

Frontend Developer Onboarding Guide

Version 3.1 - Anokha Frontend Team

Welcome & Getting Started

Welcome to Anokha Frontend Development!

This onboarding guide will help you understand our codebase, development philosophy, and best practices. Follow this guide step by step to get up to speed quickly.

Prerequisites

- Node.js (Latest LTS version)
- Git basics
- React & TypeScript fundamentals
- Basic understanding of Next.js

Quick Setup Checklist

- Clone the repository
- Run init scripts
- Run development server (`pnpm run dev`)
- Explore the `/examples/` folder for code patterns

Frontend Tech Stack

Core Technologies

- **Framework:** Next.js (App Router)
- **Language:** TypeScript
- **Styling:** Tailwind CSS + CSS Modules
- **State Management:**
 - Zustand (App state - auth, UI)
 - TanStack Query (Server state)
- **Forms:** React Hook Form, Zod
- **HTTP Client:** Axios (centralized in lib/api.ts)
- **Animations:** GSAP or Barba.js for page transitions
- **Image Optimization:** Next/Image (WebP format only)

Development Tools

- **Linting:** ESLint + Prettier
- **Bundle Analysis:** Webpack Bundle Analyzer
- **Performance:** Lighthouse CI

Frontend Guidelines & Ruleset

Team Philosophy

- **Separate logic and UI:**
 - Smart components handle logic, state, data fetching → features/
 - Dumb components are pure UI → components/
- Reuse logic via hooks when needed — shared flows, side effects
- All API calls must go through central API wrapper
- Use TanStack Query for server data
- Use Zustand for app state (auth, UI)
- Toast feedback for all user actions
- Clear, maintainable code — avoid clever over-complicated patterns
- Mobile-first, accessible frontend
- Optimize for performance — but keep it simple

Project Structure and Naming

- Use kebab-case for folders: `components/`, `features/`, `hooks/`, `app/`, `lib/`, `stores/`, `public/`, `styles/`
- Use PascalCase for React component files: `EventCard.tsx`, `Navbar.tsx`
- Use camelCase for function names: `handleSubmit`, `fetchEvents`
- Use kebab-case for images and assets: `anokha-logo.webp`, `techfair-banner.webp`

Component folder example:

```
/components/EventCard/EventCard.tsx
/components/EventCard/EventCard.module.css (if needed)
/components/EventCard/index.ts (optional re-export)
```

- `lib/constants.ts` → for enums, static config, API routes, dropdown options, etc.

CSS / Styling Rules

- Use CSS variables or Tailwind tokens for spacing and sizing. Do not hardcode px.
 - Example: `p-4` is allowed.
 - `style={{ padding: '23px' }}` is not allowed.
- Use rem or em where Tailwind is not used.
- Colors must use CSS variables. Avoid hex codes inside JSX.
- Maintain consistent spacing and rhythm.

- `globals.css` defines tokens for colors, spacing, fonts, shadows, etc.
- Use Shadcn components for forms or buttons to maintain consistency

Image Assets

- All images must be WebP format.
- Use Next/Image for all images in components and pages.
- Lazy-load images by default.
- No image larger than 500kb. Compress images.
- SVGs: inline if simple. Otherwise, serve from /public.

Components

- Components should be atomic and reusable.
- If a component is over 250 lines, split it.
- One component = one responsibility.
- Avoid deep prop drilling. Use Zustand for shared state if needed.
- No inline CSS. Use Tailwind or module CSS.
- All API-driven components and pages must display loading skeletons.
- Loading state should match layout to avoid layout shift.
- Custom 404 page is required.

SSR / CSR Usage

- Default to SSR unless the page needs client-side state or effects.
- Avoid adding `use client` at the top of every file unless required.
- Example:
 - Home page → SSR
 - Profile and event registration → CSR
- Do not force CSR unnecessarily.

Animations

- Page transitions: GSAP or Barba.js. No CSS-only full-page transitions.
- No infinite loops unless subtle (spinner is fine).
- Animations must degrade gracefully on slow devices.
- Do not block interaction.
- Pre-loader only for the Home page.

Performance

- Max bundle size target: 300–400kb per page.

- Home page LCP target: under 2.5s on 4G.
- Minimize unused JS.
- Use bundle analyzer on each major release.
- Lazy-load all third-party libraries unless on critical path.

Anti-Patterns

- No hardcoded API URLs.
- No inline styles with px.
- No uncommented console.log in PRs.
- Do not duplicate component logic. Reuse.
- Do not copy large parts of previous year's code without refactoring.
- Do not add heavy animation libraries without review.
- Do not use JPG or PNG — convert to WebP.
- No animations that interfere with accessibility.
- Always build mobile-first.

Accessibility

- All interactive elements must be keyboard accessible.
- Use aria-label, aria-describedby as needed.
- Maintain WCAG 2.1 AA contrast ratios.
- Test with keyboard navigation.
- Respect reduced motion preferences.

Documentation

- All reusable components must have a simple README comment at the top.
- If a component has more than 3 props, document the props.

API Structure & Layering Guide

Check the `/examples/` folder for sample code with detailed comments.

Folder Responsibilities

- `/lib/api.ts` → Axios instance, token handling, toast notifications, helpers
- `/services/` → Functions for API calls (defines URLs, GET/POST logic)
- `/hooks/` → React hooks that manage the state of API calls
- `/components/` → UI components — use hooks to get data and render UI

Why Use a Service Layer?

- **Separation of concerns:** API URLs and logic are not written in hooks or components.
- **Easy backend changes:** If the backend changes, only the service file needs updating.
- **Hooks focus on state:** (loading, error, cache, etc.)
- **Components focus on UI:** No API logic in UI code.
- **Modular & maintainable:** Code is easier to read, test, and update.

Flow of Logic

Component → Hook → Service → API → Backend

- **Component:** Renders UI, uses hooks for data.
- **Hook:** Manages API call state, uses service functions.
- **Service:** Defines API endpoints and methods, uses shared API helpers.
- **API:** Handles low-level HTTP logic (e.g., Axios instance, tokens).

State Management

- **TanStack Query:** For server state management
 - `useQuery()` → GET API data
 - `useMutation()` → POST / PUT
- **Zustand:** For app state (auth, UI)
- **React Hook Form:** For all form handling

Toast Feedback

- Every user action should show a toast (success or error).
- Toasts should be part of API flow/hooks — not scattered in JSX.
- Wording must be consistent.

Using AI for Development

Purpose

When using AI tools (ChatGPT, Cursor, Copilot Chat), you must first give it project context — otherwise, the AI will generate incorrect or inconsistent code.

How to use:

- Before generating any code with AI, copy and paste the prompt below as the first message.
- Do this once per chat.
- After this, you can ask the AI to generate services, hooks, components, stores, and forms, and it will follow the Anokha 2025 architecture.

AI Setup Prompt (for Anokha 2025):

You are generating code for Anokha 2025 Frontend project.

Tech Stack:

- Next.js (App Router), React 19, TypeScript
- TanStack Query v5
- Zustand
- Axios wrapper (with token injection, skipAuth, interceptors, toast)
- React Hook Form + Zod
- Tailwind CSS (with CSS vars)
- Folder structure: /src/lib, /components, /features, /hooks, /services, /stores, /types

Architecture:

- API Layering: Component → Hook → Service → API wrapper → Backend
- All Services use /lib/api.ts wrapper:
 - apiGet<T>(url, options)
 - apiPost<T>(url, payload, options)

Service Example:

```
export const ExampleService = {
  getPublicData: <T>(): Promise<T> => apiGet('/example/public', {
    skipAuth: true }),
  getProtectedData: <T>(): Promise<T> =>
    apiGet('/example/protected'),
  postProtectedAction: <T>(payload: unknown): Promise<T> =>
    apiPost('/example/action', payload),
```

```
};
```

Hooks Example:

```
export function useExampleData() {
  return useQuery<ExampleItem[]>({
    queryKey: ['examplePublic'],
    queryFn: ExampleService.getPublicData,
    staleTime: 5 * 60 * 1000,
  });
}

export function useExampleAction() {
  const queryClient = useQueryClient();

  return useMutation({
    mutationFn: (payload: ExampleActionPayload) =>
      ExampleService.postProtectedAction(payload),
    onSuccess: () => {
      toast.success('Action successful!');
      queryClient.invalidateQueries({ queryKey: ['exampleProtected']
    });
  },
  onError: () => {
    toast.error('Action failed.');
```

Components:

- Dumb: UI only, takes typed props, no API call
- Smart: Uses hook, handles loading/error/empty, maps data to dumb component
- Skeletons:

For every dumb component, create a matching skeleton component (e.g., EventCardSkeleton for EventCard) using SkeletonBlock for each visual part.

In smart components, use the skeleton component(s) during loading states.

Example:

```
// In a smart component
if (isLoading) {
  return (
    <div>
      {Array.from({ length: 3 }).map((_, i) => <EventCardSkeleton
key={i} />)}
    </div>
  );
}
```

```

    </div>
  );
}

```

Zustand Store (non-persistent):

```

export const useExampleUiStore = create<ExampleUiStoreState>((set) =>
({
  isExampleModalOpen: false,
  openExampleModal: () => set({ isExampleModalOpen: true }),
  closeExampleModal: () => set({ isExampleModalOpen: false }),
})));

```

Zustand Store (persistent):

```

export const useExampleAuthStore = create<ExampleAuthStoreState>() (
  persist((set) => ({
    token: null,
    setToken: (token) => set({ token }),
  })), { name: 'example-auth-store' })
);

```

Forms (React Hook Form + Zod):

```

const exampleFormSchema = z.object({
  title: z.string().min(1, 'Title is required'),
  description: z.string().min(5, 'Description must be at least 5
characters'),
});

```

```

type ExampleFormValues = z.infer<typeof exampleFormSchema>;

```

```

const { register, handleSubmit, formState: { errors } } =
useForm<ExampleFormValues>({
  resolver: zodResolver(exampleFormSchema),
});

```

Rules:

- Do not generate API calls inside hooks or components
- Use Service Layer for all API calls
- Hooks must use TanStack Query object form
- Forms must use RHF + Zod
- State must use Zustand (no Context, no Redux)
- Dumb/Smart component separation must be followed
- No `any` type
- Match project `/examples/` style
- Include proper imports and type definitions
- Use toast for success/error messages in mutations

- Include query invalidation in mutation success callbacks

Please remember this context for this chat.

Git Branching Rules

Our team follows a strict branching strategy to maintain code quality.

Branch Structure

- **main** → Protected, production-ready code. No direct commits.
- **dev** → Default working branch. All features branch from here.
- **feature/bugfix** → Individual work branches.

Branch Naming

- **Features:** `feat/<feature-name>` (e.g., `feat/events-page`)
- **Bug fixes:** `fix/<bug-description>` (e.g., `fix/navbar-overlap`)
- **Hotfixes:** `hotfix/<urgent-fix>` (critical fixes only)

Commit Messages

Follow conventional commits:

- `feat: add events list component`
- `fix: correct loading spinner in auth flow`
- `chore: update eslint config`
- `refactor: optimize API error handling`
- `docs: update onboarding guide`

PR Requirements

All PRs must:

- Target `dev` branch (except hotfixes)
- Pass lint and pre-commit hooks
- Follow project architecture
- Include clear commit messages
- Get approved before merge

Workflow

feat/xxx → PR → dev → PR → main