Architecture Document Final

Projeto: EduStream **Fonte analisada:** `/mnt/data/stream-learn-global-main.zip` (código do repositório enviado)

---

1. Versão Final com Todas as Decisões Tomadas

Visão Geral

EduStream é uma aplicação frontend single-page escrita em **React + TypeScript**, empacotada com **Vite**, estilizada com **TailwindCSS**, e organizada como um conjunto de páginas (Dashboard, Courses, LiveClass, Library, Assessments, Gamification, Profile, Settings, MyCourses). O projeto foca em experiência de aprendizagem online com rotas cliente, componentes de UI reutilizáveis e integração com mecanismos de busca/estado local.

Componentes Principais

- **Vite + TypeScript** — build e bundling, desenvolvimento rápido com HMR.
- **React (TSX)** — base da aplicação, arquitetura de componentes.
- **React Router (BrowserRouter)** — roteamento entre páginas (ex.: `/`, `/dashboard`, `/courses`, `/liveclass`, `/profile`).
- **TanStack React Query** — gerenciamento de chamadas de API, cache e estado assíncrono.
- **Context API** — estado global simples (ex.: `SidebarContext` para controlar abertura/fechamento do menu lateral).
- **Componentes UI customizados** — `Navbar`, `NavbarSidebar`, `NavLink`, `Toaster`, `Sonner` (notificações), e componentes de UI localizados em `src/components/ui`.
- **Hooks reutilizáveis** — ex.: `use-mobile`, `use-toast` para comportamento compartilhado.
- **Tailwind CSS + clsx + tailwind-merge** — composição de classes utilitárias e merge seguro de classes dinâmicas.
- **Assets estáticos** — imagens e ícones em `src/assets`.

Integrações Externas e Operacionais

- Dependências visíveis no `node_modules` (por exemplo: `@radix-ui`, `@tanstack/react-query`, `sonner`) indicam uso de bibliotecas para acessibilidade e UI.
- O projeto é um frontend; integrações com backend (APIs de autenticação, streaming, ou store) são acessadas via chamadas HTTP — o código atual organiza pontos de chamadas dentro das páginas e hooks (React Query como camada de abstração).

2. ADRs (Architecture Decision Records)

> Observação: ADRs foram escritas com base na análise do código-fonte presente no zip. Cada ADR contém motivação, alternativas consideradas e impacto.

ADR 001 — Uso de Vite + TypeScript

Decisão: Adotar Vite como bundler e TypeScript como linguagem. **Motivação:** Desenvolvimento rápido, HMR, builds rápidos e tipagem estática para reduzir bugs.
Alternativas Consideradas: Create React App, Webpack puro.
Impacto: Redução do tempo de build e feedback rápido durante desenvolvimento; requer configuração inicial de TS e Vite. **Status:** Aceita

ADR 002 — Gerenciamento de Estado Assíncrono com React Query

Decisão: Usar `@tanstack/react-query` para chamadas API e cache. **Motivação:** Simplificar fetching, retry, cache, sincronização de dados e estados de loading/error sem criar uma camada complexa de redux. **Alternativas Consideradas:** Redux Toolkit, Zustand, SWR.
Impacto: Código de chamadas assíncronas unificado e desacoplado; melhora performance por cache; curva de aprendizado baixa em comparação a soluções mais pesadas. **Status:** Aceita

ADR 003 — Estado Global Simples com Context API

Decisão: Usar React Context para estado global leve (ex.: `SidebarContext`).
Motivação: Simplicidade para estados pequenos e não críticos. **Alternativas Consideradas:** Redux, Zustand.
Impacto: Bom para casos simples; pode ser limitado se o app escalar para muitos estados e atualizações frequentes (rehydration/render costs). **Status:** Aceita

ADR 004 — UI com TailwindCSS e Componentes Atômicos

Decisão: Construir UI com Tailwind + bibliotecas de primitives (`@radix-ui`) e componentes locais. **Motivação:** Rapidez na composição, consistência visual e facilidade de customização. **Alternativas Consideradas:** CSS-in-JS (styled-components), frameworks de componente prontos. **Impacto:** Estilo consistente e baixo custo de manutenção; requer disciplina para classes utilitárias. **Status:** Aceita

ADR 005 — Notificações com Sonner + Toaster personalizados

Decisão: Usar `Sonner` (e um Toaster local) para notificações. **Motivação:** Melhor UX para feedback assíncrono (ações, erros, confirmações). **Alternativas Consideradas:** Toasts caseiros, bibliotecas alternativas. **Impacto:** UX consistente; ocupa pequena parcela do bundle se importado modularmente. **Status:** Aceita

---

3. Justificativas para Trade-offs Realizados

- **Simplicidade vs Flexibilidade:** Optou-se por Context API para casos simples em vez de Redux para evitar sobrecarga. Trade-off: fácil de implementar, mas pode precisar de migração se o estado crescer.
- **React Query vs Redux para dados remotos:** React Query oferece gerenciamento de cache, refetch e sincronização que reduziram a necessidade de uma store global para dados remotos — melhora produtividade e performance, porém delega lógica de normalização e atualizações complexas para cada query/mutation.
- **Tailwind (utilitário) vs componente visual pronto:** Tailwind dá velocidade e flexibilidade para criar UI customizada, mas exige consistência e disciplina nos utilitários; bibliotecas pré-construídas acelerariam entrega, porém limitariam customização.
- **Vite (dev experience) vs Webpack (maturidade/ecossistema):** Vite favorece DX e builds rápidos, mas algumas integrações muito específicas podem demandar ajustes extras comparado a Webpack.

---

4. Lições Aprendidas

- **Manter ADRs atualizadas** — registrar decisões (como as acima) evita re-trabalho e esclarece motivos futuros.
- **Isolar chamadas de API** — centralizar hooks/clients (por exemplo `apiClient.ts` e hooks `useCourses`) facilita testes e mudanças de endpoint/autenticação.
- **Limitar responsabilidades dos componentes** — separar lógica (hooks) da apresentação reduziu duplicação e facilitou testes.
- **Planejar crescimento do estado global** — se o número de estados compartilhados crescer, migrar para uma store (Zustand/Redux) pode ser necessário.

---

5. Trabalhos Futuros e Melhorias Possíveis

- **Adicionar uma camada de teste automatizado** (unit + integration) com Jest + React Testing Library.
- **Criar uma documentação de API e contratos** (OpenAPI) e uma camada de cliente gerada para reduzir mismatches.
- **Implementar SSR/SSG se necessário**: considerar Next.js caso SEO e renderização no servidor se tornem requisitos.

- **Observability**: adicionar Sentry/LogRocket e métricas de performance para rastrear erros e latência do front.
- **Melhorar CI/CD**: adicionar pipeline para lint, types, testes e build, e deploy automático (Netlify/Vercel/Cloudfront).

---

6. Anexos e Referências

- Repositório analisado: `/mnt/data/stream-learn-global-main.zip` (código fonte enviado).
- Principais arquivos verificados: `src/App.tsx`, `src/main.tsx`, `src/context/SidebarContext.tsx`, `src/pages/*`, `src/components/*`.
