Table of Contents

Development Setup

```
# Install dependencies
npm install

# Run development server
npm run dev

# Run linter
npm run lint

# Build for production
npm run build
```

Code Style & Syntax

TypeScript

- **Strict mode enabled** - All code must pass TypeScript strict checks
- **Explicit return types** - Always specify return types for functions (except inline callbacks)
- **No `any` types** - Use proper typing or `unknown` when type is truly unknown
- **Target ES2017** - Use modern JavaScript features compatible with ES2017

```
// ✅ Good
export async function getOrgs(): Promise<ActionResult<Organization[]>> {
  // ...
}

// ❌ Bad
export async function getOrgs() {
  // ...
}
```

Imports

- **Always use path aliases** with `@/` prefix
- **Group imports** in this order:
    1. React/Next.js imports
    2. Third-party libraries
    3. Local components (with path aliases)
    4. Types
    5. Utilities

```
// ✅ Good
import { useState } from "react";
import { useRouter } from "next/navigation";
import { useQuery } from "@tanstack/react-query";

import { Button } from "@/components/ui/button";
import { OrganizationList } from "@/components/dashboard/organizations/li
import { Organization } from "@/app/types/supabase";
import { cn } from "@/lib/utils";

// ❌ Bad - No path alias
import { Button } from "../../../components/ui/button";
```

Formatting

- **Indentation**: 2 spaces (no tabs)
- **Line length**: Prefer 80-100 characters (not enforced strictly)
- **Semicolons**: Required
- **Quotes**: Double quotes for strings (enforced by Prettier if configured)
- **Trailing commas**: Required in multi-line objects/arrays

TypeScript Conventions

Type Definitions

- **Location**: All shared types go in `src/app/types/`
- **Naming**: Use PascalCase for types and interfaces
- **Prefer `type` over `interface`** for consistency (unless extending is needed)

```
// ✅ Good - src/app/types/supabase.ts
export type Organization = {
  id: string;
  name: string;
  created_at: string;
};

export type OrganizationWithRole = Organization & {
  role: string;
  joined_at: string;
};

// ❌ Bad - Using interface unnecessarily
export interface Organization {
  id: string;
  name: string;
}
```

ActionResult Pattern

All server actions MUST return `ActionResult<T>` for consistent error handling:

```
import { ActionResult } from "@/app/types/action";

export async function createOrg(
  formData: FormData
): Promise<ActionResult<Organization>> {
  // Validation
  if (!name) {
    return { success: false, error: "Name is required" };
  }
```

```
  // Database operation
  const { data, error } = await supabase.from("organizations").insert({ r

  if (error) {
    return { success: false, error: error.message };
  }

  return { success: true, data };
}
```

Client-side usage:

```
const result = await createOrg(formData);

if (!result.success) {
  setError(result.error);
  return;
}

// TypeScript knows result.data is available here
console.log(result.data.id);
```

Next.js 15 Dynamic Routes

Params are Promises in Next.js 15:

```
// ✅ Good
export default async function Page({
  params,
}: {
  params: Promise<{ id: string }>;
}) {
  const { id } = await params;
  // Use id...
}

// ❌ Bad - Treating params as synchronous
export default async function Page({
  params,
}: {
  params: { id: string };
}) {
```

```
  const { id } = params; // This will fail
}
```

Component Patterns

Server vs Client Components

Default to Server Components:

- Server components are the default in Next.js App Router
- Only use `"use client"` when needed for:
  - Event handlers (onClick, onChange, etc.)
  - React hooks (useState, useEffect, etc.)
  - Browser APIs
  - TanStack Query hooks

```
// ✅ Good - Server Component (no directive)
export default async function Page() {
  const supabase = await createClient();
  const { data } = await supabase.from("organizations").select();

  return <OrganizationList organizations={data} />;
}

// ✅ Good - Client Component (explicit directive)
"use client";

export function OrganizationForm() {
  const [name, setName] = useState("");

  return <input value={name} onChange={(e) => setName(e.target.value)} />
}
```

Component Naming

- **File names**: kebab-case (e.g., `new-organization-form.tsx`)
- **Component names**: PascalCase (e.g., `NewOrganizationForm`)
- **File exports**: Named exports preferred for components

```
// ✅ Good - new-organization-form.tsx
export function NewOrganizationForm() {
  // ...
}

// ❌ Bad - Using default export
export default function NewOrganizationForm() {
  // ...
}

// ❌ Exception - Page components use default export (Next.js requirement
export default async function Page() {
  // ...
}
```

Component Structure

Order component elements consistently:

1. Imports
2. Type definitions (props, etc.)
3. Component function
4. Return statement
5. Sub-components (if any)

```
"use client";

import { Button } from "@/components/ui/button";
import { useState } from "react";

type FormProps = {
  orgId: string;
  onSubmit: () => void;
};

export function MyForm({ orgId, onSubmit }: FormProps) {
  const [name, setName] = useState("");
  const [error, setError] = useState<string | null>(null);

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();
    // Handle submit
```

```
  };

  return (
    <form onSubmit={handleSubmit}>
      {/* Form content */}
    </form>
  );
}
```

Props Typing

- **Always type props explicitly**
- **Use `type` for props**, not inline types
- **Destructure props** in function signature

```
// ✅ Good
type ButtonProps = {
  variant: "primary" | "secondary";
  children: React.ReactNode;
  onClick?: () => void;
};

export function Button({ variant, children, onClick }: ButtonProps) {
  // ...
}

// ❌ Bad - Inline prop types
export function Button({ variant, children }: { variant: string; childrer
  // ...
}
```

State Management

TanStack Query (React Query)

Use for all server data:

```
"use client";

import { useQuery, useMutation, useQueryClient } from "@tanstack/react-qu
```

```
// ✅ Good - Query for fetching
export function OrganizationList() {
  const { data, isLoading, error } = useQuery({
    queryKey: ["organizations"],
    queryFn: async () => {
      const result = await getOrgs();
      if (!result.success) throw new Error(result.error);
      return result.data;
    },
  });

  // ...
}

// ✅ Good - Mutation for updates with cache invalidation
export function NewOrganizationForm() {
  const queryClient = useQueryClient();

  const mutation = useMutation({
    mutationFn: (formData: FormData) => createOrg(formData),
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey: ["organizations"] });
    },
  });

  // ...
}
```

Query Key Conventions:

- Organizations: `["organizations"]`
- Repositories: `["repositories", organizationId]`
- Conversations: `["conversations", repoId]`
- Messages: `["messages", conversationId]`

Always invalidate queries after mutations:

```
// ✅ Good
const result = await createOrg(formData);
if (result.success) {
  queryClient.invalidateQueries({ queryKey: ["organizations"] });
}
```

```
// ❌ Bad - No cache invalidation
const result = await createOrg(formData);
router.push("/dashboard/organizations"); // Stale data!
```

Zustand

Use for UI-only state:

- Sidebar collapsed/expanded
- Modal open/closed
- Current tab/view
- Temporary selections

```
// src/lib/stores/ui-store.ts
import { create } from "zustand";

type UIStore = {
  sidebarCollapsed: boolean;
  toggleSidebar: () => void;
};

export const useUIStore = create<UIStore>((set) => ({
  sidebarCollapsed: false,
  toggleSidebar: () => set((state) => ({ sidebarCollapsed: !state.sidebar
}));
```

Store Naming:

- File: `[name]-store.ts` (e.g., `ui-store.ts`)
- Hook: `use[Name]Store` (e.g., `useUIStore`)
- Location: `src/lib/stores/`

Server Actions

Structure

All server actions live in `src/lib/services/` :

```
src/lib/services/
├── orgService.ts        # Organization CRUD
```

```
├── repoService.ts      # Repository CRUD
└── userService.ts      # User management (future)
```

Conventions

1. **Always use `"use server"` directive** at the top of the file
2. **Always return `ActionResult<T>`**
3. **Always check authentication** first
4. **Always validate inputs**
5. **Use descriptive function names** (get, create, update, delete prefix)

```
"use server";

import { createClient } from "@/utils/supabase/server";
import { ActionResult } from "@/app/types/action";
import { Organization } from "@/app/types/supabase";

export async function createOrg(
  formData: FormData
): Promise<ActionResult<Organization>> {
  // 1. Get authenticated client
  const supabase = await createClient();

  // 2. Check authentication
  const { data: { user } } = await supabase.auth.getUser();
  if (!user) {
    return { success: false, error: "User not authenticated" };
  }

  // 3. Validate inputs
  const name = formData.get("name") as string;
  if (!name?.trim()) {
    return { success: false, error: "Name is required" };
  }

  // 4. Perform database operation
  const { data, error } = await supabase
    .from("organizations")
    .insert({ name: name.trim() })
    .select()
    .single();

  // 5. Handle errors
```

```
  if (error) {
    return { success: false, error: error.message };
  }

  // 6. Return success
  return { success: true, data };
}
```

Database & Supabase

Client Usage

Three client types:

1. **Browser client** (`@/utils/supabase/client`) - For client components
2. **Server client** (`@/utils/supabase/server`) - For server components/actions
3. **Middleware client** - Only used in middleware (do not import elsewhere)

```
// ✅ Good - Client component
"use client";
import { createClient } from "@/utils/supabase/client";

export function Component() {
  const supabase = createClient();
  // ...
}

// ✅ Good - Server component
import { createClient } from "@/utils/supabase/server";

export default async function Page() {
  const supabase = await createClient();
  // ...
}

// ✅ Good - Server action
"use server";
import { createClient } from "@/utils/supabase/server";

export async function myAction() {
  const supabase = await createClient();
```

```
  // ...
}
```

Query Patterns

Always use explicit select:

```
// ✅ Good - Explicit columns
const { data } = await supabase
  .from("organizations")
  .select("id, name, created_at")
  .eq("id", orgId)
  .single();

// ❌ Bad - Select all (implicit)
const { data } = await supabase
  .from("organizations")
  .select()
  .eq("id", orgId)
  .single();
```

Join patterns for relations:

```
// ✅ Good - Joining user_organizations with organizations
const { data } = await supabase
  .from("user_organizations")
  .select("role, joined_at, organizations(id, name, created_at)")
  .eq("user_id", user.id);
```

Naming Conventions

Files

- **Components**: `kebab-case.tsx` (e.g., `new-organization-form.tsx`)
- **Services**: `camelCase.ts` (e.g., `orgService.ts`)
- **Types**: `camelCase.ts` (e.g., `supabase.ts`)
- **Pages**: `page.tsx` (Next.js convention)
- **Layouts**: `layout.tsx` (Next.js convention)

Functions

- **Components**: `PascalCase` (e.g., `OrganizationList`)
- **Hooks**: `use` prefix + `camelCase` (e.g., `useSidebar`)
- **Server actions**: `camelCase` with verb prefix (e.g., `createOrg`, `getRepos`)
- **Event handlers**: `handle` prefix + `PascalCase` (e.g., `handleSubmit`)
- **Utilities**: `camelCase` (e.g., `cn`, `formatDate`)

Variables

- **Constants**: `UPPER_SNAKE_CASE` (e.g., `MAX_FILE_SIZE`)
- **Regular variables**: `camelCase` (e.g., `userName`, `isLoading`)
- **Boolean variables**: Use `is`, `has`, `should` prefix (e.g., `isLoading`, `hasError`)

```
// ✅ Good
const MAX_FILE_SIZE = 100 * 1024 * 1024; // 100MB
const isLoading = false;
const hasError = error !== null;

// ❌ Bad
const max_file_size = 100 * 1024 * 1024;
const loading = false;
const error_state = error !== null;
```

File Organization

Component Files

```
// new-organization-form.tsx

// 1. Directives (if needed)
"use client";

// 2. Imports
import { Button } from "@/components/ui/button";
import { useState } from "react";

// 3. Types
type FormProps = {
```

```
  onSuccess?: () => void;
};

// 4. Component
export function NewOrganizationForm({ onSuccess }: FormProps) {
  // 4a. Hooks
  const [name, setName] = useState("");
  const router = useRouter();

  // 4b. Handlers
  const handleSubmit = async (e: React.FormEvent) => {
    // ...
  };

  // 4c. Render
  return (
    <form onSubmit={handleSubmit}>
      {/* JSX */}
    </form>
  );
}

// 5. Sub-components (if any, prefer separate files)
```

Service Files

```
// orgService.ts

// 1. Directive
"use server";

// 2. Imports
import { createClient } from "@/utils/supabase/server";
import { ActionResult } from "@/app/types/action";
import { Organization } from "@/app/types/supabase";

// 3. Functions grouped by resource
// GET operations
export async function getOrgs(): Promise<ActionResult<Organization[]>> {
export async function getOrgById(id: string): Promise<ActionResult<Organi

// CREATE operations
```

```
export async function createOrg(formData: FormData): Promise<ActionResult

// UPDATE operations
export async function updateOrg(id: string, data: Partial<Organization>):

// DELETE operations
export async function deleteOrg(id: string): Promise<ActionResult<void>>
```

Testing

Test Files (Future)

When tests are added:

- **Location**: Co-located with source files or in `__tests__` directory
- **Naming**: `[filename].test.ts` or `[filename].spec.ts`
- **Framework**: TBD (likely Jest + React Testing Library)

```
src/lib/services/
├── orgService.ts
└── orgService.test.ts
```

Git Workflow

Branch Naming

- **Feature**: `feature/short-description`
- **Bug fix**: `fix/short-description`
- **Chore**: `chore/short-description`

Commit Messages

Follow conventional commits format:

```
<type>(<scope>): <subject>

<body>

<footer>
```

Types:

- `feat` : New feature
- `fix` : Bug fix
- `refactor` : Code refactoring
- `docs` : Documentation changes
- `style` : Formatting changes
- `test` : Adding tests
- `chore` : Build process or tooling changes

Examples:

```
feat(orgs): add member management to organization page

- Add MembersTable component
- Implement getOrgMembers server action
- Add remove member functionality

Closes #123


fix(auth): redirect authenticated users from login page

Previously authenticated users could access /login and see the
login form. Now they are redirected to /dashboard/organizations.
```

Pull Requests

1. **Create feature branch** from `main` (or team-specific branch)
2. **Make changes** following conventions in this guide
3. **Run linter**: `npm run lint`
4. **Test locally**: `npm run dev` and verify changes
5. **Create PR** with descriptive title and summary
6. **Link issues** if applicable (Closes #123)

Questions?

If you have questions about these conventions or need clarification:

1. Check CLAUDE.md for architectural guidance
2. Look at existing code for examples

3. Ask in team discussions