful 🎯

Please note that each component has its own state, meaning when you render many components of the same type; meaning they're built with the same logic, they still would have two different presences in the components tree, so this :



will give you this :



never the same state unless you want it to be the same.

Passing Data via Props

If we wanna make components reusable, meaning, we don't wanna create a new components for each set of data; we use Props.

They're the input for our components, to define them we use the interfaces in ts :

```
interface ListGroupProps {
    items : string[];
    heading : string;
}
```

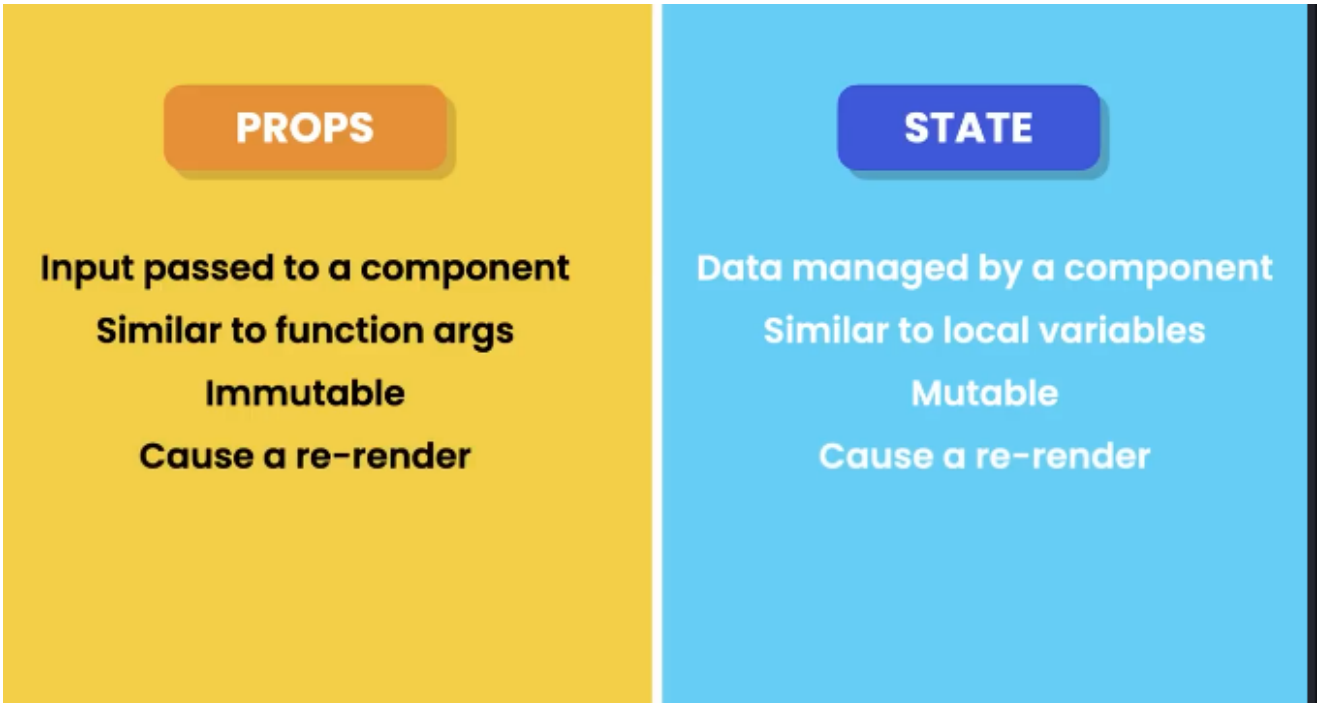
here, the ListGroupProps is an interface with the properties of a listGroup that we wanna work with later in the code.

Passing functions via Props

In a real world app, something should happen after an item is selected, like filtering some data.

We don't wanna create a specific logic inside our component, if we do so, it won't be reusable anymore!

Props vs State



immutability of props is not an obligation but it's a pattern error.

why are props immutable by conventions ?

In React, **props (short for "properties") are immutable** by convention, meaning **a component should never modify its own props**. Instead, they should be treated as **read-only values** passed down from a parent component.

1️⃣ React's One-Way Data Flow 🔁

- React follows a **one-way data flow**, meaning data flows **from parent to child**.
- If a child component modifies props, it **disrupts this flow**, making state management unpredictable.

🔹 **Example of One-Way Data Flow:**

```
function Parent() {
  return <Child name="Alice" />;
}

function Child(props) {
  return <h1>Hello, {props.name}!</h1>; // Props are read-only
}
```

✔ `props.name` is **passed from the parent** and cannot be changed inside `<Child />`.

2️⃣ Prevents Unexpected Bugs & Side Effects 🐛

If a child component modified its props, it could cause **unintended side effects** in the parent or other components.

🔴 **Wrong Approach (Trying to Mutate Props)**

```
function Child(props) {
  props.name = "Bob"; // ❌ Mutating props (This is BAD!)
  return <h1>Hello, {props.name}!</h1>;
}
```

💥 This can lead to **unexpected behavior** and debugging issues.

3️⃣ Encourages Pure Functions ✅

Functional components should be **pure functions**, meaning:

- They **always return the same output** for the same input.
- They **don't modify external values** (like props).

If props were mutable, components could behave inconsistently.

✔ **Correct Approach (Using State Instead)** If a component needs to modify data, **use state (`useState`) instead of mutating props**:

```
function Child({ initialName }) {
  const [name, setName] = React.useState(initialName);

  return (
    <div>
      <h1>Hello, {name}!</h1>
      <button onClick={() => setName("Bob")}>Change Name</button>
    </div>
  );
}
```

🔹 Here, `initialName` is **received as a prop**, but changes are managed via **state (`useState`)**.

4️⃣ Parent Should Control Data Changes 🎛️

If a child needs to update the data, it should **notify the parent** via a callback function.

✔ **Correct Approach (Lifting State Up)**

```
function Parent() {
  const [name, setName] = React.useState("Alice");

  return <Child name={name} changeName={() => setName("Bob")} />;
}

function Child({ name, changeName }) {
  return (
    <div>
      <h1>Hello, {name}!</h1>
      <button onClick={changeName}>Change Name</button>
    </div>
  );
}
```

- The child **requests** a change, but the **parent controls the update**.

Conclusion: Props = Read-Only for a Reason 🚀

✔ **Ensures predictable UI updates** 🎯
✔ **Follows React's one-way data flow** 🔄
✔ **Encourages better state management** 🏗️
✔ **Prevents hard-to-debug issues** 🐛

Passing Children

sometimes we wanna pass children to a component, just we did in our app here passing the ListGroup to the div element.

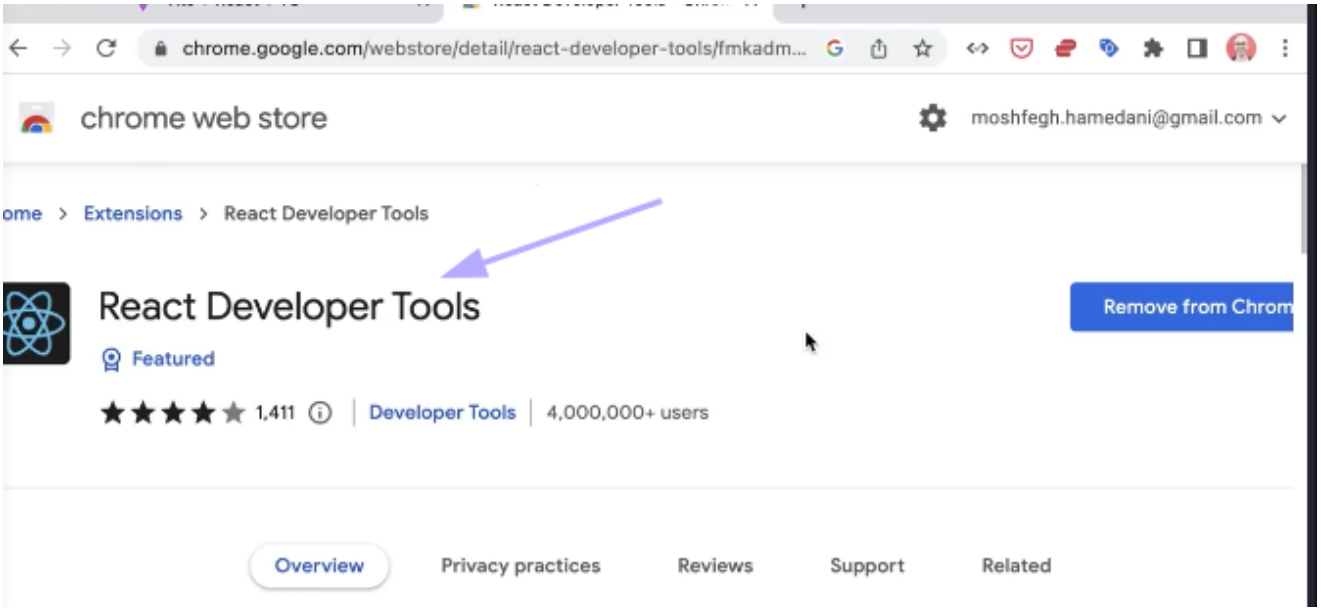
We need to create a component that accepts children!

We will make a Alert component, so go check its styling in bootstrap docs, they have two classes, `alert` for all alert elements and the second one for the color.

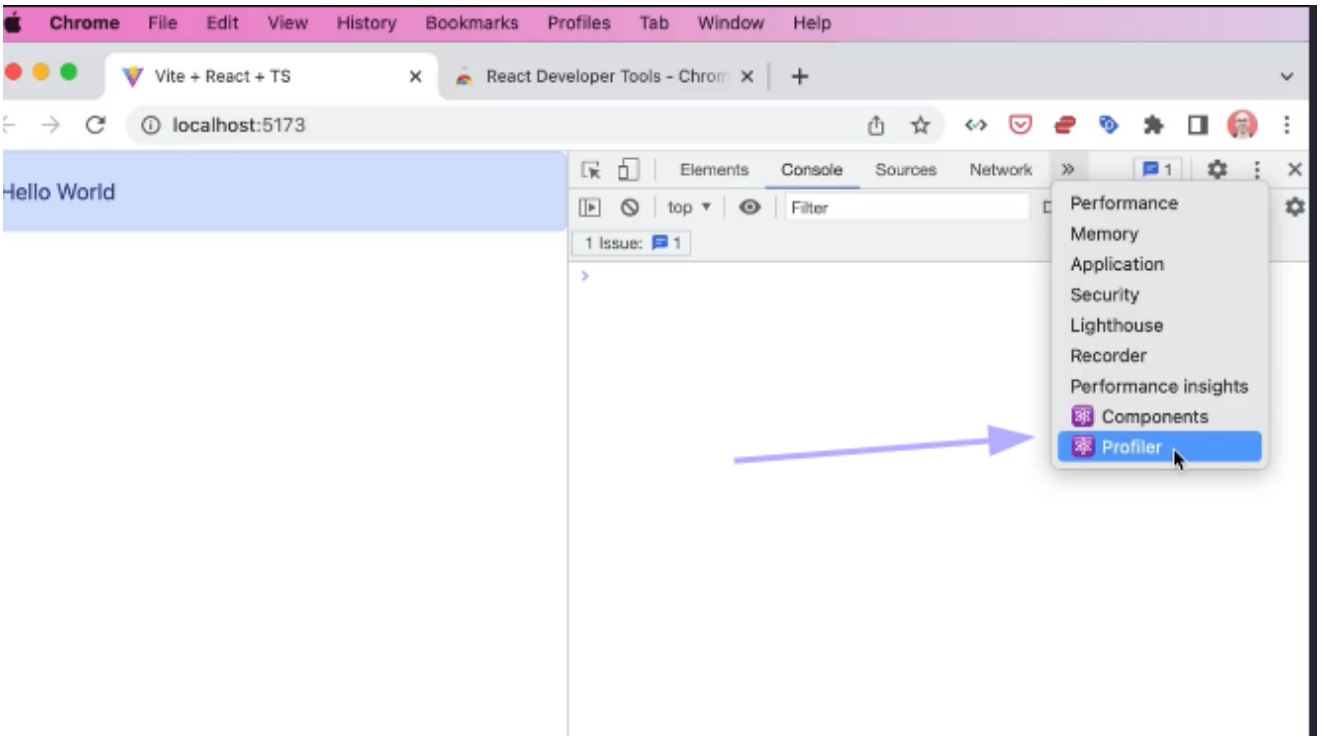
We don't wanna keep passing arguments to elements like html attributes but instead we wanna pass them as children elements.

Inspecting Components with React Dev Tools

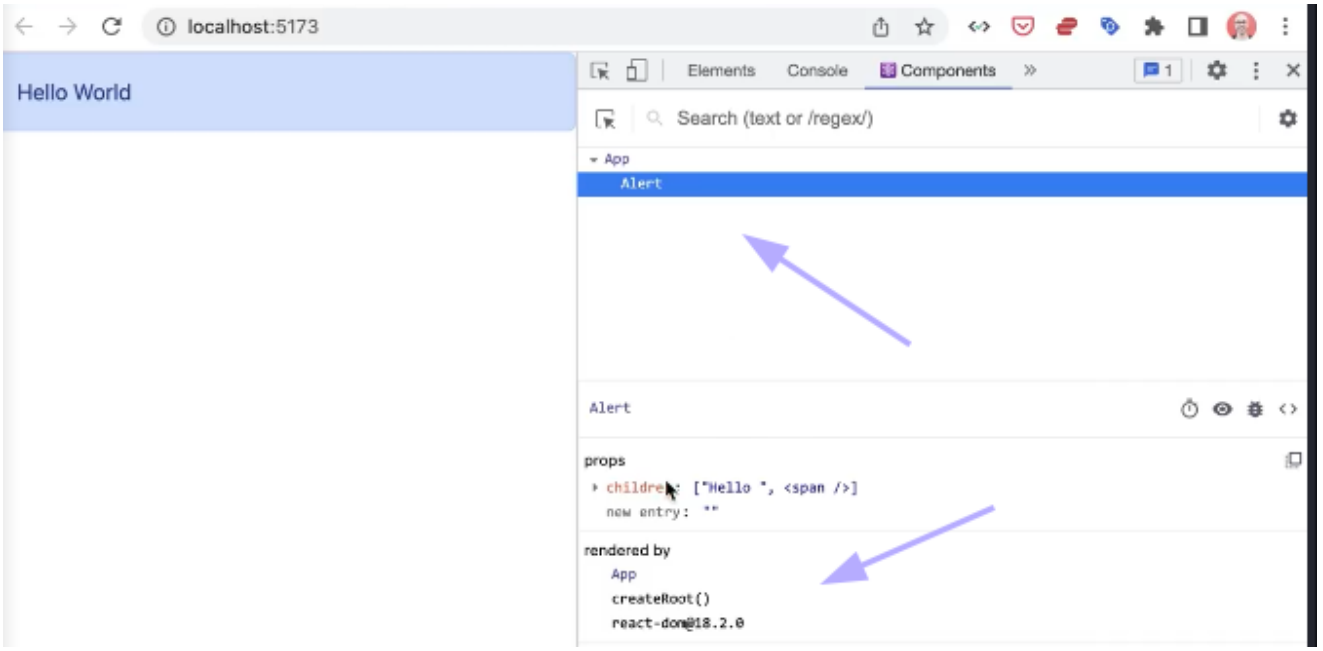
get this extension for your browser :



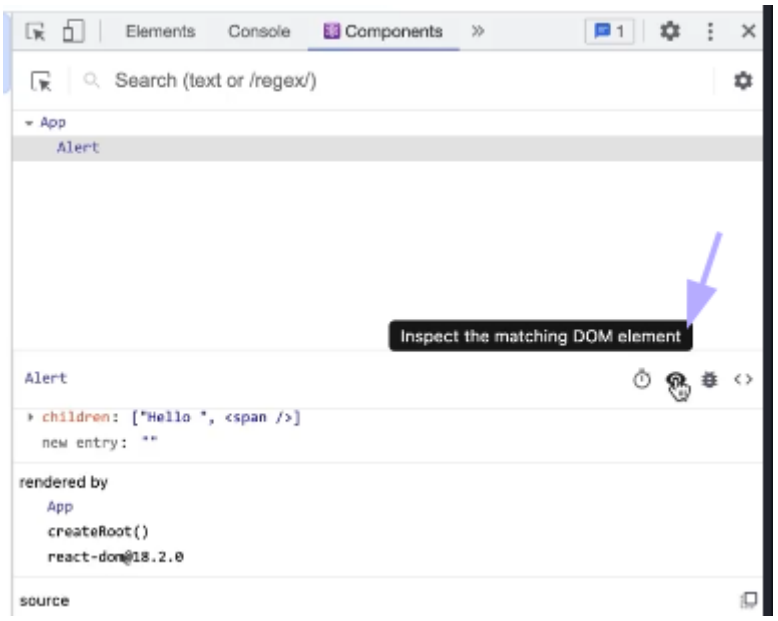
when done, you'll have new tabs in your dev tools :



in `components` tab, you can see the hierarchy of components used in your app and you can also see the props and more :



one very useful tools is the DOM element matching of a react component :



you select the component you wanna see the DOM element corresponding to and you go back to the Elements tab to check it.

And of course you can see the source code of a component too! all in this tab.

what does the profiler tab have ?

In React DevTools, the **Profiler** tab is an incredibly useful tool for tracking and improving the performance of your React application. It helps you see how often your components are re-rendering, how long each render takes, and which components are causing performance bottlenecks.

Key Features of the **Profiler** Tab:

1. **Record Rendering Performance:**
   - The Profiler tab allows you to record the performance of your React app by capturing renders over time.
   - You can start and stop recording interactions in your app to see how components are behaving during those periods.
2. **Component Renders:**
   - It shows you **which components** are rendering, and it will highlight components that are re-rendering frequently.
   - You can see **how many times** each component re-rendered during the recording session.
3. **Render Time:**
   - The Profiler shows how long each render takes. This is important for identifying slow components that might be affecting the performance of your app.
   - It displays the **"commit time"** (how long it took for React to process all updates for a render) and the **"render time"** (how long it took to render each

component).

4. **Flamegraph:**
   - You can see a **flamegraph** that visualizes the render times of all components. Components with the longest render times are shown as wider bars, helping you easily identify performance issues.
   - The flamegraph helps you spot **performance bottlenecks** by showing where React is spending the most time during re-renders.
5. **Why Did a Component Render:**
   - It also shows you why each component rendered (for example, because of a state change, prop change, etc.).
   - You can view the **"why" of the re-render** by looking at the details of each commit.
6. **Interactive Mode:**
   - The Profiler tab allows you to interact with the app while recording. For example, you can click buttons or interact with forms, and the Profiler will track how these interactions affect rendering.
   - This lets you analyze the app's performance under typical user interactions.
7. **Comparing Multiple Renders:**
   - The Profiler lets you compare **before and after** render performance, so you can track if your optimizations (like `React.memo` or `useMemo`) are actually making a difference.
   - You can see changes in render times and the number of renders between different frames.

How to Use the Profiler:

1. **Start Recording:**
   - In the Profiler tab, click the **"Record"** button to start recording render performance.
   - Perform some interactions in your app (like clicking buttons, updating state, etc.).
   - After some interactions, stop the recording.
2. **Analyze the Results:**
   - Once you stop the recording, React DevTools will display all the recorded renders.
   - You can hover over components in the flamegraph to see their render times and see which ones are taking longer than expected.
3. **Look for Optimization Opportunities:**
   - If you see a component re-rendering too often or taking too long, you can investigate and optimize the code, for example by using `React.memo`, `useMemo`, or avoiding unnecessary state updates.

Example Use Case:

- **Detecting Unnecessary Re-renders:** You might notice that some components are re-rendering every time a parent component updates, even if they don't actually need to. The Profiler will help you spot this and optimize the app, for example, by memoizing components or using `shouldComponentUpdate` to prevent unnecessary renders.

Summary:

The **Profiler** tab in React DevTools helps you analyze your app's performance by showing you:

- How often components are re-rendering.
- How long each render takes.
- Which components are causing performance issues.
- How to track and compare performance before and after optimizations.

It's a valuable tool to ensure that your React app remains fast and efficient, especially as it grows larger and more complex.

Exercise : Building a Button Component

all in the source files.

Exercise : Showing an Alert

Basically we wanna have a button that when clicked it'll show an alert on the page, and that alert must have a button that we can click to close that alert element (dismiss it).

→ It's great to use useState to toggle or changes the states of the components even if the change is slight.

DONE!