Project: Weather App

The full journey in one app: search a city, fetch real weather, show it in cards — with loading and error states and an API key kept in .env.

Introduction

Time to spend everything at once: components, props, `useState`, `useEffect`, forms and fetching — assembled into a real **Weather App**. Type a city, the app asks a real weather API, and shows the temperature, condition and details in clean cards — with honest loading and error states, the way production apps behave. *(Roman Urdu: Yeh project ab tak ke saare React ko aik asal app mein jodta hai — search, fetch, aur saaf cards.)*

Key points

- **The shape of the app:** a search form (controlled input), a state trio — `weather` (data), `loading` (waiting), `error` (failed) — and display components that receive the data as props.
- **The fetch lives in a function, the call in the form:** submitting runs `loadWeather(city)`; an effect loads a default city (Lahore) once on mount, so the page never opens empty.
- **An API key is a password:** weather services give you a personal key. It goes in a **`.env` file** as `VITE_WEATHER_KEY=...`, read via `import.meta.env.VITE_WEATHER_KEY` in Vite — and `.env` is **never** committed to GitHub (it belongs in `.gitignore`). The video shows the free-key signup.

  (Roman Urdu: API key password jaisi hai — .env mein rakho, GitHub par kabhi mat bhejo.)
- **Loading and error are first-class screens:** a spinner line while waiting, a clear "City not found" when the API answers 404, "check your connection" when the network fails. Apps feel broken when these are missing, not when errors happen.
- **The API's data is not your data's shape:** the response carries dozens of fields with API names (`main.temp`, `weather[0].description`). Read only what the UI needs, and guard every access — the first render has no data yet.
- **Small components, clear jobs:** `SearchBar` owns the input, `WeatherCard` only displays (props in, markup out), `App` owns the state and the fetch — the composition lesson from 5.2, at project scale.

Code examples

Example 1 — App: state trio and the fetch

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

const KEY = import.meta.env.VITE_WEATHER_KEY;

function App() {
  const [city, setCity] = useState("Lahore");
  const [weather, setWeather] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState("");

  async function loadWeather(name) {
    setLoading(true);
    setError("");
    try {
      const res = await fetch(
        `https://api.openweathermap.org/data/2.5/weather?q=${name}&units=metric&appid=${KEY}`
      );
      if (!res.ok) throw new Error("City not found");
      const data = await res.json();
      setWeather(data);
    } catch (err) {
      setError(err.message);
      setWeather(null);
    } finally {
      setLoading(false);
    }
  }

  useEffect(() => {
    loadWeather("Lahore");
  }, []);

  return (
    <div className="app">
      <h1>Weather App</h1>
      <SearchBar onSearch={loadWeather} />
      {loading && <p>Loading weather…</p>}
      {error && <p className="error">{error}</p>}
      {weather && !loading && <WeatherCard data={weather} />}
    </div>
```

```
  );
}
```

What it does: The full skeleton: the key is read from `.env`, the fetch function owns the loading/error choreography (start → try → success or error → always stop loading), and an effect loads Lahore once so the first screen is alive. Rendering is pure conditional display of the four possible states.

What you will see: the app opens straight onto Lahore's weather after a brief "Loading weather…"; a bad city name shows "City not found" instead of a crash.

Example 2 — SearchBar: a controlled form that calls up

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

function SearchBar({ onSearch }) {
  const [text, setText] = useState("");

  function handleSubmit(e) {
    e.preventDefault();
    if (text.trim() === "") return;
    onSearch(text.trim());
  }

  return (
    <form onSubmit={handleSubmit}>
      <input
        value={text}
        onChange={(e) => setText(e.target.value)}
        placeholder="Search a city — Karachi, Multan, Dubai…"
      />
      <button type="submit">Search</button>
    </form>
  );
}
```

What it does: The search bar owns only its input text. On submit it does not fetch anything itself — it calls the `onSearch` prop (a function passed down from App), which is how a child asks its parent to act: data flows down as props, requests flow up as callbacks.

What you will see: type "Karachi", press Search, and the weather card below reloads with Karachi — the form never reloaded the page.

Example 3 — WeatherCard: display only, props in

```
function WeatherCard({ data }) {
  return (
    <div className="card">
      <h2>{data.name}, {data.sys.country}</h2>
      <p className="temp">{Math.round(data.main.temp)}°C</p>
      <p>{data.weather[0].description}</p>
      <p>Feels like {Math.round(data.main.feels_like)}°C</p>
      <p>Humidity: {data.main.humidity}% · Wind: {data.wind.speed} m/s</p>
    </div>
  );
}
```

What it does: The card knows nothing about fetching or state — it receives the API response as one prop and reads exactly the fields it displays, rounding temperatures the way humans like them. A component this pure can be styled, tested and reused without touching the rest of the app.

What you will see: a card like "Lahore, PK — 38°C — clear sky — Feels like 40°C — Humidity: 24% · Wind: 3 m/s", changing with every search.

Quick recap

- A real project is the old pieces in formation: **controlled form → fetch function → data/loading/error state → display components**.
- API keys live in `.env` (`VITE_…` in Vite) and never reach GitHub; loading and error states are designed screens, not afterthoughts.
- Children request, parents act: the search bar calls `onSearch`, App owns the data — data down as props, requests up as callbacks.

Common mistakes

1. **Committing the API key** — keys pushed to GitHub get scraped within minutes and abused on your account. The fix: `.env` + `.gitignore` from the project's first day; if a key ever leaks, regenerate it in the provider's dashboard.
2. **Reading `data.main.temp` before data exists** — the first render has `weather === null`, and the direct access crashes the app (the famous white screen). The fix: render the card only when data is present — `{weather && <WeatherCard data={weather} />}`.
3. **No loading or error state** — a slow network looks like a dead button, and a wrong city silently shows the old weather. The fix: the state trio — every fetch owns `loading` and `error` from the start, as in Example 1.

Practice

1. Add a 3-city quick row: buttons for Lahore, Karachi and Islamabad that call `loadWeather` directly. Then add a "wind" detail and the sunrise time (the API returns `sys.sunrise` as a Unix timestamp — convert it with `new Date(...)`).
2. Give the app a memory: save the last searched city with the `useLocalStorage` hook from lesson 5.9, and load that city on mount instead of always Lahore. Verify by searching Dubai, refreshing, and seeing Dubai.

Practice quiz

1. What state trio does the Weather App keep?

- city, country, and language
- weather (data), loading (waiting), error (failed), with display components receiving the data as props
- Only weather
- Search text only

Answer: weather (data), loading (waiting), error (failed), with display components receiving the data as props

2. Where does the weather API key live, and where must it never go?

- In the code, and it may go to GitHub
- In a .env file as VITE_WEATHER_KEY, read via import.meta.env.VITE_WEATHER_KEY, and .env is never committed to GitHub
- In localStorage, and it must go to GitHub
- In the URL, always

Answer: In a .env file as VITE_WEATHER_KEY, read via import.meta.env.VITE_WEATHER_KEY, and .env is never committed to GitHub

3. Why does the Weather App guard the card with {weather && <WeatherCard data={weather} />}?

- For style only
- Because the first render has weather === null, and reading data.main.temp before data exists crashes the app into a white screen
- Because WeatherCard requires a boolean
- It does not, the guard is decoration

Answer: Because the first render has weather === null, and reading data.main.temp before data exists crashes the app into a white screen

---

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/react-weather-app-project · More free tools and notes at https://scopetreasure.com