React — Commands, Vite vs Next.js, and Folder Structure

Agenda

- Old (legacy) commands to create React apps
- Modern/new commands (Vite, Next.js) — how to use them
- What is Vite? Features, pros, cons
- What is Next.js? Features, pros, cons
- Vite vs Next.js — detailed comparison
- React (Vite) folder structure — file-by-file explanation + code snippets
- Best practices, recommendations, cheat sheet commands

Old command: Create React App (CRA)

- Create React App (CRA) was the standard official starter for many years.
- Command (legacy):

```
npx create-react-app my-app
# or
yarn create react-app my-app
# Creates a React app using Webpack + Babel.
Default structure, build & dev scripts.
```

Why CRA is considered legacy now

- Pros: Easy, batteries-included, works out-of-the-box with webpack & babel.
- Cons: Slower dev server startup and HMR compared to modern tools.
- CRA is not the preferred new starter: Vite (or Next.js for frameworks) is faster and more flexible.
- CRA supports ejecting which complicates maintenance for teams that eject.

Modern commands: Vite & Next.js

- Vite: super-fast dev tooling for modern frontends.
- Next.js: framework for React with SSR/SSG and many built-in features.

```
Vite (recommended for frontend-only projects):
  npm create vite@latest my-app
  # or
  yarn create vite my-app
  pnpm create vite@latest my-app

Next.js (full-stack React framework):
  npx create-next-app@latest my-app
  # or (pnpm)
  pnpm create next-app@latest my-app

Legacy (CRA):
  npx create-react-app my-app
```

What is Vite?

- Vite is a frontend build tool and dev server focused on speed and DX (developer experience).
- How it works (high level): dev server uses native ES modules, esbuild pre-bundles dependencies, Rollup is used for production builds.
- Designed for fast cold starts and near-instant HMR (hot module replacement).
- Use cases: Single Page Apps (SPAs), dashboards, frontend-only apps that call external APIs.

Vite — Pros

- ✅ Extremely fast dev server & instant HMR.
- ✅ Minimal configuration — sensible defaults; easy to extend with plugins.
- ✅ Small opinionated toolchain for modern JS and TypeScript.
- ✅ Works well with React, Vue, Svelte, and plain JS projects.

Vite — Cons

- ❌ No built-in routing or advanced framework features (SSR/SSG) — you add libraries as needed.
- ❌ For big sites requiring SEO (pre-rendered HTML), extra setup is needed (manual SSR or prerender).
- ❌ Some advanced optimizations need plugin/config knowledge.

What is Next.js?

- Next.js is a React framework that provides routing, SSR/SSG, API routes, and other features out-of-the-box.
- It offers file-based routing (pages/ or app/ directories), built-in image optimization, and middleware.
- Use cases: Marketing sites, blogs, e-commerce, full-stack apps where SSR/SEO matters.

Next.js — Pros

- ✅ SSR/SSG/ISR support for SEO and initial-load performance.
- ✅ File-based routing (easy to organize pages).
- ✅ Built-in API routes for simple backend endpoints.
- ✅ Many optimizations (image, fonts) and deployment options (Vercel, etc.).
- ✅ Good defaults for production with minimal config.

Next.js — Cons

- ❌ More concepts to learn (routing modes, SSR vs CSR tradeoffs, data fetching patterns).
- ❌ Slightly heavier project structure compared to a minimal Vite SPA.
- ❌ If you only need a pure frontend SPA, Next.js may feel like overkill.

Vite vs Next.js — Quick Comparison

- Dev speed: Vite is usually faster for dev HMR; Next.js dev server is slower but acceptable.
- Routing: Vite → you'd add react-router; Next.js → file-system routing built-in.
- SSR / SEO: Vite → requires custom setup; Next.js → built-in SSR/SSG.
- API Routes: Vite → none; Next.js → built-in API routes.
- Use case summary: Vite for frontend-only SPAs; Next.js for SEO or full-stack apps.

Feature checklist

- Choose Vite if: fast dev, simple SPA, you control backend (e.g., PHP/Laravel), or prefer minimalism.
- Choose Next.js if: SEO, SSG pages, built-in backend routes (API), or need hybrid rendering.
- You can migrate: start with Vite and later adopt Next.js if you need SSR/SSG.

Cheat Sheet — Commands

- Vite (React):
- Next.js:
- Legacy CRA:

```
Vite (React):
  npm create vite@latest my-app
  # Choose 'react' or 'react-ts' from the prompt
  cd my-app
  npm install
  npm run dev

Next.js:
  npx create-next-app@latest my-app
  # Follow prompts (TypeScript, ESLint, Tailwind)
  cd my-app
  npm run dev

CRA (legacy):
  npx create-react-app my-app
  cd my-app
  npm start
```

Typical project tree (Vite React)

- my-app/ ├─ public/ │ └─ index.html ├─ src/ │ ├─ assets/ # images/fonts │ ├─ components/ # reusable components │ ├─ pages/ # if using file-based routing or react-router 'pages' │ ├─ App.jsx │ ├─ main.jsx │ └─ index.css ├─ package.json └─ vite.config.js

Key files — index.html

- This is the single HTML page where React mounts into the root div.
- Vite serves this file during development and injects the module script.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>My Vite React App</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

Key files — src/main.jsx

- Entry point: renders <App /> into the #root element.
- React 18 uses createRoot from react-dom/client.

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App'
import './index.css'

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

Key files — src/App.jsx

- Main App component. From here you build your component tree.
- Keep App clean by moving UI into smaller components.

```
function App() {
  return (
    <div>
      <h1>Hello React (Vite) 🚀</h1>
    </div>
  )
}
export default App
```

Folder & file explanations

- public/: Static files served as-is (favicon, robots.txt). index.html lives here.
- src/: All source code — components, hooks, services, styles, assets.
- components/: Reusable UI components (buttons, cards, modals).
- pages/ (optional): If you use react-router, keep route-based components here.
- assets/: images, icons, fonts — import these in React components.
- services/ or api/: Code to call backend APIs (axios/fetch wrappers).
- hooks/: Custom React hooks (useAuth, useFetch, useDebounce).
- utils/: Utility functions shared across the app.
- vite.config.js: Vite-specific settings (aliases, plugins).
- .env: Environment variables (VITE_ prefix for client-side)

Best practices & tips

- Use folder conventions: components/, pages/, hooks/, services/, styles/.
- Prefer small, focused components and keep App.jsx simple.
- Use CSS Modules or utility-first CSS (Tailwind) to avoid global conflicts.
- Use absolute imports via jsconfig.json/tsconfig.json to avoid long relative paths.
- Store API endpoints and keys in .env files and never commit secrets.
- Write reusable hooks for common logic (data fetching, auth).

When to pick which (short)

- Choose Vite when: you want a fast dev experience for a frontend-only app.
- Choose Next.js when: SEO, SSG/SSR, or built-in backend routes are needed.
- If unsure: start with Vite for quick frontend UI work — migrate to Next.js later if SEO or SSR is required.