Module 3 — React + Vite: Handling Events & Rendering Lists (80 minutes)

> Audience: Kids 13–14 (continuing from Module 2). All work stays in the same `react-app` project. Today we'll organize code in `src/day3/`.

Learning Objectives

By the end of this module, students will: 1. Recall components and props from previous modules. 2. Understand **event handling** in React: attaching functions to UI events like `onClick`, `onChange`. 3. Compare event handling in plain JavaScript vs React. 4. Learn about **lists and keys** in React. 5. Build a **Todo List** app with add/delete functionality as practice.

80-Minute Agenda

- **0–10 min** — Recap: components + props quiz
- **10–25 min** — Event handling in React (theory + examples)
- **25–40 min** — Hands-on: button click counter
- **40–55 min** — Rendering lists: map(), keys, and common mistakes
- **55–75 min** — Hands-on: Todo List mini-project
- **75–80 min** — Recap, Q&A, Homework

Recap (0–10 min)

Quick verbal quiz: - What are props and why are they useful? - Can a component change its own props? (No) - How do we destructure props?

Event Handling in React (10–25 min)

Events in Everyday Life

- Pressing a light switch = triggers the **event** of turning the light on.
- Clicking a button on a website = triggers a function.

Plain JavaScript Example

```
<button id="myBtn">Click Me</button>
<script>
```

```
  const btn = document.getElementById('myBtn');
  btn.addEventListener('click', () => {
    alert('Button clicked!');
  });
</script>
```

React Example

```
function ClickExample() {
  function handleClick() {
    alert('Button clicked in React!');
  }

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

Key Differences

- In React, event names use **camelCase**: `onClick`, `onChange`, `onSubmit`.
- You pass a **function reference**, not a string.
- Example: `<button onClick={handleClick}>` ✅
- Wrong: `<button onclick="handleClick()">` ❌

Inline Functions (Quick Demo)

```
<button onClick={() => alert('Clicked!')}>Click Me</button>
```

- Great for small things. - For bigger actions, use named functions.

Hands-On: Counter Button (25–40 min)

Create `src/day3/Counter.jsx`:

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

function Counter() {
  const [count, setCount] = useState(0); // state to track clicks

  function increase() {
    setCount(c => c + 1); // update count by +1
```

```
  }

  return (
    <div className="card">
      <p>Count: {count}</p>
      <button onClick={increase}>Increase</button>
    </div>
  );
}

export default Counter;
```

Update `App.jsx`:

```
import Counter from './day3/Counter.jsx';

<main className="container">
  <Counter />
</main>
```

> **Teaching point:** State changes re-render the component automatically.

Rendering Lists in React (40–55 min)

Why Lists?

Imagine showing all students in a class, or all tasks in a Todo app.

Plain JS Example

```
const items = ['Milk', 'Eggs', 'Bread'];
const ul = document.createElement('ul');
items.forEach(item => {
  const li = document.createElement('li');
  li.textContent = item;
  ul.appendChild(li);
});
```

React Example

```
function ShoppingList() {
  const items = ['Milk', 'Eggs', 'Bread'];

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

About `key`

- React needs a unique `key` for each item.
- Keys help React efficiently update only the changed items.
- Avoid using `index` if data can be rearranged.

Example with unique IDs

```
const tasks = [
  { id: 1, text: 'Do homework' },
  { id: 2, text: 'Clean room' },
];

<ul>
  {tasks.map(task => (
    <li key={task.id}>{task.text}</li>
  ))}
</ul>
```

Hands-On: Todo List (55–75 min)

Step 1 — Create TodoList.jsx

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

function TodoList() {
  const [tasks, setTasks] = useState([]); // start with empty list
  const [input, setInput] = useState('');
```

```
  function addTask() {
    if (input.trim() === '') return; // ignore empty input

    const newTask = { id: Date.now(), text: input };
    setTasks([...tasks, newTask]); // add to list
    setInput(''); // clear input
  }

  function deleteTask(id) {
    setTasks(tasks.filter(task => task.id !== id));
  }

  return (
    <div className="card">
      <h2>Todo List</h2>
      <input
        type="text"
        value={input}
        onChange={e => setInput(e.target.value)}
        placeholder="Add new task"
      />
      <button onClick={addTask}>Add</button>

      <ul>
        {tasks.map(task => (
          <li key={task.id}>
            {task.text}
            <button onClick={() => deleteTask(task.id)}>❌</button>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default TodoList;
```

Step 2 — Use in App.jsx

```
import TodoList from './day3/TodoList.jsx';

<main className="container">
  <TodoList />
</main>
```

Teaching Notes

- `onChange` captures input text.
- `onClick` on the ❌button removes tasks.
- `useState` stores the dynamic list.

Recap & Homework (75–80 min)

Recap

- Events in React = camelCase, pass function reference.
- State updates cause re-render.
- Lists require **map()** + **unique key**.
- Todo app = events + lists together.

Homework

Enhance the TodoList: 1. Add a “Done” button that marks a task as completed (strike-through text). 2. Add a counter showing how many tasks remain.

Bonus: Save tasks to `localStorage` so they remain after page refresh (advanced).

Instructor Script (Suggested)

- “Events in React are written in camelCase like `onClick`. Always pass a function, not text.”
- “When state changes, React re-renders just what is needed.”
- “Lists in React are built with `map()`. Always add a unique `key` prop.”
- “Our Todo List combines both ideas: event handling + list rendering.”

Mini-Quiz (Ask Students)

1. Why do we need a `key` prop in list items?
2. What’s the difference between `onClick={handle}` and `onClick={handle()}`?
3. How do we remove a task in our TodoList?