Angular Testing Guide: Best Practices and Examples

Table of Contents

Component Design Patterns

Bad Pattern - Hard to Test

```
@Component({
  selector: "app-book-list",
  template: `
    <div *ngFor="let book of books">
      {{ book.title }}
      <button (click)="addToCart(book)">Add to Cart</button>
    </div>
  `,
})
export class BookListComponent {
  books: Book[] = [];

  constructor(private http: HttpClient) {
    this.loadBooks();
  }

  private loadBooks() {
    this.http.get("/api/books").subscribe((books) => {
      this.books = books;
    });
  }

  addToCart(book: Book) {
    // Complex logic here
  }
}
```

Problems with this pattern:

- Direct HTTP calls in component (hard to mock)
- Private methods (can't test directly)
- State management in component (hard to track)
- No error handling
- No loading states
- Complex logic mixed with presentation

Good Pattern - Easy to Test

```
@Component({
  selector: "app-book-list",
  template: `
    <div *ngFor="let book of books$ | async">
      {{ book.title }}
      <button (click)="onAddToCart(book)">Add to Cart</button>
    </div>
  `,
})
export class BookListComponent {
  books$ = this.store.select(selectBooks);

  constructor(private store: Store, private bookService: BookService) {}

  onAddToCart(book: Book) {
    this.store.dispatch(BookActions.addToCart({ book }));
  }
}
```

Why this is better for testing:

1. **Separation of Concerns**
   - Component only handles UI and user interactions
   - Data fetching moved to service
   - State management handled by store
   - Each piece can be tested independently
2. **Observable Pattern**
   - `books$` is an Observable, making async testing easier
   - Can test loading states
   - Can test error states
   - Can test data updates
3. **Dependency Injection**
   - Store and Service are injected
   - Easy to mock in tests
   - Clear dependencies

Service Design Patterns

Bad Pattern - Hard to Test

```
@Injectable()
export class BookService {
  constructor(private http: HttpClient) {}

  getBooks() {
    return this.http.get("/api/books");
  }
}
```

Problems:

- No error handling
- No type safety
- No data transformation
- Hard to test error scenarios

Good Pattern - Easy to Test

```
@Injectable()
export class BookService {
  constructor(private http: HttpClient, private errorHandler: ErrorHandler) {}

  getBooks(): Observable<Book[]> {
    return this.http.get<Book[]>("/api/books").pipe(
      catchError(this.errorHandler.handleError),
      map((response) => response.data)
    );
  }
}
```

Why this is better for testing:

1. **Type Safety**
   - Return type is explicitly defined
   - TypeScript can catch errors at compile time
   - Easier to write type-safe tests
2. **Error Handling**
   - Centralized error handling
   - Can test error scenarios
   - Consistent error handling across app
3. **Data Transformation**
   - Clear data flow
   - Easy to test transformations
   - Can mock HTTP responses

Store Design Patterns

Bad Pattern - Hard to Test

```
export const bookReducer = (state = initialState, action: any) => {
  switch (action.type) {
    case "ADD_BOOK":
      return [...state, action.payload];
    default:
      return state;
  }
};
```

Problems:

- No type safety
- No action creators
- No selectors
- Hard to test state changes

Good Pattern - Easy to Test

```
export const bookReducer = createReducer(
  initialState,
  on(BookActions.addBook, (state, { book }) => ({
    ...state,
    books: [...state.books, book],
  }))
);
```

Why this is better for testing:

1. **Type Safety**
   - Actions are typed
   - State is typed
   - Reducer is type-safe
2. **Action Creators**

   - Actions are created consistently
   - Easy to test action creation
   - Can mock actions in tests

3. **Selectors**

   - Can test state selection
   - Can mock selectors
   - Easy to test state updates

Template Design Patterns

Bad Pattern - Hard to Test

```
@Component({
  template: `
    <div *ngIf="isLoading">Loading...</div>
    <div *ngIf="error">{{error}}</div>
    <div *ngIf="data">{{data}}</div>
  `
})
```

Problems:

- No way to reliably select elements
- Hard to test specific states
- No clear structure

Good Pattern - Easy to Test

```
@Component({
  template: `
    <div data-testid="loading" *ngIf="isLoading$ | async">
      Loading...
    </div>
    <div data-testid="error" *ngIf="error$ | async as error">
      {{error}}
    </div>
    <div data-testid="content" *ngIf="data$ | async as data">
      {{data}}
    </div>
  `
})
```

Why this is better for testing:

1. **Test IDs**

   - Reliable element selection
   - Tests are resilient to template changes
   - Clear purpose for each element

2. **Async Pipe**

   - Handles subscription cleanup
   - Tests async states easily
   - No need to manually handle subscriptions

3. **State Management**

   - Clear loading states
   - Clear error states
   - Clear content states

Testing Examples

Component Testing

```
describe("BookListComponent", () => {
  let component: BookListComponent;
  let fixture: ComponentFixture<BookListComponent>;
  let store: MockStore;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [BookListComponent],
      providers: [createMockStore({})],
    }).compileComponents();

    fixture = createComponentFixture(BookListComponent);
    component = fixture.componentInstance;
    store = TestBed.inject(MockStore);
  });

  it("should display books", () => {
    const books = [
      { id: 1, title: "Book 1" },
      { id: 2, title: "Book 2" },
    ];

    store.overrideSelector(selectBooks, books);
    fixture.detectChanges();

    const bookElements = fixture.debugElement.queryAll(By.css('[data-testid="book-card"]'));
    expect(bookElements.length).toBe(2);
  });
});
```

Why this testing approach is good:

1. **Setup**
   - Clear test setup
   - Reusable test utilities
   - Mocked dependencies
2. **Test Structure**
   - Clear test descriptions
   - Isolated tests
   - No side effects
3. **Assertions**
   - Clear expectations
   - Easy to understand
   - Easy to maintain

Test Utilities

```
// test-utils.ts
export function createMockStore<T>(initialState: T) {
  return provideMockStore({
    initialState,
    selectors: [],
  });
}

export function createComponentFixture<T>(component: Type<T>) {
  return TestBed.createComponent(component);
}

export function getElementByTestId(fixture: ComponentFixture<any>, testId: string) {
  return fixture.debugElement.query(By.css(`[data-testid="${testId}"]`));
}
```

Why utilities are good:

1. **Reusability**
   - Common test setup
   - Consistent testing patterns
   - Less code duplication
2. **Maintainability**
   - Centralized test helpers
   - Easy to update
   - Consistent across tests
3. **Readability**
   - Clear intent
   - Self-documenting
   - Easy to understand

Key Principles to Remember

1. **Use Data Attributes**
   - Add `data-testid` attributes to important elements
   - Makes it easier to find elements in tests
2. **Keep Components Simple**
   - Move business logic to services
   - Use store for state management
   - Keep components focused on presentation
3. **Use Observables**
   - Prefer observables over direct properties
   - Makes async testing easier
   - Better state management
4. **Dependency Injection**
   - Use constructor injection
   - Makes mocking easier
   - Better separation of concerns
5. **Error Handling**
   - Centralize error handling
   - Use error states in store
   - Makes error testing easier
6. **Type Safety**
   - Use TypeScript interfaces
   - Use strict typing
   - Prevents runtime errors