Production-Grade Frontend Folder Structure Guide

Complete Guide for Clean, Maintainable, and Scalable Frontend Applications

---

Table of Contents

Introduction

This document provides a comprehensive, production-grade frontend folder structure that works across frameworks (React, Next.js, Vue, etc.). It's designed for:

- **Scalability**: From small projects to enterprise applications
- **Maintainability**: Easy to understand and modify
- **Collaboration**: Multiple developers can work efficiently
- **Testability**: Clear structure for testing

Technology Focus: Optimized for React/Next.js but principles apply universally.

Root Level Structure

```
my-app/
├── public/                 # Static assets served directly
├── src/                    # All application source code
├── .env.example            # Environment variables template
├── .env.local          # Local environment variables (gitignored)
├── .gitignore          # Git ignore rules
├── .eslintrc.json      # ESLint configuration
├── .prettierrc         # Prettier configuration
├── tsconfig.json       # TypeScript configuration
├── next.config.js      # Next.js configuration (if using Next.js)
├── package.json        # Dependencies and scripts
├── package-lock.json   # Locked dependencies
├── README.md           # Project documentation
└── tailwind.config.js  # Tailwind CSS config (if using Tailwind)
```

Key Points

- `public/` : Static files that don't need processing (favicons, robots.txt, static images)
- `src/` : All your working code lives here
- **Config files**: At root for discoverability and tooling
- `.env.example` : Template for environment variables (commit this)
- `.env.local` : Actual secrets (never commit this)

Core Source Structure

```
src/
├── app/                   # Next.js App Router (or pages/ for Pages
Router)
├── components/            # Reusable UI components
├── features/             # Feature-based modules
├── hooks/                # Global custom hooks
├── lib/                  # External library configurations
├── services/             # Global API services
├── store/                # Global state management
├── styles/               # Global styles
├── types/                # Global TypeScript types
├── utils/                # Global utility functions
├── constants/            # Application constants
├── context/              # React Context providers
└── assets/               # Internal assets
```

Complete Folder Breakdown

1. Components Directory

```
src/components/
├── ui/                     # Generic UI elements
│   ├── button/
│   │   ├── button.tsx
│   │   ├── button.test.tsx
│   │   ├── button.module.css
│   │   └── index.ts
│   ├── input/
│   │   ├── input.tsx
│   │   ├── input.test.tsx
│   │   ├── input.module.css
│   │   └── index.ts
│   ├── card/
│   ├── badge/
│   ├── dialog/
│   └── dropdown/
│
├── layout/                # Layout components
│   ├── header/
│   │   ├── header.tsx
│   │   ├── header.module.css
│   │   └── index.ts
│   ├── footer/
│   ├── sidebar/
│   ├── navbar/
│   └── container/
│
└── shared/               # Cross-feature shared components
    ├── loader/
    ├── modal/
    ├── toast/
    ├── error-boundary/
    └── pagination/
```

Three-Tier Component System:

1. `ui/` - Pure presentational, zero business logic
2. `layout/` - Structural components defining page layout
3. `shared/` - Used across features, may contain logic

---

2. Features Directory (THE GAME CHANGER)

```
src/features/
├── auth/
│   ├── components/            # Auth-specific components
│   │   ├── login-form/
│   │   │   ├── login-form.tsx
│   │   │   ├── login-form.test.tsx
│   │   │   └── index.ts
│   │   ├── register-form/
│   │   └── reset-password-form/
│   ├── hooks/                 # Feature-specific hooks
│   │   ├── useAuth.ts
│   │   ├── useLogin.ts
│   │   └── useRegister.ts
│   ├── services/              # API calls for this feature
│   │   └── authService.ts
│   ├── store/                 # State management
│   │   └── authStore.ts
│   ├── types/                 # TypeScript types
│   │   └── auth.types.ts
│   ├── utils/                 # Feature utilities
│   │   ├── validateEmail.ts
│   │   └── hashPassword.ts
│   └── index.ts               # Public API of the feature
│
├── posts/
│   ├── components/
│   │   ├── post-card/
│   │   ├── post-list/
│   │   ├── post-form/
│   │   └── post-details/
│   ├── hooks/
│   │   ├── usePosts.ts
│   │   ├── useCreatePost.ts
│   │   └── useDeletePost.ts
│   ├── services/
│   │   └── postsService.ts
│   ├── store/
│   │   └── postsStore.ts
│   ├── types/
│   │   └── post.types.ts
│   └── index.ts
│
├── profile/
│   ├── components/
│   ├── hooks/
│   ├── services/
│   └── index.ts
│
└── comments/
    ├── components/
    ├── hooks/
    ├── services/
    └── index.ts
```

---

3. Hooks Directory

```
src/hooks/
├── useDebounce.ts
├── useLocalStorage.ts
├── useMediaQuery.ts
├── useIntersectionObserver.ts
├── useClickOutside.ts
├── useCopyToClipboard.ts
└── index.ts
```

Global hooks that can be used anywhere in the application.

---

4. Library Configurations

```
src/lib/
├── axios.ts              # Axios instance setup
├── react-query.ts        # React Query/TanStack Query setup
├── firebase.ts           # Firebase configuration
├── supabase.ts          # Supabase client
└── analytics.ts         # Analytics setup (GA, Mixpanel)
```

---

5. Services Directory

```
src/services/
├── api/
│   ├── client.ts             # Base API client
│   ├── endpoints.ts          # API endpoint constants
│   └── interceptors.ts       # Request/Response interceptors
├── storage/
│   ├── localStorage.ts
│   └── sessionStorage.ts
└── analytics/
    └── tracking.ts
```

---

6. Store Directory

```
src/store/
├── slices/                   # Redux slices or Zustand stores
│   ├── userSlice.ts
│   ├── themeSlice.ts
│   └── notificationSlice.ts
├── middleware/
│   └── logger.ts
└── index.ts                  # Store configuration
```

---

7. Styles Directory

```
src/styles/
├── globals.css              # Global styles
├── variables.css            # CSS variables
├── reset.css                # CSS reset
├── themes/
│   ├── dark.css
│   └── light.css
└── utilities.css            # Utility classes
```

---

8. Types Directory

```
src/types/
├── common.types.ts          # Common shared types
├── api.types.ts             # API response types
├── env.d.ts                 # Environment variable types
└── models/
```

```
    ├── user.model.ts
    ├── post.model.ts
    └── comment.model.ts
```

---

9. Utils Directory

```
src/utils/
├── format/
│   ├── date.ts
│   ├── currency.ts
│   └── number.ts
├── validation/
│   ├── validators.ts
│   └── schemas.ts          # Zod/Yup schemas
├── helpers/
│   ├── array.ts
│   ├── string.ts
│   └── object.ts
└── index.ts
```

---

10. Constants Directory

```
src/constants/
├── routes.ts                # Application routes
├── config.ts                # App configuration
├── messages.ts              # UI messages
├── api.ts                   # API constants
└── regex.ts                 # Regular expressions
```

---

11. Context Directory

```
src/context/
├── ThemeContext.tsx
├── AuthContext.tsx
├── ModalContext.tsx
└── index.ts
```

---

12. Assets Directory

```
src/assets/
├── images/
│   ├── logo.svg
│   └── placeholder.png
├── icons/
│   ├── social/
│   └── ui/
└── fonts/
    ├── custom-font.woff2
    └── custom-font.woff
```

Detailed Explanations

Component Structure Deep Dive

Every component should follow this pattern:

```
button/
├── button.tsx              # Main component file
├── button.test.tsx         # Unit tests
├── button.module.css       # Scoped styles
├── button.stories.tsx      # Storybook stories (optional)
└── index.ts               # Barrel export
```

Example `index.ts`:

```
export { Button } from './button';
export type { ButtonProps } from './button';
```

Why this works:

- Clean imports: `import { Button } from '@/components/ui/button'`
- Encapsulation: Internal implementation hidden
- Co-location: Everything related to Button is together

Feature-Based Architecture

The `features/` folder is what separates hobby projects from production apps.

Principles:

1. **Self-contained**: Each feature has everything it needs
2. **Encapsulated**: Internal components aren't exported
3. **Scalable**: Add features without touching existing code
4. **Team-friendly**: Different developers work on different features

Example Feature Structure:

```
// features/posts/index.ts (Public API)
export { PostCard, PostList } from './components/post-card';
export { PostForm } from './components/post-form';
export { usePosts, useCreatePost } from './hooks';
export type { Post, CreatePostDto, UpdatePostDto } from './types/post.typ

// Everything else stays private!
```

Usage:

```
// Other parts of app import from feature's public API
import { PostCard, usePosts } from '@/features/posts';
```

State Management Organization

Feature-level state (stays in feature):

```
// features/auth/store/authStore.ts
import { create } from 'zustand';
import type { User } from '../types/auth.types';

interface AuthState {
  user: User | null;
  isAuthenticated: boolean;
  login: (user: User) => void;
  logout: () => void;
}

export const useAuthStore = create<AuthState>((set) => ({
  user: null,
  isAuthenticated: false,
  login: (user) => set({ user, isAuthenticated: true }),
  logout: () => set({ user: null, isAuthenticated: false }),
}));
```

Global state (in `store/slices/`):

```
// store/slices/themeSlice.ts
import { createSlice } from '@reduxjs/toolkit';

export const themeSlice = createSlice({
  name: 'theme',
  initialState: { mode: 'light' as 'light' | 'dark' },
  reducers: {
    toggleTheme: (state) => {
      state.mode = state.mode === 'light' ? 'dark' : 'light';
    },
    setTheme: (state, action) => {
      state.mode = action.payload;
    },
  },
});

export const { toggleTheme, setTheme } = themeSlice.actions;
```

API Service Pattern

Base API Client:

```
// services/api/client.ts
import axios from 'axios';

export const apiClient = axios.create({
  baseURL: process.env.NEXT_PUBLIC_API_URL,
  timeout: 10000,
  headers: {
    'Content-Type': 'application/json',
  },
});

// Request interceptor for auth token
apiClient.interceptors.request.use((config) => {
  const token = localStorage.getItem('token');
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});

// Response interceptor for error handling
apiClient.interceptors.response.use(
  (response) => response,
  (error) => {
    if (error.response?.status === 401) {
      // Handle unauthorized
      window.location.href = '/login';
    }
    return Promise.reject(error);
  }
);
```

Feature Service:

```
// features/posts/services/postsService.ts
import { apiClient } from '@/services/api/client';
import type { Post, CreatePostDto, UpdatePostDto } from '../types/post.ty

export const postsService = {
  getAll: async () => {
    const response = await apiClient.get<Post[]>('/posts');
    return response.data;
  },

  getById: async (id: string) => {
    const response = await apiClient.get<Post>(`/posts/${id}`);
    return response.data;
  },

  create: async (data: CreatePostDto) => {
    const response = await apiClient.post<Post>('/posts', data);
    return response.data;
  },

  update: async (id: string, data: UpdatePostDto) => {
    const response = await apiClient.patch<Post>(`/posts/${id}`, data);
    return response.data;
  },

  delete: async (id: string) => {
    await apiClient.delete(`/posts/${id}`);
  },
};
```

Using with React Query:

```
// features/posts/hooks/usePosts.ts
import { useQuery } from '@tanstack/react-query';
import { postsService } from '../services/postsService';

export const usePosts = () => {
  return useQuery({
    queryKey: ['posts'],
    queryFn: postsService.getAll,
  });
};

export const useCreatePost = () => {
  const queryClient = useQueryClient();

  return useMutation({
    mutationFn: postsService.create,
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey: ['posts'] });
    },
  });
};
```

Type Organization

Feature-specific types:

```
// features/posts/types/post.types.ts
export interface Post {
  id: string;
  title: string;
  content: string;
  authorId: string;
  author: {
    id: string;
    name: string;
    avatar?: string;
  };
  tags: string[];
  createdAt: Date;
  updatedAt: Date;
}

export interface CreatePostDto {
  title: string;
  content: string;
  tags?: string[];
}

export interface UpdatePostDto extends Partial<CreatePostDto> {}

export interface PostFilters {
  authorId?: string;
  tags?: string[];
  search?: string;
}
```

Global types:

```
// types/common.types.ts
export interface ApiResponse<T> {
  data: T;
  message: string;
  success: boolean;
}

export interface PaginatedResponse<T> {
  data: T[];
  total: number;
  page: number;
  pageSize: number;
}

export type Status = 'idle' | 'loading' | 'success' | 'error';

export interface ErrorResponse {
  message: string;
  code: string;
  details?: Record<string, string[]>;
}
```

Naming Conventions

File Naming

Type	Convention	Example
Components	PascalCase.tsx	`UserProfile.tsx`
Utilities	camelCase.ts	`formatDate.ts`
Hooks	camelCase.ts (use prefix)	`useAuth.ts`
Types	camelCase.types.ts	`user.types.ts`
Services	camelCase.ts	`authService.ts`
Folders	kebab-case	`user-profile/`
Styles	component.module.css	`button.module.css`

Code Naming

Type	Convention	Example
Components	PascalCase	`const UserProfile = () => {}`
Functions	camelCase	`const getUserData = () => {}`
Variables	camelCase	`const userName = 'John'`
Constants	UPPER_SNAKE_CASE	`const API_BASE_URL = '...'`
Interfaces	PascalCase (I prefix optional)	`interface User {}`
Types	PascalCase	`type Status = 'active'`
Enums	PascalCase	`enum UserRole {}`

Specific Patterns

Boolean Variables:

```
const isLoading = true;
const hasError = false;
const shouldUpdate = true;
const canEdit = false;
```

Event Handlers:

```
const handleClick = () => {};
const handleSubmit = () => {};
const handleChange = () => {};
```

Custom Hooks:

```
const useAuth = () => {};
const useDebounce = () => {};
const useLocalStorage = () => {};
```

Component Props:

```
interface ButtonProps {
  onClick: () => void;
  isDisabled?: boolean;
  variant?: 'primary' | 'secondary';
}
```

Code Examples

Complete Component Example

```
// components/ui/button/button.tsx
import React from 'react';
import styles from './button.module.css';

export interface ButtonProps {
  children: React.ReactNode;
  variant?: 'primary' | 'secondary' | 'danger';
  size?: 'small' | 'medium' | 'large';
  isDisabled?: boolean;
  isLoading?: boolean;
  onClick?: () => void;
  type?: 'button' | 'submit' | 'reset';
}

export const Button: React.FC<ButtonProps> = ({
  children,
  variant = 'primary',
  size = 'medium',
  isDisabled = false,
  isLoading = false,
  onClick,
  type = 'button',
}) => {
  return (
    <button
      type={type}
      className={`${styles.button} ${styles[variant]} ${styles[size]}`}
      disabled={isDisabled || isLoading}
      onClick={onClick}
    >
      {isLoading ? 'Loading...' : children}
    </button>
  );
};
```

```css
/* components/ui/button/button.module.css */
.button {
  border: none;
  border-radius: 4px;
  cursor: pointer;
  font-weight: 600;
  transition: all 0.2s;
}

.primary {
  background-color: #0070f3;
  color: white;
}

.primary:hover {
  background-color: #0051cc;
}

.secondary {
  background-color: #eaeaea;
  color: #000;
}

.small {
  padding: 8px 16px;
  font-size: 14px;
}

.medium {
  padding: 12px 24px;
  font-size: 16px;
}

.large {
  padding: 16px 32px;
  font-size: 18px;
}
```

```
// components/ui/button/index.ts
export { Button } from './button';
export type { ButtonProps } from './button';
```

Complete Feature Example

```
// features/posts/types/post.types.ts
export interface Post {
  id: string;
  title: string;
  content: string;
  authorId: string;
  createdAt: Date;
}

export interface CreatePostDto {
  title: string;
  content: string;
}
```

```
// features/posts/services/postsService.ts
import { apiClient } from '@/services/api/client';
import type { Post, CreatePostDto } from '../types/post.types';

export const postsService = {
  getAll: async () => {
    const { data } = await apiClient.get<Post[]>('/posts');
    return data;
  },

  create: async (postData: CreatePostDto) => {
    const { data } = await apiClient.post<Post>('/posts', postData);
    return data;
  },
};


// features/posts/hooks/usePosts.ts
import { useQuery } from '@tanstack/react-query';
import { postsService } from '../services/postsService';

export const usePosts = () => {
  return useQuery({
    queryKey: ['posts'],
    queryFn: postsService.getAll,
  });
};


// features/posts/components/post-card/post-card.tsx
import React from 'react';
import type { Post } from '../../types/post.types';
import styles from './post-card.module.css';

interface PostCardProps {
  post: Post;
  onDelete?: (id: string) => void;
}

export const PostCard: React.FC<PostCardProps> = ({ post, onDelete }) =>
  return (
    <article className={styles.card}>
      <h3>{post.title}</h3>
      <p>{post.content}</p>
      {onDelete && (
        <button onClick={() => onDelete(post.id)}>Delete</button>
      )}
    </article>
  );
};
```

```
// features/posts/index.ts (Public API)
export { PostCard } from './components/post-card';
export { usePosts } from './hooks/usePosts';
export type { Post, CreatePostDto } from './types/post.types';
```

Path Aliases Configuration

tsconfig.json:

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "@/*": ["src/*"],
      "@/components/*": ["src/components/*"],
      "@/features/*": ["src/features/*"],
      "@/hooks/*": ["src/hooks/*"],
      "@/utils/*": ["src/utils/*"],
      "@/types/*": ["src/types/*"],
      "@/services/*": ["src/services/*"],
      "@/lib/*": ["src/lib/*"],
      "@/constants/*": ["src/constants/*"]
    }
  }
}
```

Usage:

```
// Instead of messy relative imports:
import { Button } from '../../../components/ui/button';
import { formatDate } from '../../../../utils/format/date';

// Use clean absolute imports:
import { Button } from '@/components/ui/button';
import { formatDate } from '@/utils/format/date';
import { usePosts } from '@/features/posts';
```

Best Practices

1. Single Responsibility Principle

Each component, function, or file should do ONE thing well.

Bad:

```
// UserProfileWithPostsAndComments.tsx - does too much!
```

Good:

```
// UserProfile.tsx - displays user info
// UserPosts.tsx - displays user's posts
// UserComments.tsx - displays user's comments
```

---

2. Co-location

Keep related files together.

Bad:

```
components/Button.tsx
styles/Button.css
tests/Button.test.tsx
types/ButtonTypes.ts
```

Good:

```
components/ui/button/
├── button.tsx
├── button.module.css
├── button.test.tsx
└── index.ts
```

---

3. Shallow Nesting

Keep folder nesting to maximum 2-3 levels.

Bad:

```
src/components/ui/forms/inputs/text/variants/outlined/TextInput.tsx
```

Good:

```
src/components/ui/text-input/TextInput.tsx
```

4. Barrel Exports

Use `index.ts` files for clean imports.

```
// components/ui/index.ts
export { Button } from './button';
export { Input } from './input';
export { Card } from './card';

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

5. Type Safety

Always use TypeScript and type everything.

```
// Good
interface User {
  id: string;
  name: string;
  email: string;
}

const getUser = (id: string): Promise<User> => {
  // implementation
};

// Bad
const getUser = (id) => {
  // no types
};
```

6. Consistent Formatting

Use tools like Prettier and ESLint.

.prettierrc:

```
{
  "semi": true,
  "trailingComma": "es5",
  "singleQuote": true,
  "printWidth": 80,
  "tabWidth": 2
}
```

---

7. Component Composition

Build complex UIs from simple components.

```
// Good - composable
<Card>
  <CardHeader>
    <CardTitle>Title</CardTitle>
  </CardHeader>
  <CardContent>Content</CardContent>
</Card>

// Bad - monolithic
<ComplexCardWithEverything title="Title" content="Content" />
```

---

8. Error Boundaries

Wrap features in error boundaries.

```
// components/shared/error-boundary/ErrorBoundary.tsx
import React from 'react';

interface Props {
  children: React.ReactNode;
  fallback?: React.ReactNode;
}

export class ErrorBoundary extends React.Component<Props> {
  state = { hasError: false };

  static getDerivedStateFromError() {
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      return this.props.fallback || <div>Something went wrong</div>;
    }
    return this.props.children;
  }
}
```

---

9. Environment Variables

Use `.env` files properly.

```
# .env.example (commit this)
NEXT_PUBLIC_API_URL=
NEXT_PUBLIC_ANALYTICS_ID=
DATABASE_URL=

# .env.local (never commit this)
NEXT_PUBLIC_API_URL=https://api.example.com
NEXT_PUBLIC_ANALYTICS_ID=G-XXXXXXXXXX
DATABASE_URL=postgresql://...
```

---

10. Documentation

Document complex logic and public APIs.

```
/**
 * Custom hook for debouncing a value
 *
 * @param value - The value to debounce
 * @param delay - Delay in milliseconds
 * @returns Debounced value
 *
 * @example
 * const debouncedSearch = useDebounce(searchTerm, 500);
 */
export const useDebounce = <T>(value: T, delay: number): T => {
  // implementation
};
```

Quick Reference Checklist

Starting a New Project

- [ ] Set up folder structure
- [ ] Configure path aliases in `tsconfig.json`
- [ ] Set up ESLint and Prettier
- [ ] Create `.env.example` file
- [ ] Add `.gitignore` for `.env.local`
- [ ] Set up base API client
- [ ] Create common types
- [ ] Set up global styles

Adding a New Feature

- [ ] Create feature folder in `features/`
- [ ] Add `components/` subfolder
- [ ] Add `hooks/` subfolder
- [ ] Add `services/` subfolder (if needed)
- [ ] Add `types/` subfolder
- [ ] Create `index.ts` for public API
- [ ] Only export what's needed

Creating a Component

- [ ] Create component folder
- [ ] Add main component file
- [ ] Add styles file
- [ ] Add test file
- [ ] Create `index.ts` barrel export
- [ ] Define TypeScript props interface
- [ ] Add JSDoc comments if complex

Before Committing

- [ ] Run linter: `npm run lint`
- [ ] Run tests: `npm test`
- [ ] Check build: `npm run build`
- [ ] Format code: `npm run format`
- [ ] Remove console.logs
- [ ] Update documentation if needed

Conclusion

This structure is battle-tested and used in production applications across the industry. Key principles to remember:

1. **Feature-based architecture** for scalability
2. **Co-location** for maintainability
3. **Consistent naming** for clarity
4. **Type safety** for reliability
5. **Shallow nesting** for simplicity

Start simple and add complexity only when needed. Not every project needs every folder from day one. Grow your structure organically as your application grows.

Additional Resources

- **Official Docs**: Next.js, React, TypeScript documentation
- **Style Guides**: Airbnb JavaScript Style Guide
- **Tools**: ESLint, Prettier, Husky for pre-commit hooks
- **State Management**: Zustand, Redux Toolkit, Jotai
- **Data Fetching**: TanStack Query (React Query), SWR

Document Version: 1.0
Last Updated: January 2026
Maintained by: Your Team

This document is a living guide. Update it as your team's practices evolve.