Common Repository Architecture

Rakshit Pandey

intellect

Architecture Proposal

Overview:

This proposal outlines the design of a modular **monorepo architecture** intended to centralize and standardize reusable frontend assets across multiple projects.
The primary goal is to promote consistency, improve maintainability, and accelerate development by providing a shared space for:

- UI components
- Utility functions
- Business logic
- Theming tokens and configuration
- Shared tooling and documentation

By housing these resources in a single repository with a well-organized package structure, the architecture supports:

- Versioned reuse of common modules
- Framework-agnostic utilities for maximum flexibility
- Scalable onboarding for new features and teams
- Consistent quality standards via shared configs and linters

This proposal details the directory structure, naming conventions, design principles, and tooling choices required to implement and evolve this monorepo architecture effectively.

Architecture Overview:

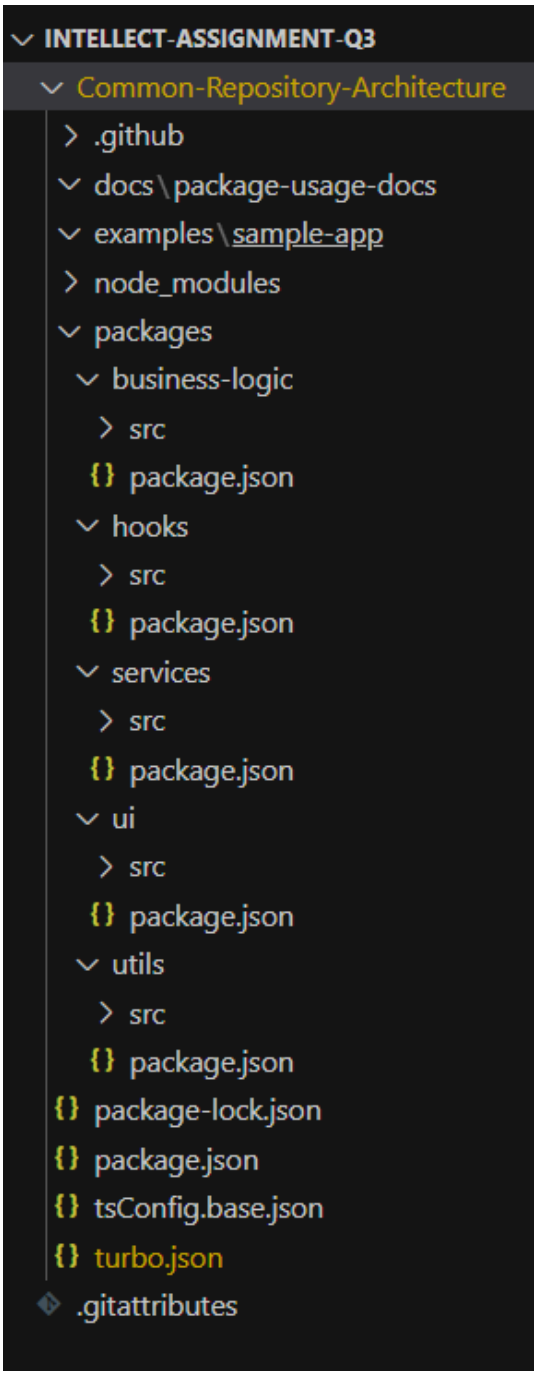
The proposed architecture follows a **modular monorepo structure** designed to facilitate scalability, reusability, and maintainability across multiple web projects. This centralized repository will serve as a shared foundation where commonly used UI components, utility functions, and business logic are developed, maintained, and consumed in a consistent manner. The architecture encourages clear separation of concerns by dividing the codebase into independently manageable packages—each with a well-defined responsibility.
The **monorepo** is organized under a packages directory that includes key modules such as:

- **ui:** A collection of atomic and composite UI components (e.g., buttons, modals, input fields), built with customization and accessibility in mind.
- **utils:** A library of framework-agnostic helper functions for formatting, validations, API handling, and error management.
- **business-logic:** Shared business logic, such as domain-specific rules, state handlers, or data transformations.
- **theme:** Centralized design tokens, color palettes, typography settings, and spacing utilities to ensure visual consistency.
- **config:** Shared ESLint, Prettier, TypeScript, and testing configurations to enforce standards across packages.

Additionally, the examples directory contains lightweight sample applications that demonstrate real-world usage of packages, useful for development and testing. The docs folder hosts internal documentation, including contribution guidelines, architecture decisions, and usage instructions.

This high-level design ensures that each unit can be developed, tested, and deployed independently, while still leveraging the benefits of a unified codebase. The architecture supports tools like **Turborepo,** Yarn/NPM Workspaces, and can be extended with CI/CD pipelines and semantic versioning. Ultimately, this structure promotes a highly maintainable and collaborative ecosystem, enabling developers to deliver consistent experiences across different projects efficiently.

Directory Structure:



Purpose of each main folder:

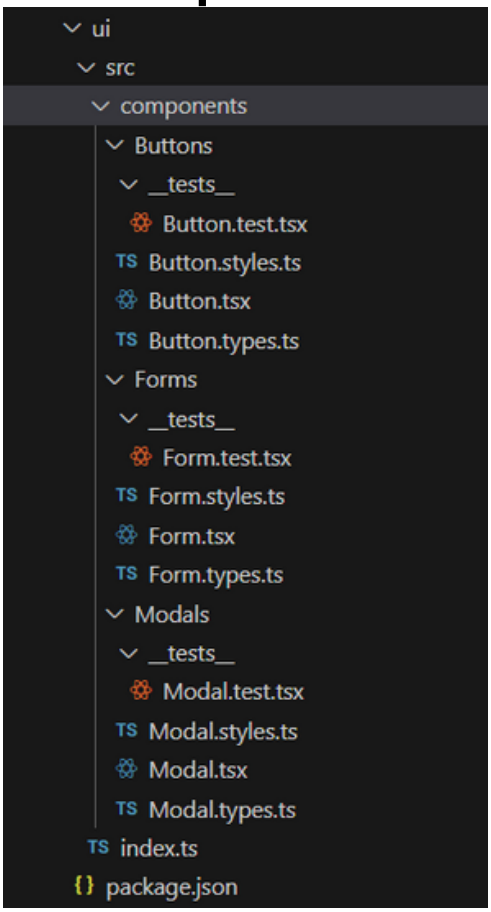
- ***packages/business-logic/*** **–** Contains core application logic and domain-specific rules.
- ***packages/hooks/*** **–** Holds reusable custom React hooks.
- ***packages/services/*** **–** Manages API calls and external service integrations.
- ***packages/ui/*** **–** Includes shared UI components used across applications.
- ***packages/utils/*** **–** Contains utility functions and helpers.
- ***examples/sample-app/*** **–** A sample app to demonstrate usage of the packages.
- ***docs/package-usage-docs/*** **–** Documentation for using different packages.
- ***.github/*** **–** GitHub-related configurations and workflows.
- ***turbo.json*** **–** Configuration for Turborepo task and cache management.
- ***tsconfig.base.json*** **–** Shared base TypeScript config for all packages

intellect

Naming Convention:

- ***Folder & File Naming***
  - Use kebab-case for all folder and file names (e.g., business-logic, sample-app, custom-hook.ts).
  - Keep names short, meaningful, and descriptive of their purpose.
- ***Package Names***
  - Follow the format: @repo-scope/package-name (e.g., @common/hooks, @common/services).
  - Use lowercase and hyphens to separate words.
- ***Component Naming (UI)***
  - React components should use PascalCase (e.g., Button.tsx, ModalHeader.tsx).
  - File and folder names for components should match the component name.
- ***Hook Naming***
  - All custom hooks must start with use (e.g., useAuth, useDebounce).
  - Keep hooks modular and single-responsibility.
- ***Utility Functions***
  - Use camelCase for function names (e.g., formatDate, capitalizeText).
  - Group related utils into files with descriptive names (e.g., date-utils.ts).
- ***Configuration Files***
  - Use standard naming like tsconfig.base.json, turbo.json, .eslintrc.json, etc.
  - All config files should be placed at the root unless scoped per package.

UI Components:



Folder Orgnaization:

- **components/:** Contains individual folders for each reusable UI element.
- **__tests__/ :** Inside each component folder: Keeps unit tests colocated with the components they test for better discoverability and maintainability.
- **index.ts:** Central export file for all UI components to simplify imports.
- **Separation of Concerns:**
  - Component.tsx: Core logic and JSX
  - Component.styles.ts: Tailwind/CSS-in-JS styles (if any)
  - Component.types.ts: TypeScript interfaces and prop definitions

intellect

Strategies for Customizable & Maintainable UI Components:

1. Props-Based Customization

- We use props to let developers change the behavior or look of components without modifying the internal code.
- Example: A Button component can accept props like variant="primary" or size="large" to control its style and size.

2. Consistent Design with Tailwind CSS

- We use Tailwind CSS for all styling, which allows us to apply consistent design patterns using utility classes. This also makes it easy to adjust styling quickly.
- Benefit: No need for custom CSS files, and the design remains uniform across all components.

3. Theme Support

- We allow global theming using Tailwind's configuration or custom design tokens. This helps in applying dark/light mode or brand-specific colors easily.
- Use Case: The same Button component can look different across apps by changing the theme, not the component code.

4. Component Isolation

- Each component does one job — it should not depend on external data or logic. This makes it easier to test, reuse, and understand.
- Good Practice: Keep logic like API calls or state outside the UI components.

5. Folder-Level Tests

- We place tests inside each component's folder under a __tests__ directory. This keeps testing close to the source and ensures every component is reliable.
- Benefit: Easy to maintain and track what is tested with the component.

6. Clear File Naming and Structure

- Each component has its own folder with separate files for logic, styles, types, and tests.

```
Button/
├── Button.tsx       // Logic
├── Button.styles.ts  // Styling (if needed)
├── Button.types.ts   // Props types
└── __tests__/
    └── Button.test.tsx
```

7. Centralized Exports

- We re-export all components through a single index.ts so other packages can easily import them.
- Example:
- [// packages/ui/src/index.ts]- export * from './components/Button/Button';

8. Documentation

- We can use tools like Storybook to document how components work and what props they accept. This helps teams understand and reuse components easily.

intellect

Utilities:

Types of Utility Functions to Include:

1. Data Formatting Utilities

Functions that help in transforming or displaying data consistently.

- formatDate(date: string): string – Converts ISO strings into readable formats.
- formatCurrency(value: number): string – Adds currency symbol, decimals.
- capitalize(text: string): string – Capitalizes the first letter.
- truncate(text: string, limit: number): string – Truncates long text with ellipsis.

2. API Handling Utilities

Reusable logic for making and managing API calls.

- apiClient – A pre-configured wrapper over fetch or axios with built-in headers, interceptors, and error handling.
- retryRequest – Retry failed requests with exponential backoff.
- serializeQueryParams – Serialize JS objects into query string format.

3. Error Handling Utilities

Functions for consistent error management across the app.

- parseError(error: unknown): string – Extracts a user-friendly message from API or runtime errors.
- logError(error: Error) – Standardized error logging (to console or remote).
- safeExecute(fn) – Wraps functions in try/catch for safe execution.

4**. Validation Utilities**

Common validation logic used across forms or APIs.

- isValidEmail, isValidPhone, isStrongPassword
- validateSchema – For validating objects against custom rules or a schema like zod or yup.

5. Local/Session Storage Utilities

Utilities for safely accessing and modifying browser storage.

- getFromLocalStorage, setToLocalStorage, clearStorage
- Automatically stringify/parse JSON values.

6. Environment/Platform Utilities

Helpers to adapt the app to different environments or platforms.

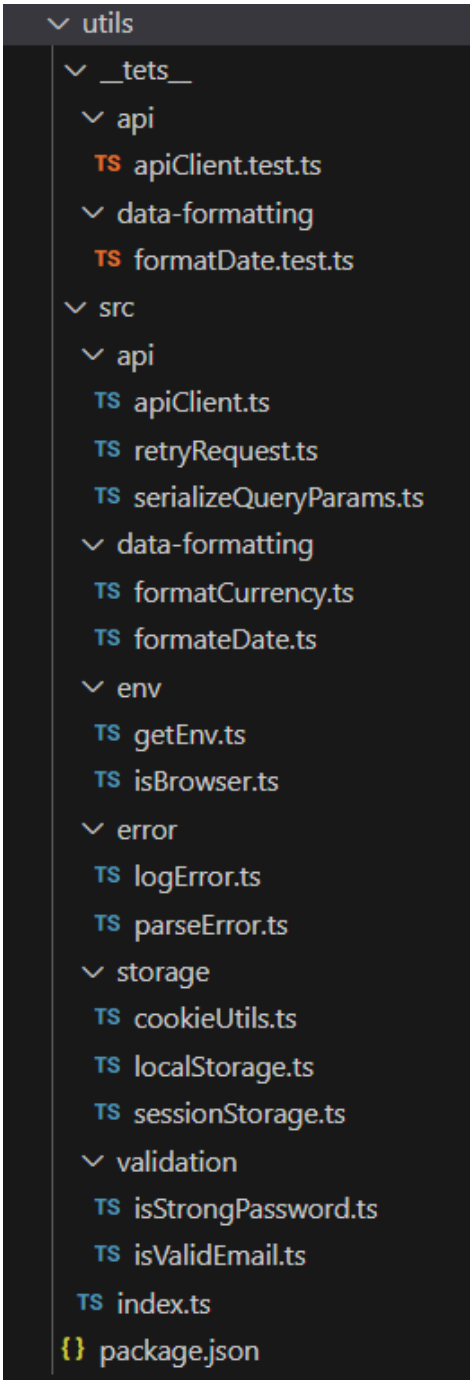
- getEnv(key: string) – Safely fetches environment variables.
- isMobile(), isBrowser(), isDevEnv()

7. State Management Helpers

- Shared Redux logic like:
  - createAsyncThunkWrapper – Standard wrapper to reduce boilerplate.
  - persistedReducer – Reusable localStorage wrapper for reducers.

intellect

Folder Structure:



Accessibility and Reusability of Utilities

1. Namespaced Package Structure

All utility functions are grouped into a standalone package (e.g., @common/utils) within the packages/ directory. This allows utilities to be easily imported into any other workspace package.

import { formatDate, handleApiError } from '@common/utils';

This makes utilities:

- Globally accessible across the entire monorepo.
- Independent of any app or UI logic, ensuring wide reusability.

2. Modular, Domain-Based Organization

Inside the utils package, utilities are separated by domain folders:

- Discoverability: Developers can easily find relevant utility functions.
- Maintainability: Adding, removing, or updating utilities doesn't affect unrelated areas.

3. Central Index File

We expose all utility functions via a single index.ts entry point. This central export file re-exports all modules:

export * from './formatting/formatDate';
export * from './api/apiClient';
export * from './errors/handleError';

Benefits:

- Simplifies imports.
- Prevents deep nested paths like @common/utils/formatting/formatDate.

4. Pure, Stateless Functions

All utilities are implemented as pure functions with no side effects. This ensures:

- Testability: Easy to write and maintain unit tests.
- Reusability: Can be used across apps, hooks, services, or business logic without modification.

5. Strict Type Safety

Each function is written in TypeScript with clear type annotations. This:

- Prevents runtime bugs.
- Improves developer experience through auto-completion and inline documentation.

6. Dedicated Testing Folder

All utility functions have corresponding unit tests in the __tests__ directory. For example: Tests are colocated logically, making it easy to maintain and extend test coverage.

7. Scalable Package Versioning

By isolating utilities in a shared package (@common/utils), it's easy to: Apply semantic versioning. Track changes across versions. Update utility usage across workspaces in a controlled way.

✳ Business Logic:

Manage Shared Business Logic:

A. Centralized Business Logic Module

I recommend maintaining a dedicated @common/business-logic package within the shared repository. This module encapsulates all reusable logic that doesn't belong to the UI layer but supports feature or domain functionality.

- 🔹 Contents of @common/business-logic:
  - Data Transformers: Functions for converting, formatting, or validating raw data.
  - Calculators: Logic for computing values (e.g., discount calculations, pricing formulas).
  - Validators: Cross-field or form validation logic.
  - Parsers: JSON or API response parsing utilities.
  - Domain-Specific Rules: For example, working days logic, permission rules, etc.
- 🔹 Benefits:
  - Encourages reuse across apps and teams.
  - Keeps business logic separate from UI concerns.
  - Easier to maintain, test, and evolve.

B. Dual Strategy for State Management

I provide two adaptable approaches under @common/state to address varying needs of frontend applications:

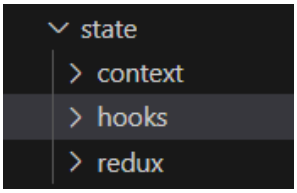
1. Lightweight State (React Context + Hooks)

- Ideal for small to medium apps.
- Encapsulates local/global state using Context API and custom hooks.
- No external dependencies.

2. Scalable State (Redux Toolkit)

- Recommended for larger applications.
- Centralized state management using Redux Toolkit.
- Optionally includes RTK Query for API state and caching.

- 🔹 Structure:



Local/global app state via React Context
Reusable hooks for accessing state
Redux slices and store setup

intellect

Keeping Business Logic Modular and Adaptable

To ensure shared business logic works across multiple projects:

- **Domain-Based Structure:** Logic is grouped by features (e.g., auth, user, notifications) for clarity and separation of concerns.
- **Pure Functions & Hooks:** Use pure utility functions and custom hooks (useAuth, useUserData) to make logic easily reusable and testable.
- **TypeScript for Contracts:** Shared types and interfaces ensure consistent data handling across apps.
- **Decoupled State Management:** Logic is not tightly coupled with any specific UI—can be used with Context, Redux Toolkit, or any other solution.
- **Environment Agnostic:** Avoid hard-coded environment/config values—inject them via context or config files.

Best Practices Guide:

Versioning

- Use Semantic Versioning (SemVer): MAJOR.MINOR.PATCH format.
- Maintain a CHANGELOG.md to track updates and breaking changes.

Documentation

- Add JSDoc/TSDoc comments for auto-completion and clarity.
- Include README.md in each package/module explaining usage.
- Use Storybook for documenting and testing UI components visually.

Testing

- Follow unit testing for all utilities, logic, and UI components.
- Use:
  - Vitest or Jest for logic tests.
  - React Testing Library for UI behavior.
- Ensure test coverage using c8 or --coverage flag.

Recommended Tools and Libraries

- **Turborepo / Nx –** for monorepo management and task orchestration.
- **Changesets –** for semantic versioning and changelog automation.
- **Redux Toolkit –** for managing and sharing state logic across apps and components efficiently.
- **React Testing Library –** for testing components based on user interaction.
- **Vitest / Jest –** for unit and integration testing.
- **ESLint –** to enforce consistent code quality and best practices.
- **Storybook –** to develop and document UI components in isolation.
- **JSDoc / TSDoc –** for inline documentation with editor support.
- **GitHub Actions / CI tools –** for automated tests, builds, and deployments.

intellect

Code Samples:

1. UI Component

File - packages/ui/components/Button/Button.tsx

```
import React from 'react';
import { ButtonProps } from './Button.types';

const Button: React.FC<ButtonProps> = ({ label, onClick, variant = 'primary' }) => {
  const baseClasses = 'font-semibold py-2 px-4 rounded';
  const variantClasses =
    variant === 'primary'
      ? 'bg-blue-600 hover:bg-blue-700 text-white'
      : 'bg-gray-300 hover:bg-gray-400 text-black';

  return (
    <button className={`${baseClasses} ${variantClasses}`} onClick={onClick}>
      {label}
    </button>
  );
};

export default Button;
```

File - packages/ui/src/components/Button/Button.types.ts

```
export interface ButtonProps {
  label: string;
  onClick: () => void;
  variant?: 'primary' | 'secondary';
}
```

File - packages/ui/src/index.ts

```
export { default as Button } from './components/Button/Button';
```

2. Utility Function

File - packages/utils/src/date/formatDate.ts)

```
export const formatDate = (date: Date, locale: string = 'en-IN'): string => {
  return new Intl.DateTimeFormat(locale, {
    day: '2-digit',
    month: 'short',
    year: 'numeric',
  }).format(date);
};
```

File - packages/utils/src/index.ts

```
export * from './date/formatDate';
```

3. Business logic

File: packages/business-logic/src/discount/calculateDiscount.ts

```
export const calculateDiscount = (price: number, percent: number): number => {
  const discount = (price * percent) / 100;
  return Math.round(price - discount);
};
```

File: packages/business-logic/src/index.ts

```
export * from './discount/calculateDiscount';
```

Usage:

File: sample-app/src/component/discount.tsx

```
import React from 'react';
import { Button } from '@intellect/ui';
import { formatDate } from '@intellect/utils';
import { calculateDiscount } from '@intellect/business-logic';

export default function App() {
  const handleClick = () => {
    console.log('Clicked!');
  };

  const formatted = formatDate(new Date());
  const discountedPrice = calculateDiscount(1000, 15);

  return (
    <div className="p-4">
      <h1 className="text-xl font-bold mb-4">Welcome</h1>
      <p>Today is {formatted}</p>
      <p>Discounted Price: ₹{discountedPrice}</p>
      <Button label="Click Me" onClick={handleClick} variant="primary" />
    </div>
  );
}
```

Update sample-app/package.json

```
{
“name": "sample-app",
"version": "1.0.0",
"private": true,"
dependencies":
{"@intellect/ui": "*,
"@intellect/utils": "*",
"@intellect/business-logic": "*""}
}
```