**Angular Coding Standards**

**1. Project Structure**

**1.1. Standard Folder Structure**

* **src/app**: Main application source folder.
  + **components/**: Reusable components.
  + **services/**: Services for business logic and API calls.
  + **models/**: Data models and interfaces.
  + **modules/**: Feature modules.
  + **pipes/**: Custom pipes.
  + **directives/**: Custom directives.
  + **shared/**: Shared components, directives, and pipes.
  + **guards/**: Route guards.
  + **interceptors/**: HTTP interceptors.
  + **core/**: Core module for singleton services and global functionality.
  + **assets/**: Static assets (images, styles, etc.).
  + **environments/**: Environment configuration files.
* **src/styles**: Global stylesheets.
* **src/assets**: Static assets like images and fonts.

**1.2. Module Organization**

* **Feature Modules**: Group related components, services, and other entities.
* **Core Module**: Singleton services used across the application.
* **Shared Module**: Common components, pipes, and directives used by multiple modules.

**2. Coding Conventions**

**2.1. Naming Conventions**

* **Component Files**: component-name.component.ts
* **Service Files**: service-name.service.ts
* **Module Files**: module-name.module.ts
* **Directive Files**: directive-name.directive.ts
* **Pipe Files**: pipe-name.pipe.ts
* **Model Files**: model-name.model.ts

**2.2. Component and Module Names**

* Use **PascalCase** for component, directive, pipe, and module names.
  + Example: UserProfileComponent, AppModule
* Use **kebab-case** for file names.
  + Example: user-profile.component.ts, app.module.ts

**3. Component Standards**

**3.1. Component Definition**

* Use OnPush change detection strategy for performance.
* Use Angular CLI for generating components to follow best practices.
* Define components with an appropriate selector following the app- prefix.
  + Example: selector: 'app-user-profile'

**3.2. Templates**

* Use **Interpolation** for one-way binding.
  + Example: <h1>{{ title }}</h1>
* Use **ng-container** for structural directives without adding to the DOM.
  + Example: <ng-container \*ngIf="condition">Content</ng-container>
* Prefer **attribute binding** over property binding.
  + Example: <img [attr.src]="imageUrl" alt="Image">
* Use **ngClass** and **ngStyle** for conditional styling.
  + Example: <div [ngClass]="{'active': isActive}">Content</div>

**3.3. Styles**

* Use **SCSS** for component styles.
* Use **BEM (Block, Element, Modifier)** methodology for class names.
  + Example: .block\_\_element--modifier
* Avoid **global styles** to prevent clashes; use **encapsulation**.

**4. Service Standards**

**4.1. Service Creation**

* Use Angular CLI to generate services.
* Services should be **singleton** and **stateless**.
* Prefer **Dependency Injection** over creating instances directly.

**4.2. HttpClient Usage**

* Use Angular’s HttpClient for HTTP requests.
* Use **RxJS** operators for handling responses.
* Define a **base URL** for API endpoints in environment files.
* Handle errors gracefully with **catchError** operator.

**5. Module Standards**

**5.1. Module Configuration**

* Import and export **CommonModule** in feature modules.
* Use **Lazy Loading** for feature modules to improve performance.
* Use **SharedModule** for commonly used components and services.

**5.2. Dependency Injection**

* Use **providedIn: 'root'** for services used across the application.
* Use **providedIn: 'any'** for services with scoped instances.

**6. Code Practices**

**6.1. TypeScript Standards**

* Use **TypeScript types** for variables and return values.
* Enable **strict mode** in tsconfig.json.
* Use **const and let** instead of var.

**6.2. Code Readability**

* Use **consistent indentation** (2 or 4 spaces).
* Follow **maximum line length** of 120 characters.
* Use **comments** to explain complex logic.

**6.3. Error Handling**

* Use **try-catch** for synchronous operations.
* Use **catchError** for handling asynchronous errors in observables.

**7. Testing Standards**

**7.1. Unit Testing**

* Use **Jasmine** and **Karma** for unit tests.
* Write tests for components, services, and pipes.
* Aim for at least **80% code coverage**.

**7.2. End-to-End Testing**

* Use **Protractor** or **Cypress** for end-to-end testing.
* Write tests for critical user journeys.
* Ensure tests cover **happy paths** and **edge cases**.

**8. Documentation Standards**

**8.1. Code Comments**

* Use **JSDoc** style comments for functions and methods.
  + Example: /\*\* This function does X \*/
* Comment **complex logic** and **algorithms**.

**8.2. Documentation Tools**

* Use **Compodoc** for generating documentation.
* Maintain a **README** file with project setup and usage instructions.

**9. Version Control Standards**

**9.1. Git Practices**

* Follow **Git Flow** for branching strategy.
* Use **descriptive commit messages**.
  + Example: feat(user-profile): add profile update feature

**9.2. Pull Requests**

* Ensure code review before merging.
* Use **code linters** and **formatters** to maintain code quality.

**10. Performance Best Practices**

**10.1. Lazy Loading**

* Load feature modules on demand to reduce initial load time.

**10.2. AOT Compilation**

* Use **Ahead-of-Time (AOT)** compilation for production builds.

**10.3. Bundle Optimization**

* Use **tree shaking** to remove unused code.
* Minimize **polyfills** to reduce bundle size.

**11. Security Best Practices**

**11.1. Input Validation**

* Validate all user inputs on the client side and server side.

**11.2. Output Encoding**

* Encode all output to prevent XSS attacks.

**11.3. Use HTTPS**

* Always use **HTTPS** for secure data transmission.

**12. Deployment Standards**

**12.1. Environment Configuration**

* Use **environment files** for configuration.
* Do not include **sensitive data** in environment files.

**12.2. CI/CD Pipelines**

* Use **Continuous Integration/Continuous Deployment** pipelines for automated builds and deployments.
* Implement **automated tests** and **code quality checks** in the pipeline.