Day 1 — React Basics (Foundations)

1️⃣ What is React? Why do we use it?

React is a JavaScript library for building user interfaces.

Created by **Meta (Facebook)** to solve one main problem:

Updating the UI efficiently when data changes.

🌟 Why React became so popular?

1. **Fast UI updates** (because of Virtual DOM)
2. **Component-based architecture** (reusable building blocks)
3. **Easy to maintain**
4. **Large community + job market**
5. **Used by top companies** (Meta, Instagram, Netflix, Airbnb)

🎯 Real-world example:

Instagram Web → Feed updates instantly → React handles those updates.

📝 Mini Cheat-Sheet (Why React?)

Reason	Meaning
Fast UI updates	Virtual DOM diffing
Easy to reuse	Components
Clean code	JSX + hooks
Flexible	Can be used in small or large projects
Huge community	More job opportunities

2️⃣ What is the DOM? (Browser DOM Basics)

🧠 Definition:

DOM = **Document Object Model**
It is a **tree representation** of your HTML page.

Example HTML:

```
<div>
 <h1>Hello</h1>
 <p>Welcome</p>
</div>
```

📦 DOM Tree Visualization:

```
Document
  └── <div>
        ├── <h1>
        │     └── "Hello"
        └── <p>
              └── "Welcome"
```

❗ Problem with DOM:

Updating the DOM directly is slow.

Example:
Imagine updating **1000 items** every few milliseconds → Browser becomes slow.

📝 Mini Cheat-Sheet (DOM)

- DOM = browser's internal structure of the page
- DOM is **slow** when you update elements frequently
- React solves this using the **Virtual DOM**

3️⃣ Virtual DOM & How React Works Internally

🔥 Biggest selling point of React = Virtual DOM

⚡ Virtual DOM = A light-weight copy of the real DOM

React uses:

- **Virtual DOM (VDOM)**
- **Diffing**
- **Reconciliation**

Let's understand with visuals.

🖼️ Real DOM vs Virtual DOM (Visualization)

Real DOM:

```
<div id="app">
  <h1>Count: 0</h1>
</div>
```

If count becomes 1, real DOM updates that <h1> element manually → **slow**.

Virtual DOM:

```
VDOM_OLD          VDOM_NEW
<h1>0</h1>  --->  <h1>1</h1>
```

React compares both using Diffing Algorithm:

DIFF RESULT:

```
Only <h1> text changed → update only that part in real DOM
```

🔁 React Update Cycle (VERY IMPORTANT Visualization)

State changes →
React creates new Virtual DOM →
Compare with old Virtual DOM →
Find minimal changes →
Update ONLY those DOM parts →
UI refreshes fast 🚀

Simple Example

Your JSX:

```
<h1>Count: {count}</h1>
```

When count = 5 → React:

1. Creates a virtual version:

```
<h1>Count: 5</h1>
```

2. Compares with old:

```
<h1>Count: 4</h1>
```

3. Updates ONLY the text → not entire screen.

📝 Mini Cheat-Sheet (Virtual DOM)

Concept	Meaning
VDOM	Copy of real DOM
Diffing	Compare old vs new VDOM
Reconciliation	Updating only required parts
Result	Faster UI, better performance

4️⃣ React Components — The Building Blocks

📦 Definition:

A component is **a reusable UI block**.

Examples:

- Button
- Navbar
- Card
- Profile
- Sidebar
- Todo item

React apps are built by **combining many small components**.

How components look (Function component)

```
function Header() {
  return <h1>Hello World</h1>;
}
export default Header;
```

Use the component:

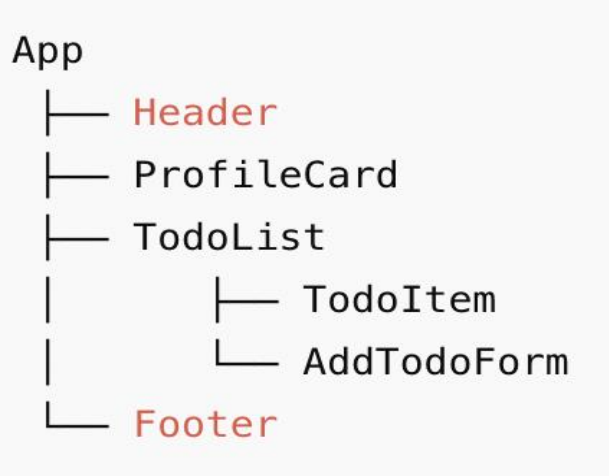
```
<Header />
```

(A component behaves like a **custom HTML tag**.)

Why components?

- Reusable
- Easy to maintain
- Easy to test
- Break down UI into small pieces
- React recommends **small, focused components**

Component Tree (Visualization)



Like a family tree → **parent & child components**.

Mini Cheat-Sheet (Components)

Component Type	Meaning
Functional	Simple JS function returning JSX
Presentational	Only UI
Stateful	Uses hooks like useState
Parent	Renders children
Child	Receives props

5 JSX — Writing HTML inside JavaScript

JSX = JavaScript XML

(it allows you to write HTML-like code inside JS)

Example:

```
return <h1>Hello {name}</h1>;
```

Why JSX?

- Cleaner syntax
- Easier to see UI structure
- Fast to build components
- Converts to JavaScript internally

JSX Rules

1. Must return one parent element

❌ Wrong:

```
return <h1 /> <p />;
```

✅ Right:

```
return (
 <div>
  <h1 />
  <p />
 </div>
);
```

2. Use camelCase for attributes

```
className="box"
onClick={handleClick}
```

3. Use curly braces for JS expressions

```
<p>Total: {price * qty}</p>
```

Example Component Using JSX

```
function Welcome(props) {
  return (
    <div className="card">
      <h2>Hello {props.name}</h2>
      <p>Welcome to React!</p>
    </div>
  );
}
```

Mini Cheat-Sheet (JSX)

Concept	Example
Expression	{count + 1}
Attribute	className="btn"
Event	onClick={fn}
Parent wrapper	...

6️⃣ Props — Passing Data Between Components

🧠 Definition:

Props = Read-only inputs passed from parent to child components.

Think of props like:

- Function arguments
- Data coming from parent to child

📦 Example: Passing props

Parent:

```
<ProfileCard name="John" age={21} />
```

Child:

```
function ProfileCard(props) {
  return (
    <div>
      <h2>{props.name}</h2>
      <p>Age: {props.age}</p>
    </div>
  );
}
```

🎨 Visualization of Props Flow

```
App Component
   ↓ props
ProfileCard
```

Like water flowing down pipes — it only flows **downwards**.

❗ Important Rules of Props:

- Props are **read-only**
- Parent → Child **only**
- Cannot modify props inside child
- Used to make components **reusable**

📝 Mini Cheat-Sheet (Props)

Term	Meaning
props	Data passed to component
Destructuring	function A({name, age}) {}
Read-only	Cannot change it inside child
Reusability	Same component, different props

7️⃣ One-Way Data Flow (MOST IMPORTANT CONCEPT)

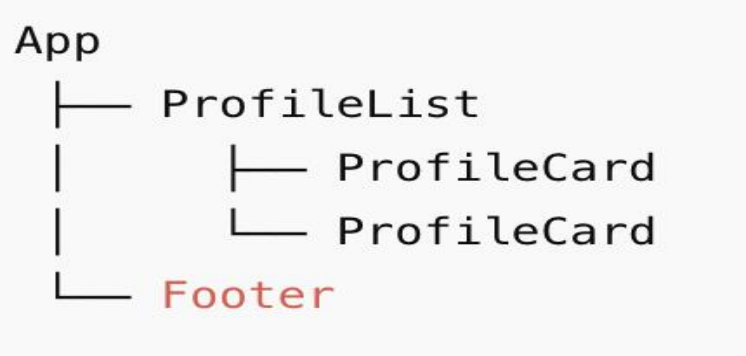
One of the BIGGEST reasons React is easy to debug.

🔄 React follows a Top → Down data flow.

Meaning:

- Data goes from **parent → child**
- NEVER from child → parent directly

🖼️ Visualization:



Data always flows this way:

```
App → ProfileList → ProfileCard
```

Not reverse.

Example

App → sends a student name → ProfileCard receives it.

```
<ProfileCard name="Alex" />
```

Inside child:

```
<h3>Hello {name}</h3>
```

Why One-Way Flow?

- Predictable UI
- Easier debugging
- Components don’t accidentally modify each other
- Data travels in a controlled way

📝 Mini Cheat-Sheet (One-Way Flow)

- Data always flows **downwards**
- Use props to send data
- Use callbacks (later) to send data upwards
- Ensures predictable UI

8️⃣ Children Prop — Wrapping and Reusing Layout

🧠 Definition:

props.children = whatever you put **inside** a component.

Example:

```
<Wrapper>
  <p>Hello</p>
</Wrapper>
```

Inside Wrapper:

```
function Wrapper({ children }) {
  return <div className="box">{children}</div>;
}
```

🖼️ Visualization of Children

```
<Wrapper>
   CHILD CONTENT
</Wrapper>

Wrapper.jsx receives:

props.children = CHILD CONTENT
```

Why Children Prop?

Used for building:

- Layout components
- Cards
- Modals
- Wrappers
- Reusable UI containers

Example (Very Common Pattern)

```
<Modal>
  <h2>Delete Item?</h2>
```

```
 <button>Confirm</button>
</Modal>
```

Modal.jsx:

```
return <div className="modal">{children}</div>;
```

Mini Cheat-Sheet (Children)

Concept	Meaning
children	Anything inside a component
Reusable layout	Wrap common structure
Example	Cards, modals, wrappers
Access	props.children

Rendering Lists & Keys

Why we use lists?

Most UIs show repeated data:

- Students
- Products
- Todo items
- Messages
- Comments

React makes this easy with .map().

Example: Rendering a list of students

Data file:

```
const students = [
 { id: 1, name: "Alex" },
 { id: 2, name: "Meera" }
];
```

Display using map:

```
{students.map(stu => (
  <p key={stu.id}>{stu.name}</p>
))}
```

! VERY IMPORTANT: Key attribute

Keys help React identify which item changed.

Wrong:

```
{students.map((s, i) => <p key={i}>{s.name}</p>)}
```

Correct:

```
key={s.id}
```

Visualization (Why keys matter)

Without keys:

- React gets confused
- Might re-render wrong items

With keys:

```
[id:1] Alex ✓
[id:2] Meera ✓
```

React tracks them correctly.

📝 Mini Cheat-Sheet (Lists + Keys)

Term	Meaning
map()	Loop JSX output
Key	Unique identifier
Why key?	Helps React track updates
Correct key	Database ID, UUID

🔟 Intro to React Hooks (useState & useEffect)

Hooks = Special functions that add features to components.

useState — For Component State

🧠 State = memory inside a component

(React re-renders UI when state changes)

useState Example:

```
const [count, setCount] = useState(0);
```

Visualization:

```
Initial state: 0
Click ➜ setCount(1)
React re-renders component
```

Counter Component Example:

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

  return (
    <>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>+</button>
    </>
  );
}
```

Mini Cheat-Sheet (useState)

Concept	Example
Declare state	useState(0)
Read state	{count}
Update state	setCount(newValue)
Causes re-render	Yes

useEffect — For Side Effects

“Side effects” = anything outside React UI logic:

- Fetching API
- Timers
- Events
- Logging
- Updating document title

Basic useEffect Example:

```
useEffect(() => {
  console.log("Component rendered");
});
```

Runs after every render.

Example with dependency:

```
useEffect(() => {
  console.log("Count changed");
}, [count]);
```

Visualization:

```
Render ↓
useEffect runs ↓
Dependency changes → re-run effect
```

📝 Mini Cheat-Sheet (useEffect)

Dependency	Meaning
[]	Run only once (on mount)
[count]	Run when count changes
No array	Run on every render

1️⃣1️⃣ Controlled vs Uncontrolled Components

A. Controlled Components (MOST IMPORTANT)

React controls the input value using **state**.

Example:

```
const [text, setText] = useState("");

<input
  value={text}
  onChange={(e) => setText(e.target.value)}
/>
```

🔍 Visualization:

```
User types → onChange → setState → UI re-renders → input gets new value
```

React is ALWAYS the single source of truth.

B. Uncontrolled Components

HTML controls the input; React just "reads" it using refs.

Example:

```
const inputRef = useRef();

<input ref={inputRef} />
```

Difference Table:

Feature	Controlled	Uncontrolled
Value managed by	React state	Browser DOM
Live validation	✔ Easy	❌ Hard

Use cases	Forms, UI	Quick input, file upload
Complexity	Slightly more	Very simple

📝 Mini Cheat-Sheet:

- **Use controlled components** for forms
- Uncontrolled only when needed (file input, simple refs)

1️⃣2️⃣ Handling Forms & Input in React

Why forms matter?

Almost every real app uses:

- Login forms
- Signup
- Search bars
- Filters
- Comments
- Todo input

🧩 Basic Form Example (Controlled Input):

```
const [name, setName] = useState("");

<form>
 <input
  value={name}
  onChange={(e) => setName(e.target.value)}
 />
```

```
  <button>Submit</button>
</form>
```

Submit Handler:

```
const handleSubmit = (e) => {
  e.preventDefault();
  console.log("Name:", name);
};
```

Visualization (Form Flow)

```
User types → state updates → UI re-renders → form submits → React handles data
```

Common Form Patterns

Pattern	Example
Controlled input	value + onChange
Submit handler	onSubmit
Validation	check state before submit
Clearing input	setState("")

Real Example (Todo Input in React)

```
const [todo, setTodo] = useState("");
const [items, setItems] = useState([]);
```

```
const addTodo = () => {
  setItems([...items, todo]);
  setTodo("");
};
```

1️⃣3️⃣ Conditional Rendering

React shows UI **based on conditions** — like turning things ON/OFF.

1. Simple if

```
{loggedIn && <Dashboard />}
```

2. Ternary Operator (Most Used)

```
isDark ? <DarkMode /> : <LightMode />
```

3. Multiple Conditions

```
status === "loading"
  ? <Spinner />
  : status === "error"
     ? <Error />
     : <Data />
```

Visualization:

```
state changes → React checks condition → shows correct UI
```

Most Practical Example:

```
{open ? <p>Content Visible</p> : <p>Hidden</p>}
```

This is exactly how **modals**, **dropdowns**, **menus**, **toggles** work.

1️⃣4️⃣ Folder Structure (Best Practices for Beginners)

Clear structure makes React easy to scale.

Beginner-friendly folder structure

```cpp
src/
  ├── components/
  │      ├── Header.jsx
  │      ├── ProfileCard.jsx
  │      ├── Counter.jsx
  │      └── TodoList.jsx
  │
  ├── data/
  │      └── students.js
  │
  ├── pages/ (optional)
  │      └── Home.jsx
  │
  ├── assets/ (optional)
  │      └── images/
  │
  ├── App.jsx
  ├── index.js
```

Why this structure?

✓ Separation of concerns

UI → components Data → data folder

✓ Reusable components

Easier to navigate, especially for students learning React.

✓ Cleaner import paths

Instead of:

```
import ProfileCard from "../../ProfileCard";
```

You get:

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

Naming Rules:

- Component files start with **capital letters**
- Use **PascalCase**
- Folder names lowercase

1️⃣5️⃣ Final Day-1 Revision Cheat Sheet 🧠 (SUPER COMPACT)

Students can revise this in **1 minute**:

⚛️ React Basics Cheat Sheet

◇ What is React?

- JS library for building UI
- Fast because of **Virtual DOM**
- Component-based

◇ DOM vs Virtual DOM

DOM	Virtual DOM
Heavy	Light
Slow updates	Fast diffing
Browser updates	JS updates first

React → updates Virtual DOM → compares → updates only changed parts.

◇ Components

- Function that returns JSX
- Reusable UI
- Props = data passed from parent

◇ One-way Data Flow

- Parent → Child
- Predictable & debuggable

◇ JSX Rules

- Must return **one parent element**
- Use {} for JS
- className instead of class

◇ Props

- Read-only

- function Card({name}) {}
- Used to pass data

◇ State (useState)

```
const [value, setValue] = useState(initial);
```

- Changes UI
- Triggers re-render

◇ useEffect

```
useEffect(() => {}, []);
```

- Runs after render
- Used for side effects
- Dependency array controls behavior

◇ Rendering Lists + Keys

```
items.map(i => <Item key={i.id} />)
```

◇ Children Prop

Wrap JSX inside components
Used for wrappers/modals/layouts

◇ Forms

- Controlled inputs recommended
- value + onChange

📝 Assignment:

Scenario (easy, real-world, single app)

Build a tiny “Student Dashboard” page that:

1. Displays a list of student Profile Cards (name, email, CGPA).
2. Lets the teacher add a quick note (a Todo item) using an input box — items show in a list.
3. Shows a simple Counter for how many students are displayed.
4. Has a Toggle button that shows/hides a help message.
5. Persists the Todo list in localStorage so it stays after refresh.

Answer:

Complete answer — step-by-step

Before starting: open your react-refresher project folder in VS Code and run:

npm run dev

Open browser at http://localhost:5173/. Keep the browser console (F12) open for logs we will use.

1) Project structure (create these files)

Create folders/files under src/:

```
src/
 components/
  Header.jsx
  ProfileCard.jsx
  Counter.jsx
  TodoList.jsx
  Toggle.jsx
  Wrapper.jsx
 data/
  students.js
 App.jsx
 main.jsx
 index.css
```

Why: Keeping components in components/ keeps code tidy and shows modularity.

2) Data: src/data/students.js

Create this array of students. This is simple JS data we will map to cards.

```
// src/data/students.js
export const students = [
 { id: 1, name: "Alex", email: "alex@example.com", cgpa: 8.9 },
 { id: 2, name: "Maria", email: "maria@example.com", cgpa: 9.1 },
 { id: 3, name: "Rahul", email: "rahul@example.com", cgpa: 8.6 }
];
```

Explain: id is important for React list keys — unique ID helps React track each item.

3) Header component: src/components/Header.jsx

A simple presentational component that shows a title.

```
// src/components/Header.jsx
export default function Header({ title }) {
 return (
  <header className="p-4 bg-slate-800 text-white rounded">
   <h1 className="text-2xl font-semibold">{title}</h1>
  </header>
 );
}
```

Explaination:

- Header is a JavaScript function that returns JSX (looks like HTML).
- {title} is a **prop** value passed by the parent (App.jsx).
- className is used for CSS classes (Tailwind or normal CSS).

4) Profile card: src/components/ProfileCard.jsx

Reusable UI block to display student info.

```
// src/components/ProfileCard.jsx
export default function ProfileCard({ name, email, cgpa }) {
 return (
  <div className="p-4 border rounded shadow-sm bg-white/5">
   <h3 className="text-lg font-medium">{name}</h3>
   <p className="text-sm text-slate-300">{email}</p>
   <p className="mt-2 text-sm">CGPA: <strong>{cgpa}</strong></p>
  </div>
```

```
  );
}
```

Explaination: Each card receives props: name, email, cgpa. Props are read-only — think of them as parameters you pass into the component.

5) Counter component: src/components/Counter.jsx

Shows useState and a useEffect to log when mounted and when count changes.

```
// src/components/Counter.jsx
import { useState, useEffect } from "react";

export default function Counter({ initial = 0 }) {
  const [count, setCount] = useState(initial);

  // Runs once when component mounts
  useEffect(() => {
    console.log("Counter mounted with initial:", initial);
  }, []); // empty array => run once on mount

  // Runs whenever count changes
  useEffect(() => {
    console.log("Counter changed, count =", count);
  }, [count]);
```

```
 return (
  <div className="p-4 mt-4 border rounded w-48">
   <h4 className="font-medium">Counter</h4>
   <div className="flex items-center gap-2 mt-2">
    <button onClick={() => setCount(c => c - 1)} className="px-3 py-1 bg-slate-700
rounded">-</button>
    <div className="px-4">{count}</div>
    <button onClick={() => setCount(c => c + 1)} className="px-3 py-1 bg-slate-700
rounded">+</button>
   </div>
  </div>
 );
}
```

Explaination:

- useState(0) creates count and setCount.
- Calling setCount tells React to update count and re-render only this component.
- useEffect with [] runs once after first render (mount).
- useEffect with [count] runs after each render where count changed.

6) TodoList (controlled input + localStorage) src/components/TodoList.jsx

This demonstrates controlled inputs, list rendering, keys, and useEffect for persistence.

```
// src/components/TodoList.jsx
import { useState, useEffect } from "react";

export default function TodoList() {
 const [text, setText] = useState("");
 const [items, setItems] = useState([]);  // Load from localStorage on mount
```

```
useEffect(() => {
  const raw = localStorage.getItem("todos_v1");
  if (raw) {
    try {
      const parsed = JSON.parse(raw);
      setItems(parsed);
      console.log("Loaded todos from localStorage");
    } catch (err) {
      console.warn("Failed to parse todos", err);
    }
  }
}, []);

// Save to localStorage whenever `items` changes
useEffect(() => {
  localStorage.setItem("todos_v1", JSON.stringify(items));
  console.log("Saved todos to localStorage, count =", items.length);
}, [items]);

const add = () => {
  if (!text.trim()) return;
  setItems(prev => [...prev, { id: Date.now(), text }]);
  setText("");
};

const remove = (id) => {
  setItems(prev => prev.filter(it => it.id !== id));
};

return (
  <div className="p-4 border rounded mt-4">
    <h4 className="font-medium">Teacher Notes (Todos)</h4>

    <div className="mt-2 flex gap-2">
      <input
        value={text}
        onChange={(e) => setText(e.target.value)}
        placeholder="Add a note..."
```

```
        className="px-2 py-1 border rounded flex-1"
      />
      <button onClick={add} className="px-3 py-1 bg-slate-700 text-white
rounded">Add</button>
    </div>

    <ul className="mt-3 space-y-2">
      {items.map(item => (
        <li key={item.id} className="p-2 border rounded bg-white/3 flex justify-between
items-center">
          <span>{item.text}</span>
          <button onClick={() => remove(item.id)} className="ml-3 px-2 py-1 bg-red-600
rounded text-sm">Delete</button>
        </li>
      ))}
    </ul>
  </div>
 );
}
```

Detailed explanation:

- value={text} and onChange={...} make the input **controlled**: the input value is always the React state text.
- setItems(prev => [...prev, newItem]) creates a new array (immutable update) — do not mutate arrays directly.
- Each list item uses key={item.id} — this helps React reconcile list changes efficiently.
- useEffect with [] loads todos once on mount. useEffect with [items] writes to localStorage whenever the items change — practical persistence.

7) Toggle component (conditional rendering & children) src/components/Toggle.jsx

Demonstrates conditional rendering & children.

```
// src/components/Toggle.jsx
import { useState, useEffect } from "react";

export default function Toggle({ children }) {
  const [on, setOn] = useState(false);

  // Demo effect with cleanup: setInterval example (here harmless)
  useEffect(() => {
    console.log("Toggle mounted");
    const id = setInterval(() => {
      // console.log("Toggle is mounted");
    }, 10000);
    return () => {
      clearInterval(id);
      console.log("Toggle unmounted and cleaned up");
    };
  }, []);

  return (
    <div className="p-3 border rounded mt-4">
      <button onClick={() => setOn(prev => !prev)} className="px-2 py-1 bg-slate-800 text-
white rounded">
        {on ? "Hide" : "Show"} help
      </button>
      <div className="mt-3">{on ? children : <em className="text-slate-400">Help is
hidden</em>}</div>
    </div>
  );
}
```

Explain: children is any JSX placed between <Toggle>...</Toggle>. Conditional {on ? children : ...} decides what to render.

8) Wrapper (example of children usage) src/components/Wrapper.jsx

```
// src/components/Wrapper.jsx
export default function Wrapper({ children }) {
  return <div className="p-4 border-2 border-dashed rounded bg-slate-
900/40">{children}</div>;
}
```

Explain: Use Wrapper to visually group content — demonstrates composition.

9) Root: src/App.jsx — wire everything together

Put everything in App to make the Student Dashboard.

```
// src/App.jsx
import { useState } from "react";
import Header from "./components/Header";
import ProfileCard from "./components/ProfileCard";
import Counter from "./components/Counter";
import TodoList from "./components/TodoList";
import Toggle from "./components/Toggle";
import Wrapper from "./components/Wrapper";
import { students } from "./data/students";

export default function App() {
  const [showToggle, setShowToggle] = useState(true);

  return (
    <div className="p-6 space-y-6 bg-slate-900 min-h-screen text-white">
      <Header title="Student Dashboard — Day 1" />

      <Wrapper>
        <p className="mb-2">Student list (ProfileCard component used with props):</p>
```

```
      <div className="grid grid-cols-1 md:grid-cols-3 gap-4">
       {students.map(s => (
        <ProfileCard key={s.id} name={s.name} email={s.email} cgpa={s.cgpa} />
       ))}
      </div>
    </Wrapper>

    <div className="flex flex-col md:flex-row gap-6">
     <Counter initial={students.length} />
     <TodoList />
    </div>

    <div>
     <button onClick={() => setShowToggle(v => !v)} className="px-3 py-1 bg-indigo-600
rounded">
       {showToggle ? "Unmount Toggle" : "Mount Toggle"}
     </button>
    </div>

    {showToggle && (
     <Toggle>
      <div className="p-2 bg-slate-800 rounded">
       <h4 className="font-medium">How to use this page</h4>
       <p className="text-sm text-slate-300">Add quick teacher notes and view student
cards.</p>
      </div>
     </Toggle>
    )}
   </div>
  );
}
```

Explain:

- students.map(...) renders one ProfileCard per student.
- key={s.id} keeps React list stable.
- Counter initial={students.length} shows how to pass props.

- The toggle mount/unmount button demonstrates cleanup logs — try clicking to see console output Toggle unmounted and cleaned up.

10) What to run and how to test (step-by-step)

1. Run dev server: npm run dev.
2. Open http://localhost:5173/.
3. Observe:
    a. Header shows page title (from Header prop).
    b. Three ProfileCards appear (one per student).
    c. Counter shows initial student count.
    d. Todo input is empty.
4. Type a todo note, click **Add**. Item appears below. Refresh page → todo stays (because of localStorage).
5. Click + or - in Counter → console shows Counter changed, count = X. Only Counter updates; ProfileCards don't re-render (verify in React DevTools).
6. Click **Unmount Toggle** button → console shows Toggle unmounted and cleaned up (demonstrates cleanup).
7. Edit students.js (change a student name) → save → browser hot reload updates that ProfileCard.

11) Teaching explanations in beginner language (why every piece is there)

A. DOM vs Virtual DOM (simple)

- The **browser DOM** is the actual page structure. Changing it is slow.
- **React creates a Virtual DOM** (a lightweight copy) and compares new vs old to update only changes — fast and efficient.

Example in app: When you click Counter +, only the Counter area changes. React finds that difference and updates the real DOM for only that part.

B. Components & Props

- Components are small functions that return UI. Reuse them.
- ProfileCard is written once and used many times with different props (name, email, cgpa).

C. State (useState)

- State is local memory for a component. useState gives you state + setter.
- When state changes, React re-renders that component.

Example: TodoList uses items state to show list. setItems triggers UI update.

D. Controlled input

- value={text} + onChange ensures React controls the input. This is safer and predictable.

E. useEffect (side effects)

- useEffect(()=>{}, []) runs once on mount (used to load todos).
- useEffect(()=>{}, [items]) runs whenever items change (used to save todos).
- useEffect can return a cleanup function to cancel timers/intervals/listeners.

Example: Toggle has an interval and cleans it up on unmount.

F. Lists & keys

- Use .map() to create repeated UI.
- Use a stable key so React can track items. IDs are best.

12) Visuals (ASCII) — helpful metaphors to tell students

Component tree

App
 ├─ Header (props: title)

```
├─ Wrapper
│  └─ ProfileCard (props: name,email,cgpa) × 3
├─ Counter (useState + useEffect)
├─ TodoList (useState + controlled input + useEffect localStorage)
└─ Toggle (children + cleanup)
```

Update flow (Virtual DOM)

User clicks + →
React creates new Virtual DOM →
Diff with old Virtual DOM →
Only Counter node changed →
Real DOM updated (Counter only)

13) Common beginner mistakes & how to fix them

- **Forgot to export a component** → export default … — Fix: add export default.
- **Wrong import path** → check ./components/ProfileCard vs ../components/….
- **Missing key in list** → React warns in console — add key={id}.
- **Mutating state directly** (items.push) — Fix: use setItems(prev => [...prev, newItem]).
- **Infinite useEffect loop** — caused by effect setting a state that is in dependency array without guard. Fix: remove unnecessary dependencies or add conditions.

14) Short checklist for students (to hand in)

- Project runs npm run dev with no errors.
- Page shows Header + 3 ProfileCards.
- Counter increments and logs effect messages.
- Add a todo, refresh — todo persists.
- Toggle can be mounted/unmounted — console shows cleanup message.
- Code organized under src/components/ and src/data/.

15) Extra: small reflection question (for students)

Explain in your own words what happens when you click **Add** in the TodoList.
(Expected answer: Controlled input triggers setText, add creates new item with setItems, React re-renders TodoList, useEffect writes items to localStorage.)

16) Quick cheats (one-liners)

- useState(initial) → local state
- useEffect(fn, []) → run once on mount
- useEffect(fn, [dep]) → run on dep change
- value={v} onChange={e => setV(e.target.value)} → controlled input
- arr.map(x => <Comp key={x.id} {...x} />) → render lists

ContextAPI, promises, async await

Toggle Component

This component lets the user **Show / Hide** some content.
It also has a **useEffect** function that runs when the component appears on the screen and cleans up when it disappears.

Let's break everything down.

1️⃣ Import the Hooks

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

✓ What are these?

- **useState** → For storing changing values (state)
- **useEffect** → For running extra code after the component appears or updates

Think of:

- **useState** = React's memory
- **useEffect** = React's "after work" area (side effects)

2️⃣ Component Starts

```
export default function Toggle({ children }) {
```

This creates a component called **Toggle**.

It receives **children**, which means anything written between:

<Toggle> ...this part... </Toggle>

3️⃣ State Variable

```
const [on, setOn] = useState(false);
```

✓ What this means?

- on → stores whether the content is visible
- setOn → a function that changes on
- useState(false) → starts with **false** — content hidden

So at the beginning:

on = false

🪝 useEffect Section (Very Important)

```
useEffect(() => {
  console.log("Toggle mounted");

  const id = setInterval(() => {
    // console.log("Toggle is mounted");
  }, 10000);

  return () => {
    clearInterval(id);
    console.log("Toggle unmounted and cleaned up");
  };
}, []);
```

🟢 What is useEffect here?

✓ It runs only once when the component is added to the screen (mounted).

Because of the **empty array []**.

This is the React life-cycle:

Component appears → useEffect runs
Component disappears → cleanup runs

🟢 What happens when mounted?

console.log("Toggle mounted");

Meaning:
React prints “Toggle mounted” in the console.

🟢 What is setInterval doing?

const id = setInterval(() => {}, 10000);

- It creates a timer that runs every 10 seconds.
- You are not doing anything inside it (commented code), so it's harmless.

Still, it creates a timer.

🟢 What is Cleanup? (VERY IMPORTANT)

Cleanup function:

```
return () => {
  clearInterval(id);
  console.log("Toggle unmounted and cleaned up");
};
```

This clean-up runs when:

- The component disappears from the UI
  (for example: if you navigate to another page)

Cleanup:

- Stops the timer with clearInterval
- Prints “Toggle unmounted and cleaned up”

✓ Why cleanup is needed?

If you don’t clean intervals, timers continue running → performance issues.

React teaches:
💡 *“Always clean up intervals, listeners, subscriptions.”*

5️⃣ UI Part (What Shows on the Screen)

```
return (
  <div className="p-3 border rounded mt-4">
```

Just styling (Tailwind). A box.

✓ Button for toggling

```
<button onClick={() => setOn(prev => !prev)} className="...">
  {on ? "Hide" : "Show"} help
</button>
```

🔥 What this means?

When button is clicked:

setOn(prev => !prev);

If on = false → becomes true
If on = true → becomes false

This is **toggle logic**.

Button text:

- If on is true → show “Hide help”
- If on is false → show “Show help”

✓ Showing or hiding the content

```
<div className="mt-3">
 {on ? children : <em className="text-slate-400">Help is hidden</em>}
</div>
```

Very Simple Logic:

on value	What you see
false	“Help is hidden”
true	children content

So when on becomes true → you see whatever you wrapped inside:

```
<Toggle>
  <p>This is the help content</p>
</Toggle>
```

FINAL SUMMARY

"This Toggle component shows and hides content using state.

- useState stores whether the content is shown or hidden.
- The button flips the value using setOn.
- If on is true → show children.
- If on is false → show 'Help is hidden'.
- We also use useEffect with [] to run code once when the component appears.
- Inside useEffect, we created an interval (timer).
- In the cleanup function, we clear the interval when the component disappears.

Assignment –

Scenario: “Classroom Helper App”

You are building a **small helper app for a classroom**.

The app helps a teacher:

- See class information
- Count students
- Add small notes
- Share a theme (light/dark) across the app
- Understand how React thinks and updates UI

🧠 App Features

1️⃣ Class Title (Props + Component)

- Show the class name at the top
  Example: **“React Basics – Day 1”**
- The title should come from App.jsx as a **prop**

📌 **Concepts used**

- Components
- Props

2️⃣ Student Counter (useState)

- Show how many students are in the class
- Buttons:
  - ➕ Add student
  - ➖ Remove student
- Count updates on button click

📌 **Concept used**

- useState

3️⃣ Notes Section (useState + useEffect)

- Teacher can type a note in an input box
- Click **Add**
- Notes appear in a list
- Notes should **stay even after page refresh**

📌 **Concepts used**

- useState → store notes
- useEffect → save & load notes from localStorage

4️⃣ Total Characters Counter (useMemo)

- Show **total number of characters** written in all notes
- This value should update **only when notes change**
- Not on every button click elsewhere

📌 **Concept used**

- useMemo

5️⃣ Theme Toggle (useContext)

- Add a **Light / Dark mode** toggle button
- Entire app color should change
- No prop drilling allowed

📌 **Concept used**

- useContext

🧱 Required Components (Simple Structure)

```
src/
 ├─ components/
 │   ├─ Header.jsx
 │   ├─ StudentCounter.jsx
 │   ├─ Notes.jsx
 │   ├─ ThemeToggle.jsx
 ├─ context/
 │   └─ ThemeContext.jsx
```

├─ App.jsx

🔗 Hook Usage Mapping (VERY IMPORTANT)

Feature	Hook Used	Why
Student count	useState	Value changes on click
Notes list	useState	Dynamic data
Save/load notes	useEffect	Side effects
Total characters	useMemo	Avoid recalculation
Theme (dark/light)	useContext	Global state