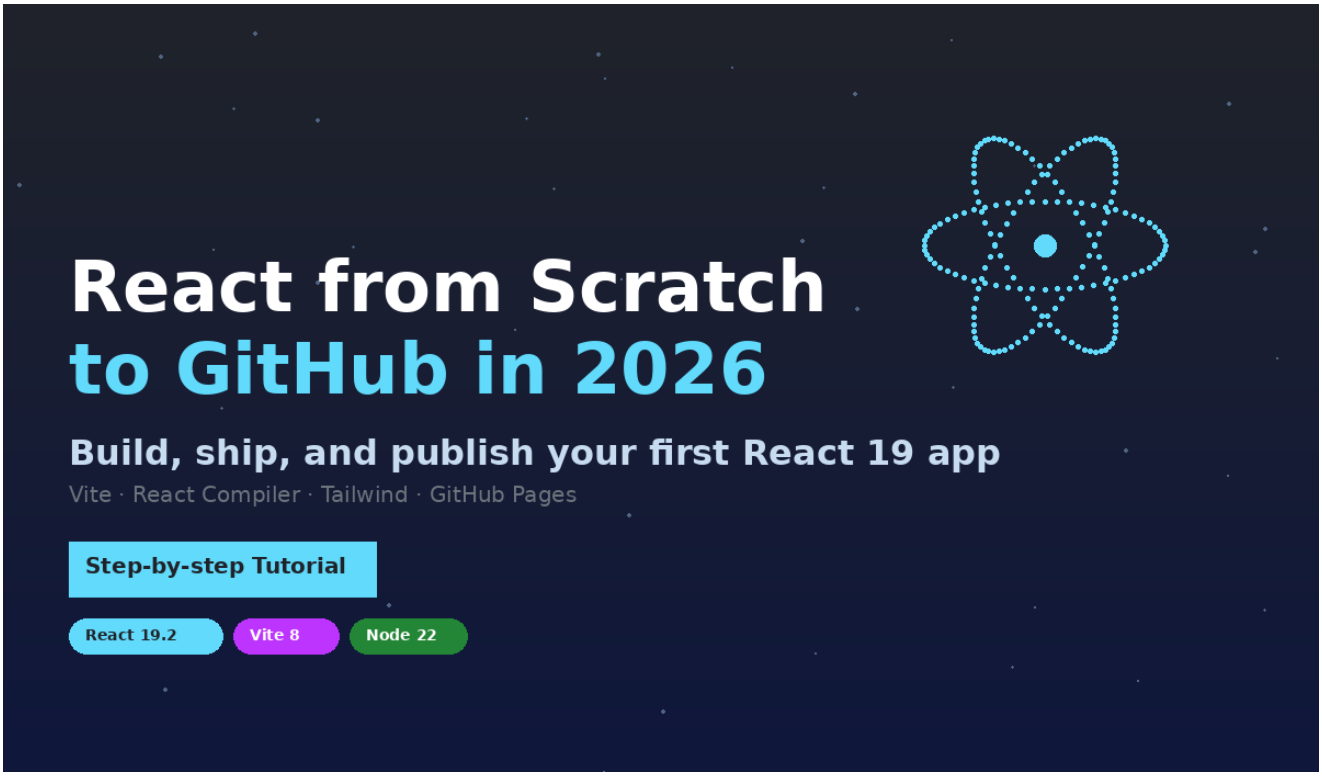

By	Suranjith Kahawanda
Publisher	EgoTechWorld (PV00352315)
Stack	React 19.2 · Vite 8 · TypeScript · Node 22
You will build	TaskMaster — a real task tracker app, deployed live
You will learn	Install · Components · Hooks · Git · GitHub Pages
Year	2026

What is React, and why bother in 2026?

React is a JavaScript library for building user interfaces. Originally released by Facebook in 2013, it has become the single most-used frontend tool in the world. In 2026, the latest stable version is **React 19.2**, and the ecosystem is more powerful and more beginner-friendly than it has ever been.

If you have ever opened Facebook, Instagram, WhatsApp Web, Netflix, Discord, Notion, or even the Anthropic Claude chat interface, you have used React. Learning it is one of the highest-leverage things a beginner web developer can do.

Why React, instead of just HTML?

Plain HTML works fine for static pages. But the moment your page needs to *react* to a user - showing a list that updates when you click a button, a form that validates as you type, or a counter that goes up - you have to write a lot of manual JavaScript. React handles all that updating for you. You describe what the screen should look like, and React figures out the changes.

What is new in React 2026

- **React Compiler 1.0** — released October 2025 — automatically optimizes your components. You no longer need to memorize useMemo and useCallback.
- **Actions and useActionState** — built-in form handling with loading states. Less code, fewer bugs.
- **Server Components** — render UI on the server for free, ship less JavaScript to the browser.
- **The React Foundation** — launched February 2026 under the Linux Foundation. React is now governed by an independent non-profit, not just Meta.

> **Don't worry about all that yet**
>
> This tutorial focuses on the parts you actually need on day one — components, props, and useState. The newer features (Compiler, Server Components, Actions) come into play on bigger projects. We will build a complete real app using just the basics.

What we are going to build

We will build **TaskMaster** - a clean task tracker where users can add tasks, mark them done, and see live counters of what is pending vs completed. By the end of this PDF, the app will be live on the public internet at `https://yourname.github.io/my-react-app/`, and you will know how to deploy any future React project the same way.

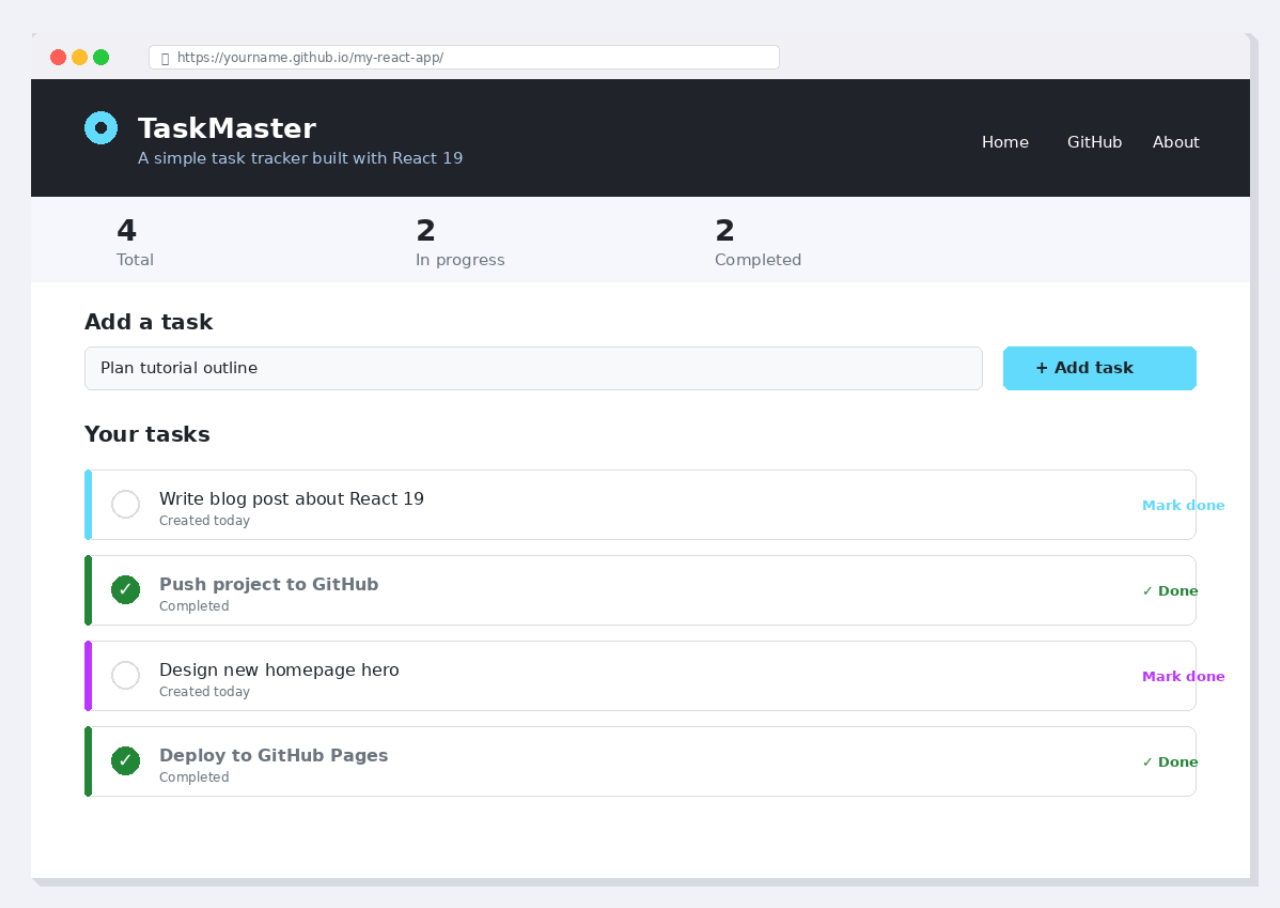

Figure 1 - The finished TaskMaster app. This is what your live site will look like by the end of the tutorial.

PART 1 · GETTING YOUR MACHINE READY

What you need before we start

Before writing a single line of React, we need three programs installed on your computer. You install them once, and they work for every project you ever build after this. Allow about 20 minutes for the whole setup if your internet is decent.

Tool	What it does	Version (2026)
Node.js	Runs JavaScript on your computer outside the browser. Required for almost every React tool.	22.x LTS
npm	Package manager. Comes free with Node. Installs React, Vite, and any other library.	10.9+
Git	Tracks every change to your code and lets you push it to GitHub.	2.45+
VS Code	The code editor. Free, made by Microsoft, used by roughly 75% of web developers.	1.96+
GitHub account	Free account at github.com — where your code will live and from where the site will be served.	Free

Step 1 - Install Node.js

Go to **https://nodejs.org** in your browser. You will see two big green download buttons. Click the one labelled **22.x LTS** (LTS means Long Term Support - the safe choice). The website detects your operating system and gives you the right installer.

1. **Windows** - download node-v22.x-x64.msi and double-click it. Click Next, Next, Next, Install. That's it.
2. **macOS** - download node-v22.x.pkg and double-click it. Allow the installer to run. Click through to finish.
3. **Linux (Ubuntu/Debian)** - run the commands below in a terminal:

```
$ curl -fsSL https://deb.nodesource.com/setup_22.x | sudo -E bash -
$ sudo apt install -y nodejs
```

Restart your terminal after installing

Whatever terminal program was open before you ran the installer needs to be **closed and reopened**. New programs are not visible to old terminal sessions. This trips up about half of all beginners.

Step 2 - Verify everything works

Open a fresh terminal (Terminal on macOS/Linux, PowerShell on Windows) and run these three commands. You should see version numbers come back, not errors:

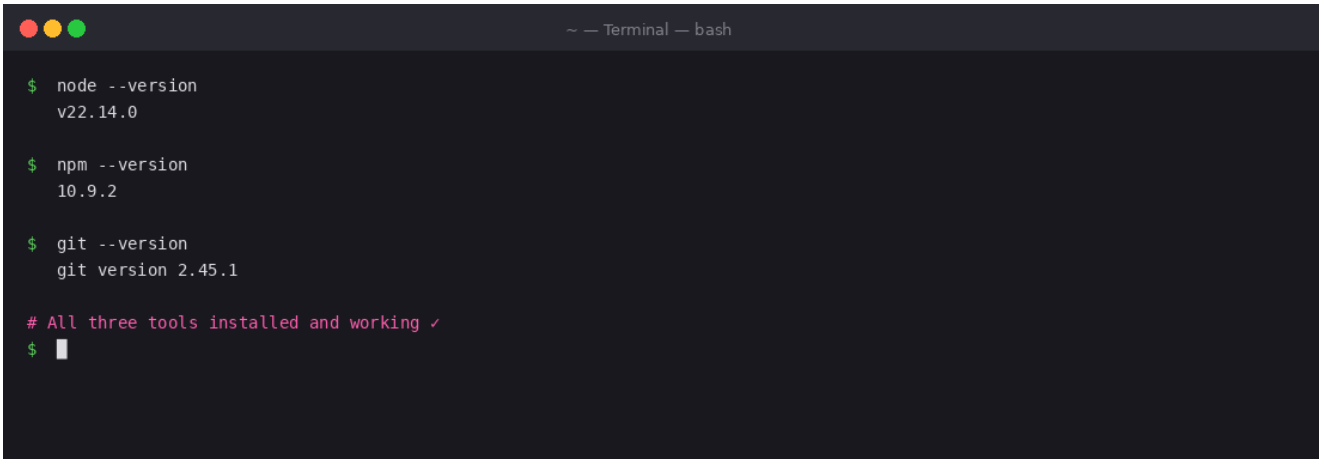

Figure 2 - Verifying that Node, npm, and Git are all installed correctly. Your numbers may differ slightly - any 22.x for Node is fine.

If you see "command not found"

It means the program is not installed, or your PATH doesn't see it. Try: (1) closing and reopening the terminal, (2) restarting your computer, (3) re-running the installer. On Windows specifically, make sure you ticked *Add to PATH* during installation.

Step 3 - Install Git

Git tracks every change to your code so you never lose work, and it is how your code gets uploaded to GitHub. If git --version already worked above, skip this step.

1. **Windows** - download from **https://git-scm.com/download/win**. During installation, accept all defaults except make sure *Use Visual Studio Code as Git's default editor* is selected if offered.
2. **macOS** - run git --version in Terminal. macOS will offer to install command line tools automatically. Click Install.
3. **Linux** - run:

```
$ sudo apt install -y git
```

Step 4 - Configure Git (one-time)

Git needs to know who you are so commits get attributed correctly. Use the **same email you will use for your GitHub account**. Run these two commands once - never again on this machine:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "you@example.com"
$ git config --global init.defaultBranch main
```

Step 5 - Install VS Code

Download from **https://code.visualstudio.com**, install, open it. We will install two helpful extensions that make working with React much smoother. In VS Code, click the **Extensions** icon on the left sidebar (it looks like four stacked squares), then search for and install:

- **ES7+ React/Redux/React-Native snippets** - type rfc + Tab to instantly create a new React component.
- **Prettier - Code formatter** - auto-formats your code so it always looks clean.
- **GitHub Pull Requests** - lets you push code to GitHub right from the editor.

Step 6 - Create your GitHub account

Go to **https://github.com** and click **Sign up**. Use the same email you configured Git with above. Pick a username you don't mind being public - it appears in every URL of every project you ever publish.

Pick your username carefully

Your live React site URL will be `https://USERNAME.github.io/project-name/`. If your GitHub username is *cool-coder-99*, that's what appears in every URL. Pick something you'd put on a CV.

PART 2 · CREATING YOUR REACT PROJECT

Why Vite, not Create React App

For years, every React tutorial started with npx create-react-app. That tool is now **officially deprecated** as of 2025. The React team themselves recommend **Vite** instead. Vite (pronounced *veet*, French for "fast") starts the development server in under a second, even on big projects. Create React App often took 30 seconds or more.

Vite 8 with Rolldown

As of March 2026, Vite 8 ships with Rolldown - a Rust-based bundler that is 10-30x faster than the old Rollup. You don't need to know what any of that means. It just means your project starts faster than you can blink.

Step 1 - Scaffold the project

Open your terminal in the folder where you keep your projects (I use ~/projects on Linux/macOS or C:\Users\You\projects on Windows). Then run:

```
$ npm create vite@latest my-react-app
```

Vite will ask you a few questions. Answer exactly like this:

1. **Project name** - press Enter to accept my-react-app, or type a different name.
2. **Select a framework** - use arrow keys to pick **React**, then press Enter.
3. **Select a variant** - pick **TypeScript + SWC**. SWC is a Rust-based compiler that's faster than Babel. TypeScript adds type safety - it catches bugs before your app even runs.
4. **Use rolldown-vite?** - pick **No**. Rolldown is stable but the regular Vite path has more tutorials online still.
5. **Install with npm and start now?** - pick **Yes**. Vite will install dependencies and launch the dev server automatically.

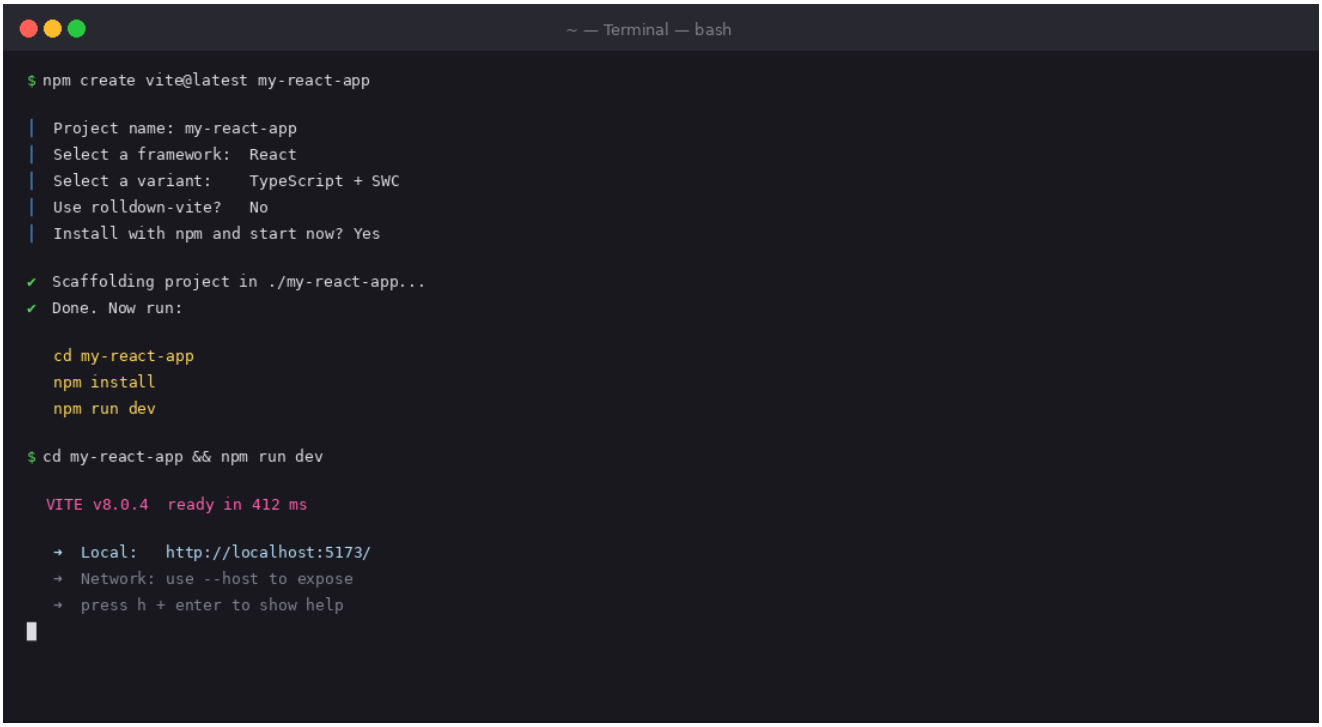

Figure 3 - The full Vite scaffolding session. This is what you should see in your terminal.

Step 2 - Open the live preview

Once the dev server starts, you will see a line like Local: http://localhost:5173/. Hold **Ctrl** (or **Cmd** on macOS) and click the URL, or just open it manually in any browser. You should see Vite's default welcome page with a spinning React logo. Congratulations - you are now running React.

> **Hot Module Replacement is magic**
>
> Leave the dev server running. In a moment we will edit a file and watch the browser update **instantly** without you refreshing - that's HMR. It saves hours of clicking refresh during a real project.

Step 3 - Open the project in VS Code

Open a second terminal window (leave the first one running the dev server). In the new terminal:

```
$ cd my-react-app
$ code .
```

The code . command opens the current folder in VS Code. If it doesn't work, just open VS Code manually and use *File > Open Folder*.

What did Vite create for us?

Take a moment to look at the file tree in VS Code's left panel. Here is what each file is for:

Vite + React project structure (TypeScript)

```
my-react-app/                    # the project root
├── node_modules/                # installed dependencies (auto)
├── public/                      # static assets (favicon, robots)
├── src/                         # your application code lives here
│   ├── assets/                  # images, svgs
│   ├── components/              # your reusable React components
│   │   ├── Header.tsx           # navigation bar
│   │   ├── TaskCard.tsx         # task list item
│   │   └── TaskForm.tsx         # add new task form
│   ├── App.tsx                  # root component
│   ├── App.css                  # app-level styles
│   ├── main.tsx                 # entry point - renders <App />
│   └── index.css                # global styles
├── .gitignore                   # files git should never commit
├── index.html                   # Vite uses this as the entry HTML
├── package.json                 # scripts + dependency list
├── tsconfig.json                # TypeScript config
└── vite.config.ts               # Vite settings
```

Figure 4 - A typical Vite + React + TypeScript project. The files marked in bold are the ones you will actually edit.

Don't touch node_modules

node_modules/ is over 200 MB and contains every package React depends on. Never edit it, never commit it to GitHub. Vite recreated it from package.json when you ran npm install - so if you delete it, running npm install again brings it back.

PART 3 · THE THREE IDEAS BEHIND REACT

Components, Props, and State

Before building TaskMaster, you need three concepts. These three are 90% of React. The remaining 10% is just learning specific hooks and patterns - which makes sense once these three are clear.

Idea 1 - Components

A **component** is just a JavaScript function that returns some HTML-looking code. That's it. The HTML-looking code is called **JSX**. React takes the JSX, turns it into real DOM elements, and puts them on the page.

The simplest possible component:

A complete React component

```
function Greeting() {
  return <h1>Hello, world!</h1>
}
```

You then use <Greeting /> anywhere in another component, and it renders the <h1>. Components can be reused as many times as you want, and they **nest** inside each other - that's how a whole app gets built up.

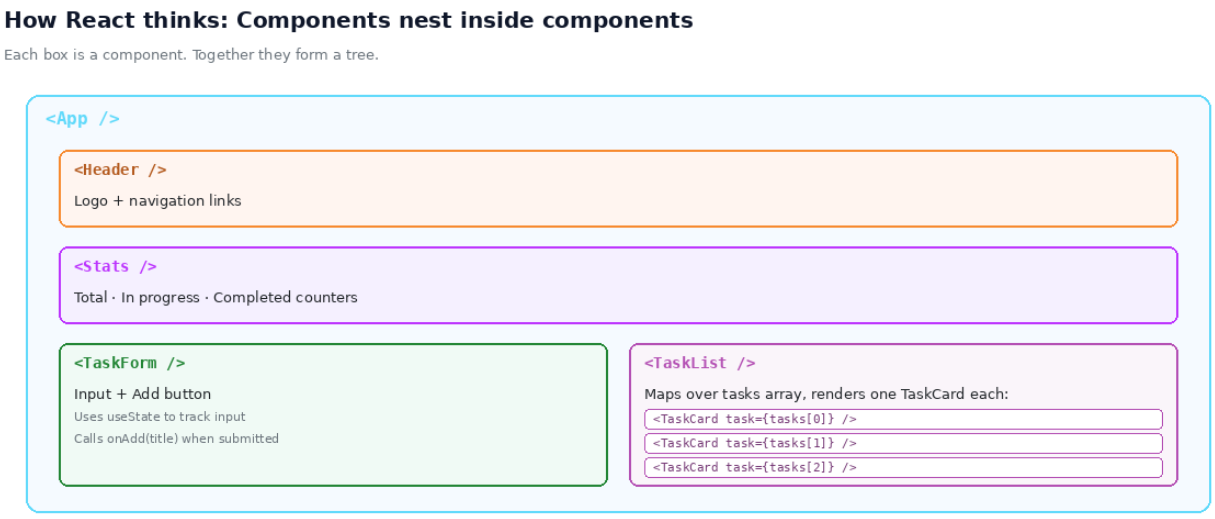

Figure 5 - The TaskMaster component tree. The whole <App /> is just smaller components nested inside it.

Idea 2 - Props

A component on its own is static. To make it dynamic, you pass **props** into it. Props are exactly like function arguments - you write them as JSX attributes and the component receives them as a parameter.

Components become reusable thanks to props

```
function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>
}

// Used like this:
<Greeting name="Suranjith" />   // → Hello, Suranjith!
<Greeting name="Anjali" />      // → Hello, Anjali!
```

Idea 3 - State

Props come from outside. **State** is data that lives inside one component and can change over time. When state changes, React automatically re-renders the component. You create state using a **hook** called useState.

A counter component using state

```
import { useState } from 'react'

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

  return (
    <div>
      <p>Count is: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  )
}
```

useState(0) creates a state variable starting at 0 and gives back two things: the current value (count) and a function to change it (setCount). Every time the button is clicked, setCount updates the value, React re-renders, and the new number appears on screen.

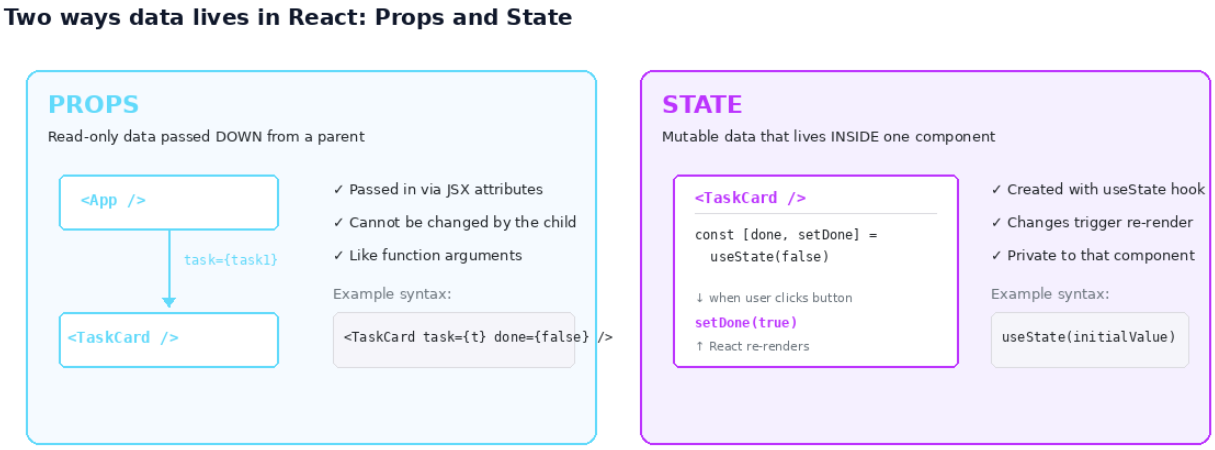

Figure 6 - Props vs State. Props flow down from parents and are read-only. State lives inside one component and is mutable.

The golden rule

Never modify state directly. Always use the setter. Don't write count = count + 1 - that won't trigger a re-render. Use setCount(count + 1). React watches the setter; it doesn't watch the variable.

PART 4 · BUILDING TASKMASTER

Step 1 - Clean out the boilerplate

Vite ships with a demo counter. We don't need it. Open src/App.tsx in VS Code and replace the entire contents with this clean starting point:

src/App.tsx — clean slate

```
import { useState } from 'react'
import './App.css'

function App() {
  return (
    <div className="app">
      <h1>TaskMaster</h1>
      <p>A simple task tracker built with React 19</p>
    </div>
  )
}

export default App
```

Save the file. Look at your browser - it has already updated. That's HMR doing its job.

Step 2 - Define the Task type

Because we picked TypeScript, we get to describe the shape of our data. This catches typos and missing fields before the app even runs. Add this above the App function:

Add to top of src/App.tsx, after the imports

```
type Task = {
  id: number
  title: string
  done: boolean
}
```

Step 3 - Add task state

We need a list of tasks that can grow over time. Inside the App function, before the return:

Inside App() — task list and input state

```
const [tasks, setTasks] = useState<Task[]>([
  { id: 1, title: 'Write blog post about React 19', done: false },
  { id: 2, title: 'Push project to GitHub', done: true },
])
const [input, setInput] = useState('')
```

Two state variables. tasks holds the array of tasks. input holds whatever the user is currently typing into the new-task box. Notice the useState<Task[]> - that's TypeScript saying "this state is an array of Task objects".

Step 4 - Add and toggle handlers

Two functions: one to add a new task, one to flip a task's done status. Both go inside App, right after the state declarations:

Add inside App() before the return statement

```
function addTask() {
  if (input.trim() === '') return
  const newTask: Task = {
    id: Date.now(),
    title: input.trim(),
    done: false,
  }
  setTasks([...tasks, newTask])
  setInput('')
}

function toggleTask(id: number) {
  setTasks(tasks.map(t =>
    t.id === id ? { ...t, done: !t.done } : t
  ))
}
```

Why Date.now() for the id?

Every task needs a unique id. Date.now() returns the current millisecond. It's unique enough for a personal app. For a real production app you would use a UUID library or a database-generated id.

Step 5 - The full UI

Now replace the return (...) block with the real interface - input box, add button, and task list:

Replace the return block in src/App.tsx

```
return (
  <div className="app">
    <header className="header">
      <h1>TaskMaster</h1>
      <p>A simple task tracker built with React 19</p>
    </header>

    <div className="stats">
      <div><strong>{tasks.length}</strong> Total</div>
      <div><strong>{tasks.filter(t => !t.done).length}</strong> In progress</div>
      <div><strong>{tasks.filter(t => t.done).length}</strong> Completed</div>
    </div>

    <section className="form">
      <input
        type="text"
        value={input}
        onChange={e => setInput(e.target.value)}
        onKeyDown={e => e.key === 'Enter' && addTask()}
        placeholder="What needs doing?"
      />
      <button onClick={addTask}>+ Add task</button>
    </section>

    <ul className="list">
      {tasks.map(task => (
        <li key={task.id} className={task.done ? 'done' : ''}>
          <span onClick={() => toggleTask(task.id)}>
            {task.done ? '✓' : '○'} {task.title}
          </span>
        </li>
      ))}
    </ul>
  </div>
)
```

What just happened? Read it once more carefully:

- tasks.map(task => ...) turns each task into an <li>. That's how React renders lists.
- key={task.id} helps React track which item is which. Always add a unique key to mapped items.
- onChange={e => setInput(e.target.value)} - every keystroke updates state, which re-renders the input with the new value. This is called a "controlled component".
- onKeyDown with e.key === 'Enter' lets the user submit by pressing Enter, not just clicking the button. Small detail, big UX win.

Step 6 - Style it

Open src/App.css and replace its contents with:

src/App.css — full replacement

```
* { box-sizing: border-box; margin: 0; padding: 0; }

body {
  font-family: system-ui, -apple-system, sans-serif;
  background: #F0F2F7;
  color: #20232A;
}

.app { max-width: 720px; margin: 0 auto; padding: 2rem 1rem; }

.header { text-align: center; margin-bottom: 2rem; }
.header h1 { font-size: 2.5rem; color: #61DAFB; }
.header p { color: #6C757D; margin-top: 0.5rem; }

.stats {
  display: flex; gap: 1rem; justify-content: center;
  background: white; padding: 1rem; border-radius: 12px;
  margin-bottom: 1.5rem;
}
.stats div { text-align: center; flex: 1; color: #6C757D; }
.stats strong { font-size: 1.8rem; color: #20232A; display: block; }

.form {
  display: flex; gap: 0.5rem; margin-bottom: 1.5rem;
}
.form input {
  flex: 1; padding: 0.75rem 1rem; font-size: 1rem;
  border: 1px solid #DEE2E6; border-radius: 8px;
  background: white;
}
.form button {
  padding: 0.75rem 1.5rem; background: #61DAFB; color: #20232A;
  border: none; border-radius: 8px; font-weight: 600;
  cursor: pointer;
}
.form button:hover { background: #4DC8E8; }

.list { list-style: none; }
.list li {
  background: white; padding: 1rem; margin-bottom: 0.5rem;
  border-radius: 8px; cursor: pointer;
  border-left: 4px solid #61DAFB;
}
.list li.done {
  border-left-color: #238636;
  text-decoration: line-through; color: #6C757D;
}
```

Save. Switch to your browser. The app should now look polished, with stats, a styled form, and a clean task list. Try adding a task. Try clicking one to mark it done. The counters at the top update automatically because they read from the same state.

If the page is blank or broken

Open the browser DevTools (right-click the page → Inspect → Console tab) and look for red error messages. The most common issue for beginners: a missing closing tag, or curly braces {} where parentheses () belong. TypeScript also underlines errors directly in VS Code — fix those first.

How it should look in VS Code

Once the file is saved, your editor should look something like this. The bracket-matching, red squigglies for errors, and syntax colours all come from VS Code + TypeScript.

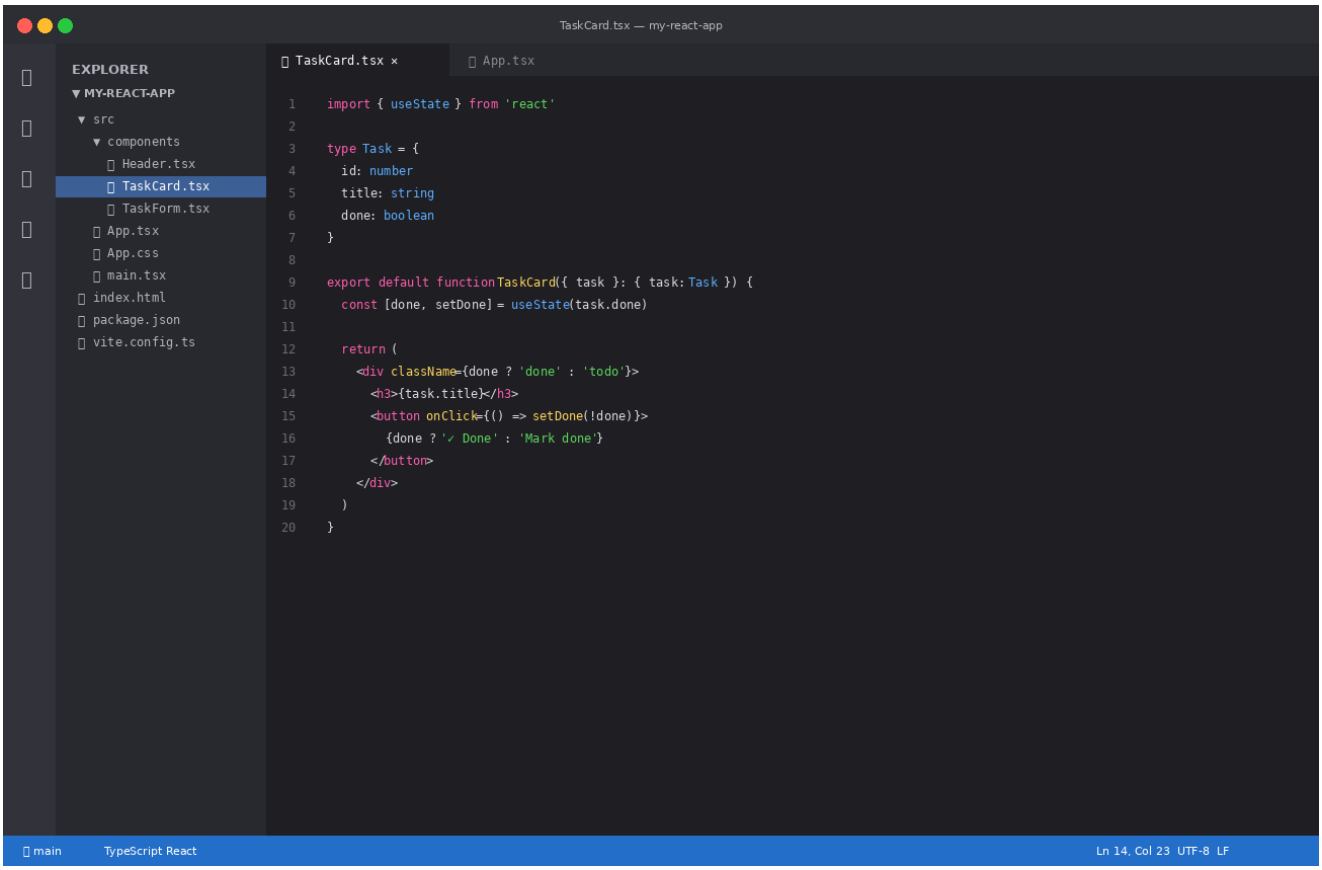

Figure 7 - VS Code with a TypeScript React component open. Notice the file tree on the left, syntax highlighting in the main pane, and the status bar showing the current Git branch.

PART 5 · TRACKING YOUR CODE WITH GIT

Why Git matters before GitHub

Git is the version control system that records every change to your project. GitHub is just a website that hosts Git repositories. You always run Git locally first, *then* push to GitHub. Many beginners try to skip Git and use GitHub's web upload - that works for one file, but becomes painful immediately on a real project.

Step 1 - Initialize Git

In your project folder, run:

```
$ git init
$ git status
```

git init creates a hidden .git folder where Git stores its history. git status shows you which files Git is currently tracking - none yet.

Step 2 - Make your first commit

Vite already created a sensible .gitignore file for you - that's why node_modules/ won't get uploaded. So we can stage and commit everything:

```
$ git add .
$ git commit -m "Initial commit: scaffold TaskMaster with Vite + React 19"
```

git add . stages all files for the next commit. git commit -m "..." saves a snapshot with a message describing the change.

Write commit messages your future self will thank you for

Bad: "stuff", "asdf", "final final v3"
Good: "Add task toggle handler", "Fix mobile layout on hero section", "Update Vite to v8". Imagine reading this list of commits in 6 months when something breaks. You will thank yourself.

The 4-step Git rhythm

Once your project is initialized, your daily Git workflow comes down to four commands. You will run these dozens of times per project:

1. git status - see what changed.
2. git add . - stage all changes.
3. git commit -m "describe what you did" - save a snapshot.
4. git push - upload commits to GitHub.

PART 6 · PUBLISHING TO GITHUB

Step 1 - Create a new GitHub repository

Log in to **github.com**. In the top-right, click the **+** icon and pick **New repository**. Fill in the form like this:

Field	Set it to	Why
Repository name	my-react-app	Match your local folder name exactly.
Description	A task tracker built with React 19 + Vite	Shows up under the repo title.
Public / Private	**Public**	Required for free GitHub Pages hosting.
Add a README	**Leave unchecked**	We will write our own from VS Code.
Add .gitignore	**Leave unchecked**	Vite already gave us one.
Add a license	**MIT** (or none)	MIT is the standard for open-source side projects.

Click **Create repository**. You will land on an empty repo page with instructions for pushing existing code. Use the section labelled *... or push an existing repository from the command line*.

Step 2 - Connect your local repo to GitHub

GitHub will show you commands like the ones below. Replace **YOURNAME** with your actual GitHub username:

```
$ git remote add origin https://github.com/YOURNAME/my-react-app.git
$ git branch -M main
$ git push -u origin main
```

On the first push, GitHub will ask you to log in. The modern way (since 2021) is via a **Personal Access Token**, not your password. Easier still: install the **GitHub CLI** (gh) and run gh auth login once - it handles authentication automatically.

Authentication trouble?

If you get "Support for password authentication was removed", you need a Personal Access Token. Go to GitHub → Settings → Developer settings → Personal access tokens → Tokens (classic) → Generate new token. Give it the repo scope. Use the token in place of your password when Git asks.

Step 3 - Verify on GitHub

Refresh your repository page on github.com. All your files - src/, package.json, vite.config.ts, etc. - should now appear. The code is safely backed up.

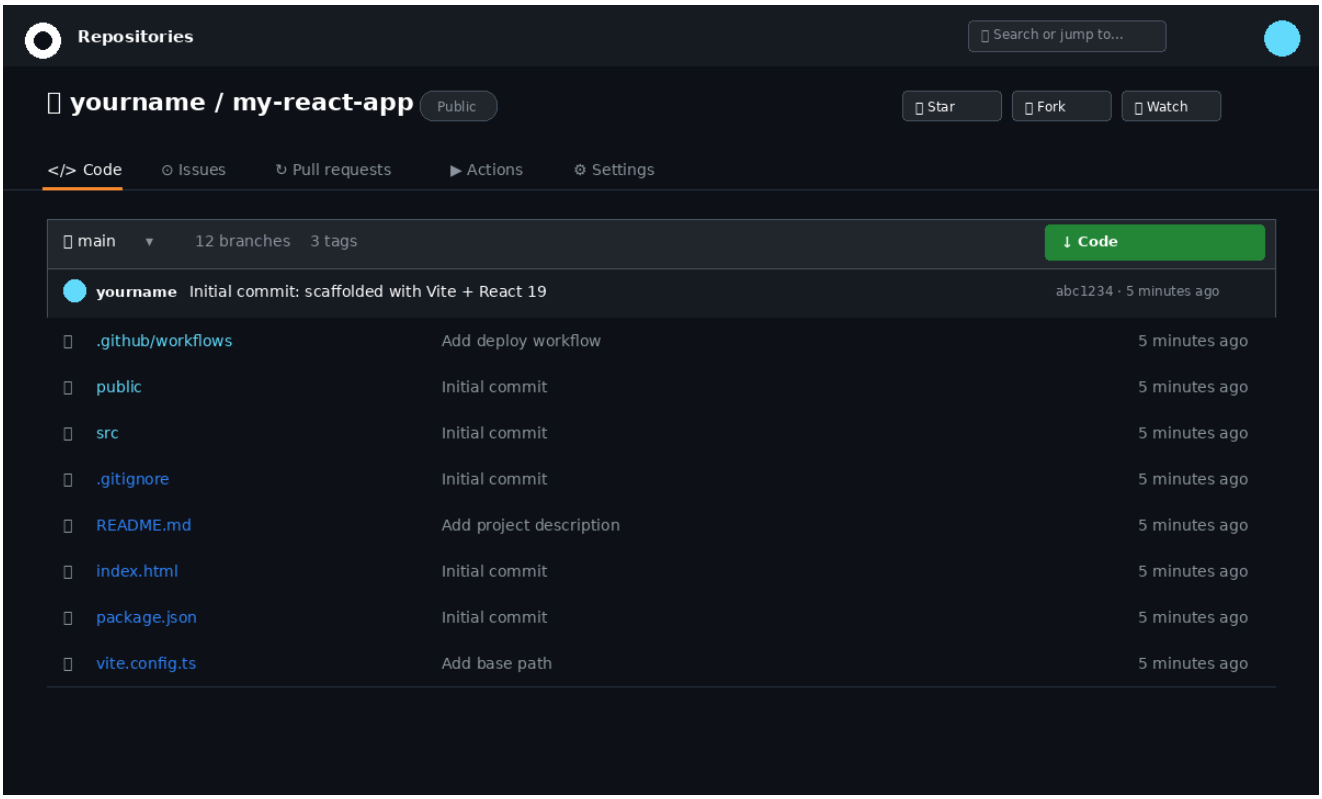

Figure 8 - Your GitHub repo after the first push. The file tree, commit message, and timestamps all show up.

PART 7 · DEPLOY LIVE ON GITHUB PAGES

How deployment works in 5 steps

Code on your laptop is invisible to the internet. To make it public, we let GitHub build and host the static site. Here's the full flow at a glance:

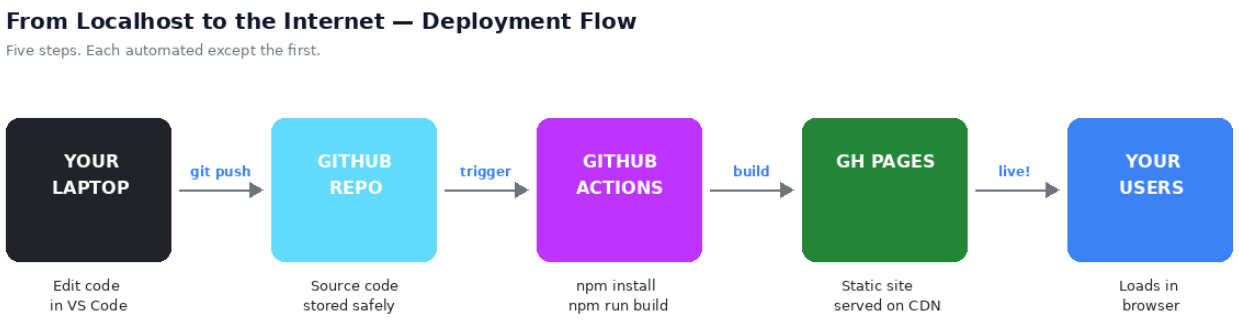

Figure 9 - From your laptop to your users in five automated steps. Once configured, every git push deploys automatically.

Step 1 - Tell Vite the right base URL

GitHub Pages serves your project from a sub-path (/my-react-app/), not from the root. Vite needs to know this so it generates correct asset paths in the built HTML. Open vite.config.ts and update it:

vite.config.ts — update with the base path

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react-swc'

export default defineConfig({
  plugins: [react()],
  base: '/my-react-app/',  // ← IMPORTANT: match repo name
})
```

Custom domain users

If you connect a custom domain like *yourname.com* to GitHub Pages later, change base back to '/'. The sub-path only applies to the default *username.github.io/repo* URL.

Step 2 - Add the GitHub Actions workflow

GitHub Actions is GitHub's built-in CI/CD. We add a *workflow* file that says "every time someone pushes to main, build the project and deploy it". Create a new file at **.github/workflows/deploy.yml** (you'll need to create the .github and workflows folders):

.github/workflows/deploy.yml

```
name: Deploy to GitHub Pages

on:
  push:
    branches: [main]
  workflow_dispatch:

permissions:
  contents: read
  pages: write
  id-token: write

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    environment:
      name: github-pages
      url: ${{ steps.deploy.outputs.page_url }}
    steps:
      - uses: actions/checkout@v4

      - uses: actions/setup-node@v4
        with:
          node-version: '22'
          cache: 'npm'

      - run: npm ci
      - run: npm run build

      - uses: actions/configure-pages@v5

      - uses: actions/upload-pages-artifact@v3
        with:
          path: ./dist

      - id: deploy
        uses: actions/deploy-pages@v4
```

What does that workflow actually do?

- **on: push: branches: [main]** - run this workflow every time someone pushes to main.
- **actions/checkout@v4** - copy the latest code into a fresh Linux machine that GitHub provides for free.
- **actions/setup-node@v4** - install Node 22 on that fresh machine.
- **npm ci** - install dependencies (faster than npm install in CI).
- **npm run build** - run Vite's production build, which outputs static files to dist/.
- **upload-pages-artifact + deploy-pages** - upload the dist/ folder to GitHub Pages. Done.

Step 3 - Push the workflow

```
$ git add .
$ git commit -m "Add deploy workflow + base path for GitHub Pages"
$ git push
```

Step 4 - Enable GitHub Pages

On your repo page, click **Settings** → **Pages** (left sidebar). Under *Build and deployment* → *Source*, change the dropdown from *Deploy from a branch* to **GitHub Actions**. Save.

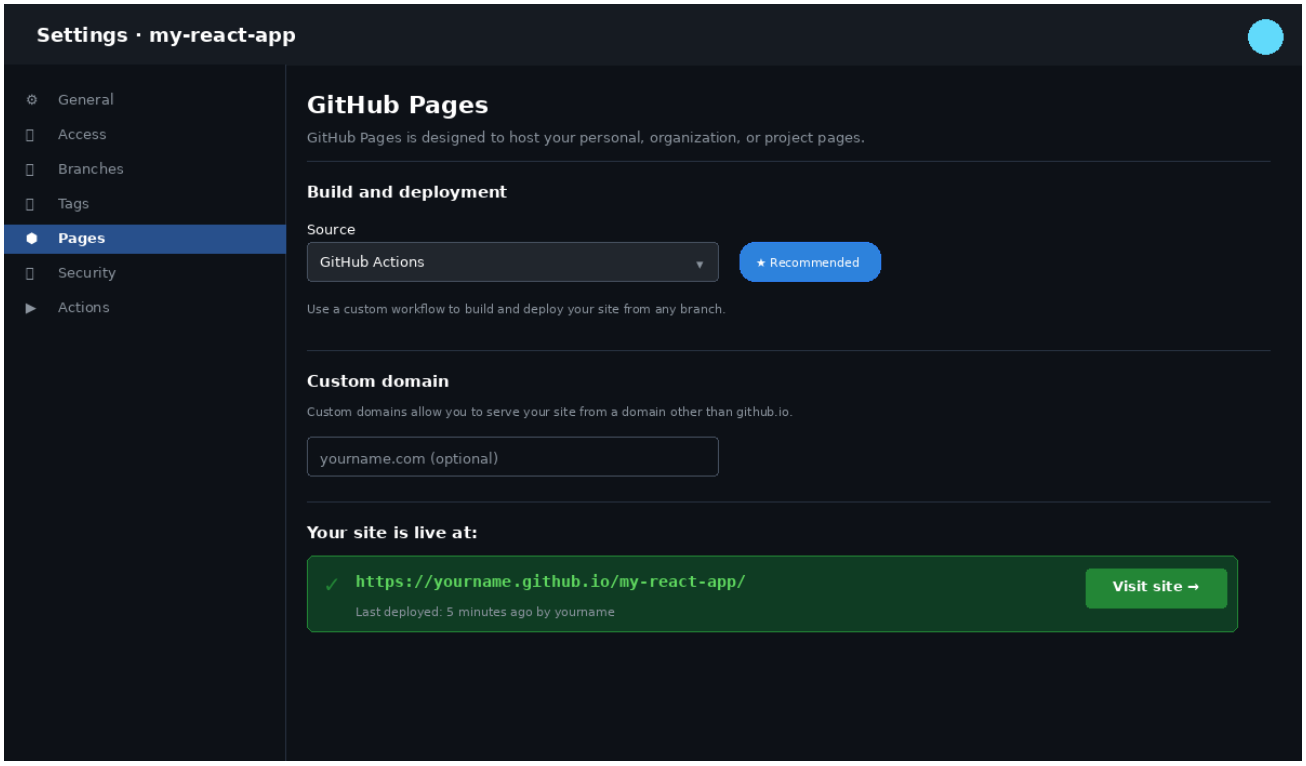

Figure 10 - The GitHub Pages settings panel. Source must be set to **GitHub Actions**, *not Deploy from a branch.*

Step 5 - Watch it deploy

Click the **Actions** tab on your repo. You should see your *Deploy to GitHub Pages* workflow running - a yellow dot means in-progress, green check means success, red X means an error. Click into the run to see live logs.

Wait for the green check (typically 60-90 seconds). Then navigate to:

Your live site

```
https://YOURNAME.github.io/my-react-app/
```

Your TaskMaster app is now live on the public internet

Anyone with that URL can use your app, on any device, anywhere in the world. **You shipped a real React app.** Send the link to a friend. Add it to your CV. Tweet it. This is the same path - Vite + React + GitHub Pages - that powers thousands of side projects and developer portfolios in 2026.

From now on, deployment is automatic

Every time you make a change locally, run git add ., commit it, and push it - the workflow rebuilds and redeploys the site automatically. Total deployment time: about 90 seconds. No FTP, no hosting bill, no SSH keys.

PART 8 · WHEN THINGS GO WRONG

Common errors and fixes

Every developer hits these. Save this section - you will re-read it on your first three projects.

Error / symptom	Likely fix
Blank white page after deploy	Almost always the base in vite.config.ts doesn't match the repo name, or trailing slashes are missing. Check the browser DevTools Console for 404s on JS or CSS files.
"Module not found" in red	You imported a file that doesn't exist. Check the spelling - filenames are case-sensitive on Linux (the GitHub build machine), even if they aren't on Windows or macOS.
App works locally but Actions build fails	The most common cause: a TypeScript error you ignored. Locally Vite warns; in CI npm run build fails hard. Run npm run build on your laptop to reproduce.
"Permission denied" on git push	Use a Personal Access Token, not your password. Or install gh CLI and run gh auth login once.
Page works but images / fonts don't load	Same root cause as the blank page - base path wrong. Asset URLs end up pointing to the root domain instead of the sub-path.
npm install hangs forever	Often a slow mirror. Try npm install --registry https://registry.npmmirror.com once, or switch to pnpm (3-10x faster, drop-in replacement).
"Cannot find name 'useState'"	Missing import at the top of your file. Add import { useState } from 'react' at the top.
Hot reload stopped working	Restart the dev server - sometimes it gets confused by renamed files. Ctrl+C then npm run dev again.

Wrap Up & Next Steps

You just installed a full modern frontend toolchain, built a working React 19 application, version-controlled it with Git, pushed it to GitHub, and configured continuous deployment to GitHub Pages - all from scratch. This same workflow is what professional teams use for client work in 2026.

Quick recap of everything you learned

- Installing Node.js 22, npm, Git, and VS Code
- Scaffolding a React + TypeScript + Vite project
- Components, JSX, props, and the useState hook
- Rendering lists with map() and unique keys
- Controlled inputs, event handlers, and event objects
- Basic CSS for a polished UI
- Git initialization, staging, committing, and pushing
- Creating a GitHub repository and connecting it to local code
- GitHub Actions workflow for automatic deployment
- Configuring Vite's base for sub-path hosting

Where to go next

- **Add features.** Edit-in-place for tasks. Filter by *All / Active / Done*. Save tasks to localStorage so they persist across reloads.
- **Try React Router.** Add a second page (e.g. /about) - this is the next core skill for any real React app.
- **Try Tailwind CSS.** Replace your hand-written CSS with Tailwind utility classes. npm install -D tailwindcss and you're off.
- **Try a backend.** Add a simple PHP or Node.js API. Or use Supabase / Firebase for instant auth + database.
- **Try Next.js.** Once you're comfortable with React, Next.js adds routing, server-side rendering, and Server Components - the production stack of choice for many teams.

One more piece of advice

Build, don't just read. Pick something simple that bothers you in your daily life - a habit tracker, an expense logger, a recipe collection - and build it in React. The point of this tutorial isn't to memorize syntax, it's to give you the permission and the toolchain to ship the next idea you have. Now go build something.

Thanks for reading

This tutorial is part of the **Web Design for Beginners** series by EgoTechWorld. The next PDF in the series covers building a Tailwind-powered portfolio site, also deployed to GitHub Pages.

More tutorials, source code, and free PDFs at **egotechworld.com**