Angular Clean Code and Testing Guide

```
## Table of Contents 1. [Clean Code Principles](#clean-code-principles) 2. [Component Design](#component-design) 3. [Service Design](#service-design) 4. [Store Design](#store-design) 5. [Testing Principles](#testing-principles) 6. [Component Testing](#component-testing) 7. [Service Testing](#service-testing) 8. [Store Testing](#store-testing) 9. [Best Practices](#best-practices)
```

Clean Code Principles

1. Single Responsibility Principle

> **Bad: Component doing too many things** ```typescript @Component({ selector: "app-book-list", template: ` {{ book.title }} Delete Loading... {{ error }} `, }) export class BookListComponent { books: Book[] = []; loading = false; error: string | null = null;
>
> constructor(private bookService: BookService, private store: Store) {}
>
> ngOnInit() { this.loadBooks(); }
>
> loadBooks() { this.loading = true; this.bookService.getBooks().subscribe({ next: (books) => {

```
this.books = books; this.loading = false; }, error: (error) =>
{ this.error = error.message; this.loading = false; }, }); }

deleteBook(id: string) {
this.bookService.deleteBook(id).subscribe({ next: () =>
this.loadBooks(), error: (error) => (this.error =
error.message), }); } }
```

```
</div>

<div class="good-example">
**Good: Separated concerns**
```typescript
@Component({
  selector: "app-book-list",
  template: `
    <app-book-list-content
      [books]="books$ | async"
      [loading]="loading$ | async"
      [error]="error$ | async"
      (deleteBook)="onDeleteBook($event)">
    </app-book-list-content>
  `,
})
export class BookListComponent {
  books$ = this.store.select(selectBooks);
  loading$ = this.store.select(selectBooksLoading);
  error$ = this.store.select(selectBooksError);

  constructor(private store: Store) {}

  ngOnInit() {
    this.store.dispatch(loadBooks());
  }

  onDeleteBook(id: string) {
    this.store.dispatch(deleteBook({ id }));
  }
}

@Component({
  selector: "app-book-list-content",
  template: `
```
```

```
    <div>
      <div *ngFor="let book of books">
        {{ book.title }}
        <button (click)="deleteBook.emit(book.id)">Delete</button>
      </div>
      <div *ngIf="loading">Loading...</div>
      <div *ngIf="error">{{ error }}</div>
    </div>
  `,
})
export class BookListContentComponent {
  @Input() books: Book[] = [];
  @Input() loading = false;
  @Input() error: string | null = null;
  @Output() deleteBook = new EventEmitter<string>();
}
```

Tip: Separating concerns makes components more maintainable and testable. The container component handles state management while the presentational component focuses on rendering.

2. Interface Segregation

Bad: Large interface ```typescript interface Book { id: string; title: string; author: string; price: number; publishedDate: Date; isbn: string; description: string; coverImage: string; category: string; rating: number; reviews: Review[]; publisher: string; pageCount: number; language: string; format: string; weight: number; dimensions: string; } ```

Good: Segregated interfaces ```typescript interface BaseBook { id: string; title: string; author: string; price: number; }

interface BookDetails extends BaseBook { publishedDate: Date; isbn: string; description: string; coverImage: string; category: string; }

```
interface BookMetadata { publisher: string; pageCount: number; language: string; format: string; weight: number; dimensions: string; }

interface BookReview { rating: number; reviews: Review[]; }

interface Book extends BookDetails, BookMetadata, BookReview {}
```

````
</div>

<div class="note">
**Note:** Interface segregation helps maintain type safety and makes t
</div>

### 3. Dependency Inversion

<div class="bad-example">
**Bad: Direct dependency**
```typescript
@Component({
  selector: "app-book-list",
  template: `...`,
})
export class BookListComponent {
  constructor(private bookService: BookService) {}
}
```
</div>

<div class="good-example">
**Good: Interface-based dependency**
```typescript
interface IBookService {
  getBooks(): Observable<Book[]>;
  deleteBook(id: string): Observable<void>;
}

@Component({
  selector: "app-book-list",
  template: `...`,
})
export class BookListComponent {
````
````

```
  constructor(@Inject("IBookService") private bookService: IBookServic
}
```

Tip: Using interfaces for dependencies makes the code more flexibl

Component Design

1. Smart and Presentational Components

Smart Component (Container)
```typescript
@Component({
  selector: "app-book-list-container",
  template: `
    <app-book-list
      [books]="books$ | async"
      [loading]="loading$ | async"
      [error]="error$ | async"
      (deleteBook)="onDeleteBook($event)">
    </app-book-list>
  `,
})
export class BookListContainerComponent {
  books$ = this.store.select(selectBooks);
  loading$ = this.store.select(selectBooksLoading);
  error$ = this.store.select(selectBooksError);

  constructor(private store: Store) {}

  onDeleteBook(id: string) {
    this.store.dispatch(deleteBook({ id }));
  }
}
```

Presentational Component (Dumb)
```typescript
@Component({
  selector: "app-book-list",
  template: `
```

```
    <div class="book-list">
      <div *ngFor="let book of books" class="book-item">
        <app-book-card
          [book]="book"
          (delete)="deleteBook.emit(book.id)">
        </app-book-card>
      </div>
      <app-loading-spinner *ngIf="loading"></app-loading-spinner>
      <app-error-message *ngIf="error" [error]="error"></app-error-mes
    </div>
  `,
})
export class BookListComponent {
  @Input() books: Book[] = [];
  @Input() loading = false;
  @Input() error: string | null = null;
  @Output() deleteBook = new EventEmitter<string>();
}
```

Note: The container component handles state management and data fe

2. Component Communication

Parent Component
```typescript
@Component({
  selector: "app-parent",
  template: `
    <app-child
      [data]="data"
      (dataChange)="onDataChange($event)">
    </app-child>
  `,
})
export class ParentComponent {
  data = { value: "test" };

  onDataChange(newData: any) {
    this.data = newData;
  }
}
```

```

<div class="code-header">Child Component</div>
```typescript
@Component({
  selector: "app-child",
  template: `
    <div>
      <input [ngModel]="data.value" (ngModelChange)="onChange($event)"
    </div>
  `,
})
export class ChildComponent {
  @Input() data: any;
  @Output() dataChange = new EventEmitter<any>();

  onChange(value: string) {
    this.dataChange.emit({ ...this.data, value });
  }
}
```

<div class="tip">
Tip: Use Input/Output decorators for parent-child communication.
</div>

Service Design

1. Base API Service

<div class="code-header">Base API Service</div>
```typescript
@Injectable({
  providedIn: "root",
})
export class ApiService {
  private readonly baseUrl = environment.apiUrl;
  private readonly defaultHeaders: HttpHeaders = new HttpHeaders({
    "Content-Type": "application/json",
  });

  constructor(private http: HttpClient, private errorHandler: ErrorHa

  get<T>(endpoint: string, params?: HttpParams): Observable<T> {
```
```

````
    return this.http
      .get<T>(`${this.baseUrl}/${endpoint}`, {
        headers: this.defaultHeaders,
        params,
      })
      .pipe(catchError(this.errorHandler.handleError));
  }

  post<T, R>(endpoint: string, data: T): Observable<R> {
    return this.http
      .post<R>(`${this.baseUrl}/${endpoint}`, data, {
        headers: this.defaultHeaders,
      })
      .pipe(catchError(this.errorHandler.handleError));
  }
}
```

### 2. Feature Service

<div class="code-header">Feature Service</div>
```typescript
@Injectable({
  providedIn: "root",
})
export class BookService {
  private readonly endpoint = "books";

  constructor(private apiService: ApiService) {}

  getBooks(): Observable<BookResponse> {
    return this.apiService.get<BookResponse>(this.endpoint);
  }

  createBook(book: CreateBookRequest): Observable<Book> {
    return this.apiService.post<CreateBookRequest, Book>(this.endpoint
  }
}
```

<div class="note">
**Note:** The base API service handles common HTTP functionality, whil
</div>
````
````

````
## Store Design

### 1. Actions

<div class="code-header">Actions</div>
```typescript
export const loadBooks = createAction("[Books] Load Books");
export const loadBooksSuccess = createAction(
  "[Books] Load Books Success",
  props<{ books: Book[] }>()
);
export const loadBooksFailure = createAction(
  "[Books] Load Books Failure",
  props<{ error: ApiError }>()
);
```

### 2. Reducer

<div class="code-header">Reducer</div>
```typescript
export interface BooksState {
  books: Book[];
  loading: boolean;
  error: ApiError | null;
}

export const initialState: BooksState = {
  books: [],
  loading: false,
  error: null,
};

export const booksReducer = createReducer(
  initialState,
  on(loadBooks, (state) => ({
    ...state,
    loading: true,
    error: null,
  })),
  on(loadBooksSuccess, (state, { books }) => ({
    ...state,
    books,
    loading: false,
````
````

```
  })),
  on(loadBooksFailure, (state, { error }) => ({
    ...state,
    error,
    loading: false,
  }))
);
```

3. Effects

Effects
```typescript
@Injectable()
export class BooksEffects {
  loadBooks$ = createEffect(() =>
    this.actions$.pipe(
      ofType(loadBooks),
      mergeMap(() =>
        this.bookService.getBooks().pipe(
          map((response: BookResponse) => loadBooksSuccess({ books: re
          catchError((error: ApiError) => of(loadBooksFailure({ error
        )
      )
    )
  );
}
```

Tip: Use effects to handle side effects like API calls. This keeps

Testing Principles

1. Arrange-Act-Assert Pattern

Component Test
```typescript
describe("BookListComponent", () => {
  // Arrange
  let component: BookListComponent;
  let fixture: ComponentFixture<BookListComponent>;
  let store: Store;
```

```
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [BookListComponent],
      providers: [provideMockStore({ initialState })],
    });
    fixture = TestBed.createComponent(BookListComponent);
    component = fixture.componentInstance;
    store = TestBed.inject(Store);
  });

  // Act & Assert
  it("should load books on init", () => {
    const storeSpy = spyOn(store, "dispatch");
    component.ngOnInit();
    expect(storeSpy).toHaveBeenCalledWith(loadBooks());
  });
});
```

2. Isolation

Service Test
```typescript
describe("ApiService", () => {
  let service: ApiService;
  let httpMock: HttpTestingController;
  let errorHandler: ErrorHandler;

  const mockErrorHandler = {
    handleError: jest.fn(),
  };

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [ApiService, { provide: ErrorHandler, useValue: mockE
    });
  });

  afterEach(() => {
    httpMock.verify();
  });
});
```

```
```

3. State Testing

State Test
```typescript
describe("BookListComponent", () => {
  it("should show loading state", () => {
    const loadingState = {
      books: {
        books: [],
        loading: true,
        error: null,
      },
    };

    TestBed.resetTestingModule();
    TestBed.configureTestingModule({
      declarations: [BookListComponent],
      providers: [provideMockStore({ initialState: loadingState })],
    });

    fixture = TestBed.createComponent(BookListComponent);
    fixture.detectChanges();

    expect(fixture.nativeElement.querySelector(".loading")).toBeTruthy
  });
});
```

Note: Testing different states helps ensure your components handle

Best Practices

1. Component Testing
- Test component creation
- Test lifecycle hooks
- Test template bindings
- Test user interactions
- Test error states
- Test loading states

```
### 2. Service Testing
- Test HTTP requests
- Test error handling
- Test data transformation
- Test service dependencies
- Test service methods

### 3. Store Testing
- Test actions
- Test reducers
- Test effects
- Test selectors
- Test state updates

### 4. General Testing Tips
- Use meaningful test descriptions
- Test one thing per test
- Use setup and teardown properly
- Mock external dependencies
- Test edge cases
- Test error scenarios
- Keep tests maintainable
- Use type-safe testing
- Follow AAA pattern
- Isolate tests

## Conclusion

This guide covers the essential aspects of writing clean code and test

1. Follow SOLID principles
2. Use proper component architecture
3. Implement proper service design
4. Follow store patterns
5. Write comprehensive tests
6. Follow testing best practices
7. Keep code maintainable
8. Use type safety
9. Handle errors properly
10. Test edge cases

By following these guidelines, you can create maintainable, testable,
```