Coding Standards for Angular Projects

1. Pages/Components

- **Avoid Deep Nesting:** Keep the component and template structure as flat as possible for better readability and maintainability
- **Use Signal for Reactive Properties:** Use Angular's new signal type for reactive properties to improve reactivity and performance.
- **Optimize Change Detection:** Always use `ChangeDetectionStrategy.OnPush` in components for better performance and reduced unnecessary checks.
- **File Size Limit:** Limit each file to a maximum of 200 lines to promote readability and maintainability.
- **Reusable Components:** Implement reusable components for isolated UI elements or logic that can be reused across different parts of the application.
- **Error Handling:** Always anticipate and handle expected errors (e.g., API failures, user input errors) to improve reliability.
- **Button Disable on API Calls:** All buttons triggering API calls must be disabled upon click and display a loading state to prevent multiple submissions.
- **Responsive Design:** Ensure UI components and pages are fully responsive and adapt properly across all target devices (desktop, tablet, mobile).
- **Separate Logic into Units:** Isolate complex logic into separate classes or use cases to keep components focused on UI concerns and ensure reusability.
- **Standardize with PrimeNG:** Prefer using PrimeNG components as the standard UI library and apply custom styles where necessary for consistency.

2. App Styles & SCSS

- **Tailwind Integration:**
  - You may use Tailwind CSS classes in the templates for quick styling, but follow these rules:
    - Short and Concise: Use Tailwind directly in templates only when the classes are short and easy to read.
    - Use @apply for Long Classes: If the Tailwind classes are long or complex, move them into SCSS files and apply them using the @apply directive.
  - **SCSS Organization:** Organize SCSS files by grouping styles into clear sections like color, layout, fonts, etc., with each section separated by a line.
- **Design System:** Define a basic theme or minimalistic design system using variables (colors, spacing, typography) to maintain consistency across the app.
- **Avoid Repetition:** Do not repeat styles in different components. Create

shared classes for common styles and reuse them.

- **Component Animations:** Leverage Angular's built-in animation API to add animations to components for smooth transitions and improved UX.

3. Logic and Code Structure

- **Keep Code Simple and Expressive:** Write code that is easy to read and understand. Avoid overcomplicating solutions.
- **Avoid Nested Conditionals:** Minimize nested if statements to improve clarity and prevent deeply indented, hard-to-read code.
- **No Nested switch or ternary Operators:** Never nest switch statements or ternary operators, as they lead to hard-to-follow logic. Use intermediate variables if necessary.
- **Limit Long Ternary Operations:** Avoid overly long ternary operations. Extract complex ternary conditions to well-named variables.
- **Avoid any Type:** Never use the any type in TypeScript. Always define precise types for variables and functions.
- **Prefer Simple, Pure Functions:** Write small, pure functions with a single responsibility. Functions should not have side effects.
- **Minimize Code Nesting:** Try to keep code nesting minimal for better readability and maintainability.
- **Avoid Magic Numbers:** Replace hardcoded numbers with constants that clearly express their meaning.
- **Centralize Constants:** Store constant values (strings, numbers, enums) in a central place, such as a constants file or an enum.

Naming Conventions:

- **Constants:** Use `ALLCAPS` (e.g., `MAX_LIMIT`, `API_URL`).
- **Variables:** Use `camelCase` (e.g., `userDetails`, `apiResponse`).
- **Files and Folders:** Use `kebab-case` for files and folder names (e.g., `user-profile.component.ts`, `auth-service.ts`).
- **Clarity in Naming:** Stick to established conventions and avoid ambiguous or misleading names. Names should convey the purpose clearly.
- **Commenting:** Provide brief comments for non-obvious variables or logic, explaining the "why" behind complex decisions or algorithms.
- **Avoid Redundancy:** Eliminate unnecessary or redundant words in names (e.g., `PaginationHelper` should just be `Pagination`).
- **Functions:** Name functions after the action they perform (e.g., `fetchData`, `updateUser`).
- **Variables:** Name variables using concise, meaningful nouns (e.g., `userList`, `totalCount`).
- **Classes:** Use singular, descriptive nouns for class names (e.g., `UserProfile`, `InvoiceManager`). Avoid acronyms unless it's an established convention.

4. Recommended Design Patterns

Solid Priciples

- **Single Responsibility Principle (SRP):** Each class or component should have one job or reason to change.
- **Open/Closed Principle (OCP):** Components should be open for extension but closed for modification. This allows for flexible future changes.
- **Liskov Substitution Principle (LSP):** You should be able to replace a class with its subclass without affecting the program's behavior.
- **Interface Segregation Principle (ISP):** Clients should not be forced to implement interfaces they don't use. Keep interfaces focused and small.
- **Dependency Inversion Principle (DIP):** High-level modules should not depend on low-level modules. Both should depend on abstractions (interfaces).

Behavioral Design Patterns

```
-   **Memento**
-   **State**
-   **Iterator**
-   **Strategy**
-   **Chain of Responsibility**
-   **Template Method**
-   **Command**
-   **Mediator**
-   **Observer**
-   **Visitor**
```

Structural Design Patterns

```
-   **Composite**
-   **Adapter**
-   **Decorator**
-   **Facade**
-   **Flyweight**
-   **Brdige**
-   **Proxy**
```

Creational Design Patterns

```
-   **Prototype**
-   **Singleton**
-   **Factory Method**
-   **Abstract Factory**
-   **Builder**
```

Other Principles:

- **KISS (Keep It Simple, Stupid)** Always strive for simplicity. Avoid over-engineering solutions.
- **DRY (Don't Repeat Yourself)** Avoid repeating logic. Extract common functionality into reusable functions or services. - Don't repeat yourself

5. Additional Best Practices

- **Testing:** Always write unit tests for your services, components, and use cases. Ensure your code is testable by following principles like Dependency Injection and separation of concerns.
- **Error Boundaries:** Use Angular's built-in error handling mechanisms to catch and log errors appropriately.
- **Modularization:** Split your application into smaller, feature-based modules to improve scalability and maintainability.
- **Documentation:** Use clear and concise documentation for complex modules, functions, or components. Aim for clarity but avoid over-documenting obvious parts of the code which most of it should be.