React Notebook

~ BY SAGAR BISWAS

Part 25: Controlled Component | Collect Form Data

❓ What is a Controlled Component?

A **controlled component** in React is a form element (like <input>) whose value is **controlled by React state** via useState.
This means we can monitor, validate, or even modify input **as users type**.

src/components/forms/Form1.js

```
import React, { useState } from "react";
import "./Form1.css";

export default function Form1() {
  // Declaring state variables for each input field
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [confirmPassword, setConfirmPassword] = useState("");

  // Handles form submission
  const handleSubmit = (e) => {
    e.preventDefault(); // prevent the submission(default behavior) of the form
    const password = e.target.password.value;
    const confirmPassword = e.target.confirmPassword.value;

    const userInfo = {
      name,
      email,
      password,
      confirmPassword,
    };

    /*console.log( "\n..:: Form's Data:\nname\t\t\t: ", name, "\nemail\t\t\t: ", email,
"\npassword\t\t: ", password, "\nconfirmPassword\t: ", confirmPassword );*/

    console.log(userInfo);
    console.log("\n");
    if (password !== confirmPassword) {
      alert("Passwords do not match!");
      return;
    }
  };
  // Handlers for each input field to update their respective states
  const handleNameChange = (e) => {
    console.log(e.target.value);
    setName(e.target.value);
  };
  const handleEmailChange = (e) => {
    console.log(e.target.value);
    setEmail(e.target.value);
  };
  const handlePasswordChange = (e) => {
    console.log(e.target.value);
    setPassword(e.target.value);
  };
  const handleConfirmPasswordChange = (e) => {
    console.log(e.target.value);
    setConfirmPassword(e.target.value);
  };
  return (
    <div className="form1-container">
      <h1>Registration Form</h1>
      <form onSubmit={handleSubmit}>
        {/* Input field for name */}
        <div className="input-container">
          <label htmlFor="name">Name: </label>
          <input
            type="text"
            id="name"
            name="name" // Allows access via e.target.name
            value={name}
            required
            onChange={handleNameChange}
          />
          {/* name="name" used to access the value of the input field */}
        </div>
        {/* Input field for email */}
        <div className="input-container">
          <label htmlFor="email">Email: </label>
          <input
            type="email"
            id="email"
            name="email"
            value={email}
            required
            onChange={handleEmailChange}
          />
        </div>
        {/* Input field for password */}
        <div className="input-container">
          <label htmlFor="password">Password: </label>
          <input
            type="password"
            id="password"
            name="password"
            value={password}
            required
            onChange={handlePasswordChange}
          />
        </div>
        {/* Input field for confirming password */}
        <div className="input-container">
          <label htmlFor="confirmPassword">Confirm Password: </label>
          <input
            type="password"
            id="confirmPassword"
            name="confirmPassword"
            value={confirmPassword}
            required
            onChange={handleConfirmPasswordChange}
          />
        </div>
        <button type="submit">Register</button>
      </form>
    </div>
  );
}
```

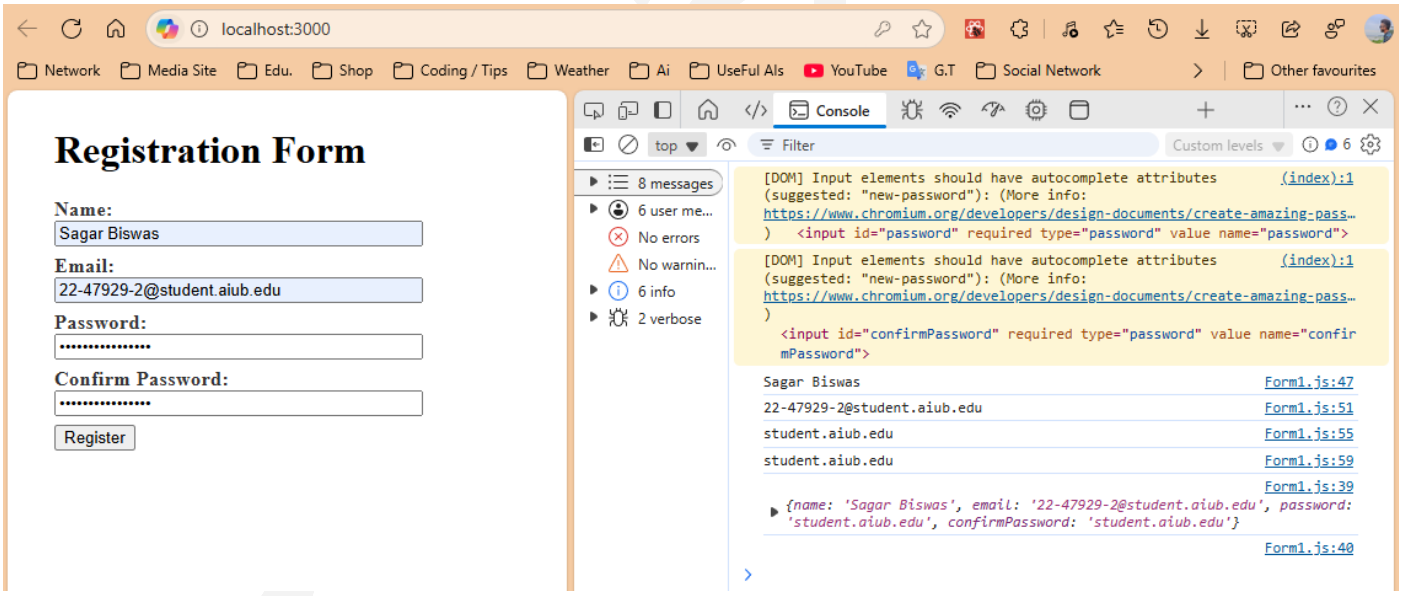
src/components/forms/Form1.css

```
.form1-container {
  margin: 30px; /* Adds space around the form container */
}

.input-container {
  margin-bottom: 20px;      /* Adds space below each input block */
  display: flex;            /* Enables flexbox layout */
  flex-direction: column;   /* Stack label and input vertically */
  font-weight: 600;         /* Makes label text slightly bolder */
  color: #333;              /* Dark gray color for labels */
  letter-spacing: 0.5px;    /* Adds spacing between letters for better readability */
  width: 300px;             /* Fixes input field width */
}
```

📂 src/App.js

```
import Form1 from './components/forms/Form1';

function App() {
  return (
    <div>
      <Form1 />
    </div>
  );
}

export default App;
```

Output:



Registration Form

Name:	Sagar Biswas
Email:	22-47929-2@student.aiub.edu
Password:	•••••••••••••••••
Confirm Password:	•••••••••••••••••

Register

```
[DOM] Input elements should have autocomplete attributes          (index):1
(suggested: "new-password"): (More info:
https://www.chromium.org/developers/design-documents/create-amazing-pass…
)   <input id="password" required type="password" value name="password">
[DOM] Input elements should have autocomplete attributes          (index):1
(suggested: "new-password"): (More info:
https://www.chromium.org/developers/design-documents/create-amazing-pass…
)
  <input id="confirmPassword" required type="password" value name="confir
  mPassword">
Sagar Biswas                                                    Form1.js:47
22-47929-2@student.aiub.edu                                     Form1.js:51
student.aiub.edu                                                Form1.js:55
student.aiub.edu                                                Form1.js:59
                                                                Form1.js:39
 ▶ {name: 'Sagar Biswas', email: '22-47929-2@student.aiub.edu', password:
 'student.aiub.edu', confirmPassword: 'student.aiub.edu'}
                                                                Form1.js:40
```

🔍 **What we Learned:**

- Controlled components use useState to control form fields.
- Input values are updated via onChange handlers.
- Form submission is handled using onSubmit and e.preventDefault().
- Clean separation of logic (.js) and style (.css).

🏳 Part 26: useState with Object

🎯 Purpose (Why use useState with an object?):

When you're handling **multiple related state values** — like multiple form fields — using an **object** inside useState keeps your code:

- More **organized** ✅
- Easier to **read and manage** ✅
- Cleaner for **updating multiple values together** ✅

Instead of having separate useState() for name, email, password, etc., we can store all in **one object**, reducing redundancy.

📂 **src\components\forms\Form1.js**

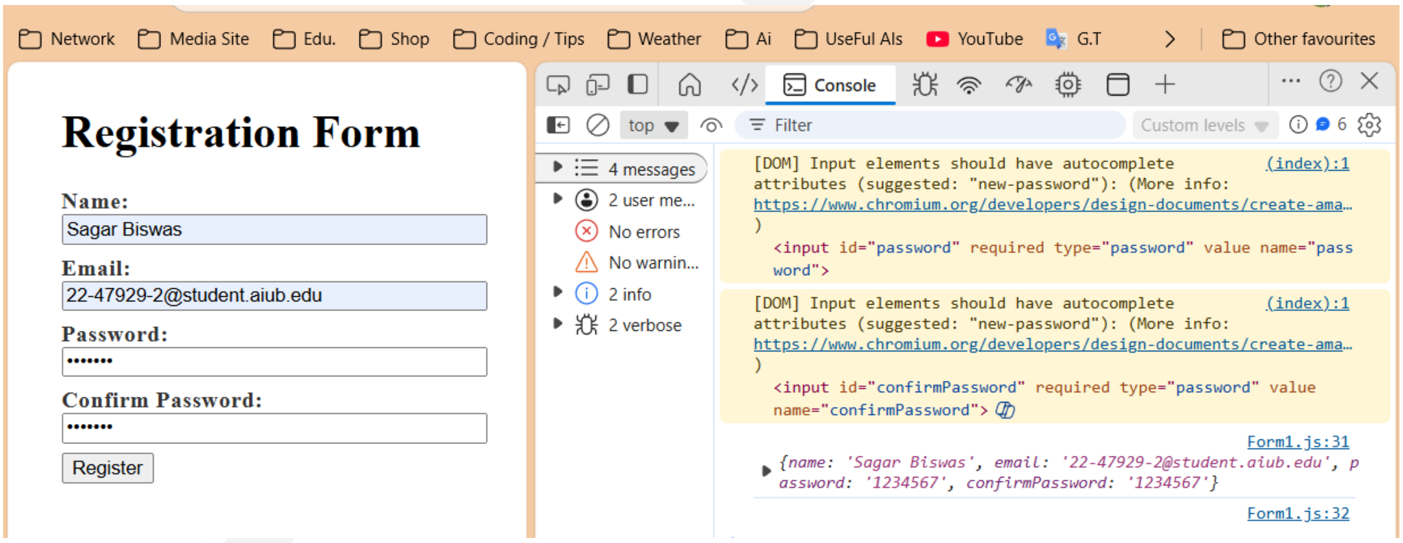
```
import React, { useState } from "react";
import "./Form1.css";

export default function Form1() {
  // ✅ useState with an object to manage multiple related fields together
  const [userInfo, setUserInfo] = useState({
    name: "",
    email: "",
    password: "",
    confirmPassword: "",
  });

  // ✅ Destructuring object properties for easier access
  const { name, email, password, confirmPassword } = userInfo;

  // ✅ Form submission handler
  const handleSubmit = (e) => {
    e.preventDefault(); // Prevents page reload

    // Getting values from form inputs (though already in state)
    const password = e.target.password.value;
    const confirmPassword = e.target.confirmPassword.value;

    // Collecting user data
    const userData = {
      name,
      email,
      password,
      confirmPassword,
    };

    console.log(userData);
    console.log("\n");

    if (password !== confirmPassword) {
      alert("Passwords do not match!");
      return;
    }
  };

  // ✅ Handles input changes dynamically using input `name`
  const handleReset = (e) => {

    // name is the name of the input field
    // if (name === "name") {
    //   setUserInfo({ ...userInfo, name: e.target.value });
    // } else if (name === "email") {
    //   setUserInfo({ ...userInfo, email: e.target.value });
    // } else if (name === "password") {
    //   setUserInfo({ ...userInfo, password: e.target.value });
    // } else if (name === "confirmPassword") {
    //   setUserInfo({ ...userInfo, confirmPassword: e.target.value });
    // }

    // Dynamically updating the specific field using computed property name
    setUserInfo({ ...userInfo, [e.target.name]: e.target.value });
  };

  return (
    <div className="form1-container">
      <h1>Registration Form</h1>

      <form onSubmit={handleSubmit}>
        {/* 👇 Name input field */}
        <div className="input-container">
          <label htmlFor="name">Name: </label>
          <input
            type="text"
            id="name"
            name="name"
            {/* name="name" used to access the value of the input field */}
            value={name}
            required
            onChange={handleReset}
          />
        </div>

        {/* 👇 Email input field */}
        <div className="input-container">
          <label htmlFor="email">Email: </label>
          <input
            type="email"
            id="email"
            name="email"
            value={email}
            required
            onChange={handleReset}
          />
        </div>

        {/* 👇 Password input field */}
        <div className="input-container">
          <label htmlFor="password">Password: </label>
          <input
            type="password"
            id="password"
            name="password"
            value={password}
            required
            onChange={handleReset}
          />
        </div>

        {/* 👇 Confirm Password input field */}
        <div className="input-container">
          <label htmlFor="confirmPassword">Confirm Password: </label>
          <input
            type="password"
            id="confirmPassword"
            name="confirmPassword"
```

```
            value={confirmPassword}
            required
            onChange={handleReset}
          />
        </div>

        {/* 👇 Submit Button */}
        <button type="submit">Register</button>
      </form>
    </div>
  );
}
```

📂 **src\components\forms\Form1.css**

```
      .form1-container {
        margin: 30px;
      }

      .input-container {
        margin-bottom: 20px;
        display: flex;
        flex-direction: column;
        margin-bottom: 7px;
        font-weight: 600;
        color: #333;
        letter-spacing: 0.5px;
        width: 300px;
      }
```

Output:



Registration Form

Name: Sagar Biswas

Email: 22-47929-2@student.aiub.edu

Password: •••••••

Confirm Password: •••••••

Register

```
[DOM] Input elements should have autocomplete          (index):1
attributes (suggested: "new-password"): (More info:
https://www.chromium.org/developers/design-documents/create-ama…
)
  <input id="password" required type="password" value name="pass
  word">
[DOM] Input elements should have autocomplete          (index):1
attributes (suggested: "new-password"): (More info:
https://www.chromium.org/developers/design-documents/create-ama…
)
  <input id="confirmPassword" required type="password" value
  name="confirmPassword">
                                                      Form1.js:31
 ▸ {name: 'Sagar Biswas', email: '22-47929-2@student.aiub.edu', p
   assword: '1234567', confirmPassword: '1234567'}
                                                      Form1.js:32
```

❓ **Why This Is Better:**

Without Object	With Object
Multiple useState() calls	One useState()
Separate onChange functions	One reusable handler
More code repetition	Cleaner, scalable, DRY code

Part 27: Passing Data from Child to Parent Component

❓ **What we'll Learn:**

1. **Passing data from Parent to Child** → using props (Top → Bottom ✅)
2. **Passing data from Child to Parent** → using function props + state lifting (Bottom → Top ✅)

src/components/state_lifting/Child.js

```
const Child = (props) => {
  const data = "I am from child component"; // This data will be passed to the parent

  // ☑ Calling the parent's function and sending data to it.
  // handleChildData() is a function passed from parent to child
  props.handleChildData(data);

  return (
    <div>
      <h1>I am from child component</h1>
      {/* ☑ This displays data passed from parent to child */}
      <p>{props.data}</p>
    </div>
  );
};

export default Child;
```

src/App.js

```
import Child from './components/state_lifting/Child';

function App() {
  const data = "I am from parent (App.js)"; // This data will be sent to the child

  // ☑ This function will receive data from the child
  const handleChildData = (childData) => {
    console.log("childData from child component is: ", childData);
  };

  return (
    <div>
      {/* ☑ Parent to Child data via `data` prop */}
      {/* ☑ Child to Parent function via `handleChildData` prop */}
      <Child data={data} handleChildData={handleChildData} />
    </div>
  );
}

export default App;
```

src/index.js

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

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

```
// ❌ React.StrictMode causes double rendering of certain lifecycle effects in development
mode
// ✅ We are avoiding it here to prevent double console log
root.render(<App />);
```

☠️ **Notes:**

Flow	Description
Parent ➔ Child	Use props to send values down (data={value})
Child ➔ Parent	Use function passed via props & call it in the child (props.handleFunc())

🪄 **Why Console Log Appears Twice?**

- React's **Strict Mode** renders components **twice** in development to help detect side effects.

we'll see:

- childData from child component is: I am from child component
- childData from child component is: I am from child component
- ✅ To prevent this, **remove <React.StrictMode>** from index.js.

🏳️ Part 28: More on State Lifting (Child → Parent Data Communication)

📌 **Purpose:**

- Practice sending data **from child to parent** via a function passed as a **prop**.
- Understand how **state lifting** allows **App.js (parent)** to manage shared data across components like NewTodo and Todo.

📂 **src/components/newTodo.js**

```
import React from "react";

// ✅ newTodo is a child component of App
export default function NewTodo(props) {
  const data = "I am from newTodo (child) component";

  // ✅ Calling the function passed from parent and sending data
  props.handleTodo(data); // this will run every time the component renders

  return (
    <div>
      <h1>{"newTodo's h1 heading: " + props.title}</h1>
      <p>{"newTodo's p paragraph: " + props.handleTodo}</p>
    </div>
  );
}
```

✅ props.handleTodo(data) calls the parent function from the child to **lift data up**.

📂 **src/App.js**

```
import NewTodo from './components/newTodo';

// ✅ Todo is another child component
const Todo = (props) => {
  return (
    <div>
      <h1>{"Todo's h1 heading: " + props.title}</h1>
    </div>
  );
};

// ✅ App is the parent component of both NewTodo and Todo
function App() {
  // ✅ Function to receive data from child (NewTodo)
  const handleTodo = (title) => {
    console.log("catching newTodo's data by handleTodo function of App.js:", title);
  };

  return (
    <div>
      {/* Passing props to both child components */}
      <NewTodo
        title="initializing newTodo props from parent (App.js) component"
        handleTodo={handleTodo} // This function is passed down to NewTodo
      />
      <Todo
        title="initializing Todo props from parent (App.js) component"
      />
    </div>
  );
}

export default App;
```

📂 **src/index.js**

```
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>
);
```

⚠️ Why You See Console Log Twice

Output:

catching newTodo's data by handleTodo function of App.js: I am from newTodo (child) component

catching newTodo's data by handleTodo function of App.js: I am from newTodo (child) component

- This **happens due to <React.StrictMode>**, which intentionally **invokes render logic twice in development** to detect side effects.
- ✅ **Solution:** Remove <React.StrictMode> in index.js for clean single logging:

- root.render(<App />);

🧠 What is State Lifting?

Concept	Meaning
State Lifting	Moving state **upward to a common ancestor** to share between siblings.
Why?	To allow **child components** to communicate with each other via parent.

✅ Final Summary:

- You passed data from **NewTodo (child)** to **App (parent)** using a function prop.
- You sent data from App to **Todo** using regular props.
- **App.js acts as a shared central state.**

🏳 Part 29: A Basic Todo App | State Lifting Principle

🔥 Concept Recap:

- **State Lifting** means **lifting shared state to the nearest common ancestor** so that multiple child components can access or update it.
- Here, the main todos state lives in Home.js, and two child components:
  - NewTodo adds new todos.
  - TodoS displays them.

📂 src/components/Home.js

```
import React, { useState } from "react";
import TodoS from "./TodoS";
import NewTodo from "./NewTodo";

// ✅ Home component is the parent that manages the lifted todos state.
const Home = () => {
  const todoSArray = ["todo1", "todo2", "todo3"]; // initial data

  const [todoS, setTodoS] = useState(todoSArray); // todos state

  // ✅ Function to handle new todo coming from child (NewTodo)
  const handleNewTodo = (newTodo) => {
    setTodoS([...todoS, newTodo]); // adds new todo to the existing array
    console.log(newTodo); // optional debug log
  };

  return (
    <div>
      <NewTodo onTodo={handleNewTodo} />  {/* child → parent data */}
      <TodoS todoSArray={todoS} />        {/* parent → child props */}
    </div>
  );
};
```

```
export default Home;
```

src/components/NewTodo.js

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

// ✅ NewTodo is a child that sends data to parent via props.onTodo
const NewTodo = (props) => {
  const [todo, setTodo] = useState(""); // local state for input

  const handleInputChange = (e) => {
    setTodo(e.target.value); // update local input value
  };

  const handleSubmit = (e) => {
    e.preventDefault(); // prevent page refresh
    props.onTodo(todo); // send the value to the parent
    setTodo(""); // clear input field
  };

  return (
    <form onSubmit={handleSubmit}>
      <label htmlFor="todo">Todo: </label>
      <input
        type="text"
        placeholder="Enter a new todo"
        value={todo} // Controlled input: value is synced with local state 'todo'
        onChange={handleInputChange}
      />
      <button type="submit">Add</button>
    </form>
  );
};

export default NewTodo;
```

src/components/TodoS.js

```
import React from "react";
import Todo from "./Todo";

// ✅ TodoS receives the full todos array and renders each item with <Todo />
const TodoS = (props) => {
  return (
    <div>
      {props.todoSArray.map((todo, index) => (
        <Todo key={index} todo={todo} index={index} />
        // ✅ key is required when rendering a list
      ))}
    </div>
  );
};

export default TodoS;
```

src/components/Todo.js

```
import React from "react";

// ✅ Todo displays a single todo item
const Todo = (props) => {
  return (
    <div>
      <p>
        the [todo{props.index + 1}] is: {props.todo} // index started from 0
      </p>
    </div>
  );
};

export default Todo;
```

📂 src/App.js

```
import Home from "./components/Home";

// ✅ App is the root component that renders Home
function App() {
  return (
    <div>
      <Home />
    </div>
  );
}

export default App;
```

🔁 How It Flows (State Lifting Structure):

```
App.js
 └── Home.js      ← State is lifted here (todos)
      ├── NewTodo.js     → Sends new data (child → parent)
      └── TodoS.js       → Receives and displays todos
           └── Todo.js → Displays each todo item
```

📌 Summary:

Feature	Description
State Lifting	State is managed in Home.js, shared with NewTodo and TodoS
Child to Parent Data	NewTodo sends data using props.onTodo()
Parent to Child Props	Home passes todoSArray down to TodoS, which maps and sends each todo to Todo
Modular & Clean	Each component has a single responsibility

🏳️ Part 30: A Complete Todo App

☠️ Features: Add, Edit, Delete Todo | Dynamic Priority Styling | Input Reset | useState-based UI Control

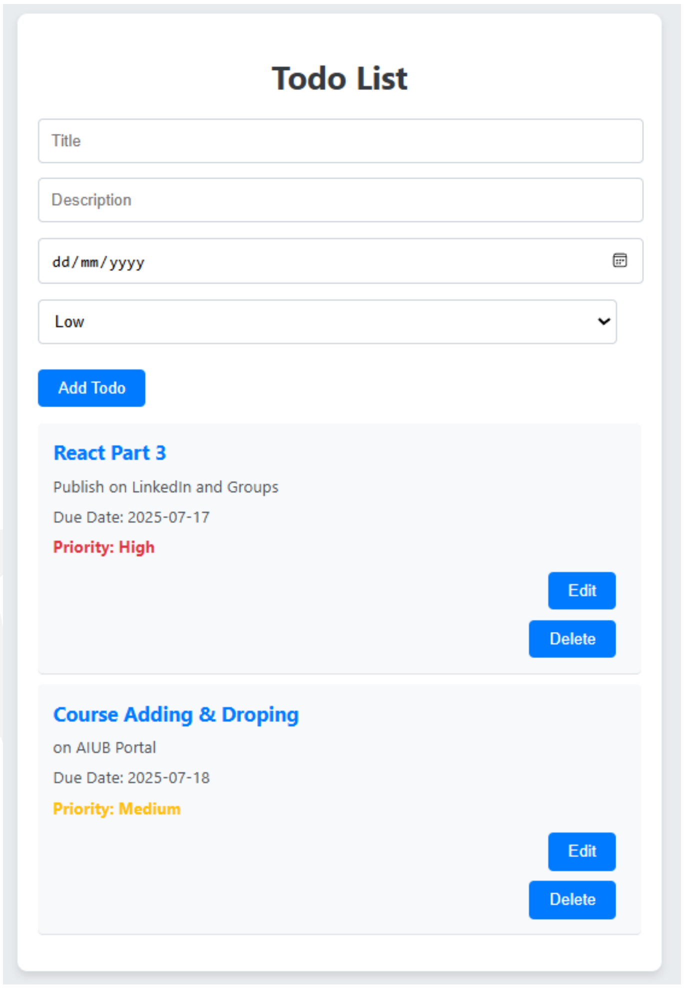
📂 1. src/App.js

```
import React, { useState } from "react";
import "./App.css";

function App() {
  const [todos, setTodos] = useState([]);              // All todos
  const [title, setTitle] = useState("");              // Title input field
  const [description, setDescription] = useState("");   // Description field
  const [dueDate, setDueDate] = useState("");           // Due Date field
  const [priority, setPriority] = useState("Low");      // Priority selector; by default set to "Low"
  const [editIndex, setEditIndex] = useState(null);     // Used to track which todo is being edited

  // ✅ Add or Update Todo
  const handleAddTodo = () => {
    if (editIndex !== null) { // If editIndex is null, we're adding a new todo; if it's a number, we're editing
an existing todo
      // 👉 Update logic
      const updatedTodos = todos.map((todo, index) =>
        index === editIndex ? { title, description, dueDate, priority } : todo // if index matches editIndex,
then update that todo; else return the existing todo
      );
      setTodos(updatedTodos);  // Update the todos state with the modified array
      setEditIndex(null);              // Reset edit index after update
    } else {
      // 👉 Add new todo
      setTodos([...todos, { title, description, dueDate, priority }]); // ...todos used to spread the existing
todos and add a new todo object at the end

    }

    // ✅ Clear input fields after submit
    setTitle("");
    setDescription("");
    setDueDate("");
    setPriority("Low"); // Reset priority to default
  };

  // ✏️ Edit Todo
  const handleEditTodo = (index) => {
    const todo = todos[index];
    setTitle(todo.title);
    setDescription(todo.description);
    setDueDate(todo.dueDate);
    setPriority(todo.priority);
    setEditIndex(index);
  };

  // ❌ Delete Todo
  const handleDeleteTodo = (index) => {
    const updatedTodos = [...todos];   // Copy todos
    updatedTodos.splice(index, 1);     // Remove one item at index
    setTodos(updatedTodos);            // Update state
  };

  return (
    <div className="App">
      <h1>Todo List</h1>

      {/* 📥 Input Fields */}
      <input
        type="text"
        placeholder="Title"
        value={title} // value used to bind the input field to the state
        onChange={(e) => setTitle(e.target.value)}
      />

      <input
        type="text"
        placeholder="Description"
        value={description}
        onChange={(e) => setDescription(e.target.value)}
      />

      <input
        type="date"
        value={dueDate}
        onChange={(e) => setDueDate(e.target.value)}
      />

      <select value={priority} onChange={(e) => setPriority(e.target.value)}>
        <option value="Low">Low</option>
        <option value="Medium">Medium</option>
        <option value="High">High</option>
      </select>

      {/* ➕ Add / 🔁 Update Button */}
      <button onClick={handleAddTodo}>
        {editIndex !== null ? "Update" : "Add"} Todo {/* if editIndex is not null means if editIndex is a
number, show "Update" else show "Add" */}
      </button>

      {/* 📝 Todo List */}
      <ul>
        {todos.map((todo, index) => (
          <li key={index}>
            <h2>{todo.title}</h2>
            <p>{todo.description}</p>
            <p>Due Date: {todo.dueDate}</p>
            <p className={`priority-${todo.priority.toLowerCase()}`}>
              Priority: {todo.priority}
            </p>

            <button onClick={() => handleEditTodo(index)}>Edit</button>
            <button onClick={() => handleDeleteTodo(index)}>Delete</button>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

📂 **2. src/App.css**

```
/* Basic Reset and Theme */
body {
  font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
  background-color: #e9ecef;
  margin: 0;
  padding: 0;
}

/* Main Container */
.App {
  max-width: 600px;
  margin: 50px auto;
  padding: 20px;
  background-color: #ffffff;
  box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1); /* subtle shadow */
  border-radius: 10px;
}

/* Heading */
h1 {
  text-align: center;
  color: #343a40;
  margin-bottom: 20px;
}

/* Input Styles */
input[type="text"],
input[type="date"],
select {
  width: calc(100% - 24px);
  padding: 12px;
  margin-bottom: 15px;
  border: 1px solid #ced4da;
  border-radius: 5px;
  font-size: 16px;
}

/* Button Styles */
button {
  padding: 10px 20px;
  margin-right: 10px;
  background-color: #007bff; /* blue */
  color: #ffffff;
  border: none;
  border-radius: 5px;
  cursor: pointer;
  font-size: 16px;
  transition: background-color 0.3s ease;
}

button:hover {
  background-color: #0056b3; /* dark blue */
}

/* Todo List */
ul {
  list-style-type: none;
  padding: 0;
}

li {
  padding: 15px;
  border-bottom: 1px solid #dee2e6;
  display: flex;
  flex-direction: column;
  align-items: flex-start;
  background-color: #f8f9fa;
  border-radius: 5px;
  margin-bottom: 10px;
  transition: background-color 0.3s ease;
}

li:hover {
  background-color: #e2e6ea;
}

/* Title */
li h2 {
  margin: 0 0 5px 0;
  color: #007bff;
  font-size: 20px;
}

/* Description and Date */
li p {
  margin: 5px 0;
  color: #495057;
  font-size: 16px;
}

/* Priority Styles */
li p.priority-high {
  color: #dc3545; /* red */
  font-weight: bold;
}

li p.priority-medium {
  color: #ffc107; /* yellow */
  font-weight: bold;
}

li p.priority-low {
  color: #28a745; /* green */
  font-weight: bold;
}
```

Output:



React Part 3
Publish on LinkedIn and Groups
Due Date: 2025-07-17
Priority: High

Course Adding & Droping
on AIUB Portal
Due Date: 2025-07-18
Priority: Medium

📊 Output Features

- ✅ Add, edit, and delete todos dynamically
- ✅ Input fields are cleared after submission
- ✅ Priority-based color styling
- ✅ Dynamic button label ("Add" or "Update")
- ✅ Fully styled and responsive layout

🧠 Learning Highlights

Concept	Purpose
useState([])	Manage the todos array
editIndex	Track which item is being edited
splice(index, 1)	Remove specific todo
map()	Display todos dynamically
dynamic classes	Priority-based styling
value + onChange	Controlled form inputs
null for initial state	Easy to check if editing

-------------------------------- X --------------------------------