React Project Setup & Optimization Guide

Best Practices for Building Scalable and Performant React Applications

***Performance starts with architecture**. Use this guide as a baseline checklist whenever you create or refactor a React project. It's easier to build performance in from the beginning than to optimize it later. A well-optimized setup minimizes rework, simplifies scaling, and ensures long-term maintainability.*

Let your React app stay fast as it grows by following these key considerations and optimizations during the initial project setup.

1. Project Architecture & Structure

Goals: Maintain scalability, modularity, and ease of maintenance.
Best Practices:

- Organize folders by **features or domains** (/features/users, /features/auth).
- Separate **UI components**, **hooks**, and **API services** clearly.
- Set up **path aliases** (@/components, @/utils) via tsconfig.json or vite.config.ts.
- Enforce consistent formatting using **ESLint** and **Prettier**.
- Keep components small and focused on a single responsibility.

Benefits: Easier scalability and reduced coupling between modules.

2. Build Tool & Framework Selection

Goals: Use a modern build system optimized for speed and DX (developer experience).
Best Practices:

- Use **Vite** or **Next.js** for fast builds and hot reloads.
- Configure **TypeScript** for static type safety.
- Maintain separate environment files (.env.development, .env.production).
- Enable **SSR** or **SSG** in Next.js where SEO or initial render performance matters.

Benefits: Faster build times, smaller bundles, and improved initial load performance.

3. Dependency Management

Goals: Keep the dependency footprint light and purposeful.
Best Practices:

- Regularly audit dependencies using npm ls or pnpm why.
- Avoid large utility libraries if native solutions exist.
- Use **tree-shakeable** imports (lodash-es, date-fns).
- Prefer modular and lightweight UI libraries (e.g., **ShadCN**, **Chakra UI**, **Radix UI**).

Benefits: Reduced bundle size and fewer runtime dependencies to maintain.

4. Routing Strategy

Goals: Load only what's needed, when it's needed.
Best Practices:

- Implement **code splitting** and **lazy loading** via React.lazy and Suspense.
- Split code by routes or feature modules.
- Preload critical routes using Next.js prefetch or React Router's loader.

Benefits: Shorter initial load time and improved navigation performance.

5. State Management Setup

Goals: Maintain a predictable and performant state flow.
Best Practices:

- Start simple: use local state and Context selectively.
- For larger apps, use **Redux Toolkit**, **Zustand**, or **Recoil**.
- Avoid global state for UI-specific data.
- Normalize state to prevent redundant updates.

Benefits: Fewer unnecessary re-renders and a clean, predictable data flow.

6. Styling & Theming Strategy

Goals: Use scalable and efficient styling techniques.
Best Practices:

- Prefer **utility-first CSS** frameworks like **TailwindCSS**.
- Remove unused CSS with **PurgeCSS** (built-in with Tailwind).
- Co-locate styles with components for maintainability.
- Use CSS variables for global theming instead of prop drilling.

Benefits: Smaller CSS bundles and faster style computation.

7. TypeScript & Linting Configuration

Goals: Prevent bugs and anti-patterns that affect performance.
Best Practices:

- Enable "strict": true mode in tsconfig.json.
- Use ESLint plugins like eslint-plugin-react-hooks and eslint-plugin-import.
- Enforce naming, import order, and hook dependency rules.
- Integrate linting and formatting in CI/CD pipelines.

Benefits: Early detection of re-render issues and improved team consistency.

8. Testing & CI/CD Foundation

Goals: Automate quality and performance checks early.
Best Practices:

- Set up **Jest** and **React Testing Library** for unit/integration tests.
- Add **Cypress** for E2E testing if needed.
- Integrate **Lighthouse** and **Web Vitals** reports in CI/CD.
- Enforce pre-commit hooks (husky, lint-staged).

Benefits: Prevents regressions and ensures consistent performance across releases.

9. Deployment & Hosting Setup

Goals: Deliver optimized builds efficiently to end users.
Best Practices:

- Host on CDNs like **Vercel**, **Netlify**, or **CloudFront**.
- Enable **Gzip** or **Brotli** compression.
- Set up caching and long-term asset versioning.
- Optimize and serve images through a CDN (Next.js next/image).

Benefits: Faster page loads, reduced network latency, and better real-world UX.