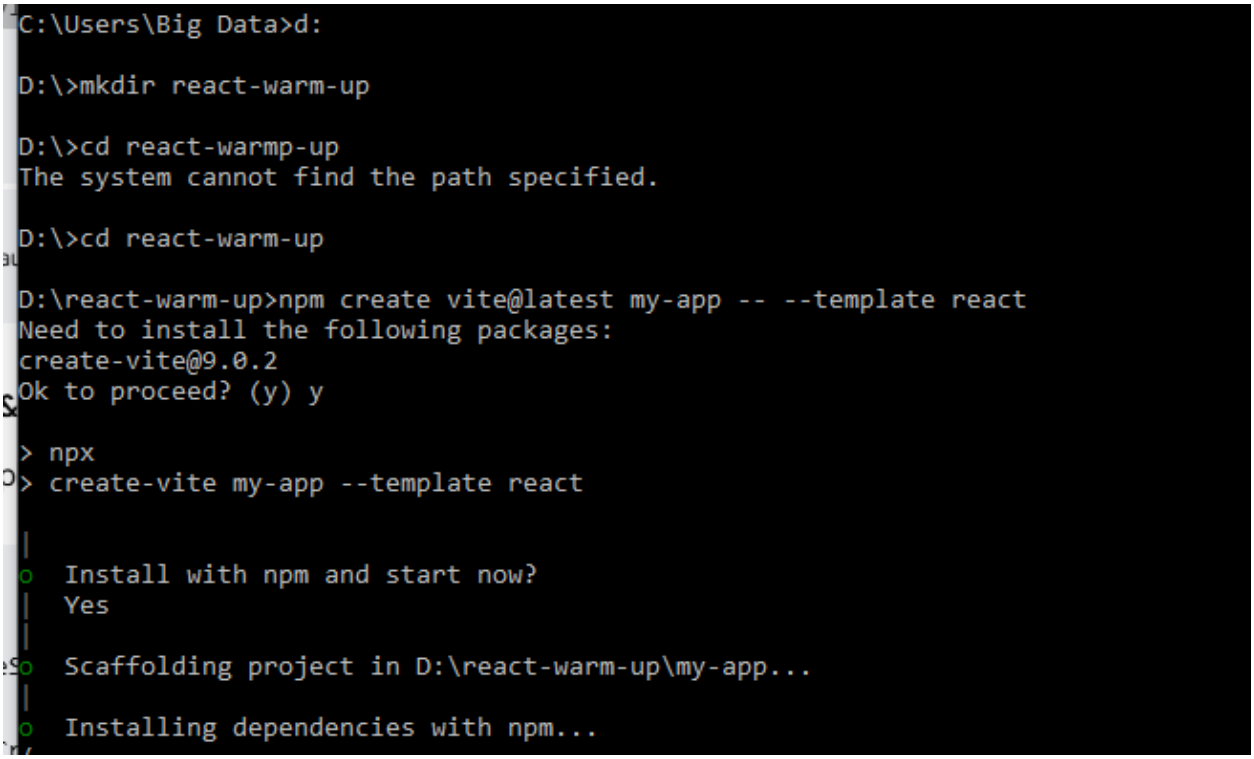
The Practice Code react js code:-

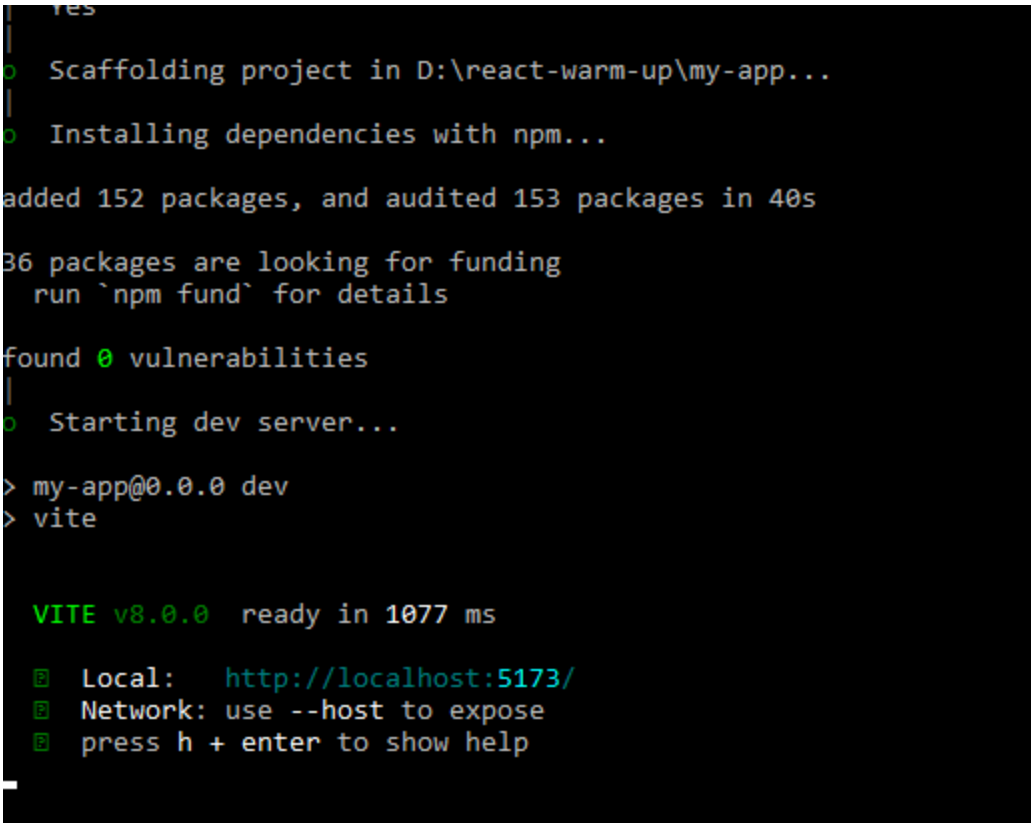
1. Setup Command (Terminal)

Students should memorize the Vite setup as it's much faster than older methods:

```
npm create vite@latest my-app -- --template react
```

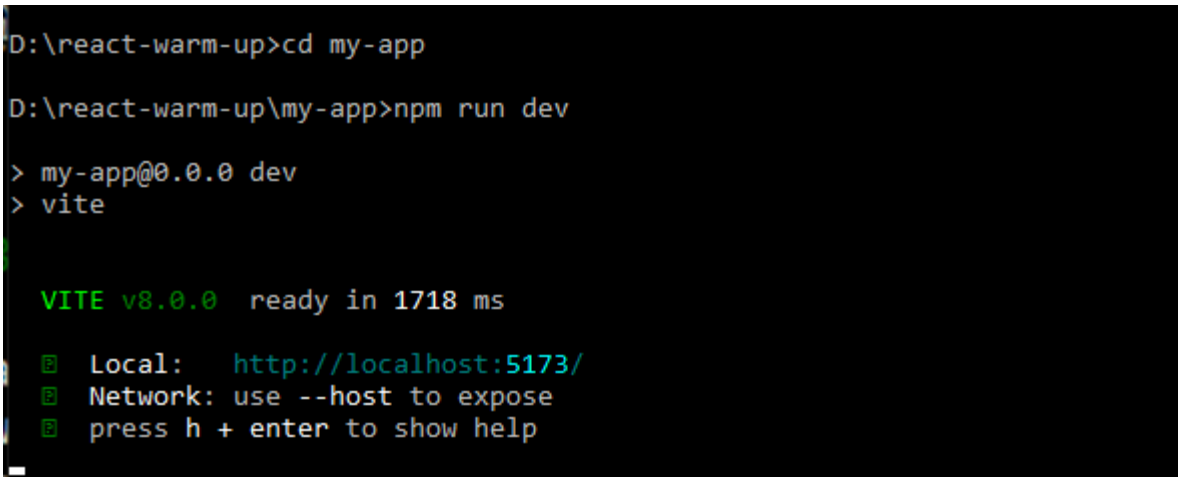


And you will see



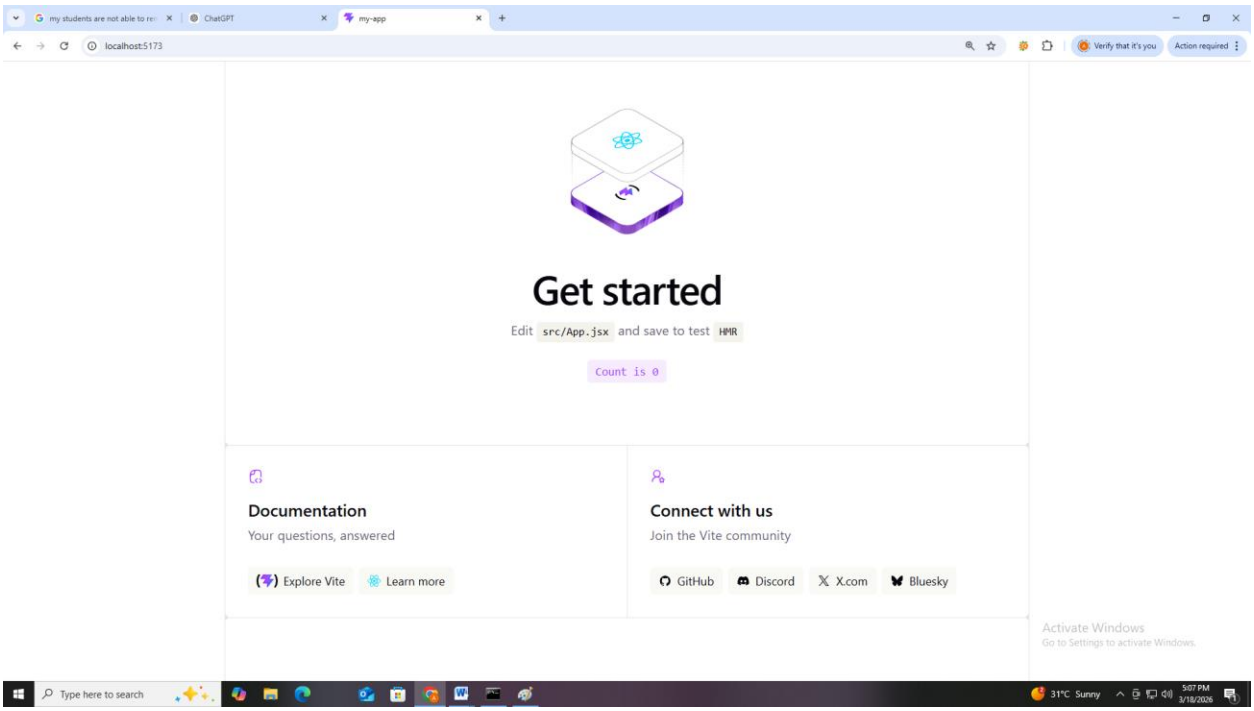
To Run :-

```
cd my-app
npm run dev
```



output:-

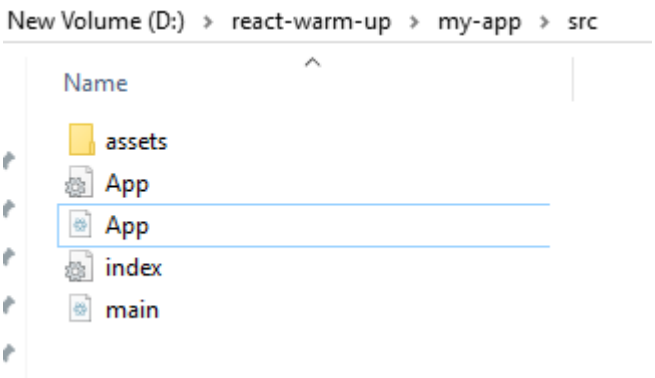
http://localhost:5173/



2. The "State & Props" Component

In `App.jsx`, replace everything with this fundamental pattern:

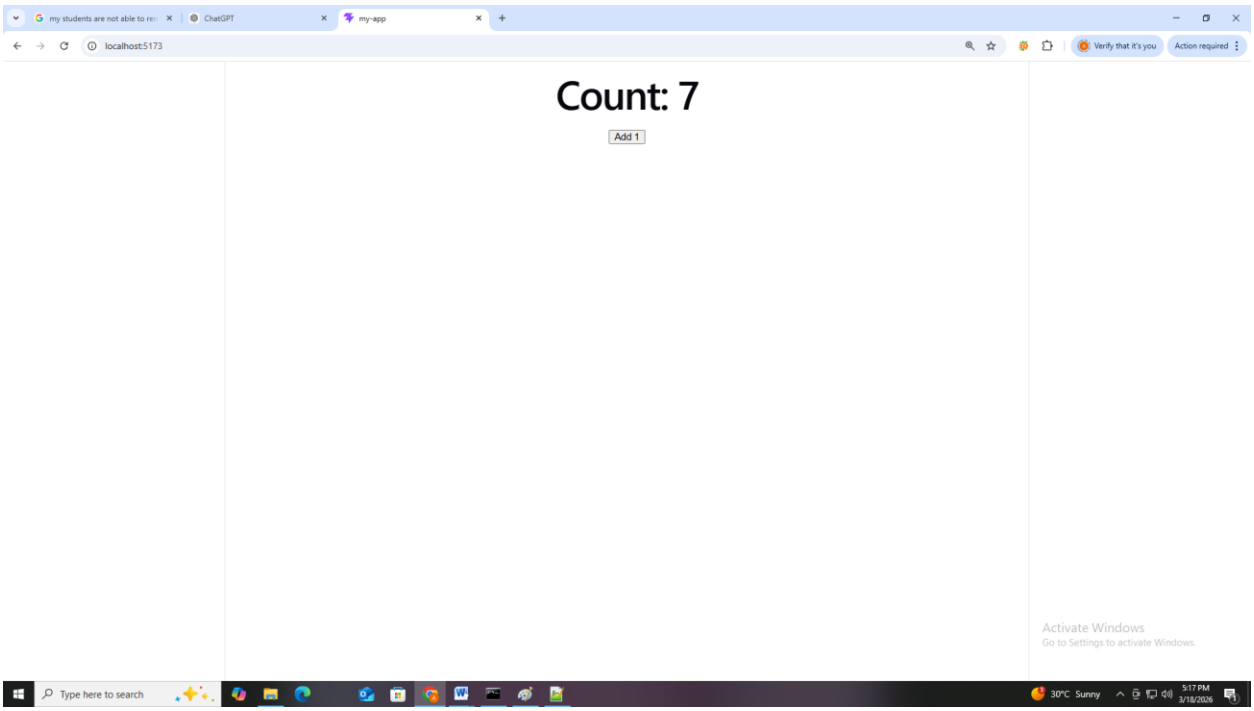
Now change code of App.jsx file :-




```
import { useState, useEffect } from 'react'


// Step A: Create a Child Component to practice Props
```

```
function Display(props) {
  return <h1>Count: {props.value}</h1>
}

export default function App() {
  // Step B: Initialize State
  const [count, setCount] = useState(0)

  // Step C: Practice Side Effects
  useEffect(() => {
    console.log("Component loaded!")
  }, []) // Empty dependency array means it runs once

  return (
    <div>
      <Display value={count} />
      <button onClick={() => setCount(count + 1)}>
        Add 1
      </button>
    </div>
  )
}
```

Output:-



Note to understand App.jsx code in details :-

- **Props** is short for **"Properties."**
- They are **Arguments** passed into React components (just like arguments in a JavaScript function).
- **Crucial Rule:** Props are **Read-Only** (Immutable). A child component can *receive* a prop, but it is not allowed to *change* it. Only the parent can change the value.

Parent (`App`) stays exactly the same—only the way the **Child (`Profile`)** handles the data changes.

The Common Parent (The Data Sender)

Students should write this first. It sends two "labels" (props): `name` and `role`.

```
// This Parent component works for BOTH examples below
export default function App() {
  return (
    <div>
      <h1>Team Directory</h1>
      {/* Sending 'name' and 'role' to the child */}
      <Profile name="Rahul" role="Developer" />
      <Profile name="Sneha" role="Designer" />
    </div>
  );
}
```

Version 1: The "Old" Way (Using `props` Object)

Concept: We receive one big object called `props` and use "Dot Notation" to look inside it.

```
function Profile(props) {
  // We have to type 'props.' every single time
  return (
    <div style={{ border: "1px solid black", margin: "10px" }}>
      <h2>Name: {props.name}</h2>
      <p>Role: {props.role}</p>
    </div>
  );
}
```

Use code with caution.

Version 2: The "Modern" Way (Destructuring)

Concept: We "unpack" the object immediately in the parentheses. This is the industry standard.

```
// We pick out exactly what we need using { }
function Profile({ name, role }) {
  // No 'props.' prefix needed here!
```

```
  return (
    <div style={{ border: "1px solid blue", margin: "10px" }}>
      <h2>Name: {name}</h2>
      <p>Role: {role}</p>
    </div>
  );
}
```

Practice Plan for Students (Step-by-Step):-

1. **Type Version 1:** Have them run the code. Ask them to add a third prop, like `city`.

   Notice they have to change the code in **two** places: the Parent and the `props.city` line.

2. **Convert to Version 2:** Have them delete the `Profile` function and rewrite it using `{ name, role, city }`.

3. **The "Ah-ha!" Moment:** Point out how much cleaner the code looks without `props.props.props.` everywhere

Here is the breakdown of `const [count, setCount] = useState(0)` and how the button actually changes it.

1. The Anatomy of the Line

Think of this line as a **contract** with React:

Part	Name	Real-World Analogy
`count`	**The State Variable**	The **Display Screen** on a calculator. It shows the current value.
`setCount`	**The Setter Function**	The **Remote Control**. It is the *only* way to change what's on the screen.
`useState`	**The Hook**	The **Memory Chip**. It tells React: "Remember this number even if the page refreshes."
`(0)`	**Initial Value**	The **Starting Point**. When the app first opens, the number is 0.

And :-

"Trigger and an Action." Here is the breakdown of every character in that line:-

```
<button onClick={() => setCount(count + 1)}>
        Add 1
      </button>
```

1. The Breakdown

Code Segment	Name	Plain-English Meaning
`<button ... >`	**HTML Tag**	Creates the physical button on the screen.
`onClick={...}`	**Event Listener**	"Hey React, **listen** for when the user clicks this button."
`() =>`	**Arrow Function**	"When the click happens, **then** run the code inside these brackets."
`setCount(...)`	**The Setter**	"Use the 'Remote Control' to change the memory."
`count + 1`	**The Logic**	"Take the current number and add 1 to it."
`Add 1`	**Label**	The text the user actually sees on the button.

This 3-week plan is designed to turn React syntax into **muscle memory**. The rule for students is: **No Copy-Pasting.** They must type every bracket and semicolon.

Week 1: The "Counter" Phase (State & Hooks)

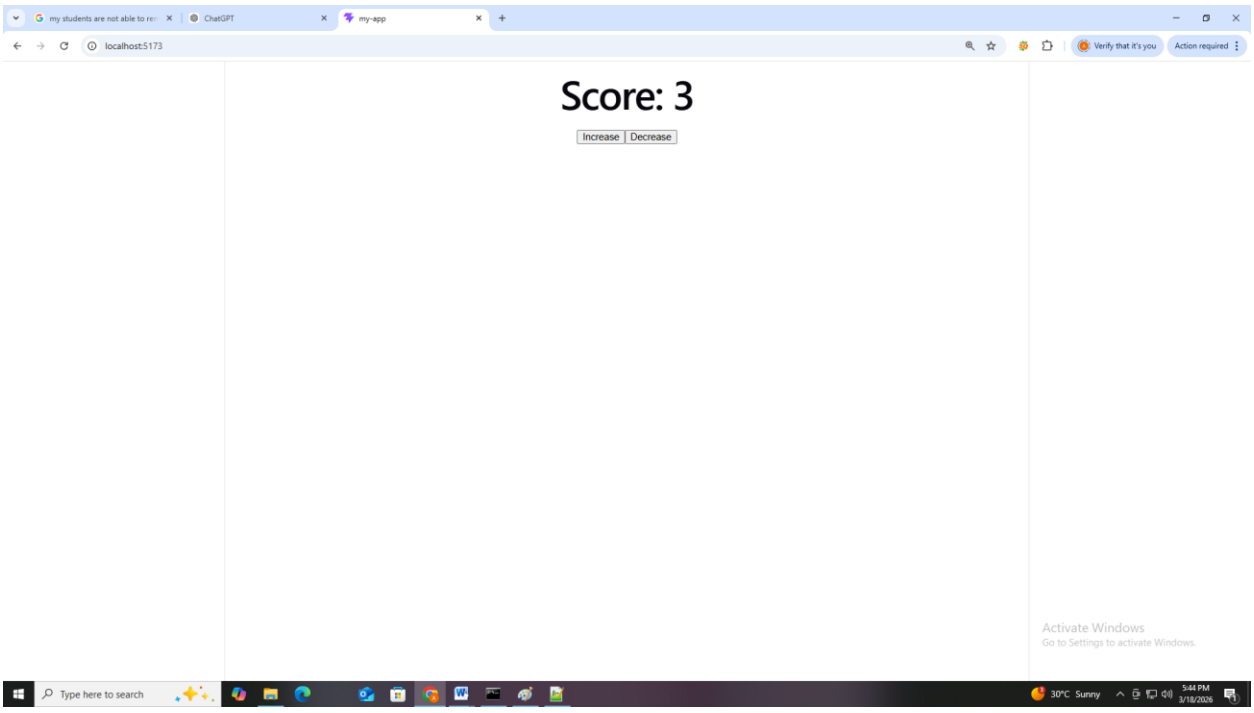
Goal: Master `useState`, `onClick` events, and the basic component structure.

- **Step 1:** Import `useState` at the very top.
- **Step 2:** Initialize a `count` variable and a `setCount` function.
- **Step 3:** Create a button that uses an **arrow function** in the `onClick` handler.

The Drill Code (`App.jsx`):-

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

function App() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <h1>Score: {count}</h1>
      <button onClick={() => setCount(count + 1)}>Increase</button>
      <button onClick={() => setCount(count - 1)}>Decrease</button>
    </div>
  );
}

export default App;
```

Output:-


my students are not able to re
ChatGPT
my-app
localhost:5173
Verify that it's you
Action required
Score: 3
Increase
Decrease
Activate Windows
Go to Settings to activate Windows.
Type here to search
30°C Sunny
5:44 PM
3/18/2026

Week 2: The "Input" Phase (Controlled Components)

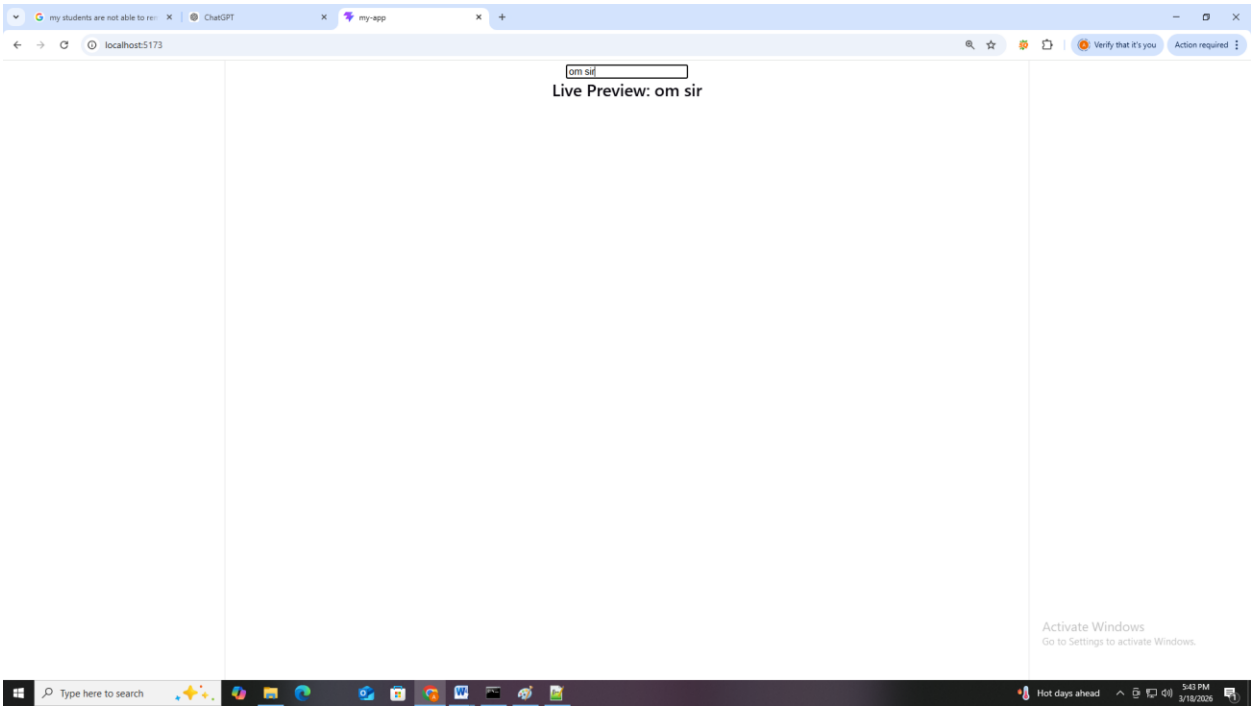
Goal: Understand how React handles forms and real-time data binding using `e.target.value`.

- **Step 1:** Create a state variable called `name` (initial value should be an empty string `""`).
- **Step 2:** Create an `<input>` tag.
- **Step 3:** Link the `value` of the input to your state.
- **Step 4:** Use `onChange` to update the state as the user types.

The Drill Code (`App.jsx`):

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

function App() {
  const [text, setText] = useState("");

  return (
    <div>
      <input
        type="text"
        placeholder="Type something..."
        value={text}
        onChange={(e) => setText(e.target.value)}
      />
      <h2>Live Preview: {text}</h2>
    </div>
  );
}

export default App;
```

Output:-


my students are not able to rea
ChatGPT
my-app
localhost:5173
Verify that it's you
Action required
om sir
Live Preview: om sir
Activate Windows
Go to Settings to activate Windows.
Type here to search
Hot days ahead
5:43 PM
3/18/2026

To handle **multiple inputs** (like Name, Email, and Password), you have two choices. You can either create many `useState` hooks or use **one single Object** to store everything.
Using **one Object** is the professional way because it keeps your code clean and scalable.

The "Modern Object" Method (Week 2 - Advanced)

Goal: Master the `[e.target.name]: e.target.value` trick. This allows one function to handle 100 inputs.

Step 1: The Initialization (The "Form State")

Instead of a string, we start with an **Object** `{ }`.

```
const [formData, setFormData] = useState({
  username: "",
  email: "",
  password: ""
});
```

Step 2: The "Universal" Handler

This is the "magic" function. It looks at the **`name`** attribute of the input to decide which part of the state to update.

```
const handleChange = (e) => {
  const { name, value } = e.target; // Get name and value from the input
  setFormData({
    ...formData, // Keep the old data (Spread Operator)
    [name]: value // Update ONLY the one that changed
  });
};
```

Use code with caution.

The Full Practice Code (`App.jsx`)

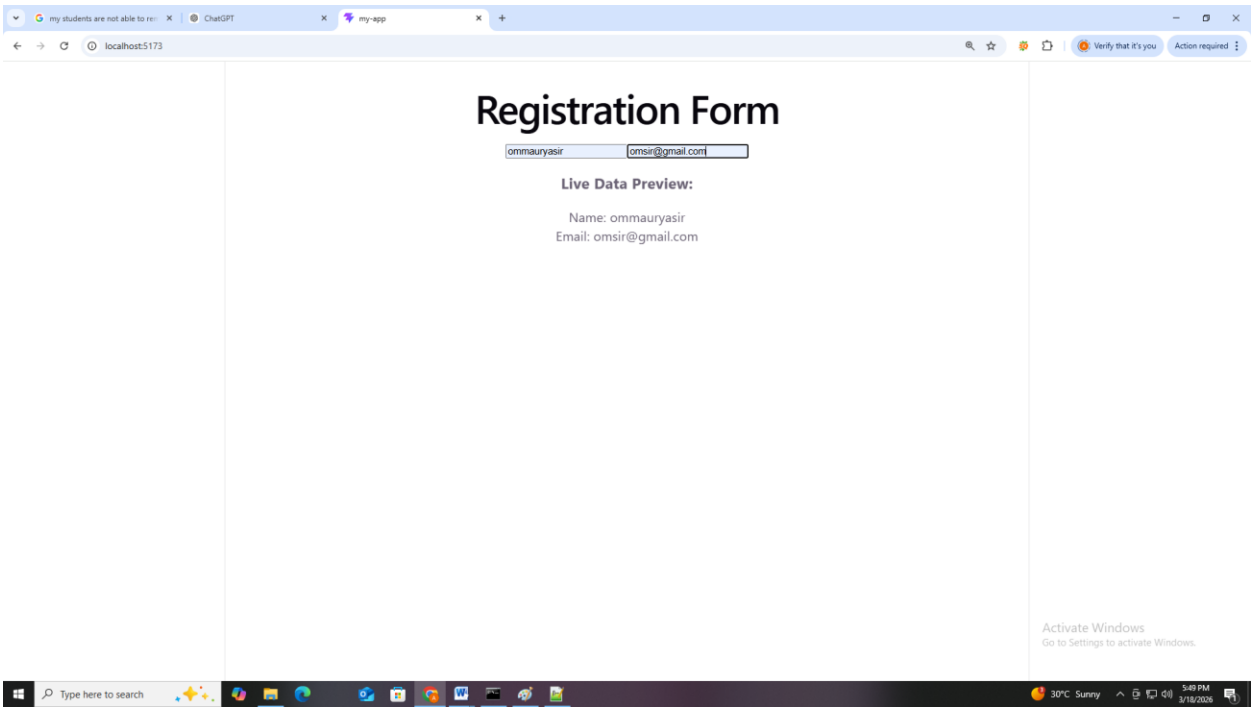
```
import { useState } from 'react';

function App() {
  // 1. Initial State as an Object
  const [formData, setFormData] = useState({
    username: "",
    email: ""
  });

  const handleChange = (e) => {
    // 2. The Universal Update Logic
    setFormData({
      ...formData,
      [e.target.name]: e.target.value
    });
  };

  return (
    <div style={{ padding: '20px' }}>
      <h1>Registration Form</h1>

      {/* Input 1: Name */}
      <input
        type="text"
        name="username"  // IMPORTANT: This must match the state key
        placeholder="Enter Name"
        value={formData.username}
        onChange={handleChange}
      />

      {/* Input 2: Email */}
      <input
        type="email"
        name="email"     // IMPORTANT: This must match the state key
        placeholder="Enter Email"
        value={formData.email}
```

```
        onChange={handleChange}
      />

      <div style={{ marginTop: '20px' }}>
        <h3>Live Data Preview:</h3>
        <p>Name: {formData.username}</p>
        <p>Email: {formData.email}</p>
      </div>
    </div>
  );
}

export default App;
```

Breakdown details :-

1. **The `name` Attribute:** Explain that the `name="username"` on the HTML tag acts like a **label** so the `handleChange` function knows where to put the text.
2. **The `...formData` (Spread Operator):** This is vital. Tell them: "If you don't use `...formData`, when you type the Email, the Name will disappear! The spread operator **copies** the old data first."
3. **The `[ ]` Brackets:** Explain that `[e.target.name]` is "Computed Property Name." It's like saying: "Find the key that matches the input's name."

Output:-



Week 3: The "List" Phase (Mapping & Keys)

Goal: Master the `.map()` function and understand why `key` props are required in React.

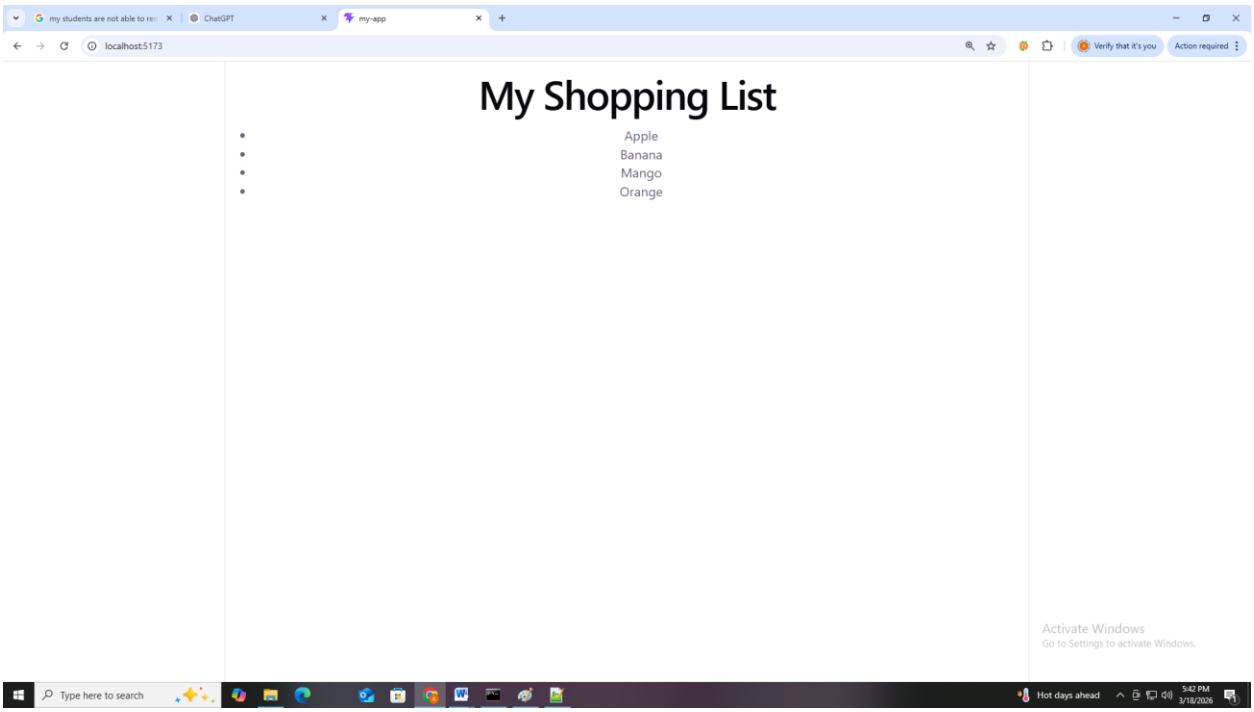
- **Step 1:** Create an array of strings (e.g., a list of fruits or names).
- **Step 2:** Use `{items.map()}` inside your JSX.
- **Step 3:** Return a `<li>` for every item.
- **Step 4: Crucial:** Assign a unique `key` to each list item.

The Drill Code (`App.jsx`):-

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

```
function App() {
  const [fruits] = useState(["Apple", "Banana", "Mango", "Orange"]);

  return (
    <div>
      <h1>My Shopping List</h1>
      <ul>
        {fruits.map((fruit, index) => (
          <li key={index}>{fruit}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

oputput:-



Week 4 Lifecycle & API" Phase with Axios :-

Vite is perfect for beginners because it's fast and simple. Let's turn your code into a **hands-on practice project using Vite**, step by step.

Step 1: Create a Vite React App

Open terminal and run:

```
npm create vite@latest my-app
```

Then:

```
cd my-app
npm install
npm install axios
npm run dev
```

Open the URL shown (usually `http://localhost:5173`)

Step 2: Clean the Project

Go to `src/App.jsx` and **replace everything** with your code.

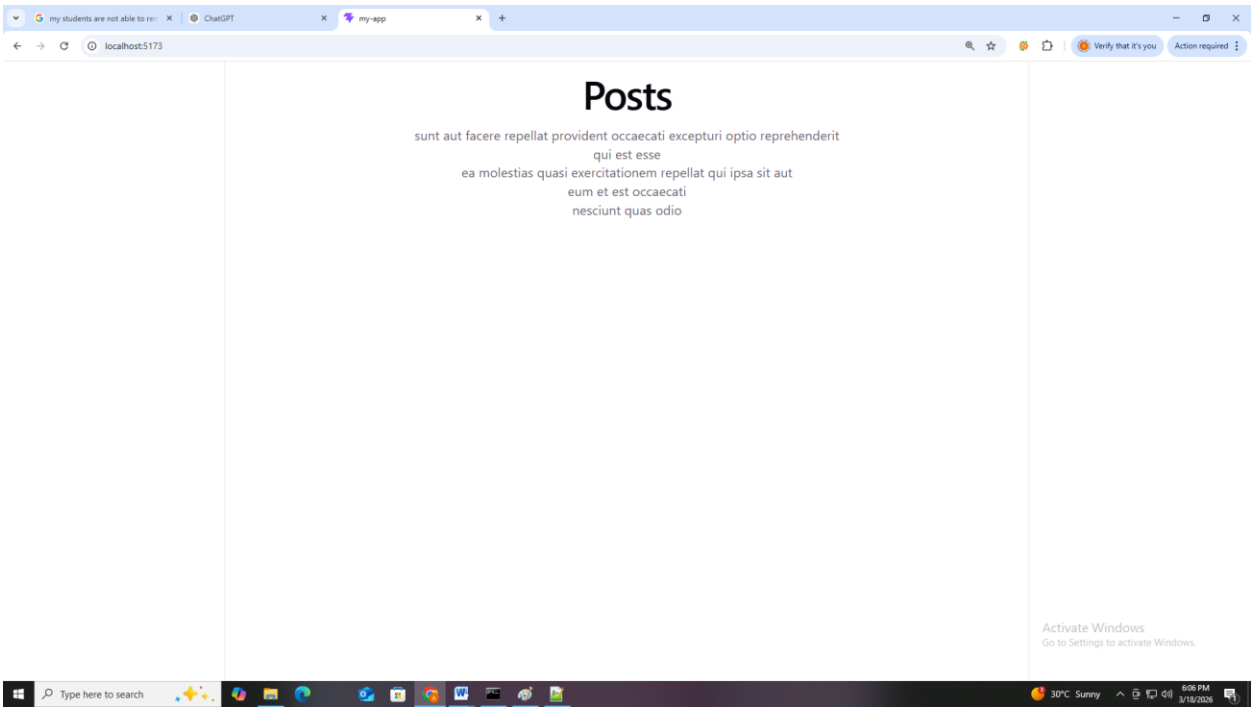
Step 3: Start With a Simple Version (Practice First)

Before your full code, try this basic version:

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

function App() {
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/posts')
      .then(res => res.json())
      .then(data => setPosts(data.slice(0, 5)));
  }, []);

  return (
    <div>
      <h1>Posts</h1>
      {posts.map(post => (
        <p key={post.id}>{post.title}</p>
      ))}
    </div>
  );
}

export default App;
```

 Run it and confirm it works.

Output:-



Here's your **same code converted to Axios version** (clean and beginner-friendly)

Axios Version (App.jsx):-

```
import { useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    axios
      .get('https://jsonplaceholder.typicode.com/posts')
      .then((response) => {
        setPosts(response.data.slice(0, 5));
      })
      .catch((error) => {
        console.error("Error fetching data:", error);
```

```
      });
  }, []);

  return (
    <div>
      <h1>Posts</h1>
      {posts.map(post => (
        <p key={post.id}>{post.title}</p>
      ))}
    </div>
  );
}

export default App;
```

□ What Changed (Simple Explanation)

1. Import Axios

```
import axios from 'axios';
```

---

2. Replace `fetch()` with `axios.get()`

```
axios.get('https://jsonplaceholder.typicode.com/posts')
```

---

3. No need for `.json()`

□ fetch:

```
res.json()
```

□ axios:

```
response.data
```

□ Quick Practice (Do This)

Try modifying:

□ Show 3 posts only

```
slice(0, 3)
```

□ Log data

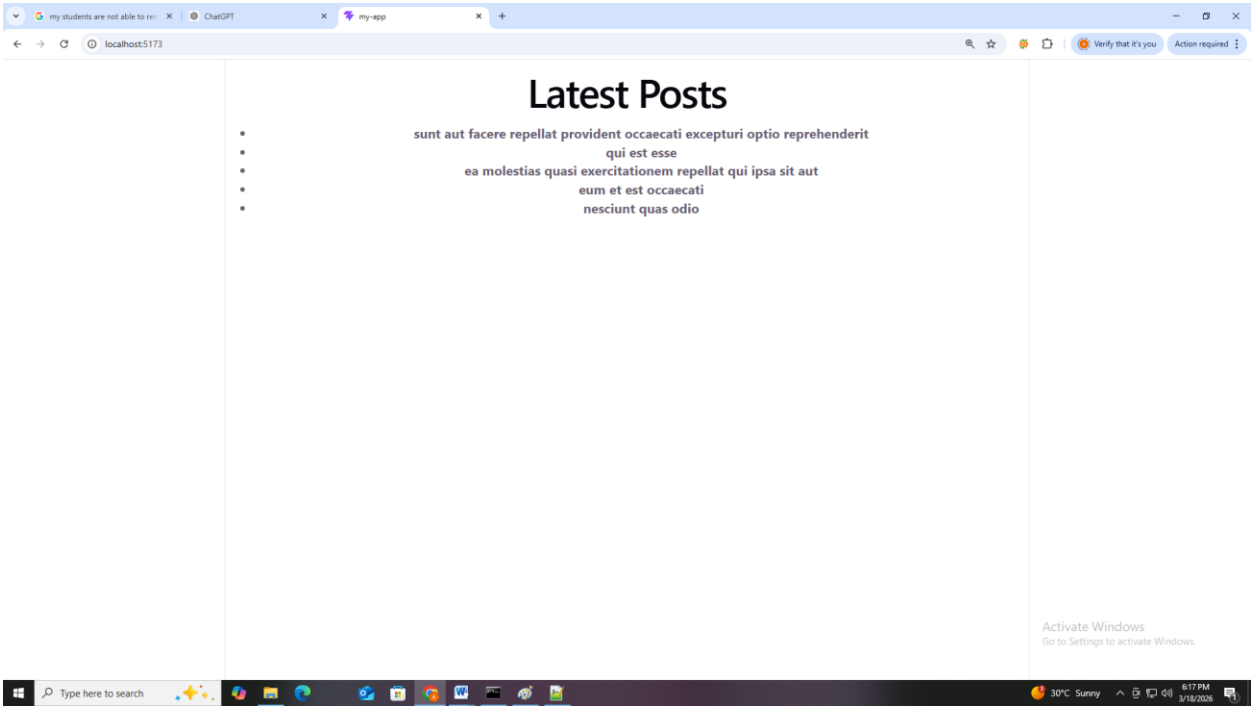
```
console.log(response.data);
```

Best way for practice axios version:-

```
import { useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [posts, setPosts] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const fetchData = async () => {
      try {
        // Correct endpoint
        const response = await axios.get(
          'https://jsonplaceholder.typicode.com/posts'
        );

        // Ensure data exists
        if (response.data) {
          setPosts(response.data.slice(0, 5)); // first 5 posts
        }

        setLoading(false);
      } catch (error) {
```

```
      console.error("Data fetch failed!", error);
      setLoading(false);
    }
  };

  fetchData();
}, []);

if (loading) return <h1>Loading data from server...</h1>;

return (
  <div>
    <h1>Latest Posts</h1>
    <ul>
      {posts.map((post) => (
        <li key={post.id}>
          <strong>{post.title}</strong>
        </li>
      ))}
    </ul>
  </div>
);
}

export default App;
```

output:-


my students are not able to ren
ChatGPT
my-app
localhost:5173
Verify that it's you
Action required
Latest Posts
sunt aut facere repellat provident occaecati excepturi optio reprehenderit
qui est esse
ea molestias quasi exercitationem repellat qui ipsa sit aut
eum et est occaecati
nesciunt quas odio
Activate Windows
Go to Settings to activate Windows.
Type here to search
30°C Sunny
6:17 PM
3/18/2026

Week 5: The "Routing" Phase:-

Goal: Turn the single-page app into a multi-page website using React Router.

- **Step 1:** Install the library: `npm install react-router-dom`.
- **Step 2:** Practice the `BrowserRouter`, `Routes`, and `Route` components.
- **Step 3:** Use the `<Link>` component instead of `<a>` tags to prevent page refreshes.

The Drill Code (`main.jsx` / `App.jsx`):

```
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';


function Home() {
  return <h1>Home Page</h1>;
}


function About() {
  return <h1>About Page</h1>;
}


function App() {
  return (
    <BrowserRouter>
      <div>
        <nav style={{ marginBottom: '20px' }}>
          <Link to="/">Home</Link> | <Link to="/about">About</Link>
        </nav>


        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/about" element={<About />} />
```

```
        </Routes>
      </div>
    </BrowserRouter>
  );
}


export default App;
```

The goal is to make it easy to understand how routing works in React without overwhelming them with professional structure.

Step-by-Step Beginner Practice: React Router

Step 1: Create a Vite React App

Open your terminal and run:

```
npm create vite@latest my-router-app
cd my-router-app
npm install
npm install react-router-dom
npm run dev
```

- This installs React + Vite + React Router

□ Step 1: Import React Router

In your `App.jsx`, import the necessary components:

```
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';
import Home from './pages/Home';
import About from './pages/About';
```

- `BrowserRouter` → Wraps your app to enable routing
- `Routes` → Container for all route definitions
- `Route` → Defines individual routes
- `Link` → Navigation links (instead of `<a>` tags)

□ Step 2: Wrap the App with BrowserRouter

Inside `App.jsx`, **wrap everything in `<BrowserRouter>`**:

```
function App() {
  return (
    <BrowserRouter>
      {/* App content will go here */}
    </BrowserRouter>
  );
}

export default App;
```

- This ensures `<Link>` and `<Routes>` work
- If you forget this → `useContext(...) is null` error

□ Step 3: Add a Navigation Menu

Inside `<BrowserRouter>`, add a simple menu using `<Link>`:

```
<nav>
  <Link to="/">Home</Link> |{" "}
  <Link to="/about">About</Link>
</nav>
```

- `to="/"` → navigates to Home
- `to="/about"` → navigates to About
- **No page reload** happens, unlike `<a>`

□ Step 4: Define Routes

Below the navigation, add your `<Routes>`:

```
<Routes>
  <Route path="/" element={<Home />} />
  <Route path="/about" element={<About />} />
</Routes>
```

- `path` → the URL path
- `element` → the component to render

□ Step 5: Full App.jsx Code

```
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';
import Home from './pages/Home';
import About from './pages/About';

function App() {
  return (
    <BrowserRouter>
      <div>
        <nav>
          <Link to="/">Home</Link> |{" "}
          <Link to="/about">About</Link>
        </nav>

        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/about" element={<About />} />
        </Routes>
      </div>
    </BrowserRouter>
  );
```

```
}

export default App;
```

□ Step 6: Create Simple Page Components

src/pages/Home.jsx:-

```
function Home() {
  return <h1>Home Page</h1>;
}

export default Home;
```

src/pages/About.jsx:-

```
function About() {
  return <h1>About Page</h1>;
}

export default About;
```

□ Step 7: Run the App

```
npm run dev
```

Visit:

- `/` → Home Page
- `/about` → About Page

Click the links and notice **no page reload** — this is client-side routing.

□ Step 8: Teaching Notes

- Emphasize **BrowserRouter must wrap Links & Routes**
- Explain the difference between `<a>` and `<Link>`
- Mention that **Option 2 is fine for small apps**, but in real projects, `BrowserRouter` should go in `main.jsx`

Step 9 (Optional Beginner Challenge)

1. Add a **Contact Page**: `/contact`
2. Add a **Not Found Page** for unknown routes using `*` path:

```
<Route path="*" element={<h1>Page Not Found</h1>} />
```

Let's make a **clear step-by-step diagram** for **Option 1 (Professional Way)** so beginners can visualize how React Router works in a Vite React app.

□ React Router (Option 1) Flow Diagram

```
main.jsx
 ├─ <StrictMode>              ← Optional, helps with React warnings
 │
 └─ <BrowserRouter>           ← Provides routing context to the app
      │
      └─ <App />              ← Your main app component
           │
           ├─ <nav>           ← Navigation links (<Link>)
           │    ├─ <Link to="/">Home</Link>
           │    └─ <Link to="/about">About</Link>
           │
           └─ <Routes>        ← Defines all routes
                ├─ <Route path="/" element={<Home />} />
                └─ <Route path="/about" element={<About />} />
                          │
                          └─ Home.jsx / About.jsx → Rendered components
```

How to Read This Diagram

1. **main.jsx**
   - Entry point of your Vite app
   - Wraps everything in `<BrowserRouter>` → necessary for routing
2. **BrowserRouter**
   - Provides React Router context
   - All `<Link>` and `<Routes>` inside it will work
3. **App.jsx**
   - Contains **navigation menu** (Links) and **route definitions** (Routes)
4. **Routes + Route**
   - `<Routes>` = container for all route definitions
   - `<Route path="/" element={<Home />} />` = "when URL is `/`, render `Home`"
5. **Page Components**
   - `Home.jsx` and `About.jsx` are just simple components rendered by routes

---

□ Quick Beginner Check

- Click a `<Link>` → URL changes without reloading the page □

- Type URL `/about` → About page appears
- Remove `<BrowserRouter>` → `useContext(...) is null` error appears

Step-by-Step Guide with Code

Step 1: Create Vite React App

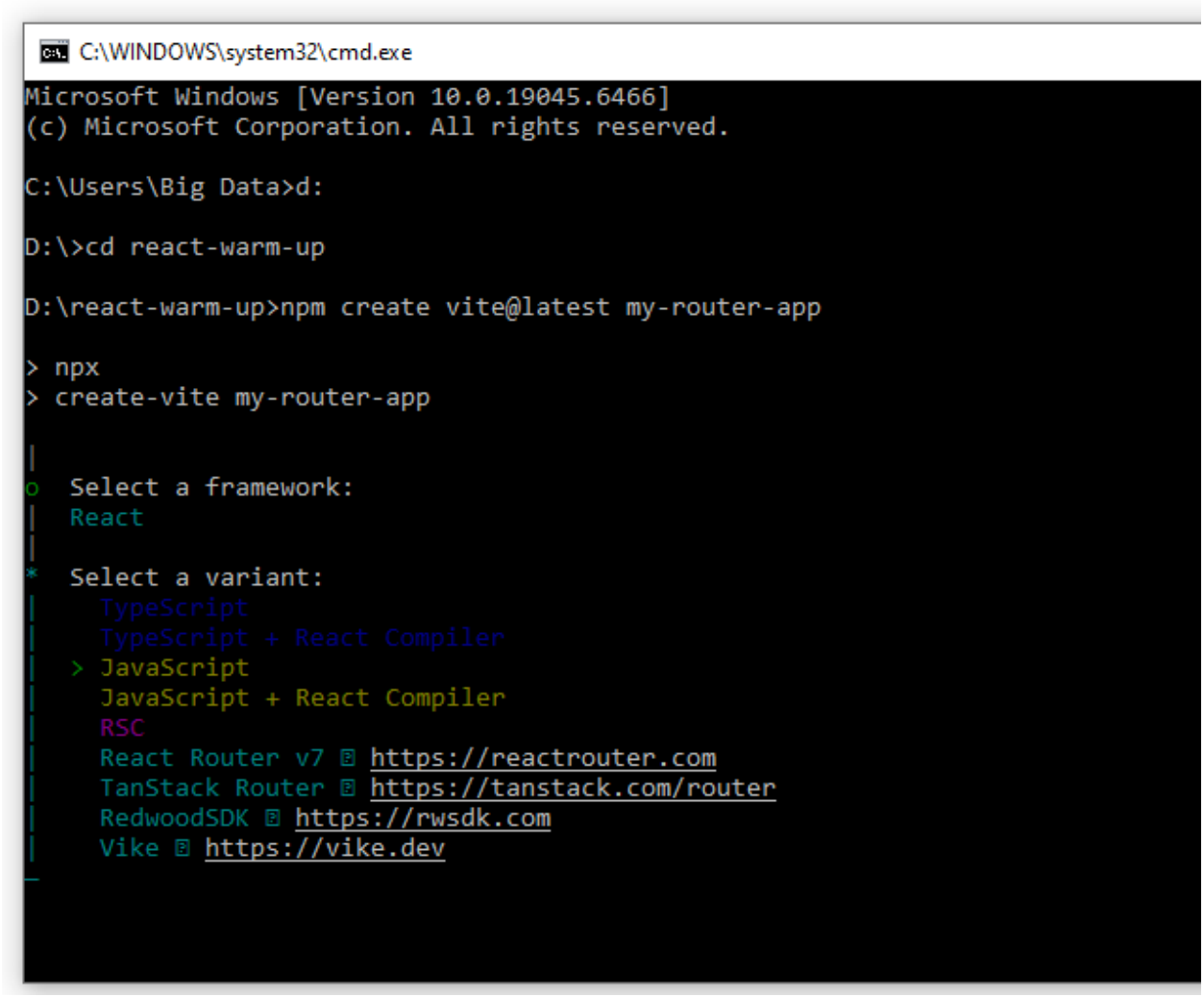
Open terminal:

```
npm create vite@latest my-router-app
```

- When prompted:
  - **Project name** → `my-router-app`
  - **Select framework** → `React`
  - **Select variant** → `JavaScript` (not TypeScript)

 This ensures your files are `.jsx` instead of `.tsx`.

```
As shown below ok.
```




```
cd my-router-app
npm install
npm install react-router-dom
npm run dev
```

- `react-router-dom` is needed for routing
- Open browser at `http://localhost:5173`

Step 2: Setup `main.jsx` (Professional Way)

Create a **root React DOM render** that wraps your app with `<BrowserRouter>`:

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'
import { BrowserRouter } from 'react-router-dom'    // □ Import BrowserRouter
```

```
createRoot(document.getElementById('root')).render(
  <StrictMode>
    <BrowserRouter>                    {/*  Wrap App with BrowserRouter */}
      <App />
    </BrowserRouter>
  </StrictMode>
)
```

 Key: **BrowserRouter must wrap `<App />`** so `<Link>` and `<Routes>` work.

Step 3: Create Page Components

Inside `src/pages/`, create:

Home.jsx:-

```
function Home() {
  return <h1>Home Page</h1>;
}

export default Home;
```

About.jsx:-

```
function About() {
  return <h1>About Page</h1>;
}

export default About;
```

(Optional: you can also create `Contact.jsx` later)

Step 4: Setup `App.jsx`

```
import { Routes, Route, Link } from 'react-router-dom'
import Home from './pages/Home'
import About from './pages/About'

function App() {
```

```
  return (
    <div style={{ padding: '20px' }}>
      {/* Navigation Menu */}
      <nav style={{ marginBottom: '20px' }}>
        <Link to="/">Home</Link> | <Link to="/about">About</Link>
      </nav>

      {/* Route Definitions */}
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </div>
  )
}

export default App
```

 Key points:

- `<Link>` prevents page reload (SPA behavior)
- `<Routes>` contains all `<Route>`
- `element` prop = component to render

Step 5: Test Your App

1. Run server:

```
npm run dev
```

2. Open browser:

- `/` → Home Page
- `/about` → About Page
- Click links → no page reload

Everything works!

Step 6: Add a "Not Found" Page (Optional)

NotFound.jsx

```
function NotFound() {
  return <h1>404 - Page Not Found</h1>
}

export default NotFound
```

Update `App.jsx` routes:

```
<Routes>
  <Route path="/" element={<Home />} />
  <Route path="/about" element={<About />} />
  <Route path="*" element={<NotFound />} /> {/* Catch-all */}
</Routes>
```

- Any unknown route now shows 404 page

Step 7: Beginner Practice Tasks

1. Add **Contact Page** `/contact`
2. Style nav links with CSS
3. Add a **reload button** to practice calling a function on click
4. Console log the current URL in `useEffect` to see SPA behavior

□ Summary

- **main.jsx** → BrowserRouter wraps the app □
- **App.jsx** → Contains `<Routes>` and `<Link>` □
- **Pages** → Components for each route □
- **Optional** → NotFound and additional pages □