FRONTEND REFACTOR

Next.js → Vite + React

Migration PRD & Implementation Guide

Version 1.0	Stack: Vite + React	Auth: Custom JWT	API: React Query

---

1. Overview & Goals

This document outlines the complete plan for migrating the frontend from Next.js (App Router) to Vite + React. The primary driver is simplification: the app uses zero SSR or SSG, meaning Next.js complexity is overhead with no benefit.

Migration Goals

- Eliminate Next.js overhead (App Router, server/client split, SSR config)
- Adopt a clean, feature-based folder structure from scratch
- Replace Auth.js (NextAuth) with a lightweight custom JWT auth layer
- Replace SWR with React Query (TanStack Query) for data fetching
- Maintain full parity with existing features, no regressions
- Improve DX: faster HMR, simpler mental model, easier onboarding

Why Not Stay on Next.js?

Concern	Next.js (Current)	Vite + React (Target)
Rendering	All pages are 'use client' — SSR unused	Pure CSR by design, no wasted config
Routing	App Router conventions + layouts overhead	React Router v7 — explicit, simple
Auth	NextAuth tightly coupled to Next.js	Custom useAuth hook, full control
Data Fetching	SWR + manual token threading	React Query with centralized config
Bundle Config	Next.js opinionated build pipeline	Vite — fast, transparent, flexible
Mental Model	Server/client boundary confusion	Pure React, no framework magic

2. Target Tech Stack

Category	Package	Replaces
Bundler	`Vite + @vitejs/plugin-react`	Next.js build system
Routing	`React Router v7`	Next.js App Router / file-based routing
Data Fetching	`@tanstack/react-query v5`	SWR (useSWR)
Auth	`Custom hooks + Context`	NextAuth / Auth.js
Global State	`Zustand (if needed)`	React Context sprawl
HTTP Client	`Native fetch (via React Query)`	Custom fetcher with SWR
Toast	`react-hot-toast (keep)`	react-hot-toast (unchanged)
Language	`TypeScript (keep)`	TypeScript (unchanged)

3. Target Folder Structure

Feature-based structure: each domain owns its components, hooks, and API calls. Shared code lives in /shared. Pages are thin route-level components only.

```
src/
  features/                  # one folder per domain
    auth/
      components/             # LoginForm, ProtectedRoute
      hooks/                  # useAuth.ts
      api.ts                  # login(), logout(), getMe()
      AuthProvider.tsx         # token context + React Query
    dashboard/
      components/
      hooks/
      api.ts
    [other-features]/          # same pattern per feature
  shared/
    hooks/                    # useApiFetch.ts, useDebounce.ts
    components/               # Button, Modal, Table, etc.
    fetcher.ts                # base fetch utilities
    types.ts                  # HookReply, ApiError, etc.
    api-url.ts                # BASE_URL constants
  pages/                      # thin route components
    LoginPage.tsx
    DashboardPage.tsx
```

```
  router/
    index.tsx                 # all routes defined here
    ProtectedRoute.tsx
  lib/
    queryClient.ts            # React Query config
  App.tsx
  main.tsx
```

---

4. Auth Migration (NextAuth → Custom)

Since the Python backend issues JWT bearer tokens, Auth.js was only acting as a token storage and session wrapper. All of that is replaced by a simple AuthProvider + useAuth hook.

4.1 What Auth.js Was Doing

- Storing the JWT token in a session
- Exposing useSession() hook to get the token
- Handling the session lifecycle (sign in / sign out)

All of this is replicated in ~80 lines of custom code with full control.

4.2 Token Storage Strategy

Recommended	Ask your Python BE to set an httpOnly cookie on successful login. The browser handles it automatically — you never touch the token in JS.

Alternative	Store token in localStorage. Simpler to implement but slightly less secure (XSS risk). Acceptable for internal tools.

4.3 AuthProvider Implementation

```
// src/features/auth/AuthProvider.tsx
const AuthContext = createContext<AuthContextType>(null!)

export function AuthProvider({ children }) {
  const [token, setToken] = useState<string | null>(
    () => localStorage.getItem('token')   // or read from cookie
  )

  const login = (newToken: string) => {
    localStorage.setItem('token', newToken)
    setToken(newToken)
  }
```

```
  const logout = () => {
    localStorage.removeItem('token')
    setToken(null)
  }

  return (
    <AuthContext.Provider value={{ token, login, logout }}>
      {children}
    </AuthContext.Provider>
  )
}

export const useAuth = () => useContext(AuthContext)
```

4.4 ProtectedRoute

```
// src/router/ProtectedRoute.tsx
export function ProtectedRoute() {
  const { token } = useAuth()
  if (!token) return <Navigate to='/login' replace />
  return <Outlet />
}
```

---

5. Data Fetching Migration (SWR → React Query)

The existing useApiFetch hook is preserved almost identically, but backed by React Query instead of SWR. All error handling logic, token threading, and loading state management carry over.

5.1 React Query Setup

```
// src/lib/queryClient.ts
import { QueryClient } from '@tanstack/react-query'

export const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      retry: 1,
      staleTime: 1000 * 60 * 5,   // 5 minutes
      refetchOnWindowFocus: false,
    }
  }
})

// src/main.tsx
```

```
<QueryClientProvider client={queryClient}>
  <AuthProvider>
    <RouterProvider router={router} />
  </AuthProvider>
</QueryClientProvider>
```

5.2 useApiFetch Hook (Migrated)

```
// src/shared/hooks/useApiFetch.ts
export function useApiFetch<T>(
  fetchUrl: string,
  isReadyCondition = true,
  options?: UseQueryOptions<T, ApiError>
): HookReply<T> {
  const navigate = useNavigate()
  const { pathname } = useLocation()
  const { token, logout } = useAuth()

  const { data, error, isLoading, isRefetching, refetch } =
    useQuery<T, ApiError>({
      queryKey: [fetchUrl, token],
      queryFn: () => token
        ? fetcher([fetchUrl, token])
        : fetcherNoToken(fetchUrl),
      enabled: isReadyCondition,
      ...options,
    })

  if (error && pathname !== '/login') {
    handleApiError(error, navigate, logout)
  }

  return {
    data,
    error,
    isLoading: !isReadyCondition || isLoading,
    isValidating: isRefetching,
    refetch,  // was: mutate
  }
}
```

5.3 Migration Mapping (SWR → React Query)

SWR (Before)	React Query (After)
`mutate()`	`refetch()`

`isValidating`	`isRefetching`
`useSWR(key, fetcher)`	`useQuery({ queryKey, queryFn })`
`SWRConfiguration`	`UseQueryOptions`
`swrConfig.onError`	`queryClient.setDefaultOptions or in hook`
`useSession() → token`	`useAuth() → token`
`useRouter() (next)`	`useNavigate() (react-router-dom)`
`usePathname() (next)`	`useLocation().pathname`

---

6. Routing (File-Based → Explicit)

Next.js App Router uses a file-based routing convention. React Router v7 uses a single explicit route config — simpler and easier to read at a glance.

6.1 Router Setup

```
// src/router/index.tsx
const router = createBrowserRouter([
  {
    path: '/login',
    element: <LoginPage />,
  },
  {
    element: <ProtectedRoute />,   // wraps all auth-required routes
    children: [
      { path: '/', element: <DashboardPage /> },
      { path: '/settings', element: <SettingsPage /> },
      // add all your existing pages here
    ]
  }
])
```

6.2 Next.js → React Router Mapping

Next.js	React Router v7
`useRouter().push('/x')`	`useNavigate()('/x') or navigate('/x')`
`usePathname()`	`useLocation().pathname`
`useParams()`	`useParams() (same API)`
`useSearchParams()`	`useSearchParams() (same API)`
`<Link href='/x'>`	`<Link to='/x'>`

`redirect()` (server)	`navigate('/x')` in effect/handler
`app/page.tsx` (route)	Route in `router/index.tsx`
`app/layout.tsx`	Wrapper component around `<Outlet />`

---

7. Implementation Phases

Phase 1 — Project Bootstrap

Estimated: 1 day

1	Scaffold new project: npm create vite@latest my-app -- --template react-ts
2	Install core dependencies: react-router-dom, @tanstack/react-query, react-hot-toast, zustand
3	Configure vite.config.ts: path aliases (@/ → src/), proxy to Python BE for local dev
4	Set up tsconfig.json path aliases to match vite config
5	Create the folder structure: features/, shared/, pages/, router/, lib/
6	Set up .env with VITE_API_URL (replaces Next.js environment variables)

Phase 2 — Auth Layer

Estimated: 1 day

1	Create src/features/auth/AuthProvider.tsx with token state, login(), logout()
2	Create src/features/auth/hooks/useAuth.ts exporting the context hook
3	Create src/features/auth/api.ts with login API call to Python BE
4	Build LoginPage that calls login() and stores token on success
5	Create src/router/ProtectedRoute.tsx that checks token and redirects to /login
6	Wrap App.tsx with AuthProvider + QueryClientProvider
7	Test login → token stored → protected route accessible → logout clears token

Phase 3 — Data Fetching Layer

Estimated: 1 day

1	Create src/lib/queryClient.ts with React Query config (staleTime, retry, refetchOnWindowFocus)
2	Migrate src/shared/fetcher.ts — update to work without Next.js context
3	Migrate src/shared/types.ts — HookReply, ApiError types
4	Create src/shared/hooks/useApiFetch.ts using React Query (see Section 5.2)

5	Create handleApiError() utility with all existing error handling (401/402/403/404/5xx)
6	Replace useRouter/usePathname (next) with useNavigate/useLocation (react-router)
7	Test a single feature end-to-end with the new fetching layer before migrating all

Phase 4 — Feature Migration

Estimated: 2-3 days (depends on feature count)

1	Migrate features one at a time, starting with the simplest (least API calls)
2	For each feature: copy components → update imports → replace next/* with react-router-dom/*
3	Replace all useSession() references with useAuth()
4	Replace all mutate() with refetch() where applicable
5	Replace all next/link <Link href=> with react-router <Link to=>
6	Remove all 'use client' directives (not needed in Vite)
7	Test each feature before moving to the next one

Phase 5 — Routing & Polish

Estimated: 0.5 day

1	Define all routes in src/router/index.tsx based on existing Next.js pages
2	Set up nested routing with ProtectedRoute wrapping authenticated routes
3	Configure 404 catch-all route
4	Add React Query Devtools for development (remove in prod build)
5	Final end-to-end test of all features and auth flow

---

8. Pre-Launch Checklist

- All 'use client' directives removed
- All next/* imports replaced with react-router-dom/* equivalents
- useSession() fully replaced by useAuth()
- All SWR mutate() calls replaced with React Query refetch()
- Environment variables using VITE_ prefix (not NEXT_PUBLIC_)
- Login → redirect → protected pages flow working
- Logout clears token and redirects to /login
- 401/403/402 error handling triggers correct behavior
- Python BE API calls working via VITE_API_URL

- vite.config.ts proxy configured for local development
- Production build (vite build) runs without errors
- React Query Devtools removed from production build

---

9. Environment Variables

Next.js uses NEXT_PUBLIC_ prefix for client-side env vars. Vite uses VITE_ prefix. Update all .env files accordingly.

```
# .env (Vite)
VITE_API_URL=http://localhost:8000
VITE_AUTH_API_URL=http://localhost:8000/auth

# Access in code
const BASE_URL = import.meta.env.VITE_API_URL
// was: process.env.NEXT_PUBLIC_API_URL
```
