**PM0: Greenfield Architecture Design**

**Strategic Planning Protocol - Built Right From Scratch**

**EXECUTIVE PHILOSOPHY**

**Core Principle:** Build the simplest thing that could possibly work, then evolve based on real usage patterns.

**Anti-Patterns to Avoid:**

* ❌ Over-engineering before understanding actual needs
* ❌ Premature abstraction ("we might need this later")
* ❌ Configuration complexity that should be code
* ❌ "Demo mode" as an afterthought bolted onto production code
* ❌ Build-time dependencies that should be runtime

**Design Mantras:**

* ✅ **Explicit over implicit** - No magic, clear data flow
* ✅ **Fail fast, fail loud** - Errors should be obvious and actionable
* ✅ **Testability from day one** - Architecture enables easy testing
* ✅ **Local-first development** - Work offline, sync when ready
* ✅ **Progressive enhancement** - Basic features work everywhere, advanced features where possible

**UNDERSTANDING PM0 ESSENCE**

**What PM0 Actually Is**

A **strategic planning protocol** that helps teams:

1. Define strategic scenarios ("what if we pursue strategy X?")
2. Map dependencies and relationships
3. Track progress and outcomes
4. Collaborate on strategic decisions

**Core Value Propositions**

1. **Clarity** - Visualize complex strategy decisions
2. **Alignment** - Teams see the same strategic picture
3. **Agility** - Quickly model and compare strategic options
4. **Memory** - Institutional knowledge captured and accessible

**Critical User Journeys**

1. **Solo strategist** - Individual exploring "what if" scenarios
2. **Team collaboration** - Multiple people contributing to strategy
3. **Presentation mode** - Sharing strategic analysis with stakeholders
4. **Historical review** - Looking back at past strategic decisions

**GREENFIELD ARCHITECTURE**

**Technology Stack (Simplified & Justified)**

// Core Stack

{

"frontend": "React 18 + TypeScript", // Industry standard, mature

"routing": "React Router v6", // Dead simple, no magic

"state": "Zustand + React Query", // Minimal boilerplate, clear patterns

"database": "Supabase (Postgres)", // Real-time, auth built-in, great DX

"styling": "Tailwind CSS", // Fast iteration, consistent design

"deployment": "Vercel", // Zero-config deploys

"build": "Vite", // Fast, modern, simple config

}

// Notably Absent

{

"NO\_REDUX": "Too much boilerplate for our needs",

"NO\_NEXT": "Don't need SSR, adds complexity",

"NO\_GRAPHQL": "REST + Supabase RPC is simpler",

"NO\_MICROSERVICES": "Monolith until we have scaling problems",

"NO\_MONO\_REPO": "Single app, single repo",

}

**File Structure (Organized by Feature)**

pm0/

├── src/

│ ├── app/ # App-level concerns

│ │ ├── App.tsx # Root component

│ │ ├── Router.tsx # Route definitions

│ │ └── providers.tsx # Context providers

│ │

│ ├── features/ # Feature modules (vertical slices)

│ │ ├── auth/ # Authentication

│ │ │ ├── components/

│ │ │ ├── hooks/

│ │ │ ├── api.ts # Auth API calls

│ │ │ ├── store.ts # Auth state

│ │ │ └── index.ts # Public exports

│ │ │

│ │ ├── scenarios/ # Scenario management

│ │ │ ├── components/

│ │ │ │ ├── ScenarioCard.tsx

│ │ │ │ ├── ScenarioEditor.tsx

│ │ │ │ └── ScenarioList.tsx

│ │ │ ├── hooks/

│ │ │ │ ├── useScenarios.ts

│ │ │ │ ├── useCreateScenario.ts

│ │ │ │ └── useUpdateScenario.ts

│ │ │ ├── api.ts

│ │ │ ├── store.ts

│ │ │ ├── types.ts

│ │ │ └── index.ts

│ │ │

│ │ ├── workspaces/ # Workspace/project management

│ │ ├── dependencies/ # Dependency mapping

│ │ ├── timeline/ # Timeline views

│ │ └── collaboration/ # Real-time collaboration

│ │

│ ├── shared/ # Shared across features

│ │ ├── components/ # Reusable UI components

│ │ │ ├── Button.tsx

│ │ │ ├── Card.tsx

│ │ │ ├── Modal.tsx

│ │ │ └── ...

│ │ ├── hooks/ # Reusable hooks

│ │ │ ├── useDebounce.ts

│ │ │ ├── useLocalStorage.ts

│ │ │ └── useMediaQuery.ts

│ │ ├── layouts/ # Layout components

│ │ │ ├── AppLayout.tsx

│ │ │ └── AuthLayout.tsx

│ │ └── utils/ # Pure utility functions

│ │ ├── date.ts

│ │ ├── string.ts

│ │ └── validation.ts

│ │

│ ├── lib/ # Core infrastructure

│ │ ├── supabase/

│ │ │ ├── client.ts # Supabase client singleton

│ │ │ ├── types.ts # Generated DB types

│ │ │ └── hooks.ts # Base Supabase hooks

│ │ ├── config/

│ │ │ └── env.ts # Environment configuration

│ │ └── query/

│ │ └── client.ts # React Query configuration

│ │

│ └── main.tsx # Entry point

│

├── supabase/

│ ├── migrations/ # Database migrations

│ │ ├── 20240101\_initial\_schema.sql

│ │ ├── 20240102\_add\_scenarios.sql

│ │ └── ...

│ ├── seed.sql # Seed data for development

│ └── config.toml # Supabase config

│

├── tests/

│ ├── e2e/ # Playwright tests

│ ├── integration/ # Integration tests

│ └── unit/ # Unit tests (colocated preferred)

│

├── docs/

│ ├── architecture.md

│ ├── features.md

│ └── development.md

│

├── .env.example

├── package.json

├── tsconfig.json

├── vite.config.ts

└── README.md

**Key Architectural Decisions**

**1. Feature-Based Organization (Not Layer-Based)**

**Why:** Features change together. When you work on "scenarios", you touch components, hooks, API, state, and types all related to scenarios. Organizing by feature keeps related code close.

// ✅ GOOD: Feature-based

features/scenarios/

├── components/ScenarioEditor.tsx

├── hooks/useScenarios.ts

├── api.ts

├── store.ts

└── types.ts

// ❌ BAD: Layer-based

components/ScenarioEditor.tsx

hooks/useScenarios.ts

api/scenarios.ts

store/scenarios.ts

types/scenarios.ts

**2. Database-Driven Types (Single Source of Truth)**

// lib/supabase/client.ts

import { createClient } from '@supabase/supabase-js';

import type { Database } from './types'; // Generated from DB schema

export const supabase = createClient<Database>(

import.meta.env.VITE\_SUPABASE\_URL,

import.meta.env.VITE\_SUPABASE\_ANON\_KEY

);

// Types flow FROM database TO application

export type Scenario = Database['public']['Tables']['scenarios']['Row'];

export type ScenarioInsert = Database['public']['Tables']['scenarios']['Insert'];

export type ScenarioUpdate = Database['public']['Tables']['scenarios']['Update'];

**Why:** The database schema is the source of truth. Generate TypeScript types from it, don't maintain them separately.

**3. No "Demo Mode" - Use Real Database with Seed Data**

// ❌ OLD APPROACH: Separate demo mode with fake client

const client = USE\_DEMO\_DATA ? null : createClient(...);

// ✅ NEW APPROACH: Use real Supabase, seed with demo data

// Everyone uses the same code paths

// Development: Local Supabase with seed data

// Production: Hosted Supabase

**Why:**

* Eliminates entire class of bugs (demo mode doesn't match production)
* Simplifies code (no branching logic)
* Better testing (test against real database)
* Faster development (everyone uses same stack)

**Local Development:**

# Start local Supabase (includes Postgres, Auth, Storage)

supabase start

# Apply migrations

supabase db reset

# Seed with demo data

supabase db seed

# App connects to localhost:54321 automatically

npm run dev

**4. Configuration: Simple & Explicit**

// lib/config/env.ts

// NO Zod, NO runtime validation, NO complex transforms

// Just simple, typed environment access

const requiredEnv = (key: string): string => {

const value = import.meta.env[key];

if (!value) {

throw new Error(

`Missing required environment variable: ${key}\n` +

`Check your .env file or deployment configuration.`

);

}

return value;

};

export const env = {

supabase: {

url: requiredEnv('VITE\_SUPABASE\_URL'),

anonKey: requiredEnv('VITE\_SUPABASE\_ANON\_KEY'),

},

sentry: {

dsn: import.meta.env.VITE\_SENTRY\_DSN || null,

},

app: {

isDev: import.meta.env.DEV,

isProd: import.meta.env.PROD,

},

} as const;

**Why:**

* Fails fast on missing config (at module load)
* No silent failures with empty strings
* TypeScript knows exactly what's available
* No runtime overhead validating on every page load

**5. State Management: Zustand for Local, React Query for Server**

// features/scenarios/store.ts

import { create } from 'zustand';

// LOCAL UI state only (selected items, filters, UI toggles)

export const useScenariosStore = create<{

selectedScenarioId: string | null;

viewMode: 'list' | 'grid' | 'timeline';

setSelectedScenarioId: (id: string | null) => void;

setViewMode: (mode: 'list' | 'grid' | 'timeline') => void;

}>((set) => ({

selectedScenarioId: null,

viewMode: 'list',

setSelectedScenarioId: (id) => set({ selectedScenarioId: id }),

setViewMode: (mode) => set({ viewMode: mode }),

}));

// features/scenarios/hooks/useScenarios.ts

import { useQuery } from '@tanstack/react-query';

import { supabase } from '@/lib/supabase/client';

// SERVER state via React Query (data, loading, error)

export function useScenarios(workspaceId: string) {

return useQuery({

queryKey: ['scenarios', workspaceId],

queryFn: async () => {

const { data, error } = await supabase

.from('scenarios')

.select('\*')

.eq('workspace\_id', workspaceId);

if (error) throw error;

return data;

},

});

}

**Why:**

* Clear separation: UI state vs server state
* React Query handles caching, refetching, optimistic updates
* Zustand handles local UI preferences
* No confusion about where state lives

**6. API Layer: Thin Wrappers, Not Abstractions**

// features/scenarios/api.ts

import { supabase } from '@/lib/supabase/client';

import type { ScenarioInsert, ScenarioUpdate } from '@/lib/supabase/types';

// Simple, typed wrappers around Supabase client

export const scenariosApi = {

async list(workspaceId: string) {

const { data, error } = await supabase

.from('scenarios')

.select('\*')

.eq('workspace\_id', workspaceId)

.order('created\_at', { ascending: false });

if (error) throw error;

return data;

},

async create(scenario: ScenarioInsert) {

const { data, error } = await supabase

.from('scenarios')

.insert(scenario)

.select()

.single();

if (error) throw error;

return data;

},

async update(id: string, updates: ScenarioUpdate) {

const { data, error } = await supabase

.from('scenarios')

.update(updates)

.eq('id', id)

.select()

.single();

if (error) throw error;

return data;

},

async delete(id: string) {

const { error } = await supabase

.from('scenarios')

.delete()

.eq('id', id);

if (error) throw error;

},

};

**Why:**

* Supabase client is already well-designed, don't hide it
* Type safety from database types
* Easy to test (mock Supabase client)
* Easy to debug (see actual queries)
* No "Repository pattern" complexity

**DATABASE SCHEMA DESIGN**

**Core Principles**

1. **Normalized but practical** - Don't over-normalize
2. **Use Postgres features** - JSONB, arrays, triggers, RLS
3. **Soft deletes** - Keep audit trails
4. **Timestamps everywhere** - created\_at, updated\_at on all tables
5. **UUIDs for IDs** - Distributed-friendly, non-guessable

**Schema Overview**

-- Core entities

CREATE TABLE workspaces (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

name TEXT NOT NULL,

description TEXT,

owner\_id UUID REFERENCES auth.users NOT NULL,

created\_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),

updated\_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),

deleted\_at TIMESTAMPTZ

);

CREATE TABLE workspace\_members (

workspace\_id UUID REFERENCES workspaces NOT NULL,

user\_id UUID REFERENCES auth.users NOT NULL,

role TEXT NOT NULL CHECK (role IN ('owner', 'editor', 'viewer')),

created\_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),

PRIMARY KEY (workspace\_id, user\_id)

);

CREATE TABLE scenarios (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

workspace\_id UUID REFERENCES workspaces NOT NULL,

name TEXT NOT NULL,

description TEXT,

status TEXT NOT NULL DEFAULT 'draft' CHECK (status IN ('draft', 'active', 'completed', 'archived')),

metadata JSONB DEFAULT '{}', -- Flexible for custom fields

created\_by UUID REFERENCES auth.users NOT NULL,

created\_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),

updated\_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),

deleted\_at TIMESTAMPTZ

);

CREATE TABLE dependencies (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

scenario\_id UUID REFERENCES scenarios NOT NULL,

from\_item TEXT NOT NULL, -- What depends on something

to\_item TEXT NOT NULL, -- What is depended upon

dependency\_type TEXT NOT NULL CHECK (dependency\_type IN ('blocks', 'requires', 'relates\_to')),

created\_at TIMESTAMPTZ NOT NULL DEFAULT NOW()

);

-- Indexes for common queries

CREATE INDEX idx\_scenarios\_workspace ON scenarios(workspace\_id) WHERE deleted\_at IS NULL;

CREATE INDEX idx\_scenarios\_created\_by ON scenarios(created\_by) WHERE deleted\_at IS NULL;

CREATE INDEX idx\_dependencies\_scenario ON dependencies(scenario\_id);

CREATE INDEX idx\_workspace\_members\_user ON workspace\_members(user\_id);

-- Automatic updated\_at trigger

CREATE OR REPLACE FUNCTION update\_updated\_at()

RETURNS TRIGGER AS $$

BEGIN

NEW.updated\_at = NOW();

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER scenarios\_updated\_at

BEFORE UPDATE ON scenarios

FOR EACH ROW

EXECUTE FUNCTION update\_updated\_at();

-- Row Level Security (RLS)

ALTER TABLE workspaces ENABLE ROW LEVEL SECURITY;

ALTER TABLE scenarios ENABLE ROW LEVEL SECURITY;

ALTER TABLE dependencies ENABLE ROW LEVEL SECURITY;

-- Users can only see workspaces they're members of

CREATE POLICY "Users can view their workspaces"

ON workspaces FOR SELECT

USING (

owner\_id = auth.uid() OR

EXISTS (

SELECT 1 FROM workspace\_members

WHERE workspace\_id = workspaces.id

AND user\_id = auth.uid()

)

);

-- Similar policies for scenarios, dependencies, etc.

**Why This Schema:**

* Simple and extensible
* Leverages Postgres features (JSONB for flexibility)
* RLS provides security at database level
* Soft deletes preserve history
* Automatic timestamps reduce bugs

**IMPLEMENTATION ROADMAP**

**Sprint 0: Foundation (1 week)**

**Goal:** Deployable "Hello World" with auth

// Deliverables:

1. Repository setup

2. Vite + React + TypeScript configured

3. Supabase project created

4. Authentication working (magic link)

5. Basic routing (/login, /dashboard)

6. Deployed to Vercel

7. CI/CD pipeline (lint, type-check, test)

// Tech tasks:

- Initialize Vite project

- Configure Tailwind CSS

- Setup Supabase (local + hosted)

- Create initial migration (users, workspaces)

- Implement auth flow

- Deploy to Vercel

- Setup GitHub Actions

**Sprint 1: Core Data Model (1 week)**

**Goal:** Create, read, update, delete scenarios

// Deliverables:

1. Database schema for scenarios

2. Scenario CRUD operations

3. Scenario list page

4. Scenario detail/editor page

5. Real-time updates working

6. Basic error handling

// Tech tasks:

- Migration: scenarios table

- Generate TypeScript types

- Build scenariosApi

- Create useScenarios hook

- Build ScenarioList component

- Build ScenarioEditor component

- Setup React Query

- Test real-time subscriptions

**Sprint 2: Collaboration Features (1 week)**

**Goal:** Multiple users can work together

// Deliverables:

1. Workspace members management

2. Permissions (owner/editor/viewer)

3. Real-time collaboration indicators

4. Activity feed

5. User profiles

// Tech tasks:

- Migration: workspace\_members

- Implement RLS policies

- Build member invitation flow

- Add presence indicators

- Create activity log

- Build user profile page

**Sprint 3: Dependencies & Relationships (1 week)**

**Goal:** Map dependencies between elements

// Deliverables:

1. Dependency creation

2. Dependency visualization

3. Impact analysis ("what depends on this?")

4. Dependency types (blocks, requires, relates)

// Tech tasks:

- Migration: dependencies table

- Build dependency UI

- Create graph visualization

- Implement dependency queries

- Add dependency validation

**Sprint 4: Polish & Launch (1 week)**

**Goal:** Production-ready application

// Deliverables:

1. Onboarding flow

2. Documentation

3. Performance optimization

4. Error tracking (Sentry)

5. Analytics

6. Beta launch

// Tech tasks:

- Build onboarding wizard

- Write user documentation

- Optimize bundle size

- Add loading states everywhere

- Setup Sentry

- Add analytics events

- Load testing

- Security audit

**DEVELOPMENT WORKFLOW**

**Local Development**

# Terminal 1: Start Supabase (Postgres, Auth, Storage, etc.)

supabase start

# Terminal 2: Start Vite dev server

npm run dev

# Terminal 3: Watch tests

npm run test:watch

**Database Workflow**

# Create a new migration

supabase migration new add\_scenarios\_table

# Edit: supabase/migrations/YYYYMMDD\_add\_scenarios\_table.sql

# Apply locally

supabase db reset

# Generate TypeScript types

npm run typegen

# Commit migration to git

git add supabase/migrations/

git commit -m "feat: add scenarios table"

**Feature Development Workflow**

# 1. Create feature branch

git checkout -b feat/scenario-editor

# 2. Create migration if needed

supabase migration new scenario\_metadata

# 3. Build feature

# - Create types

# - Create API functions

# - Create hooks

# - Create components

# - Write tests

# 4. Test locally

npm run test

npm run typecheck

npm run lint

# 5. Push and create PR

git push origin feat/scenario-editor

gh pr create

# 6. Merge after CI passes

gh pr merge --squash --delete-branch

**Deployment**

# Push to main triggers automatic deployment

git push origin main

# Vercel builds and deploys automatically

# Supabase migrations run automatically via CLI in build step

**TESTING STRATEGY**

**Test Pyramid**

/\

/E2E\ Few: Critical user journeys

/------\

/Integ. \ Some: Feature workflows

/----------\

/ Unit \ Many: Pure functions, utilities

/--------------\

**What to Test (and What Not To)**

**✅ DO Test:**

* Pure functions (utilities, validators, formatters)
* Business logic (scenario validation, dependency rules)
* Hooks with complex logic
* Critical user paths (auth flow, create scenario, invite member)
* API error handling

**❌ DON'T Test:**

* Implementation details (component internal state)
* Third-party libraries (Supabase, React Query)
* Trivial code (getters, setters)
* UI styling
* Database queries (trust Supabase)

**Example Tests**

// tests/unit/utils/validation.test.ts

describe('validateScenarioName', () => {

it('accepts valid names', () => {

expect(validateScenarioName('Q1 Strategy')).toBe(true);

});

it('rejects empty names', () => {

expect(validateScenarioName('')).toBe(false);

});

it('rejects names over 100 characters', () => {

expect(validateScenarioName('a'.repeat(101))).toBe(false);

});

});

// tests/integration/scenarios/create.test.tsx

describe('Create Scenario', () => {

it('creates a new scenario', async () => {

const user = await createTestUser();

const { getByRole, getByLabelText } = render(

<App />,

{ user }

);

// Navigate to scenarios page

await user.click(getByRole('link', { name: /scenarios/i }));

// Click "New Scenario"

await user.click(getByRole('button', { name: /new scenario/i }));

// Fill form

await user.type(getByLabelText(/name/i), 'Test Scenario');

await user.type(getByLabelText(/description/i), 'Test description');

// Submit

await user.click(getByRole('button', { name: /create/i }));

// Verify scenario appears

await waitFor(() => {

expect(screen.getByText('Test Scenario')).toBeInTheDocument();

});

});

});

// tests/e2e/scenarios.spec.ts

test('complete scenario workflow', async ({ page }) => {

// Login

await page.goto('/login');

await page.fill('[name="email"]', 'test@example.com');

await page.click('button:has-text("Send magic link")');

// ... handle magic link email ...

// Create scenario

await page.click('text=New Scenario');

await page.fill('[name="name"]', 'Q1 Strategy');

await page.click('button:has-text("Create")');

// Verify created

await expect(page.locator('text=Q1 Strategy')).toBeVisible();

// Edit scenario

await page.click('text=Q1 Strategy');

await page.fill('[name="description"]', 'Updated description');

await page.click('button:has-text("Save")');

// Verify updated

await expect(page.locator('text=Updated description')).toBeVisible();

});

**PERFORMANCE CONSIDERATIONS**

**Bundle Size Optimization**

// vite.config.ts

export default defineConfig({

build: {

rollupOptions: {

output: {

manualChunks: {

'react-vendor': ['react', 'react-dom', 'react-router-dom'],

'ui-vendor': ['lucide-react'], // Icon library

'data-vendor': ['@tanstack/react-query', 'zustand'],

'supabase-vendor': ['@supabase/supabase-js'],

},

},

},

},

});

**Target:** <200KB initial bundle, <500KB total

**Database Query Optimization**

// ✅ GOOD: Select only needed columns

const { data } = await supabase

.from('scenarios')

.select('id, name, status') // Only what we need

.eq('workspace\_id', workspaceId);

// ❌ BAD: Select everything

const { data } = await supabase

.from('scenarios')

.select('\*'); // Returns unnecessary data

// ✅ GOOD: Use indexes

CREATE INDEX idx\_scenarios\_workspace\_status

ON scenarios(workspace\_id, status)

WHERE deleted\_at IS NULL;

// ✅ GOOD: Paginate long lists

const { data } = await supabase

.from('scenarios')

.select('\*')

.range(0, 49) // First 50 items

.order('created\_at', { ascending: false });

**Real-time Subscription Management**

// Only subscribe to data you're actively viewing

function ScenarioDetail({ id }: { id: string }) {

useEffect(() => {

// Subscribe when component mounts

const subscription = supabase

.channel(`scenario:${id}`)

.on('postgres\_changes', {

event: '\*',

schema: 'public',

table: 'scenarios',

filter: `id=eq.${id}`,

}, handleChange)

.subscribe();

// Unsubscribe when component unmounts

return () => {

subscription.unsubscribe();

};

}, [id]);

}

**SECURITY BEST PRACTICES**

**1. Row Level Security (RLS) - Database First**

-- Security at the database level (not application level)

-- Even if someone bypasses your frontend, they can't access unauthorized data

-- Users can only read scenarios from their workspaces

CREATE POLICY "Users can read their workspace scenarios"

ON scenarios FOR SELECT

USING (

workspace\_id IN (

SELECT workspace\_id FROM workspace\_members

WHERE user\_id = auth.uid()

)

);

-- Users can only modify scenarios if they're editors/owners

CREATE POLICY "Editors can update scenarios"

ON scenarios FOR UPDATE

USING (

workspace\_id IN (

SELECT workspace\_id FROM workspace\_members

WHERE user\_id = auth.uid()

AND role IN ('owner', 'editor')

)

);

**2. Validation - Client and Server**

// Client-side: Fast feedback for users

function ScenarioForm() {

const [name, setName] = useState('');

const nameError = name.length > 100 ? 'Name too long' : null;

// ... show nameError to user

}

// Server-side: PostgreSQL constraints (can't be bypassed)

CREATE TABLE scenarios (

name TEXT NOT NULL CHECK (length(name) <= 100),

status TEXT NOT NULL CHECK (status IN ('draft', 'active', 'completed'))

);

**3. Authentication - Supabase Auth**

// Use Supabase Auth (don't roll your own)

// Supports: Magic links, OAuth, passwords, etc.

// Login with magic link

const { error } = await supabase.auth.signInWithOtp({

email: 'user@example.com',

});

// Check auth status

const { data: { user } } = await supabase.auth.getUser();

// Protected route component

function ProtectedRoute({ children }: { children: ReactNode }) {

const { data: { user }, isLoading } = useUser();

if (isLoading) return <LoadingSpinner />;

if (!user) return <Navigate to="/login" />;

return <>{children}</>;

}

**4. API Key Security**

// ✅ GOOD: Use environment variables

const supabase = createClient(

import.meta.env.VITE\_SUPABASE\_URL,

import.meta.env.VITE\_SUPABASE\_ANON\_KEY // Anon key, not service key!

);

// ❌ BAD: Never commit keys to git

const supabase = createClient(

'https://xxx.supabase.co',

'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...' // Hardcoded!

);

// The anon key is safe to expose (it's in your frontend bundle)

// RLS policies protect your data, not the anon key

**MONITORING & OBSERVABILITY**

**Error Tracking - Sentry**

// lib/sentry.ts

import \* as Sentry from '@sentry/react';

if (import.meta.env.PROD) {

Sentry.init({

dsn: import.meta.env.VITE\_SENTRY\_DSN,

integrations: [

new Sentry.BrowserTracing(),

new Sentry.Replay(),

],

tracesSampleRate: 0.1, // 10% of transactions

replaysSessionSampleRate: 0.1,

replaysOnErrorSampleRate: 1.0, // 100% of errors

});

}

// Wrap app

export function App() {

return (

<Sentry.ErrorBoundary fallback={ErrorFallback}>

<Router />

</Sentry.ErrorBoundary>

);

}

**Analytics - Simple Event Tracking**

// lib/analytics.ts

export const analytics = {

track(event: string, properties?: Record<string, any>) {

// Development: Just log

if (import.meta.env.DEV) {

console.log('[Analytics]', event, properties);

return;