BootStrap:

Bootstrap is a **popular front-end framework** that helps you build responsive, mobile-first websites quickly and efficiently. It provides a rich set of pre-designed HTML, CSS, and JavaScript components—like buttons, forms, modals, and navigation bars—so you don't have to start from scratch every time.

You can get started in a few ways:

- **CDN**: Add Bootstrap directly to your HTML via a link and script tag.
- **Package Managers**: Install it using npm (npm install bootstrap) if you're working with Node.js or bundlers like Webpack or Vite.
- **Customization**: Use Sass to customize Bootstrap's variables and include only the components you need.

Array listing with Map function:

The map() function in JavaScript is a powerful way to **transform arrays**. It creates a new array by applying a callback function to each element of the original array—without modifying the original.

Here's a simple example:

```
const numbers = [1, 2, 3, 4, 5];

const doubled = numbers.map(num => num * 2);

console.log(doubled); // Output: [2, 4, 6, 8, 10]
```

You can also use it to list out items, say in a React component:

```
const fruits = ['Apple', 'Banana', 'Cherry'];


return (
  <ul>
    {fruits.map((fruit, index) => (
      <li key={index}>{fruit}</li>
    ))}
  </ul>
);
```

Nested list in React.JS:

To render a **nested list in React**, especially when dealing with hierarchical data, you can use **recursive components**. Here's a clean example using a nested array of objects:

Sample Data

```
const data = [
  {
    id: 1,
    name: 'Fruits',
    children: [
      { id: 2, name: 'Apple' },
      { id: 3, name: 'Banana' },
    ],
  },
  {
    id: 4,
    name: 'Vegetables',
    children: [
      {
        id: 5,
        name: 'Leafy',
        children: [{ id: 6, name: 'Spinach' }],
      },
    ],
  },
];
```

Recursive Component

```
function NestedList({ items }) {
  return (
    <ul>
      {items.map(item => (
        <li key={item.id}>
          {item.name}
          {item.children && <NestedList items={item.children} />}
        </li>
      ))}
    </ul>
  );
}
```

Usage

```
function App() {
  return <NestedList items={data} />;
}
```

Reuse component in Loop:

In **App.js**:

```
function App() {
    return (
      <div className="App">
        <h1 style={{ color: "red" }}>Hello React JS</h1>
        <h2>Reuse Component with List</h2>
            {
                users.map((item, i)=>
                        <Reuse data = {item} />
                )
            }
      </div>
    );
}
export default App;
```

In **Reuse.js:**

```
function Reuse(props){
    return(
        <div>
            <span>{props.data.name}</span>
            <span>{props.data.city}</span>
            <span>{props.data.contact}</span>
        </div>
    )
}

export default Reuse;
```

Fragments in React:

In React, **Fragments** let you group multiple elements without adding extra nodes to the DOM. This is especially useful when a component needs to return multiple elements, but you don't want to clutter the HTML with unnecessary <div> wrappers.

Two Ways to Use Fragments

1. **Shorthand Syntax**

```
return (
  <>
    <h1>Title</h1>
    <p>Description</p>
  </>
);
```

2. **Explicit Syntax (with key support)**

```
import React from 'react';


return (
  <React.Fragment key={item.id}>
    <h1>{item.title}</h1>
    <p>{item.description}</p>
  </React.Fragment>
);
```

Why Use Fragments?

- **Cleaner DOM**: No extra wrapper elements.
- **Better performance**: Fewer nodes to render.
- **Key support**: Only with <React.Fragment> (not the shorthand <>).

Lifting state up in React:

In React, **lifting state up** means moving state from a child component to a common parent so that multiple components can share and coordinate that state.

Why lift state up?

Imagine two sibling components need to access or update the same data. Instead of duplicating state in both, you:

1. **Move the state to their nearest common ancestor**.
2. **Pass the state and updater functions down via props**.

This ensures a **single source of truth**, keeping your UI consistent and easier to debug.

Example

Let’s say you have two input fields that need to stay in sync:

```
function Parent() {
  const [text, setText] = useState('');
  return (
    <>
      <Input label="First" value={text} onChange={setText} />
      <Input label="Second" value={text} onChange={setText} />
    </>
  );
}

function Input({ label, value, onChange }) {
  return (
    <div>
      <label>{label}</label>
      <input value={value} onChange={e => onChange(e.target.value)} />
    </div>
  );
}
```

Here, the Parent holds the state, and both Input components receive it via props. Any change in one input updates the parent’s state, which then flows down to both inputs—keeping them in sync.