ChatGPT

HD Connex: Roadmap for Production and TypeScript Integration

The following guide consolidates the recommendations provided for turning the current HTML-embedded React prototype into a robust, deployable web application. Each recommendation addresses the build issues caused by integrating multiple optional features under TypeScript's strict mode and offers strategies for keeping those features while ensuring a smooth build process.

1. Adopt a modern build system

- **Extract your React code from the HTML file.** Inline Babel and development UMD bundles of React slow down the page and produce warnings. The React team recommends using the minified production bundles when deploying [1]. A real build tool such as **Vite** or **Next.js 15** compiles JSX ahead of time, bundles dependencies and optimises for performance.
- **Prefer Next.js** over bare React. Next.js 15 (with Turbopack) offers fast builds, file-based routing and server-side rendering, making it ideal for a multi-page application. It has excellent TypeScript support and removes the need for client-side routers.

2. Initialise a TypeScript-ready project

- Use `npx create-next-app@latest --ts` (or `npx create-react-app --template typescript`) to scaffold the application. These templates generate a `tsconfig.json` and configure ESLint, Prettier and test tooling out of the box.
- Preserve your existing UI and logic by porting each React component into its own `.tsx` file and gradually refactoring any JavaScript files.

3. Configure `tsconfig.json` for gradual migration

- Start with a baseline configuration that supports JavaScript files and prevents strict checking of third-party types:

```
{
  "compilerOptions": {
    "target": "es5",
    "lib": ["esnext"],
    "allowJs": true,
    "skipLibCheck": true,
    "esModuleInterop": true,
    "strict": true,
    "module": "esnext",
    "moduleResolution": "node",
    "isolatedModules": true,
    "noEmit": true
  },
  "include": ["src"]
}
```

This configuration lets you keep JavaScript files during migration and skips type checking of third-party libraries [2]. The `esModuleInterop` flag enables default imports from CommonJS modules [2].

- Replace the single `"strict": true` with individual strict flags – `noImplicitAny`, `noImplicitThis`, `alwaysStrict`, `strictBindCallApply`, `strictNullChecks`, `strictFunctionTypes` and `strictPropertyInitialization` – to turn them on or off individually [3]. This makes it easier to temporarily relax specific checks while you fix type errors.

4. Create explicit types for your components and state

- Define interfaces or types for props and state. For example, a lead capture form might have:

```
interface LeadData {
  name: string;
  email: string;
  phone?: string;
  description: string;
}
interface LeadCaptureProps {
  onSubmit: (data: LeadData) => void;
}
const LeadCaptureForm: React.FC<LeadCaptureProps> = ({ onSubmit }) => { … };
```

- Use generics to type your hooks (`useState<LeadData[]>([])`, `useReducer<ReducerState, ReducerAction>`). This prevents implicit `any` errors and helps the compiler infer types.

5. Model optional features properly

- Represent optional dashboards and pages with union types or discriminated unions. For instance, define a `DashboardView` union (e.g., `'admin' | 'contractor' | 'billing'`) and switch on this type to render the appropriate page.
- Use TypeScript utility types such as `Partial<T>` or `Record<K, T>` to type objects where only some properties are present. This satisfies `strictNullChecks` and reduces runtime checks.

6. Externalise configuration and feature flags

- Move constants like phone numbers, email addresses and API endpoints into environment variables (`.env.local`) and import them via a typed configuration module.

- Implement a simple feature-flag module that exports booleans for optional features. This allows you to enable or disable parts of the UI without introducing undefined values or conditional imports.

7. Use a form library with TypeScript support

Libraries such as `react-hook-form` provide typed form state and validation. They reduce boilerplate and prevent common errors with uncontrolled inputs. For a multi-step onboarding process, they support conditional fields and dynamic validation rules.

8. Skip third-party type checking during migration

Leaving `"skipLibCheck": true` in `tsconfig.json` skips type checking of `node_modules`, which eliminates errors from poorly typed dependencies [2]. Once your codebase is stable, you can remove this option to catch upstream type issues.

9. Build a backend API for dynamic data

Instead of hard-coding contractor profiles, reviews and leads in the front end, implement a simple REST or GraphQL API (using Express, FastAPI or Next.js API routes) with a database. Typed API responses can be consumed via `fetch` or a client like `React Query` for caching and error handling.

10. Introduce state management where necessary

For complex workflows (lead capture, contractor onboarding, billing), consider `useReducer` with context or a library such as Redux Toolkit. Both options have strong TypeScript support and allow you to co-locate actions and reducers, avoiding deeply nested prop drilling.

By following this roadmap – moving to a modern build system, gradually enabling strict TypeScript flags, defining explicit types and modelling optional features correctly – you can preserve all of your existing functionality while resolving the build errors that strict type checking uncovers.

---

[1] Optimizing Performance – React
https://legacy.reactjs.org/docs/optimizing-performance.html

[2] [3] "Strict Mode" TypeScript config options - DEV Community
https://dev.to/jsdev/strict-mode-typescript-j8p