NextFramework

A Modern React/Next.js Enterprise Framework

A comprehensive guide for creating structured, maintainable, and scalable frontend applications

Organization Name

May 15, 2025

Contents

Chapter 1

Introduction

1.1 Purpose of This Guide

Welcome to our organization's comprehensive React/Next.js development framework! This guide aims to standardize how we build frontend applications using modern JavaScript technologies while maintaining the structure and organization that Angular traditionally provides.

Whether you're transitioning from Angular or starting fresh with React, this guide will help you write clean, maintainable, and scalable code that follows consistent patterns across our organization.

1.2 Why Next.js and React?

React's component-based architecture offers flexibility and performance advantages, while Next.js provides an opinionated framework that adds server-side rendering, routing, and build optimization. Together, they create a powerful foundation for building modern web applications.

1.3 Philosophy

Our approach combines React's flexibility with Angular's structured methodology. We believe in:

- **Consistency over creativity** - Follow established patterns for predictable codebases
- **Organization over chaos** - Keep code structured and navigable
- **Types over guesswork** - Use TypeScript throughout for type safety
- **Components as building blocks** - Create focused, reusable components
- **Standards over preferences** - Follow team conventions, not personal preferences

Remember

This guide isn't about restricting creativity, but about establishing conventions that allow us to collaborate effectively on large-scale applications.

Chapter 2

Core Tech Stack

2.1 Foundation

2.1.1 Next.js

Next.js serves as our primary application framework, providing:

- Server-side rendering for improved performance and SEO
- App Router for type-safe, file-based routing
- API Routes for backend functionality
- Optimized build system with automatic code splitting
- Image optimization through next/image

Tip

Use the App Router for new projects. The Pages Router is still supported but offers fewer modern features.

2.1.2 React

We use React 18+ for our component architecture, taking advantage of:

- Concurrent rendering features
- React Server Components (with Next.js App Router)
- Hooks for state management and side effects

2.1.3 TypeScript

TypeScript is non-negotiable in our stack. Configure it with strict mode enabled:

```
{
  "compilerOptions": {
    "target": "es5",
    "lib": ["dom", "dom.iterable", "esnext"],
    "allowJs": true,
    "skipLibCheck": true,
    "strict": true,
    "noEmit": true,
```

```
    "esModuleInterop": true,
    "module": "esnext",
    "moduleResolution": "bundler",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "jsx": "preserve",
    "incremental": true,
    "plugins": [
      {
        "name": "next"
      }
    ],
    "paths": {
      "@/*": ["./src/*"]
    }
  },
  "include": ["next-env.d.ts", "**/*.ts", "**/*.tsx", ".next/types/**/*.ts"],
  "exclude": ["node_modules"]
}
```

Listing 2.1: tsconfig.json

2.2 State Management

2.2.1 React Context API

Use React Context for:

- Component-level state sharing
- Theme providers
- Feature-specific state when Zustand would be overkill

2.2.2 Zustand

Zustand is our recommended global state management solution, offering:

- Minimal boilerplate compared to Redux
- Easy integration with TypeScript
- No need for context providers or reducers
- Excellent performance with selective re-renders

```
// src/stores/userStore.ts
import { create } from 'zustand';
import type { User } from '@/types';

interface UserState {
  user: User | null;
  isLoading: boolean;
  error: Error | null;
  fetchUser: (id: string) => Promise<void>;
  updateUser: (userData: Partial<User>) => Promise<void>;
}

export const useUserStore = create<UserState>((set) => ({
  user: null,
```

```
  isLoading: false,
  error: null,
  fetchUser: async (id) => {
    set({ isLoading: true });
    try {
      const response = await fetch(`/api/users/${id}`);
      const user = await response.json();
      set({ user, isLoading: false });
    } catch (error) {
      set({ error: error as Error, isLoading: false });
    }
  },
  updateUser: async (userData) => {
    set({ isLoading: true });
    try {
      // Implementation
      set(state => ({
        user: state.user ? { ...state.user, ...userData } : null,
        isLoading: false
      }));
    } catch (error) {
      set({ error: error as Error, isLoading: false });
    }
  }
}));
```

Listing 2.2: Zustand Store Example

When to Use What

- **Local component state**: React's useState
- **Shared component tree state**: React Context
- **Application-wide state**: Zustand

2.3 Data Fetching

2.3.1 SWR / React Query

These libraries offer similar capabilities - we recommend SWR for its simplicity and React Query for more complex scenarios.

Key features:

- Cache management
- Automatic revalidation
- Loading and error states
- Pagination and infinite scrolling
- Optimistic updates

```
// src/hooks/useUser.ts
import useSWR from 'swr';
import { apiClient } from '@/lib/api';
import type { User } from '@/types';

export function useUser(id: string) {
```

```
  const { data, error, isLoading, mutate } = useSWR<User>(
    id ? `/users/${id}` : null,
    () => apiClient.get(`/users/${id}`)
  );

  return {
    user: data,
    isLoading,
    error,
    mutate
  };
}
```

Listing 2.3: SWR Example

2.3.2 tRPC

For TypeScript projects that control both frontend and backend, tRPC provides end-to-end type safety:

- Share types between client and server without code generation
- Auto-completion for API endpoints
- Runtime validation with Zod integration

2.4 Form Management

2.4.1 React Hook Form

Our preferred library for form handling thanks to:

- Minimal re-renders for better performance
- Uncontrolled components by default
- Easy validation using Zod schemas
- TypeScript support for form values

```
import { useForm } from "react-hook-form";
import { zodResolver } from "@hookform/resolvers/zod";
import { z } from "zod";

const userSchema = z.object({
  name: z.string().min(2, "Name must be at least 2 characters"),
  email: z.string().email("Invalid email address"),
  age: z.number().min(18, "Must be at least 18 years old")
});

type UserFormValues = z.infer<typeof userSchema>;

export function UserForm() {
  const { register, handleSubmit, formState: { errors } } = useForm<
    UserFormValues>({
    resolver: zodResolver(userSchema)
  });

  const onSubmit = (data: UserFormValues) => {
    console.log(data);
```

```
  };

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <div>
        <label>Name</label>
        <input {...register("name")} />
        {errors.name && <p>{errors.name.message}</p>}
      </div>

      <div>
        <label>Email</label>
        <input {...register("email")} />
        {errors.email && <p>{errors.email.message}</p>}
      </div>

      <div>
        <label>Age</label>
        <input type="number" {...register("age", { valueAsNumber: true })} />
        {errors.age && <p>{errors.age.message}</p>}
      </div>

      <button type="submit">Submit</button>
    </form>
  );
}
```

Listing 2.4: React Hook Form with Zod

2.5 UI Components

2.5.1 shadcn/UI with Tailwind CSS

We use shadcn/UI for our component library because:

- Components are copied into your project rather than installed as dependencies
- Built on Radix UI primitives for accessibility
- Customizable with Tailwind CSS
- TypeScript support out of the box

2.5.2 Tailwind CSS

Tailwind is our preferred styling solution:

- Utility-first approach that scales well
- No context switching between files
- Easy theming with design tokens
- Built-in responsive design utilities

Style Guidelines

Use Tailwind's class composition for repeated patterns:

```
// Button.tsx
const baseStyles = "px-4 py-2 rounded focus:outline-none focus:ring-2";
const variants = {
  primary: "bg-blue-600 hover:bg-blue-700 text-white",
  secondary: "bg-gray-200 hover:bg-gray-300 text-gray-800"
};

export function Button({
  variant = "primary",
  className,
  ...props
}: ButtonProps) {
  return (
    <button
      className={`${baseStyles} ${variants[variant]} ${className}`}
      {...props}
    />
  );
}
```

2.6 Testing Framework

2.6.1 Vitest

We prefer Vitest for unit testing due to:

- Fast execution with native ESM support
- Compatible with Jest's API
- Built-in TypeScript support
- Watch mode with instant feedback

2.6.2 React Testing Library

For component testing, React Testing Library ensures we test from the user's perspective:

- Focus on user interactions, not implementation details
- Encourages accessible markup
- Works with any component regardless of its internal structure

```
import { render, screen, fireEvent } from '@testing-library/react';
import { UserForm } from './UserForm';

describe('UserForm', () => {
  it('displays validation errors when form is submitted with empty fields',
    async () => {
    render(<UserForm />);

    // Submit the empty form
    fireEvent.click(screen.getByRole('button', { name: /submit/i }));

```

```
    // Check for validation messages
    expect(await screen.findByText(/name must be at least/i)).toBeInTheDocument
    ();
    expect(await screen.findByText(/invalid email address/i)).toBeInTheDocument
    ();
  });

  it('calls onSubmit with form data when valid', async () => {
    const onSubmitMock = jest.fn();
    render(<UserForm onSubmit={onSubmitMock} />);

    // Fill out the form
    fireEvent.change(screen.getByLabelText(/name/i), { target: { value: 'John
    Doe' } });
    fireEvent.change(screen.getByLabelText(/email/i), { target: { value: '
    john@example.com' } });
    fireEvent.change(screen.getByLabelText(/age/i), { target: { value: '25' }
    });

    // Submit the form
    fireEvent.click(screen.getByRole('button', { name: /submit/i }));

    // Verify onSubmit was called with the expected data
    expect(onSubmitMock).toHaveBeenCalledWith({
      name: 'John Doe',
      email: 'john@example.com',
      age: 25
    });
  });
});
```

Listing 2.5: Component Test Example

2.6.3 Playwright

For end-to-end testing, we use Playwright:

- Cross-browser testing (Chromium, Firefox, WebKit)
- Powerful API for modern web testing
- Visual testing capabilities
- GitHub Actions integration

Chapter 3

Project Structure

3.1 Base Directory Structure

Our recommended project structure follows a feature-first approach while maintaining clear separation of concerns:

```
src/
          app/                    # Next.js app router pages
          components/             # Shared components
                ui/                 # Base UI components
                features/           # Feature-specific components
          hooks/                  # Custom React hooks
          lib/                    # Utility functions and shared code
          services/               # API and external service integrations
          stores/                 # State management stores
          types/                  # TypeScript type definitions
          utils/                  # Utility functions
```

Listing 3.1: Base Directory Structure

3.2 Feature-Based Organization

For complex applications, we organize routes by feature using Next.js route groups:

```
src/
          app/
                (dashboard)/        # Group dashboard-related routes
                      users/
                            page.tsx
                            [id]/
                                page.tsx
                      settings/
                          page.tsx
                (auth)/             # Group authentication-related routes
                      login/
                            page.tsx
                      register/
                          page.tsx
                (public)/           # Group public-facing routes
                    about/
                          page.tsx
                    contact/
                        page.tsx
```

Listing 3.2: App Router Structure

3.3 Component Organization

We organize components in two ways:

1. **Simple components**: Single-file components when they're straightforward
2. **Complex components**: Directory-based approach when they have multiple parts or styles

```
UserProfile/
          UserProfile.tsx           # Main component file
          UserProfileHeader.tsx     # Sub-component
          UserProfileDetails.tsx    # Sub-component
          UserProfile.test.tsx      # Test file
          index.ts                  # Re-export main component
```

Listing 3.3: Complex Component Structure

Barrel Files

Use barrel files (index.ts) to simplify imports:

```
// components/ui/index.ts
export * from './Button';
export * from './Card';
export * from './Input';

// Usage in another file
import { Button, Card, Input } from '@/components/ui';
```

Chapter 4

Component Architecture

4.1 Component Design Principles

4.1.1 Single Responsibility

Each component should have a single responsibility. If a component grows too complex, break it down into smaller, focused components.

4.1.2 Props Interface First

Always define your component's props interface before implementing the component:

```
interface UserCardProps {
  user: User;
  showDetails?: boolean;
  onEdit?: (userId: string) => void;
}

export function UserCard({ user, showDetails = false, onEdit }: UserCardProps)
    {
  // Implementation
}
```

Listing 4.1: Props-First Approach

4.1.3 Component Structure

Follow this order for a consistent component structure:

1. Import statements (grouped by external/internal)
2. Type definitions and interfaces
3. Custom hooks (if only used in this component)
4. Component function
5. Local subcomponents (if small enough to keep in the same file)
6. Export statement

```
// External imports
import { useState, useEffect } from 'react';
import { format } from 'date-fns';

```

```
// Internal imports
import { Avatar } from '@/components/ui';
import { useUserData } from '@/hooks/useUserData';
import type { User } from '@/types';

// Props interface
interface UserProfileProps {
  userId: string;
  showActivity?: boolean;
}

// Component function
export function UserProfile({ userId, showActivity = false }: UserProfileProps) {
  const { user, isLoading, error } = useUserData(userId);
  const [activeTab, setActiveTab] = useState('info');

  if (isLoading) return <Loading />;
  if (error) return <ErrorDisplay message={error.message} />;

  return (
    <div className="user-profile">
      <ProfileHeader user={user} />
      <TabNav activeTab={activeTab} onChange={setActiveTab} />

      {activeTab === 'info' && <UserInfo user={user} />}
      {activeTab === 'posts' && <UserPosts userId={user.id} />}
      {showActivity && activeTab === 'activity' && <UserActivity userId={user.id} />}
    </div>
  );
}

// Local subcomponents
function ProfileHeader({ user }: { user: User }) {
  return (
    <div className="flex items-center gap-4 mb-6">
      <Avatar src={user.avatar} alt={user.name} size="lg" />
      <div>
        <h2 className="text-2xl font-bold">{user.name}</h2>
        <p className="text-gray-500">Member since {format(new Date(user.joinedAt), 'MMMM yyyy')}</p>
      </div>
    </div>
  );
}

function TabNav({ activeTab, onChange }: { activeTab: string; onChange: (tab: string) => void }) {
  // Implementation
}
```

Listing 4.2: Component Structure Example

4.2 Container and Presentational Pattern

Separate data-fetching logic from presentation to improve component reusability:

```
// UserProfileContainer.tsx (Container Component)
import { UserProfile } from './UserProfile';
import { useUserData } from '@/hooks/useUserData';

```

```tsx
interface UserProfileContainerProps {
  userId: string;
  showActivity?: boolean;
}

export function UserProfileContainer({ userId, showActivity }:
    UserProfileContainerProps) {
  const { user, isLoading, error } = useUserData(userId);

  if (isLoading) return <Loading />;
  if (error) return <ErrorDisplay message={error.message} />;

  return <UserProfile user={user} showActivity={showActivity} />;
}

// UserProfile.tsx (Presentational Component)
interface UserProfileProps {
  user: User;
  showActivity?: boolean;
}

export function UserProfile({ user, showActivity }: UserProfileProps) {
  // Pure presentation logic, no data fetching
  return (
    // Render UI based on props
  );
}
```

Listing 4.3: Container/Presentational Pattern

When to Use This Pattern

Use this pattern when:

- A component needs to be reused with different data sources
- You want to separate concerns for better testing
- The component has complex data-fetching logic

Chapter 5

State Management

5.1 Local Component State

Use React's built-in hooks for component-level state:

```
function Counter() {
  // Simple state
  const [count, setCount] = useState(0);

  // Complex state
  const [state, dispatch] = useReducer(
    (state, action) => {
      switch (action.type) {
        case 'increment':
          return { ...state, count: state.count + 1 };
        case 'decrement':
          return { ...state, count: state.count - 1 };
        default:
          return state;
      }
    },
    { count: 0 }
  );

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>

      <p>Reducer Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>Increment</button
    >
    </div>
  );
}
```

Listing 5.1: Local State Management

5.2 Shared Component State

For state shared across multiple components but not application-wide, use React Context:

```
// ThemeContext.tsx
import { createContext, useContext, useState, ReactNode } from 'react';

type Theme = 'light' | 'dark';
```

```

interface ThemeContextType {
  theme: Theme;
  toggleTheme: () => void;
}

const ThemeContext = createContext<ThemeContextType | undefined>(undefined);

export function ThemeProvider({ children }: { children: ReactNode }) {
  const [theme, setTheme] = useState<Theme>('light');

  const toggleTheme = () => {
    setTheme(prevTheme => prevTheme === 'light' ? 'dark' : 'light');
  };

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}

export function useTheme() {
  const context = useContext(ThemeContext);
  if (context === undefined) {
    throw new Error('useTheme must be used within a ThemeProvider');
  }
  return context;
}
```

Listing 5.2: React Context Example

5.3 Global Application State

For application-wide state, use Zustand with a clear store organization:

```
// src/stores/authStore.ts
import { create } from 'zustand';
import { persist } from 'zustand/middleware';
import type { User } from '@/types';

interface AuthState {
  user: User | null;
  isAuthenticated: boolean;
  token: string | null;

  // Actions
  login: (email: string, password: string) => Promise<void>;
  logout: () => void;
  updateProfile: (data: Partial<User>) => Promise<void>;
}

export const useAuthStore = create<AuthState>()(
  persist(
    (set) => ({
      user: null,
      isAuthenticated: false,
      token: null,

      login: async (email, password) => {
        // Implementation
        set({ user: userData, isAuthenticated: true, token: userToken });
```

```
      },

      logout: () => {
        set({ user: null, isAuthenticated: false, token: null });
      },

      updateProfile: async (data) => {
        // Implementation
        set(state => ({
          user: state.user ? { ...state.user, ...data } : null
        }));
      }
    }),
    {
      name: 'auth-storage',
      partialize: (state) => ({ user: state.user, token: state.token }),
    }
  )
);
```

Listing 5.3: Zustand Store Organization

Store Organization Tips

- Create separate stores for different domains (auth, users, products)
- Use middleware like persist for persistent storage
- Include both state and actions in the same store
- Keep stores small and focused

Chapter 6

Data Fetching and API Integration

6.1 API Client

Create a standardized API client to handle requests consistently:

```
// src/lib/api.ts
interface RequestOptions extends RequestInit {
  params?: Record<string, string>;
}

class ApiClient {
  private baseUrl: string;

  constructor(baseUrl: string) {
    this.baseUrl = baseUrl;
  }

  async get<T>(endpoint: string, options: RequestOptions = {}): Promise<T> {
    return this.request<T>(endpoint, { ...options, method: 'GET' });
  }

  async post<T>(endpoint: string, data?: any, options: RequestOptions = {}):
    Promise<T> {
    return this.request<T>(endpoint, {
      ...options,
      method: 'POST',
      body: data ? JSON.stringify(data) : undefined
    });
  }

  async put<T>(endpoint: string, data?: any, options: RequestOptions = {}):
    Promise<T> {
    return this.request<T>(endpoint, {
      ...options,
      method: 'PUT',
      body: data ? JSON.stringify(data) : undefined
    });
  }

  async delete<T>(endpoint: string, options: RequestOptions = {}): Promise<T> {
    return this.request<T>(endpoint, { ...options, method: 'DELETE' });
  }

  private async request<T>(endpoint: string, options: RequestOptions = {}):
    Promise<T> {
    const { params, headers, ...restOptions } = options;

    // Build URL with query parameters
```

```
    const url = new URL(endpoint, this.baseUrl);
    if (params) {
      Object.entries(params).forEach(([key, value]) => {
        url.searchParams.append(key, value);
      });
    }

    // Set default headers
    const defaultHeaders = {
      'Content-Type': 'application/json',
      'Accept': 'application/json',
    };

    // Merge with custom headers
    const mergedHeaders = { ...defaultHeaders, ...headers };

    // Make the request
    const response = await fetch(url.toString(), {
      ...restOptions,
      headers: mergedHeaders,
    });

    // Handle errors
    if (!response.ok) {
      const error = await response.json().catch(() => ({}));
      throw new Error(error.message || `API Error: ${response.status}`);
    }

    // Handle empty responses
    if (response.status === 204) {
      return {} as T;
    }

    // Parse JSON response
    return response.json();
  }
}

export const apiClient = new ApiClient(process.env.NEXT_PUBLIC_API_URL || '/api
    ');
```

Listing 6.1: API Client Implementation

6.2 Data Fetching with SWR

Create reusable hooks for data fetching:

```
// src/hooks/useUsers.ts
import useSWR from 'swr';
import { apiClient } from '@/lib/api';
import type { User, PaginatedResponse } from '@/types';

interface UseUsersOptions {
  page?: number;
  limit?: number;
  search?: string;
}

export function useUsers(options: UseUsersOptions = {}) {
  const { page = 1, limit = 10, search = '' } = options;

  const { data, error, isLoading, mutate } = useSWR<PaginatedResponse<User>>(
```

```
    `/users?page=${page}&limit=${limit}&search=${search}`,
    () => apiClient.get(`/users`, { params: { page: String(page), limit: String
    (limit), search } })
  );

  return {
    users: data?.data || [],
    totalCount: data?.meta?.total || 0,
    isLoading,
    error,
    mutate
  };
}
```

Listing 6.2: SWR Data Fetching Hook

6.3 Two-Way Data Binding

For Angular-like two-way data binding, implement custom form hooks:

```
// src/hooks/useFormField.ts
import { useState, useCallback } from 'react';

export function useFormField<T>(initialValue: T) {
  const [value, setValue] = useState<T>(initialValue);

  const handleChange = useCallback((e: React.ChangeEvent<HTMLInputElement>) =>
    {
    const newValue = e.target.type === 'checkbox'
      ? e.target.checked as unknown as T
      : e.target.value as unknown as T;
    setValue(newValue);
  }, []);

  return {
    value,
    setValue,
    onChange: handleChange,
    reset: useCallback(() => setValue(initialValue), [initialValue])
  };
}

// Usage example
function ProfileForm() {
  const nameField = useFormField('');
  const emailField = useFormField('');
  const agreeField = useFormField(false);

  return (
    <form>
      <div>
        <label>Name</label>
        <input type="text" {...nameField} />
      </div>
      <div>
        <label>Email</label>
        <input type="email" {...emailField} />
      </div>
      <div>
        <label>
          <input type="checkbox" {...agreeField} />
          I agree to terms
```

```
          </label>
        </div>
      </form>
    );
}
```

Listing 6.3: Two-Way Binding Implementation

Chapter 7

Dependency Injection Alternative

7.1 Service Pattern

Create service classes with React Context to mimic Angular's dependency injection:

```
// src/services/AuthService.ts
import { createContext, useContext, ReactNode } from 'react';
import type { User } from '@/types';
import { apiClient } from '@/lib/api';

class AuthService {
  private tokenKey = 'auth_token';

  async login(email: string, password: string): Promise<User> {
    const response = await apiClient.post<{ user: User; token: string }>('/auth
    /login', { email, password });
    localStorage.setItem(this.tokenKey, response.token);
    return response.user;
  }

  async logout(): Promise<void> {
    try {
      await apiClient.post('/auth/logout');
    } finally {
      localStorage.removeItem(this.tokenKey);
    }
  }

  getToken(): string | null {
    return localStorage.getItem(this.tokenKey);
  }

  isAuthenticated(): boolean {
    return this.getToken() !== null;
  }
}

// Create a singleton instance
const authService = new AuthService();

// Create context
const AuthServiceContext = createContext<AuthService | undefined>(undefined);

// Provider component
export function AuthServiceProvider({ children }: { children: ReactNode }) {
  return (
    <AuthServiceContext.Provider value={authService}>
      {children}
```

```
    </AuthServiceContext.Provider>
  );
}

// Hook for accessing the service
export function useAuthService() {
  const context = useContext(AuthServiceContext);
  if (context === undefined) {
    throw new Error('useAuthService must be used within an AuthServiceProvider
    ');
  }
  return context;
}

// Direct export for use outside of React components
export { authService };
```

Listing 7.1: Service Pattern Implementation

7.2 Service Registry

For more complex applications, create a service registry:

```
// src/services/index.ts
import { AuthService } from './AuthService';
import { UserService } from './UserService';
import { PaymentService } from './PaymentService';

interface Services {
  auth: AuthService;
  user: UserService;
  payment: PaymentService;
}

// Create singleton instances
const services: Services = {
  auth: new AuthService(),
  user: new UserService(),
  payment: new PaymentService()
};

// Provider that makes all services available
import { createContext, useContext, ReactNode } from 'react';

const ServiceContext = createContext<Services | undefined>(undefined);

export function ServiceProvider({ children }: { children: ReactNode }) {
  return (
    <ServiceContext.Provider value={services}>
      {children}
    </ServiceContext.Provider>
  );
}

export function useServices() {
  const context = useContext(ServiceContext);
  if (context === undefined) {
    throw new Error('useServices must be used within a ServiceProvider');
  }
  return context;
}

```

```
40 // Direct export for use outside React components
41 export { services };
```

Listing 7.2: Service Registry Pattern

Chapter 8

Documentation Standards

8.1 Component Documentation

Document components using JSDoc comments:

```
/**
 * A user profile card that displays user information and stats.
 *
 * @example
 * ```
 * <UserProfileCard
 *   user={user}
 *   showStats={true}
 *   onEdit={() => console.log('Edit clicked')}
 * />
 * ```
 */
interface UserProfileCardProps {
  /** The user object containing profile data */
  user: User;
  /** Whether to show user statistics */
  showStats?: boolean;
  /** Callback fired when edit button is clicked */
  onEdit?: (userId: string) => void;
  /** CSS class to apply to the component */
  className?: string;
}

export function UserProfileCard({
  user,
  showStats = false,
  onEdit,
  className
}: UserProfileCardProps) {
  // Implementation
}
```

Listing 8.1: Component Documentation Example

8.2 Hook Documentation

Document custom hooks with clear usage examples:

```
/**
 * Hook for managing pagination state and logic.
 *
```

```
 * @param initialPage - The initial page number (default: 1)
 * @param initialLimit - The initial items per page (default: 10)
 * @param totalItems - The total number of items across all pages
 *
 * @returns Pagination state and helper functions
 *
 * @example
 * ```
 * const { page, limit, totalPages, setPage, setLimit } = usePagination({
 *   initialPage: 1,
 *   initialLimit: 25,
 *   totalItems: 100
 * });
 * ```
 */
export function usePagination({
  initialPage = 1,
  initialLimit = 10,
  totalItems
}: {
  initialPage?: number;
  initialLimit?: number;
  totalItems: number;
}) {
  // Implementation
}
```

Listing 8.2: Hook Documentation Example

8.3 Code Comments

Follow these guidelines for code comments:

- Comment complex logic that isn't immediately obvious
- Focus on why, not what (the code shows what, comments explain why)
- Keep comments up-to-date when changing code
- Use TODO comments for temporary solutions that need revisiting

```
// Good comment - explains why
// Use a timeout to prevent excessive API calls when the user types quickly
const debouncedSearch = useDebounce(searchTerm, 300);

// Bad comment - just repeats what the code does
// Set state to true
setIsLoading(true);

// Good TODO comment
// TODO: Replace with proper authentication once the API is ready
const user = mockUserData;
```

Listing 8.3: Effective Code Comments

Chapter 9

Best Practices for Angular Developers

9.1 Transitioning from Angular to React

If you're coming from an Angular background, here are key differences to keep in mind:

- React uses one-way data binding vs. Angular's two-way binding
- Components are functions, not classes (for most modern React code)
- Hooks replace lifecycle methods
- JSX instead of Angular templates
- No built-in dependency injection
- CSS is often component-scoped rather than using global styles or Angular's encapsulation

9.2 Angular Concepts in React

Angular vs. React Equivalents

Angular	**React/Next.js Equivalent**
NgModules	Next.js pages/app directories
Components	React functional components
Services	Custom hooks or Context providers
Dependency Injection	React Context + Service classes
NgRx	Zustand or Redux Toolkit
ngModel	Custom form hooks
Pipes	Custom format functions or libraries
Directives	Higher-order components or hooks

9.3 Building a Mental Model

Think of your React/Next.js application this way:

- Components are pure functions that transform props into UI

- Hooks are the way to add state and side effects to components
- Context provides a way to pass data through the component tree
- Next.js provides the structure and routing that Angular provides

Chapter 10

CLI and Workflow Tools

10.1 Project Setup

Use the following command to create a new Next.js project with our recommended configuration:

```
# Create new project
npx create-next-app@latest my-app --typescript --eslint --use-npm --tailwind --
    app

# Install additional dependencies
cd my-app
npm install zustand @tanstack/react-query zod react-hook-form @hookform/
    resolvers @radix-ui/react-* date-fns
```

Listing 10.1: Project Setup

10.2 Custom CLI Commands

Create npm scripts in package.json to standardize common tasks:

```
{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "lint": "next lint",
    "test": "vitest",
    "test:ui": "vitest --ui",
    "test:coverage": "vitest run --coverage",
    "e2e": "playwright test",
    "analyze": "ANALYZE=true next build",
    "generate:component": "node scripts/generate-component.js",
    "generate:page": "node scripts/generate-page.js",
    "generate:hook": "node scripts/generate-hook.js"
  }
}
```

Listing 10.2: NPM Scripts

10.3 Git Hooks

Set up pre-commit and pre-push hooks for quality control:

```
# Install Husky
npm install --save-dev husky lint-staged
```

```

# Set up Husky
npx husky install
npm set-script prepare "husky install"

# Add pre-commit hook
npx husky add .husky/pre-commit "npx lint-staged"
```

Listing 10.3: Git Hooks Setup

Configure lint-staged in package.json:

```
{
  "lint-staged": {
    "*.{ts,tsx}": [
      "eslint --fix",
      "prettier --write"
    ],
    "*.{json,css,md}": [
      "prettier --write"
    ]
  }
}
```

Listing 10.4: Lint Staged Configuration

Chapter 11

Appendix: Templates and Examples

11.1 Component Templates

11.1.1 Basic Component

```tsx
import { ReactNode } from 'react';
import cn from 'classnames';

interface CardProps {
  title?: string;
  children: ReactNode;
  className?: string;
}

export function Card({ title, children, className }: CardProps) {
  return (
    <div className={cn('rounded-lg border border-gray-200 p-4', className)}>
      {title && <h3 className="text-lg font-medium mb-2">{title}</h3>}
      {children}
    </div>
  );
}
```

Listing 11.1: Basic Component Template

11.1.2 Data Container Component

```tsx
import { useState, useEffect } from 'react';
import { Loading, ErrorMessage } from '@/components/ui';
import { UserList } from '@/components/features/users/UserList';
import { useUsers } from '@/hooks/useUsers';

interface UserListContainerProps {
  initialPage?: number;
  pageSize?: number;
  searchTerm?: string;
}

export function UserListContainer({
  initialPage = 1,
  pageSize = 10,
  searchTerm = ''
}: UserListContainerProps) {
  const [page, setPage] = useState(initialPage);
  const [search, setSearch] = useState(searchTerm);
```

```

  const { users, totalCount, isLoading, error } = useUsers({
    page,
    limit: pageSize,
    search
  });

  const handleSearch = (term: string) => {
    setSearch(term);
    setPage(1); // Reset to first page when searching
  };

  if (isLoading) return <Loading />;
  if (error) return <ErrorMessage message={error.message} />;

  return (
    <UserList
      users={users}
      totalUsers={totalCount}
      currentPage={page}
      pageSize={pageSize}
      onPageChange={setPage}
      onSearch={handleSearch}
      searchTerm={search}
    />
  );
}
```

Listing 11.2: Data Container Template

11.2 Custom Hook Templates

11.2.1 Data Fetching Hook

```
import { useSWR } from 'swr';
import { apiClient } from '@/lib/api';

export function useResource<T>(
  endpoint: string | null,
  options = {}
) {
  const { data, error, isLoading, mutate } = useSWR<T>(
    endpoint,
    endpoint ? () => apiClient.get<T>(endpoint) : null,
    options
  );

  return {
    data,
    isLoading,
    error,
    mutate,
    isError: !!error
  };
}
```

Listing 11.3: Data Fetching Hook Template

11.2.2 Form Hook

```
import { useState, useCallback, FormEvent } from 'react';

interface UseFormOptions<T> {
  initialValues: T;
  onSubmit: (values: T) => void | Promise<void>;
  validate?: (values: T) => Partial<Record<keyof T, string>>;
}

export function useForm<T extends Record<string, any>>({
  initialValues,
  onSubmit,
  validate
}: UseFormOptions<T>) {
  const [values, setValues] = useState<T>(initialValues);
  const [errors, setErrors] = useState<Partial<Record<keyof T, string>>>({});
  const [isSubmitting, setIsSubmitting] = useState(false);

  const handleChange = useCallback((
    e: React.ChangeEvent<HTMLInputElement | HTMLSelectElement |
    HTMLTextAreaElement>
  ) => {
    const { name, value, type } = e.target as HTMLInputElement;
    setValues(prev => ({
      ...prev,
      [name]: type === 'checkbox' ? (e.target as HTMLInputElement).checked :
    value
    }));
  }, []);

  const handleSubmit = async (e: FormEvent) => {
    e.preventDefault();

    if (validate) {
      const validationErrors = validate(values);
      if (Object.keys(validationErrors).length > 0) {
        setErrors(validationErrors);
        return;
      }
    }

    setErrors({});
    setIsSubmitting(true);

    try {
      await onSubmit(values);
    } finally {
      setIsSubmitting(false);
    }
  };

  return {
    values,
    errors,
    isSubmitting,
    handleChange,
    handleSubmit,
    setValue: (name: keyof T, value: any) => {
      setValues(prev => ({ ...prev, [name]: value }));
    },
    reset: () => setValues(initialValues)
  };
```

```
}
```

Listing 11.4: Form Hook Template

11.3 Service Template

```
// src/services/UserService.ts
import { createContext, useContext, ReactNode } from 'react';
import { apiClient } from '@/lib/api';
import type { User, PaginatedResponse } from '@/types';

class UserService {
  async getUsers(options: { page?: number; limit?: number; search?: string } =
    {}): Promise<PaginatedResponse<User>> {
    const { page = 1, limit = 10, search = '' } = options;
    return apiClient.get<PaginatedResponse<User>>('/users', {
      params: {
        page: String(page),
        limit: String(limit),
        search
      }
    });
  }

  async getUser(id: string): Promise<User> {
    return apiClient.get<User>(`/users/${id}`);
  }

  async createUser(data: Omit<User, 'id'>): Promise<User> {
    return apiClient.post<User>('/users', data);
  }

  async updateUser(id: string, data: Partial<User>): Promise<User> {
    return apiClient.put<User>(`/users/${id}`, data);
  }

  async deleteUser(id: string): Promise<void> {
    return apiClient.delete<void>(`/users/${id}`);
  }
}

// Create singleton instance
const userService = new UserService();

// Create context
const UserServiceContext = createContext<UserService | undefined>(undefined);

// Provider component
export function UserServiceProvider({ children }: { children: ReactNode }) {
  return (
    <UserServiceContext.Provider value={userService}>
      {children}
    </UserServiceContext.Provider>
  );
}

// Hook for accessing the service
export function useUserService() {
  const context = useContext(UserServiceContext);
  if (context === undefined) {
    throw new Error('useUserService must be used within a UserServiceProvider')
    ;
```

```
  }
  return context;
}

// Direct export for use outside of React components
export { userService };
```

Listing 11.5: Service Template

11.4 Store Template

```
// src/stores/todoStore.ts
import { create } from 'zustand';
import { persist } from 'zustand/middleware';
import type { Todo } from '@/types';

interface TodoState {
  // State
  todos: Todo[];
  isLoading: boolean;
  error: Error | null;

  // Actions
  fetchTodos: () => Promise<void>;
  addTodo: (text: string) => Promise<void>;
  toggleTodo: (id: string) => Promise<void>;
  deleteTodo: (id: string) => Promise<void>;
}

export const useTodoStore = create<TodoState>()(
  persist(
    (set, get) => ({
      todos: [],
      isLoading: false,
      error: null,

      fetchTodos: async () => {
        set({ isLoading: true, error: null });
        try {
          const response = await fetch('/api/todos');
          const todos = await response.json();
          set({ todos, isLoading: false });
        } catch (error) {
          set({ error: error as Error, isLoading: false });
        }
      },

      addTodo: async (text) => {
        set({ isLoading: true, error: null });
        try {
          const response = await fetch('/api/todos', {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ text, completed: false })
          });
          const newTodo = await response.json();
          set(state => ({
            todos: [...state.todos, newTodo],
            isLoading: false
          }));
        } catch (error) {
```

```
          set({ error: error as Error, isLoading: false });
        }
      },

      toggleTodo: async (id) => {
        const todo = get().todos.find(t => t.id === id);
        if (!todo) return;

        set({ isLoading: true, error: null });
        try {
          const response = await fetch(`/api/todos/${id}`, {
            method: 'PATCH',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ completed: !todo.completed })
          });
          const updatedTodo = await response.json();
          set(state => ({
            todos: state.todos.map(t => t.id === id ? updatedTodo : t),
            isLoading: false
          }));
        } catch (error) {
          set({ error: error as Error, isLoading: false });
        }
      },

      deleteTodo: async (id) => {
        set({ isLoading: true, error: null });
        try {
          await fetch(`/api/todos/${id}`, { method: 'DELETE' });
          set(state => ({
            todos: state.todos.filter(t => t.id !== id),
            isLoading: false
          }));
        } catch (error) {
          set({ error: error as Error, isLoading: false });
        }
      }
    }),
    {
      name: 'todo-storage',
      partialize: (state) => ({ todos: state.todos })
    }
  )
);
```

Listing 11.6: Zustand Store Template

11.5 API Route Template

```
// src/app/api/users/[id]/route.ts
import { NextResponse } from 'next/server';
import { z } from 'zod';
import { prisma } from '@/lib/prisma';

// Validation schema
const userUpdateSchema = z.object({
  name: z.string().min(2).optional(),
  email: z.string().email().optional(),
  role: z.enum(['USER', 'ADMIN']).optional()
});

```

```
// GET handler
export async function GET(
  request: Request,
  { params }: { params: { id: string } }
) {
  try {
    const user = await prisma.user.findUnique({
      where: { id: params.id }
    });

    if (!user) {
      return NextResponse.json(
        { error: 'User not found' },
        { status: 404 }
      );
    }

    return NextResponse.json(user);
  } catch (error) {
    console.error('Error fetching user:', error);
    return NextResponse.json(
      { error: 'Failed to fetch user' },
      { status: 500 }
    );
  }
}

// PATCH handler
export async function PATCH(
  request: Request,
  { params }: { params: { id: string } }
) {
  try {
    const body = await request.json();

    // Validate input
    const result = userUpdateSchema.safeParse(body);
    if (!result.success) {
      return NextResponse.json(
        { error: result.error.format() },
        { status: 400 }
      );
    }

    // Check if user exists
    const existingUser = await prisma.user.findUnique({
      where: { id: params.id }
    });

    if (!existingUser) {
      return NextResponse.json(
        { error: 'User not found' },
        { status: 404 }
      );
    }

    // Update user
    const updatedUser = await prisma.user.update({
      where: { id: params.id },
      data: result.data
    });

    return NextResponse.json(updatedUser);
```

```
  } catch (error) {
    console.error('Error updating user:', error);
    return NextResponse.json(
      { error: 'Failed to update user' },
      { status: 500 }
    );
  }
}

// DELETE handler
export async function DELETE(
  request: Request,
  { params }: { params: { id: string } }
) {
  try {
    // Check if user exists
    const existingUser = await prisma.user.findUnique({
      where: { id: params.id }
    });

    if (!existingUser) {
      return NextResponse.json(
        { error: 'User not found' },
        { status: 404 }
      );
    }

    // Delete user
    await prisma.user.delete({
      where: { id: params.id }
    });

    return NextResponse.json({ success: true });
  } catch (error) {
    console.error('Error deleting user:', error);
    return NextResponse.json(
      { error: 'Failed to delete user' },
      { status: 500 }
    );
  }
}
```

Listing 11.7: Next.js API Route Template

11.6 Test Templates

11.6.1 Component Test

```
// src/components/ui/Button/Button.test.tsx
import { render, screen, fireEvent } from '@testing-library/react';
import { Button } from './Button';

describe('Button', () => {
  it('renders correctly with default props', () => {
    render(<Button>Click me</Button>);

    const button = screen.getByRole('button', { name: /click me/i });
    expect(button).toBeInTheDocument();
    expect(button).toHaveClass('bg-blue-600'); // Primary variant default
    expect(button).not.toBeDisabled();
  });
```

```

  it('applies variant styles correctly', () => {
    render(<Button variant="secondary">Secondary Button</Button>);

    const button = screen.getByRole('button', { name: /secondary button/i });
    expect(button).toHaveClass('bg-gray-200');
    expect(button).not.toHaveClass('bg-blue-600');
  });

  it('handles disabled state', () => {
    render(<Button disabled>Disabled Button</Button>);

    const button = screen.getByRole('button', { name: /disabled button/i });
    expect(button).toBeDisabled();
    expect(button).toHaveClass('opacity-50');
  });

  it('calls onClick handler when clicked', () => {
    const handleClick = vi.fn();
    render(<Button onClick={handleClick}>Clickable Button</Button>);

    const button = screen.getByRole('button', { name: /clickable button/i });
    fireEvent.click(button);

    expect(handleClick).toHaveBeenCalledTimes(1);
  });

  it('does not call onClick when disabled', () => {
    const handleClick = vi.fn();
    render(<Button onClick={handleClick} disabled>Disabled Button</Button>);

    const button = screen.getByRole('button', { name: /disabled button/i });
    fireEvent.click(button);

    expect(handleClick).not.toHaveBeenCalled();
  });
});
```

Listing 11.8: Component Test Template

11.6.2 Hook Test

```
// src/hooks/useCounter.test.ts
import { renderHook, act } from '@testing-library/react';
import { useCounter } from './useCounter';

describe('useCounter', () => {
  it('initializes with default value', () => {
    const { result } = renderHook(() => useCounter());

    expect(result.current.count).toBe(0);
  });

  it('initializes with provided value', () => {
    const { result } = renderHook(() => useCounter(10));

    expect(result.current.count).toBe(10);
  });

  it('increments the counter', () => {
    const { result } = renderHook(() => useCounter());

```

```
    act(() => {
      result.current.increment();
    });

    expect(result.current.count).toBe(1);
  });

  it('decrements the counter', () => {
    const { result } = renderHook(() => useCounter(5));

    act(() => {
      result.current.decrement();
    });

    expect(result.current.count).toBe(4);
  });

  it('resets the counter to initial value', () => {
    const { result } = renderHook(() => useCounter(5));

    act(() => {
      result.current.increment();
      result.current.increment();
      result.current.reset();
    });

    expect(result.current.count).toBe(5);
  });

  it('allows setting custom value', () => {
    const { result } = renderHook(() => useCounter());

    act(() => {
      result.current.setValue(42);
    });

    expect(result.current.count).toBe(42);
  });
});
```

Listing 11.9: Hook Test Template

Chapter 12

Conclusion

12.1 Key Takeaways

As you adopt this framework and its patterns, remember these key principles:

- **Consistency is key** - Following established patterns makes codebases easier to maintain
- **Type safety provides confidence** - TypeScript and Zod ensure your code works as expected
- **Component boundaries matter** - Well-defined interfaces between components lead to more maintainable code
- **Tests are not optional** - They provide confidence when refactoring and help document code behavior
- **Documentation is part of the code** - Good documentation makes your code more valuable to the team

12.2 Continuous Improvement

This framework is not set in stone. We encourage contributions and suggestions to improve our development practices:

- Share common patterns you discover
- Advocate for improvements to the existing standards
- Create reusable components and hooks for the team to use
- Keep up with the ecosystem and suggest updates

12.3 Learning Resources

To deepen your understanding of these patterns and technologies, we recommend the following resources:

- Official Next.js Documentation: `https://nextjs.org/docs`
- React Documentation: `https://react.dev/`
- TypeScript Handbook: `https://www.typescriptlang.org/docs/handbook/intro.html`

- Zustand Documentation: `https://github.com/pmndrs/zustand`
- SWR Documentation: `https://swr.vercel.app/`
- React Hook Form Documentation: `https://react-hook-form.com/`

Final Thought

Remember that these patterns are tools, not rules. Use them to solve problems efficiently, but don't be afraid to adapt when needed. The best code is the one that solves the problem effectively while remaining maintainable by the team.