React 18 JSX and Components

Introduction to ReactJS

JSX-JavaScript XML

JSX stands for JavaScript XML. JSX allows us to write HTML in React it makes easier to write and add HTML in React.

JSX allows us to write HTML elements in JavaScript and place them in the DOM without any createElement() and/or appendChild() methods. JSX converts HTML tags into react elements.

Here are two examples. The first uses JSX and the second does not.

Using JSX (React way)

Open `src/main.jsx` and replace with:

```
import ReactDOM from 'react-dom/client'
import React from 'react'


// JSX example
const myElement = <h1>I Love JSX!</h1>

const root = ReactDOM.createRoot(document.getElementById('root'))
root.render(myElement)
```

👉 Run `npm run dev` and open the browser — you'll see **"I Love JSX!"**.

Without JSX (using `React.createElement`)

Still in `src/main.jsx`, try:

```
import ReactDOM from 'react-dom/client'

// Non-JSX example
const myElement = React.createElement('h1', {}, 'I do not use JSX!')

const root = ReactDOM.createRoot(document.getElementById('root'))
root.render(myElement)
```

👉 Refresh, and you'll see **"I do not use JSX!"**.

Why the difference?

- In the **JSX example**, `<h1>I Love JSX!</h1>` is **syntactic sugar** that Vite + Babel transforms into `React.createElement('h1', {}, 'I Love JSX!')` during build.
- In the **non-JSX example**, you write `React.createElement` directly.

So both render the same thing — but JSX is cleaner and preferred.

As you can see in the first example, JSX allows us to write HTML directly within the JavaScript code.

Expressions in JSX (React 18 + Vite)

With JSX you can write **JavaScript expressions** inside curly braces `{ }`.
These expressions can be variables, properties, or any valid JavaScript code. React evaluates the expression and renders the result in the UI.

Example 1: Simple Expression

src/main.jsx

```jsx
import ReactDOM from 'react-dom/client'

// Example with an inline expression
const myElement = <h1>React is {5 + 5} times better with JSX</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'))
root.render(myElement)
```

✅ Output in browser:
React is 10 times better with JSX

Example 2: Using Variables

src/main.jsx

```jsx
import ReactDOM from 'react-dom/client'

const user = "Salah"
const age = 30

const myElement = <h2>Hello, my name is {user} and I am {age} years old.</h2>

const root = ReactDOM.createRoot(document.getElementById('root'))
root.render(myElement)
```

✅ Output:
Hello, my name is Salah and I am 30 years old.

Example 3: Calling Functions

src/main.jsx

```
import ReactDOM from 'react-dom/client'

function formatName(first, last) {
  return `${first} ${last}`
}

const myElement = <h2>Welcome, {formatName("Salah", "Falou")}!</h2>

const root = ReactDOM.createRoot(document.getElementById('root'))
root.render(myElement)
```

✅ Output:
Welcome, Salah Falou!

📌 **Key Point:**

- Anything between `{ }` in JSX is **evaluated as JavaScript**.
- You can use numbers, strings, variables, object properties, function calls, or even inline expressions.
- But you cannot put **statements** (like `if`, `for`) directly inside JSX — only expressions.

🧩 One Top Level Element in JSX

In React, **every component or JSX snippet must return a single top-level element**.
This ensures React can properly manage the virtual DOM tree.

Example 1: Wrapping with a `<div>`

If you want to render multiple elements (like two paragraphs), wrap them inside a parent element:

```
import ReactDOM from 'react-dom/client'

const myElement = (
  <div>
```

```
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </div>
)

const root = ReactDOM.createRoot(document.getElementById('root'))
root.render(myElement)
```

✅ Output:

```
I am a paragraph.
I am a paragraph too.
```

Example 2: Using a Fragment (<> </>)

Sometimes you don't want to add an unnecessary `<div>` to the DOM.
In that case, use a **fragment** (`<> ... </>`):

```
import ReactDOM from 'react-dom/client'

const myElement = (
  <>
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </>
)

const root = ReactDOM.createRoot(document.getElementById('root'))
root.render(myElement)
```

✅ Output is exactly the same — but **without an extra `<div>`** in the DOM.

📌 **Key Point:**

- JSX **must return one parent element**.
- You can use `<div>` (or any container tag) **or** a **fragment** (`<> </>`) if you don't want to add extra HTML nodes.
- This is one of the most common syntax errors beginners encounter in React.

Attribute `class` vs `className` in JSX

In plain HTML, you write:

```
<h1 class="myclass">Hello World</h1>
```

⚠️ But in JSX, you **cannot** use the attribute `class`.
This is because `class` is a reserved word in JavaScript.

Instead, React uses the `className` attribute.
When JSX is compiled, `className` is automatically translated back into the standard `class` attribute in the real DOM.

✅ Example in React 18

```
import ReactDOM from 'react-dom/client'

const myElement = <h1 className="myclass">Hello World</h1>

const root = ReactDOM.createRoot(document.getElementById('root'))
root.render(myElement)
```

👉 The browser DOM will still render:

```
<h1 class="myclass">Hello World</h1>
```

📌 Key Point:

- Always use `className` in JSX.
- It behaves exactly like `class` in HTML but avoids conflicts with JavaScript syntax.
- This is one of the most common beginner mistakes when moving from HTML to React.

Create a Component Manually

In React, components can be created either as **class components** or **functional components**.
Although functional components with hooks are preferred today, class components are still supported.

Create a New Component File

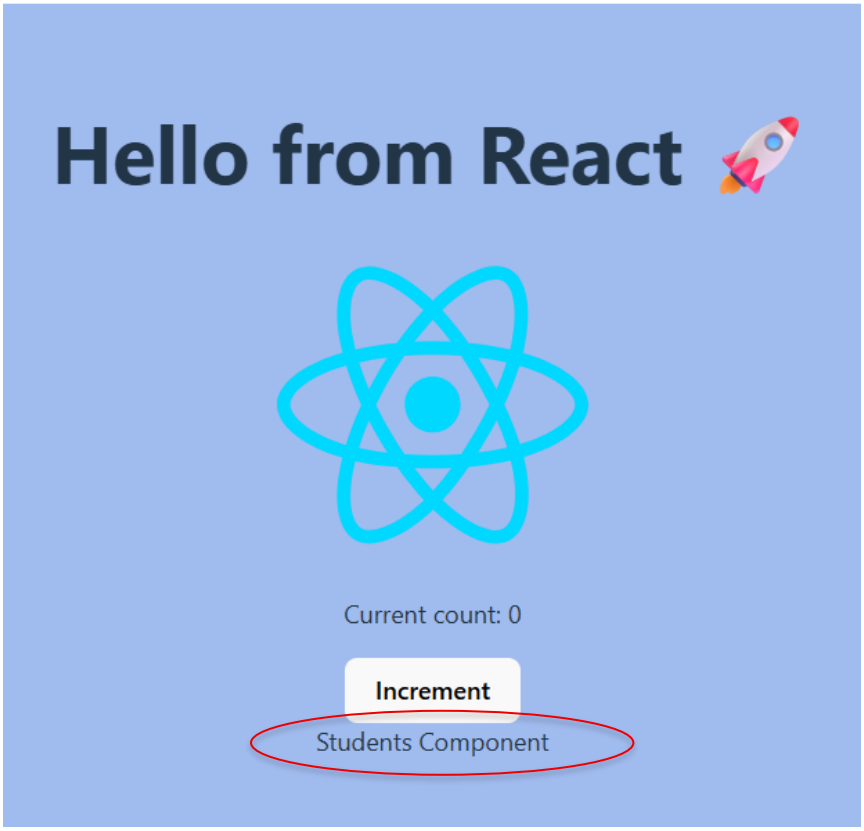
Inside the `/src` folder, create a new file called **`Student.js`**
(note: file names usually start with a capital letter by convention).

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

export default class Student extends Component {
  render() {
    return (
      <div>
        Students Component
      </div>
    )
  }
}
```

Use the Component Inside `App.jsx`

```jsx
import { useState } from "react";
import "./App.css"; // import the stylesheet
import reactLogo from "./assets/react.svg"; // React logo

import Student from "./Student";
function App() {
  const [count, setCount] = useState(0);

  return (
    <div className="container" >
      <h1>Hello from React 🚀</h1>
      <img src={reactLogo} style={{ width: '200px', height: '200px' }} />
      <p>Current count: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
      <Student/>
    </div>
  );
}

export default App;
```

We obtain:

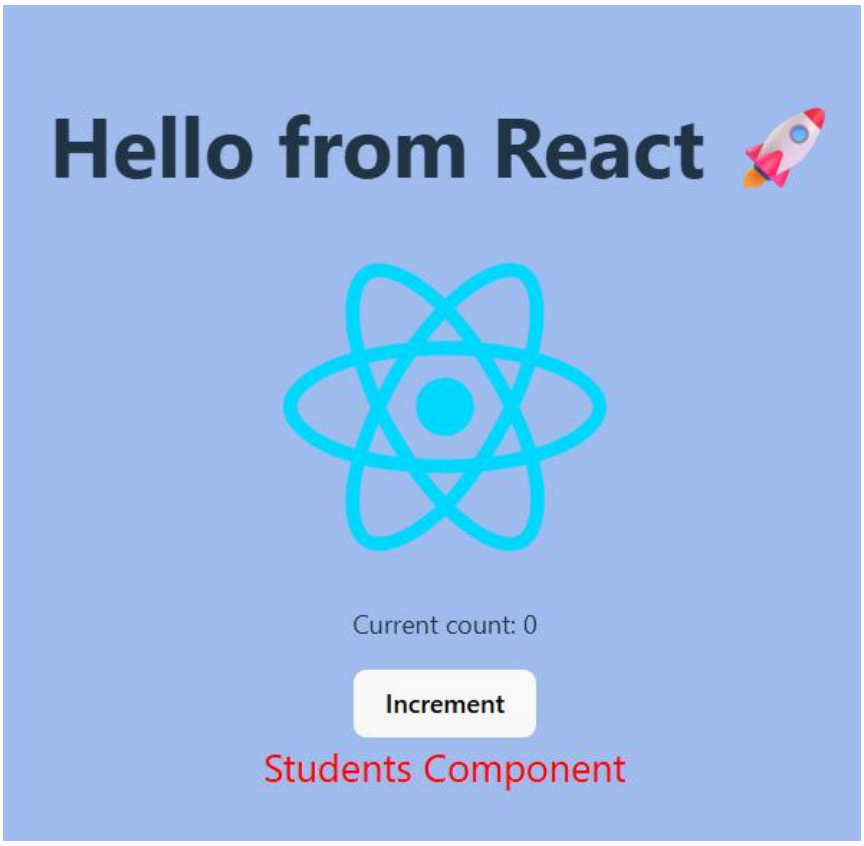


Adding Inline Styles

You can pass inline styles in JSX using an **object** with camelCase keys:

```
return (
  <div style={{ color: 'red', fontSize: '24px' }}>
    Students Component
  </div>
)
```

✅ The `Students` component will now render styled text.



📌 **Key Points:**

- A React app is made of **components** that can be reused.
- Components should start with a **capital letter**.
- Styling in JSX is done with an **object** (`{}`), and property names use camelCase.
- In React 18 projects (like those created with Vite), always render your root component (`App`) using `createRoot` in `main.jsx`.

Recommended: Functional Components

While class components still work, the **React team recommends using functional components with Hooks** for new projects.
They are simpler, more concise, and integrate directly with modern features like `useState`, `useEffect`, and `useContext`.

☑ Students Component (Functional Version)

```
function Students() {
  return (
    <div style={{ color: 'red', fontSize: '24px' }}>
      Students Component
    </div>
  )
}

export default Students
```

And in **App.js**, you use it exactly the same way:

Handling Click Events in React

In React, you can handle user interactions such as button clicks by attaching **event handler functions** to elements.
Event handlers are written in **camelCase** (`onClick`, not `onclick`) and passed inside curly braces `{ }`.

Example: Students Component with Click Event

```
function Student() {
  const name = 'Salah'

  function clickHandler() {
    alert('Button clicked, name: ' + name)
  }

  return (
    <div style={{ color: 'red', fontSize: '24px' }}>
      Students Component
      <button onClick={clickHandler}>Click Me</button>
    </div>
  )
}

export default Student
```

Explanation

- `onClick={clickHandler}` → React will call the function **only when the button is clicked**, not immediately.
- **Note**: React event handlers are written inside curly braces, the function is used without ().
- The function is defined inside the component and can access local variables (`name`).
- Functional components are the **recommended way** to write React components in React 18 because they are simpler and use hooks for state and lifecycle.

✅ When you run this code, clicking the button will show an alert:

Class Component and `setState`

In React class components, the **state** is an object that holds data specific to that component.
The function **`setState()`** is used to schedule an update to the state.

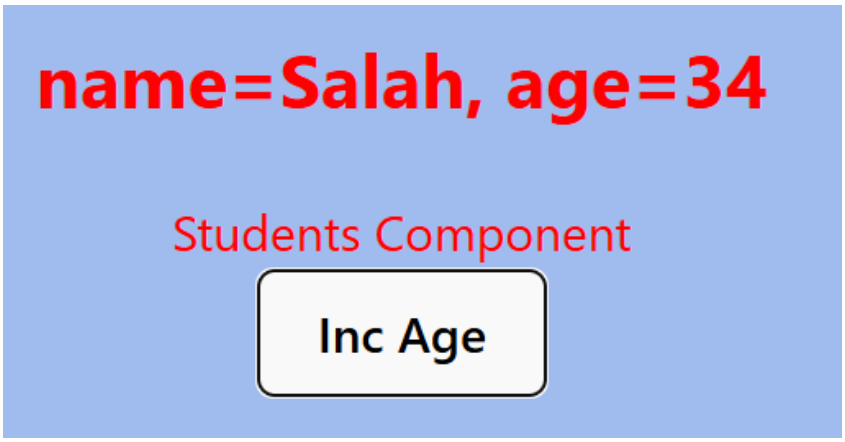
- When state changes, React re-renders the component.
- Calls to `setState()` are **asynchronous**.
  - You should not rely on `this.state` immediately after calling `setState`.
  - React may batch multiple updates together to improve performance.

Example: Increment Age with `setState`

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

export default class Students extends Component {
  state = {
    name: 'Salah',
    age: 30
  }

  clickHandler = () => {
    this.setState({
      age: this.state.age + 1
    })
  }

  render() {
    return (
      <div style={{ color: 'red', fontSize: '24px' }}>
        <h2>
          name={this.state.name}, age={this.state.age}
        </h2>
        Students Component
        <div>
          <button onClick={this.clickHandler}>Inc Age</button>
        </div>
      </div>
```

```
    )
  }
}
```

✅ When you click the button, the `age` property in state increases by 1, and React re-renders the component with the new value.



Executing Code After `setState`

If you want to run some code **after the state has been updated and the component has re-rendered**, pass a callback function as the **second argument** to `setState`:

```
this.setState(
  { age: this.state.age + 1 },
  () => alert("newAge = " + this.state.age)
)
```

Here:

- The callback runs **after** the state update is applied.
- The alert will display the **new age** correctly.

📌 **Key Points for Students:**

- `setState` is asynchronous, so React may delay updates for efficiency.
- Use the **callback parameter** if you need to run code immediately after state changes.
- State changes always trigger a **re-render**.

Create a List Dynamically (Functional Component)

The objective is to define an array of students and display it using the **`map()`** function.
In React, you can build collections of elements and include them in JSX inside curly braces `{ }`.

Example: Students Component with a List

```
import { useState } from 'react'

export default function Students() {
  const [data] = useState([
    { id: 1, name: 'salah', email: 'st1@gmail.com' },
    { id: 2, name: 'omar', email: 'st2@gmail.com' },
    { id: 3, name: 'hadi', email: 'st3@gmail.com' },
    { id: 4, name: 'Saiid', email: 'st4@gmail.com' }
  ])

  const listItems = data.map((stud) => (
    <div key={stud.id}>{stud.name}</div>
  ))

  return <div>{listItems}</div>
}
```

✅ The component loops through the array of student objects and renders each `name` inside a `<div>`.

🔑 Keys in Lists

React requires a **`key`** attribute for elements inside lists.

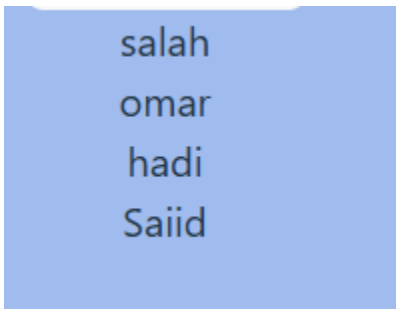
- Keys help React identify which items have changed, been added, or been removed.
- Keys must be **unique** among siblings.
- In this example, we use the `id` property as the key.

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

📌 Key Points for Students:

- Use `map()` to dynamically generate lists of elements.
- Always provide a **unique key** for each list item.
- The key should come from the data itself (like an ID), not from `index` unless no stable ID exists.

After running, we obtain:



Passing Data with Props

We will now move the **student array** into the parent component (`App.jsx`) and pass it down to the child component (`Students.jsx`) via **props**.

🔎 State vs Props

- **State**: Data managed inside a component (like variables that can change).
- **Props** (short for "properties"): Data passed **from parent to child**, similar to function parameters.

✅ App.jsx (Parent Component)

```
import { useState } from 'react'
import './App.css'
import Students from './Students'

function App() {
  const [data] = useState([
    { id: 1, name: 'salah', email: 'st1@gmail.com' },
    { id: 2, name: 'omar', email: 'st2@gmail.com' },
    { id: 3, name: 'hadi', email: 'st3@gmail.com' },
    { id: 4, name: 'Saiid', email: 'st4@gmail.com' }
  ])

  return (
    <div className="App">
      <h1>Hello from React</h1>
      {/* Passing data as props */}
      <Students students={data} />
    </div>
  )
}

export default App
```

✅ Students.jsx (Child Component)

```
export default function Students({ students }) {
  const listItems = students.map((stud) => (
    <div key={stud.id}>{stud.name}</div>
  ))

  return <div>{listItems}</div>
}
```

✅ **Students.jsx (Same code Using props)**

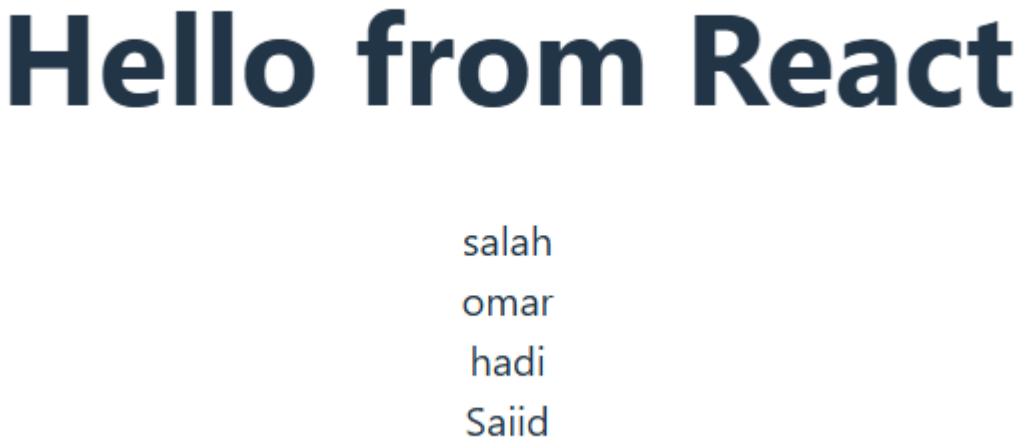
```
export default function Students(props) {
  const listItems = props.students.map((stud) => (
    <div key={stud.id}>{stud.name}</div>
  ))

  return <div>{listItems}</div>
```

```
}
```

📌 Key Points

- Data is stored in the parent (`App.jsx`) using **state**.
- Data is passed down to the child (`Students.jsx`) using **props**.
- The child cannot modify props — it can only use them to display information.
- This flow ensures **unidirectional data flow** (parent → child), which keeps React apps predictable.
- Using `props` (`props.students`) makes it very explicit that data comes **from the parent**.
- Using destructuring (`{ students }`) is just a shorter way to write it, but both are correct.

After running we obtain:



➕ Add Students to the List

We will now add functionality to **dynamically insert new students** into the list.

Step 1 — Create `AddStudent.jsx`

```jsx
import { useState } from 'react'
import { v4 as uuidv4 } from 'uuid'   // install with: npm i uuid

export default function AddStudent(props) {
  const [name, setName] = useState('newStudent')
  const [email, setEmail] = useState('std@gmail.com')

  const handleAddButton = (e) => {
    e.preventDefault()
```

```
    props.addStudentHandler({
      id: uuidv4(),
      name: name,
      email: email
    })
    setName('')
    setEmail('')
  }

  return (
    <table>
      <tbody>
        <tr>
          <td>Name</td>
          <td>
            <input
              type="text"
              value={name}
              onChange={(e) => setName(e.target.value)}
            />
          </td>
        </tr>
        <tr>
          <td>Email</td>
          <td>
            <input
              type="text"
              value={email}
              onChange={(e) => setEmail(e.target.value)}
            />
          </td>
        </tr>
        <tr>
          <td></td>
          <td>
            <button onClick={handleAddButton}>Add</button>
          </td>
        </tr>
      </tbody>
    </table>
  )
}
```

✅ Here, we use **`useState`** to manage the form inputs for `name` and `email`.
✅ When the button is clicked, the new student object is passed up to the parent via `props.addStudentHandler`.

Step 2 — Update `App.jsx`

```jsx
import { useState } from 'react'
import './App.css'
import Students from './Students'
import AddStudent from './AddStudent'

function App() {
  const [data, setData] = useState([
    { id: 1, name: 'salah', email: 'st1@gmail.com' },
    { id: 2, name: 'omar', email: 'st2@gmail.com' },
    { id: 3, name: 'hadi', email: 'st3@gmail.com' },
    { id: 4, name: 'Saiid', email: 'st4@gmail.com' }
  ])

  const addStudent = (student) => {
    setData([...data, student])
  }

  return (
    <div className="App">
      <h1>Hello from React</h1>
      <AddStudent addStudentHandler={addStudent} />
      <Students students={data} />
    </div>
  )
}

export default App
```

✅ The parent `App` manages the `data` state (the student list).
✅ It passes the **function `addStudent`** as a prop to `AddStudent`.
✅ When a new student is added, the `data` array updates and the UI re-renders automatically.

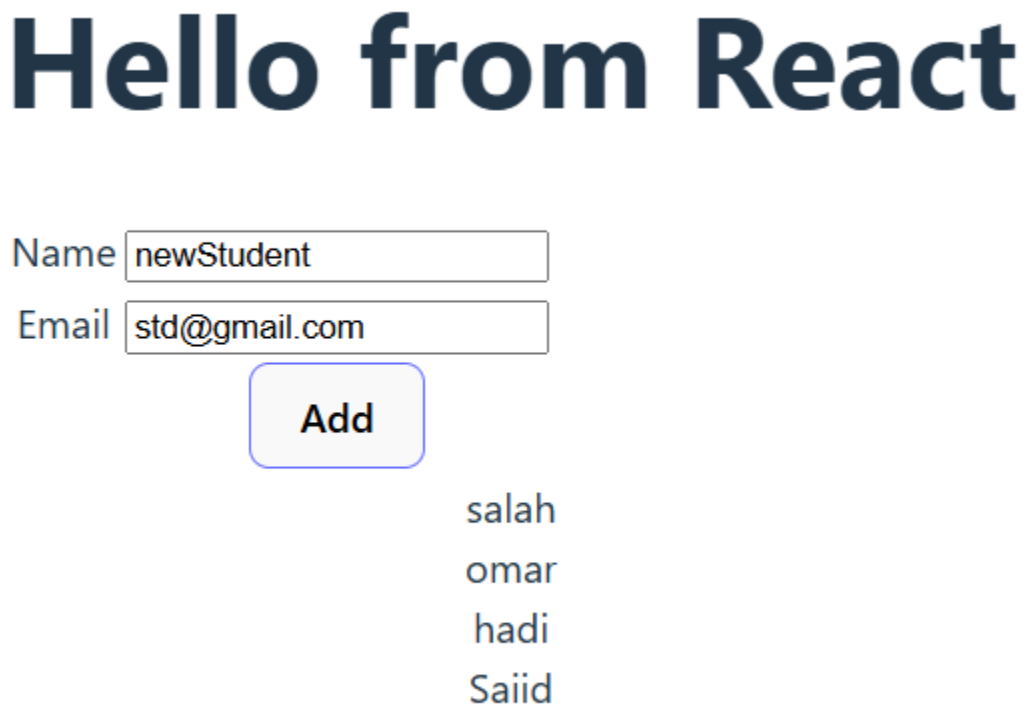
🔑 Notes

- We use **UUID** (`uuidv4()`) to generate a unique ID instead of `Math.random()`.
- Using the **spread operator `[...data, student]`** ensures immutability, which is important for React state updates.
- This flow demonstrates **lifting state up**: the parent holds the state, while the child components (`AddStudent`, `Students`) only interact through props.

📌 **Key Points**

- **Parent (`App`) manages state** (`data`).
- **Child (`AddStudent`) sends new data up** via a function passed as props.
- **Child (`Students`) receives data down** via props and renders it.
- This is the **unidirectional data flow** model of React.

After running we obtain:



Fetching Data from a Placeholder

Instead of manually writing an array of students, we often want to load data from an API or external source. In React, we do this using:

- **`useEffect`** – runs side effects when the component mounts (like fetching data).
- **`fetch()`** – to request data from an API endpoint.
- **`setData()`** – to store the fetched data in state.

We typically:

1. Initialize state as an empty array.
2. Use `useEffect` to call the API **once** when the component mounts.
3. Parse the response (`response.json()`).
4. Update state with the fetched data.

5. Render the data as usual.

✅ Updated Code: Fetch Students from JSONPlaceholder

```
import { useState, useEffect } from 'react'
import './App.css'
import Students from './Students'
import AddStudent from './AddStudent'

function App() {
  const [data, setData] = useState([]) // initially empty

  // ✅ Fetch data from placeholder when component mounts
  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/users')
      .then((response) => response.json())
      .then((users) => {
        // Map only the needed fields (id, name, email)
        const studentsData = users.map((user) => ({
          id: user.id,
          name: user.name,
          email: user.email,
        }))
        setData(studentsData)
      })
      .catch((error) => {
        console.error('Error fetching data:', error)
      })
  }, [])

  // ✅ Add new student manually
  const addStudent = (student) => {
    setData([...data, student])
  }

  return (
    <div className="App">
      <h1>Hello from React</h1>
      <AddStudent addStudentHandler={addStudent} />
      <Students data={data} />
    </div>
  )
}

export default App
```

🔍 Explanation of New Parts

- **`useEffect`**: Runs after the first render (componentDidMount equivalent).
- **`fetch('https://jsonplaceholder.typicode.com/users')`**: Requests 10 fake user objects.
- **`.then(response => response.json())`**: Converts the raw response into JSON.
- **`map()`**: Extracts only the fields we need (`id`, `name`, `email`).
- **`setData(studentsData)`**: Stores the result in state → causes a re-render.

After running we obtain: