# Module 1 — React + Vite: Course Introduction & Environment Setup (80 minutes)

> Audience: Kids 13–14 (beginners who know basic HTML/CSS/JS). We'll build one single React app for the whole course and add daily code under `src/day1`, `src/day2`, …

---

## Learning Objectives (Plain Language)

By the end of this module, students will be able to: 1. Explain what React is and how it differs from "plain" JavaScript and jQuery-style DOM updates. 2. Set up a modern React project using **Vite** (fast dev server) instead of Create React App. 3. Run a React app locally, open it in the browser, and understand its **project structure**. 4. Create and render their **first React component**, use **JSX**, **fragments (`<>...</>`)**, `className`, and **self-closing tags**. 5. Describe, at a high level, **props** & **state** (we'll use props today; we'll just preview state and go deeper in later modules).

---

## 80-Minute Agenda (Timeboxed)

- **0–10 min** — Course intro: what React is, why we use it, how the course works
- **10–25 min** — Environment setup (Node + editor) & creating the Vite project
- **25–35 min** — Project structure tour: where files live and what they do
- **35–55 min** — Core React concepts: Components, JSX, fragments, `className`, self-closing tags, props vs state (at a glance)
- **55–75 min** — Hands-on: build your first component(s) under `src/day1/`
- **75–80 min** — Q&A + recap + mini-quiz

> **Instructor note:** Keep the pace light, celebrate small wins, and show the live preview updating.

---

## Course Structure & Rules (for Students)

- We keep **one** React project for the whole course.
- Each day's code goes in a new folder: `src/day1`, `src/day2`, `src/day3`, …
- We'll commit early and often (optional if Git is available).
- Ask questions anytime. No question is silly here ✨.

---

## React vs Plain JavaScript (Real-World Analogy)

- **Plain JS** DOM updates are like rearranging your room by moving each chair, desk, and lamp yourself.

- **React** is like having a smart helper: you describe how the room *should look*, and React figures out the minimum changes to make it so. This is faster and causes fewer errors in big apps.

### Quick Compare

**Plain JS example (imperative):**

```
<!-- index.html -->
<div id="root"></div>
<script>
  // Create an element and inject it
  const root = document.getElementById('root');
  const title = document.createElement('h1'); // create an <h1>
  title.textContent = 'Hello from plain JS!'; // set its text
  root.appendChild(title); // add it to the page
</script>
```

**React example (declarative):**

```
// We describe UI using JSX; React renders it for us
function App() {
  return (
    <h1>Hello from React!</h1> // JSX element that React turns into real DOM
  );
}
```

- **Imperative** = tell the browser *how* to do each step. - **Declarative** = tell React *what* the UI should look like; React does the steps.

---

## Environment Setup (10–25 min)

### 1) Install Node.js & npm

- Check: open a terminal and run:

  ```
  node -v    # shows Node version, e.g., v20.x
  npm -v     # shows npm version, e.g., 10.x
  ```

- If you don’t get versions, install Node LTS (ask instructor if unsure).

### 2) Install a Code Editor

- We'll use **Visual Studio Code** (VS Code). Any editor works, but VS Code is kid-friendly and has great extensions.

### 3) Create a React app with Vite (not CRA)

> **Why Vite?** It's super fast, uses modern tooling, and is perfect for learning.

**Commands (choose npm or yarn/pnpm; we'll use npm here):**

```
# 1) Create project (choose React + JavaScript when prompted)
npm create vite@latest react-app

# 2) Move into the project folder
cd react-app

# 3) Install dependencies
npm install

# 4) Start the dev server
npm run dev
```

- Open the shown local URL (e.g., http://localhost:5173) in your browser.

> **Instructor tip:** If students see a blank page, check the terminal for errors. Ensure `npm install` ran successfully.

---

## Project Structure Tour (25–35 min)

After Vite creates the project, you'll see something like:

```
react-app/
├─ index.html              # The single HTML file; Vite injects the app here
├─ package.json            # Project metadata + scripts (dev/build)
├─ vite.config.js          # Vite configuration
└─ src/
   ├─ main.jsx             # App entry; mounts React into #root
   ├─ App.jsx              # Root component
   ├─ assets/              # Images, icons, etc.
   └─ day1/                # ← We'll create this for today's code
```

### How React starts (high level)

- `index.html` contains `<div id="root"></div>` — React will render your app *inside* this div.
- `src/main.jsx` tells React to render `<App />` into `#root`.

`src/main.jsx` **(with comments):**

```
// main.jsx — entry file for the React app
import React from 'react'; // React library (needed for JSX features under the
hood)
import ReactDOM from 'react-dom/client'; // React DOM renderer
import App from './App.jsx'; // Root component we render
import './index.css'; // Global styles

// Find the #root element in index.html and create a React root
ReactDOM.createRoot(document.getElementById('root')).render(
  // StrictMode helps catch mistakes in development
  <React.StrictMode>
    <App /> {/* Render our root component */}
  </React.StrictMode>
);
```

---

## React & JSX Basics (35–55 min)

### What is a Component?

A **component** is a small, reusable piece of UI (like a LEGO block). You build big UIs by combining small components. - **Functional component** = a plain JS function that returns JSX.

**Example:**

```
// A functional component that returns some JSX
function WelcomeMessage() {
  return (
    <h2>Welcome to our React course! 🐻‍❄️</h2> // JSX element
  );
}
```

### What is JSX?

JSX looks like HTML but lives inside JavaScript. It lets us write UI like:

```
const title = <h1>Hello JSX!</h1>; // JSX value assigned to a variable
```

- Under the hood, JSX is compiled to JavaScript function calls. - **Expressions** go inside `{}`:

```
const name = 'Ada';
const greet = <p>Hello, {name}!</p>; // {name} is a JS expression
```

### `className` instead of `class`

In JSX, we write `className` because `class` is a reserved keyword in JS and React maps `className` to the DOM `class` attribute.

```
// Correct in JSX
<h1 className="headline">Hello</h1>
```

### Self-closing tags

If a tag has no children, close it like `<img />` or `<input />`. Custom components can also be self-closed: `<WelcomeMessage />`.

```
<img src="/logo.png" alt="Logo" />
<WelcomeMessage />
```

### Fragments (`<> ... </>`)

Fragments let you return **multiple** elements without adding an extra wrapper `<div>` to the DOM.

```
function Info() {
  return (
    <>
      <h3>Title</h3>
      <p>Some text without adding an extra wrapper div.</p>
    </>
  );
}
```

You can also write `<React.Fragment> ... </React.Fragment>`; both are the same.

**Props vs State (today: concept only)**

- **Props**: Inputs to a component (read-only), passed from parent → child.
- **State**: A component's own memory (can change over time). We'll preview it briefly but go deep later.

---

## Hands-On (55–75 min): Your First Components (in `src/day1/`)

### Step 1 — Create `src/day1/` and a simple component

Create folder `src/day1/` and file `HelloCard.jsx`:

```
// src/day1/HelloCard.jsx
// A simple, reusable component that accepts props

// We define a function component that reads props (like inputs)
function HelloCard({ name, hobby }) {
  return (
    <div className="card"> {/* Use className in JSX */}
      <h2>Hi, {name}! 🪳</h2> {/* Insert JS values using {} */}
      <p>I heard you enjoy: {hobby}</p>
      {/* This component is pure — it just shows what it gets via props */}
    </div>
  );
}

export default HelloCard; // Export so other files can import it
```

### Step 2 — Use the component inside `App.jsx`

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

```
// src/App.jsx
// Root component that composes other components
import HelloCard from './day1/HelloCard.jsx'; // Import our new component

function App() {
  const studentName = 'Amina'; // Example data
  const studentHobby = 'drawing';

  return (
    <>
      {/* A fragment allows returning multiple siblings without extra DOM */}
      <header className="app-header">
```

```
        <h1>React + Vite — Day 1</h1>
      </header>

      <main className="container">
        {/* Pass props from parent (App) to child (HelloCard) */}
        <HelloCard name={studentName} hobby={studentHobby} />

        {/* You can reuse the same component with different props */}
        <HelloCard name="Blerim" hobby="football" />
        <HelloCard name="Elira" hobby="piano" />

        {/* Self-closing syntax for custom components */}
        {/* <HelloCard /> would render with undefined props; we’ll keep props
required */}
      </main>
    </>
  );
}

export default App;
```

**Optional style (create `src/index.css` or use the existing one):**

```
/* src/index.css — quick styles for Day 1 */
body { font-family: system-ui, Arial, sans-serif; margin: 0; }
.app-header { padding: 16px; background: #282c34; color: white; }
.container { padding: 16px; display: grid; gap: 12px; }
.card { border: 1px solid #e5e7eb; border-radius: 12px; padding: 16px; }
```

> **Run it:** The browser should show a heading and three HelloCard boxes with names and hobbies.

### Step 3 — (Preview) Tiny State Example (we’ll go deeper later)

Create `src/day1/LikeButton.jsx` just to *see* state in action:

```
// src/day1/LikeButton.jsx
// PREVIEW of state using useState (we’ll properly cover this later)
import { useState } from 'react'; // Hook for state in function components

function LikeButton() {
  const [likes, setLikes] = useState(0); // likes = current value, setLikes =
updater
```

```
  // When the button is clicked, increment the counter
  function handleClick() {
    setLikes(l => l + 1); // update based on previous value
  }

  return (
    <button className="card" onClick={handleClick}>
      ☁️ Likes: {likes}
    </button>
  );
}

export default LikeButton;
```

Now import into `App.jsx` and render under the cards:

```
import LikeButton from './day1/LikeButton.jsx';
// ...inside <main> after the HelloCards...
<LikeButton />
```

> **Teach point:** `onClick` uses **camelCase** in JSX and expects a **function**, not a string.

---

## Why These JSX Rules Exist (Kid-Friendly Explanations)

- `className`: React uses JavaScript, where `class` is a keyword. So we say `className` to avoid confusion and map to the real `class` HTML attribute.
- **Self-closing**: Cleaner and consistent syntax when no children exist. Browsers accept it, and React expects it.
- **Fragments**: Prevents unnecessary extra `<div>` wrappers that might break CSS layouts.

---

## Troubleshooting Cheatsheet

- **White screen?** Open the browser console (F12) → “Console” tab → read the error message.
- `npm run dev` **not found?** Run `npm install` first.
- **Port in use?** Close other dev servers or accept Vite’s offer to use a new port.
- **Typos in imports?** Paths are case-sensitive on many systems.

---

## Mini-Quiz (2–3 quick questions)

1. Why do we use `className` instead of `class` in JSX?
2. What do fragments (`<>...</>`) let us return from a component?

3. In React, what’s the difference between **props** and **state**?

---

## Recap (What We Did Today)

- Installed Node & VS Code, created a React + Vite app.
- Understood the project structure and how React renders.
- Learned JSX basics, fragments, `className`, self-closing tags, and the idea of components.
- Built `HelloCard` and previewed a stateful `LikeButton`.

---

## Instructor Script (Suggested Talking Points)

- “React helps us build **interactive** websites by describing the UI we want. Think LEGO.”
- “JSX looks like HTML but lives **inside JavaScript**. Use `{}` for dynamic values.”
- “We say `className`, not `class`. Also `onClick`, not `onclick`.”
- “Components are **reusable**. Change the props, get a different result.”
- “We’ll organize code by day: today’s folder is `src/day1`.”

---

## Homework (Optional)

Create a new component `StudentBadge.jsx` in `src/day1/` that shows: - A student’s **name**, **age**, and **favorite subject** passed in via **props**. - Use a **fragment** and at least one **self-closing** tag (e.g., `<img />` or your own `<Separator />`). - Style using `className` in JSX.

**Example signature:**

```
<StudentBadge name="Amina" age={13} subject="Art" />
```

> We’ll review and expand this in Module 2 (Props deep-dive + Component patterns).

---

## Appendix: Plain JS vs React (One More Side-by-Side)

**Plain JS:**

```
<div id="root"></div>
<script>
  function render(name) {
    const root = document.getElementById('root');
    root.textContent = ''; // clear
    const h1 = document.createElement('h1');
    h1.textContent = `Hello, ${name}!`;
    root.appendChild(h1);
  }
  render('World');
</script>
```

**React:**

```
import { createRoot } from 'react-dom/client';

function App({ name }) {
  return <h1>Hello, {name}!</h1>; // Declarative: describe the result
}

createRoot(document.getElementById('root')).render(<App name="World" />);
```

> React handles updates efficiently; you focus on *what* the UI should be, not *how* to mutate the DOM.