Comprehensive Guide to Testing Angular Applications with Jasmine and Karma

Table of Contents

Core Concepts

Testing Angular Features

Advanced Testing

Best Practices & Patterns

Additional Resources

Introduction

What is Jasmine?

Jasmine is a behavior-driven development (BDD) framework for testing JavaScript code. It provides:

- A test runner
- Assertion library
- Mocking and spying capabilities
- Clean, readable syntax

What is Karma?

Karma is a test runner that:

- Executes tests in real browsers
- Watches files for changes
- Provides test coverage reports
- Integrates with CI/CD pipelines

Why Test Angular Applications?

- **Reliability**: Catch bugs early in development
- **Refactoring confidence**: Make changes without fear
- **Documentation**: Tests serve as living documentation
- **Quality assurance**: Maintain code quality over time
- **Faster debugging**: Isolate issues quickly

Quick Start Guide

New to Angular testing? Start here:

1. Set up your test environment
2. Learn Jasmine fundamentals (matchers, spies, etc.)

3. Write your first component test
4. Understand asynchronous testing
5. Follow best practices for maintainable tests
6. Check the Q&A section for common questions

↑ Back to Top

Setup and Configuration

Initial Setup

Angular CLI projects come pre-configured with Jasmine and Karma. No additional setup is required.

```
# Run tests
ng test

# Run tests with code coverage
ng test --code-coverage

# Run tests in headless mode (CI/CD)
ng test --browsers=ChromeHeadless --watch=false
```

Karma Configuration

karma.conf.js

```
module.exports = function(config) {
  config.set({
    basePath: '',
```

```
    frameworks: ['jasmine', '@angular-devkit/build-angula
    plugins: [
      require('karma-jasmine'),
      require('karma-chrome-launcher'),
      require('karma-jasmine-html-reporter'),
      require('karma-coverage'),
      require('@angular-devkit/build-angular/plugins/karm
    ],
    client: {
      jasmine: {
        random: false, // Run tests in order
        seed: 42, // Set seed for reproducible random ord
        stopSpecOnExpectationFailure: false
      },
      clearContext: false
    },
    jasmineHtmlReporter: {
      suppressAll: true
    },
    coverageReporter: {
      dir: require('path').join(__dirname, './coverage'),
      subdir: '.',
      reporters: [
        { type: 'html' },
        { type: 'text-summary' },
        { type: 'lcovonly' }
      ],
      check: {
        global: {
          statements: 80,
          branches: 80,
          functions: 80,
          lines: 80
        }
      }
    },
```

```
    reporters: ['progress', 'kjhtml'],
    browsers: ['Chrome'],
    restartOnFileChange: true,
    singleRun: false
  });
};
```

TypeScript Configuration for Tests

tsconfig.spec.json

```
{
  "extends": "./tsconfig.json",
  "compilerOptions": {
    "outDir": "./out-tsc/spec",
    "types": ["jasmine", "node"]
  },
  "include": [
    "**/*.spec.ts",
    "**/*.d.ts"
  ]
}
```

↑ Back to Top

Jasmine Fundamentals

Test Structure

```
describe('Test Suite Name', () => {
  // Setup
  beforeEach(() => {
    // Runs before each test
  });

  afterEach(() => {
    // Runs after each test
  });

  beforeAll(() => {
    // Runs once before all tests
  });

  afterAll(() => {
    // Runs once after all tests
  });

  it('should do something', () => {
    // Arrange
    const value = 10;

    // Act
    const result = value * 2;

    // Assert
    expect(result).toBe(20);
  });
});
```

Matchers

```
describe('Jasmine Matchers', () => {
  it('should demonstrate equality matchers', () => {
    expect(2 + 2).toBe(4); // Strict equality (===)
    expect({ name: 'John' }).toEqual({ name: 'John' }); /
    expect(true).toBeTruthy();
    expect(false).toBeFalsy();
    expect(null).toBeNull();
    expect(undefined).toBeUndefined();
    expect('value').toBeDefined();
  });

  it('should demonstrate comparison matchers', () => {
    expect(10).toBeGreaterThan(5);
    expect(5).toBeLessThan(10);
    expect(10).toBeGreaterThanOrEqual(10);
    expect(5).toBeLessThanOrEqual(5);
    expect(0.1 + 0.2).toBeCloseTo(0.3, 2);
  });

  it('should demonstrate string matchers', () => {
    expect('Hello World').toContain('World');
    expect('test@example.com').toMatch(/\w+@\w+\.\w+/);
  });

  it('should demonstrate array matchers', () => {
    expect([1, 2, 3]).toContain(2);
    expect([1, 2, 3]).toEqual(jasmine.arrayContaining([2,
  });

  it('should demonstrate object matchers', () => {
    const obj = { name: 'John', age: 30, city: 'NYC' };
    expect(obj).toEqual(jasmine.objectContaining({ name:
  });

  it('should demonstrate exception matchers', () => {
```

```
    expect(() => { throw new Error('Error!'); }).toThrow(
    expect(() => { throw new Error('Error!'); }).toThrowE
  });
});
```

Test Helpers

```
describe('Test Helpers', () => {
  it('should skip this test', () => {
    pending('Reason for skipping');
  });

  xit('should also skip this test', () => {
    // Test code
  });

  fit('should only run this test (focused)', () => {
    // Only this test will run
  });

  fdescribe('Focused Suite', () => {
    // Only tests in this suite will run
  });
});
```

↑ Back to Top

Testing Components

Basic Component Testing

```
// user-profile.component.ts
import { Component, input, output, signal } from '@angula
import { CommonModule } from '@angular/common';

export interface User {
  id: number;
  name: string;
  email: string;
}

@Component({
  selector: 'app-user-profile',
  imports: [CommonModule],
  template: `
    <div class="user-profile">
      <h2>{{ user().name }}</h2>
      <p>{{ user().email }}</p>
      <button (click)="onDelete()">Delete</button>
      @if (showDetails()) {
        <div class="details">
          <p>User ID: {{ user().id }}</p>
        </div>
      }
    </div>
  `,
  styles: [`
    .user-profile { padding: 1rem; }
    .details { margin-top: 1rem; }
  `]
})
export class UserProfileComponent {
  user = input.required<User>();
  showDetails = input(false);
  userDeleted = output<number>();
```

```
  onDelete(): void {
    this.userDeleted.emit(this.user().id);
  }
}
```

```
// user-profile.component.spec.ts
import { ComponentFixture, TestBed } from '@angular/core/
import { UserProfileComponent, User } from './user-profil
import { DebugElement } from '@angular/core';
import { By } from '@angular/platform-browser';

describe('UserProfileComponent', () => {
  let component: UserProfileComponent;
  let fixture: ComponentFixture<UserProfileComponent>;
  let compiled: HTMLElement;

  const mockUser: User = {
    id: 1,
    name: 'John Doe',
    email: 'john@example.com'
  };

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [UserProfileComponent]
    }).compileComponents();

    fixture = TestBed.createComponent(UserProfileComponen
    component = fixture.componentInstance;
    compiled = fixture.nativeElement;
  });

  it('should create', () => {
    expect(component).toBeTruthy();
```

```
  });

  it('should display user name and email', () => {
    // Arrange
    fixture.componentRef.setInput('user', mockUser);

    // Act
    fixture.detectChanges();

    // Assert
    const h2 = compiled.querySelector('h2');
    const p = compiled.querySelector('p');
    expect(h2?.textContent).toContain('John Doe');
    expect(p?.textContent).toContain('john@example.com');
  });

  it('should show details when showDetails is true', () =
    // Arrange
    fixture.componentRef.setInput('user', mockUser);
    fixture.componentRef.setInput('showDetails', true);

    // Act
    fixture.detectChanges();

    // Assert
    const details = compiled.querySelector('.details');
    expect(details).toBeTruthy();
    expect(details?.textContent).toContain('User ID: 1');
  });

  it('should hide details when showDetails is false', () =
    // Arrange
    fixture.componentRef.setInput('user', mockUser);
    fixture.componentRef.setInput('showDetails', false);

    // Act
```

```
    fixture.detectChanges();

    // Assert
    const details = compiled.querySelector('.details');
    expect(details).toBeNull();
  });

  it('should emit userDeleted event when delete button is
    // Arrange
    fixture.componentRef.setInput('user', mockUser);
    fixture.detectChanges();
    let emittedId: number | undefined;

    component.userDeleted.subscribe((id: number) => {
      emittedId = id;
    });

    // Act
    const button = compiled.querySelector('button') as HTM
    button.click();
    fixture.detectChanges();

    // Assert
    expect(emittedId).toBe(1);
  });

  it('should use DebugElement for queries', () => {
    // Arrange
    fixture.componentRef.setInput('user', mockUser);
    fixture.detectChanges();

    // Act
    const buttonDebugElement: DebugElement = fixture.debug
      By.css('button')
    );
```

```
    // Assert
    expect(buttonDebugElement.nativeElement.textContent).
  });
});
```

Testing Component with Dependencies

```
// user-list.component.ts
import { Component, OnInit, signal } from '@angular/core'
import { CommonModule } from '@angular/common';
import { UserService } from './user.service';
import { User } from './user.model';

@Component({
  selector: 'app-user-list',
  imports: [CommonModule],
  template: `
    <div class="user-list">
      @if (loading()) {
        <p>Loading...</p>
      }
      @if (error()) {
        <p class="error">{{ error() }}</p>
      }
      @if (users().length > 0) {
        <ul>
          @for (user of users(); track user.id) {
            <li>{{ user.name }}</li>
          }
        </ul>
      }
    </div>
  `
})
```

```
export class UserListComponent implements OnInit {
  private userService = inject(UserService);

  users = signal<User[]>([]);
  loading = signal(false);
  error = signal<string | null>(null);

  ngOnInit(): void {
    this.loadUsers();
  }

  loadUsers(): void {
    this.loading.set(true);
    this.userService.getUsers().subscribe({
      next: (users) => {
        this.users.set(users);
        this.loading.set(false);
      },
      error: (err) => {
        this.error.set('Failed to load users');
        this.loading.set(false);
      }
    });
  }
}
```

```
// user-list.component.spec.ts
import { ComponentFixture, TestBed } from '@angular/core/
import { UserListComponent } from './user-list.component'
import { UserService } from './user.service';
import { of, throwError } from 'rxjs';
import { User } from './user.model';

describe('UserListComponent', () => {
```

```
  let component: UserListComponent;
  let fixture: ComponentFixture<UserListComponent>;
  let userService: jasmine.SpyObj<UserService>;

  const mockUsers: User[] = [
    { id: 1, name: 'John Doe', email: 'john@example.com'
    { id: 2, name: 'Jane Smith', email: 'jane@example.com
  ];

  beforeEach(async () => {
    const userServiceSpy = jasmine.createSpyObj('UserServ

    await TestBed.configureTestingModule({
      imports: [UserListComponent],
      providers: [
        { provide: UserService, useValue: userServiceSpy
      ]
    }).compileComponents();

    fixture = TestBed.createComponent(UserListComponent);
    component = fixture.componentInstance;
    userService = TestBed.inject(UserService) as jasmine.
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('should load users on init', () => {
    // Arrange
    userService.getUsers.and.returnValue(of(mockUsers));

    // Act
    fixture.detectChanges(); // ngOnInit is called

    // Assert
```

```
    expect(userService.getUsers).toHaveBeenCalled();
    expect(component.users()).toEqual(mockUsers);
    expect(component.loading()).toBe(false);
  });

  it('should display loading message', () => {
    // Arrange
    userService.getUsers.and.returnValue(of(mockUsers));
    component.loading.set(true);

    // Act
    fixture.detectChanges();

    // Assert
    const compiled = fixture.nativeElement;
    expect(compiled.textContent).toContain('Loading...');
  });

  it('should display users after loading', () => {
    // Arrange
    userService.getUsers.and.returnValue(of(mockUsers));

    // Act
    fixture.detectChanges();

    // Assert
    const compiled = fixture.nativeElement;
    const listItems = compiled.querySelectorAll('li');
    expect(listItems.length).toBe(2);
    expect(listItems[0].textContent).toContain('John Doe'
    expect(listItems[1].textContent).toContain('Jane Smit
  });

  it('should handle error when loading fails', () => {
    // Arrange
    const error = new Error('Network error');
```

```
    userService.getUsers.and.returnValue(throwError(() =>

    // Act
    fixture.detectChanges();

    // Assert
    expect(component.error()).toBe('Failed to load users'
    expect(component.loading()).toBe(false);
  });

  it('should display error message', () => {
    // Arrange
    userService.getUsers.and.returnValue(throwError(() =>
    fixture.detectChanges();

    // Act
    const compiled = fixture.nativeElement;
    const errorElement = compiled.querySelector('.error')

    // Assert
    expect(errorElement?.textContent).toContain('Failed t
  });
});
```

See Also:

- Testing with Signals - Modern state management in components
- Mocking and Spies - How to mock service dependencies
- Best Practices - Writing maintainable component tests

↑ Back to Top

Testing Services

Basic Service Testing

```
// calculator.service.ts
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class CalculatorService {
  add(a: number, b: number): number {
    return a + b;
  }

  subtract(a: number, b: number): number {
    return a - b;
  }

  multiply(a: number, b: number): number {
    return a * b;
  }

  divide(a: number, b: number): number {
    if (b === 0) {
      throw new Error('Cannot divide by zero');
    }
    return a / b;
  }
}
```

```
// calculator.service.spec.ts
import { TestBed } from '@angular/core/testing';
import { CalculatorService } from './calculator.service';
```

```
describe('CalculatorService', () => {
  let service: CalculatorService;

  beforeEach(() => {
    TestBed.configureTestingModule({});
    service = TestBed.inject(CalculatorService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });

  describe('add', () => {
    it('should add two positive numbers', () => {
      expect(service.add(2, 3)).toBe(5);
    });

    it('should add negative numbers', () => {
      expect(service.add(-2, -3)).toBe(-5);
    });

    it('should add zero', () => {
      expect(service.add(5, 0)).toBe(5);
    });
  });

  describe('subtract', () => {
    it('should subtract two numbers', () => {
      expect(service.subtract(5, 3)).toBe(2);
    });
  });

  describe('multiply', () => {
    it('should multiply two numbers', () => {
      expect(service.multiply(3, 4)).toBe(12);
    });
```

```
  });

  describe('divide', () => {
    it('should divide two numbers', () => {
      expect(service.divide(10, 2)).toBe(5);
    });

    it('should throw error when dividing by zero', () =>
      expect(() => service.divide(10, 0)).toThrowError('C
    });
  });
});
```

Service with HTTP Dependencies

```
// user.service.ts
import { Injectable, inject } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable, throwError } from 'rxjs';
import { catchError, map } from 'rxjs/operators';
import { environment } from '../environment/environment';

export interface User {
  id: number;
  name: string;
  email: string;
}

@Injectable({
  providedIn: 'root'
})
export class UserService {
  private http = inject(HttpClient);
  private apiUrl = `${environment.apiUrl}/users`;
```

```
  getUsers(): Observable<User[]> {
    return this.http.get<User[]>(this.apiUrl).pipe(
      catchError(this.handleError)
    );
  }

  getUserById(id: number): Observable<User> {
    return this.http.get<User>(`${this.apiUrl}/${id}`).pi
      catchError(this.handleError)
    );
  }

  createUser(user: Omit<User, 'id'>): Observable<User> {
    return this.http.post<User>(this.apiUrl, user).pipe(
      catchError(this.handleError)
    );
  }

  updateUser(id: number, user: Partial<User>): Observable
    return this.http.put<User>(`${this.apiUrl}/${id}`, us
      catchError(this.handleError)
    );
  }

  deleteUser(id: number): Observable<void> {
    return this.http.delete<void>(`${this.apiUrl}/${id}`)
      catchError(this.handleError)
    );
  }

  private handleError(error: any): Observable<never> {
    console.error('An error occurred:', error);
    return throwError(() => new Error('Something went wro
```

```
  }
}
```

```
// user.service.spec.ts
import { TestBed } from '@angular/core/testing';
import { HttpClientTestingModule, HttpTestingController }
import { UserService, User } from './user.service';
import { environment } from '../environment/environment';

describe('UserService', () => {
  let service: UserService;
  let httpMock: HttpTestingController;
  const apiUrl = `${environment.apiUrl}/users`;

  const mockUsers: User[] = [
    { id: 1, name: 'John Doe', email: 'john@example.com'
    { id: 2, name: 'Jane Smith', email: 'jane@example.com
  ];

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [UserService]
    });

    service = TestBed.inject(UserService);
    httpMock = TestBed.inject(HttpTestingController);
  });

  afterEach(() => {
    httpMock.verify(); // Verify no outstanding HTTP requ
  });

  it('should be created', () => {
```

```
    expect(service).toBeTruthy();
  });

  describe('getUsers', () => {
    it('should return an array of users', () => {
      // Arrange & Act
      service.getUsers().subscribe(users => {
        // Assert
        expect(users).toEqual(mockUsers);
        expect(users.length).toBe(2);
      });

      const req = httpMock.expectOne(apiUrl);
      expect(req.request.method).toBe('GET');
      req.flush(mockUsers);
    });

    it('should handle error when request fails', () => {
      // Arrange
      const errorMessage = 'Network error';

      // Act
      service.getUsers().subscribe({
        next: () => fail('should have failed'),
        error: (error) => {
          // Assert
          expect(error.message).toBe('Something went wron
        }
      });

      const req = httpMock.expectOne(apiUrl);
      req.error(new ProgressEvent('Network error'));
    });
  });

  describe('getUserById', () => {
```

```
    it('should return a single user', () => {
      // Arrange
      const mockUser = mockUsers[0];

      // Act
      service.getUserById(1).subscribe(user => {
        // Assert
        expect(user).toEqual(mockUser);
      });

      const req = httpMock.expectOne(`${apiUrl}/1`);
      expect(req.request.method).toBe('GET');
      req.flush(mockUser);
    });
  });

  describe('createUser', () => {
    it('should create a new user', () => {
      // Arrange
      const newUser = { name: 'Bob Wilson', email: 'bob@e
      const createdUser = { id: 3, ...newUser };

      // Act
      service.createUser(newUser).subscribe(user => {
        // Assert
        expect(user).toEqual(createdUser);
      });

      const req = httpMock.expectOne(apiUrl);
      expect(req.request.method).toBe('POST');
      expect(req.request.body).toEqual(newUser);
      req.flush(createdUser);
    });
  });

  describe('updateUser', () => {
```

```
    it('should update an existing user', () => {
      // Arrange
      const updates = { name: 'John Updated' };
      const updatedUser = { ...mockUsers[0], ...updates };

      // Act
      service.updateUser(1, updates).subscribe(user => {
        // Assert
        expect(user).toEqual(updatedUser);
      });

      const req = httpMock.expectOne(`${apiUrl}/1`);
      expect(req.request.method).toBe('PUT');
      expect(req.request.body).toEqual(updates);
      req.flush(updatedUser);
    });
  });

  describe('deleteUser', () => {
    it('should delete a user', () => {
      // Act
      service.deleteUser(1).subscribe(response => {
        // Assert
        expect(response).toBeUndefined();
      });

      const req = httpMock.expectOne(`${apiUrl}/1`);
      expect(req.request.method).toBe('DELETE');
      req.flush(null);
    });
  });
});
```

See Also:

- Testing HTTP Requests - Mocking HTTP calls in services
- Mocking and Spies - Creating service mocks
- Testing Components - Testing components that use services

↑ Back to Top

Testing Directives

Attribute Directive

```
// highlight.directive.ts
import { Directive, ElementRef, input, effect } from '@an

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  private el = inject(ElementRef);

  appHighlight = input<string>('yellow');

  constructor() {
    effect(() => {
      this.el.nativeElement.style.backgroundColor = this.
    });
  }
}
```

```
// highlight.directive.spec.ts
import { Component } from '@angular/core';
```

```
import { ComponentFixture, TestBed } from '@angular/core/
import { HighlightDirective } from './highlight.directive

@Component({
  imports: [HighlightDirective],
  template: `
    <p appHighlight>Default highlight</p>
    <p [appHighlight]="'lightblue'">Custom highlight</p>
  `
})
class TestComponent {}

describe('HighlightDirective', () => {
  let fixture: ComponentFixture<TestComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [TestComponent]
    }).compileComponents();

    fixture = TestBed.createComponent(TestComponent);
    fixture.detectChanges();
  });

  it('should apply default yellow background', () => {
    const p = fixture.nativeElement.querySelector('p:firs
    expect(p.style.backgroundColor).toBe('yellow');
  });

  it('should apply custom background color', () => {
    const p = fixture.nativeElement.querySelector('p:last
    expect(p.style.backgroundColor).toBe('lightblue');
  });
});
```

Structural Directive

```
// unless.directive.ts
import { Directive, TemplateRef, ViewContainerRef, input,

@Directive({
  selector: '[appUnless]'
})
export class UnlessDirective {
  private templateRef = inject(TemplateRef<any>);
  private viewContainer = inject(ViewContainerRef);

  appUnless = input<boolean>(false);

  constructor() {
    effect(() => {
      if (this.appUnless()) {
        this.viewContainer.clear();
      } else {
        this.viewContainer.createEmbeddedView(this.templa
      }
    });
  }
}
```

```
// unless.directive.spec.ts
import { Component } from '@angular/core';
import { ComponentFixture, TestBed } from '@angular/core/
import { UnlessDirective } from './unless.directive';

@Component({
  imports: [UnlessDirective],
  template: `
```

```
    <div *appUnless="condition">Content</div>
  `
})
class TestComponent {
  condition = false;
}

describe('UnlessDirective', () => {
  let fixture: ComponentFixture<TestComponent>;
  let component: TestComponent;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [TestComponent]
    }).compileComponents();

    fixture = TestBed.createComponent(TestComponent);
    component = fixture.componentInstance;
  });

  it('should display content when condition is false', ()
    component.condition = false;
    fixture.detectChanges();

    const div = fixture.nativeElement.querySelector('div'
    expect(div).toBeTruthy();
    expect(div.textContent).toContain('Content');
  });

  it('should hide content when condition is true', () =>
    component.condition = true;
    fixture.detectChanges();

    const div = fixture.nativeElement.querySelector('div'
    expect(div).toBeNull();
```

```
  });
});
```

↑ Back to Top

Testing Pipes

Pure Pipe

```
// truncate.pipe.ts
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'truncate'
})
export class TruncatePipe implements PipeTransform {
  transform(value: string, limit: number = 50, ellipsis:
    if (!value) {
      return '';
    }

    if (value.length <= limit) {
      return value;
    }

    return value.substring(0, limit) + ellipsis;
  }
}
```

```
// truncate.pipe.spec.ts
import { TruncatePipe } from './truncate.pipe';
```

```
describe('TruncatePipe', () => {
  let pipe: TruncatePipe;

  beforeEach(() => {
    pipe = new TruncatePipe();
  });

  it('should create an instance', () => {
    expect(pipe).toBeTruthy();
  });

  it('should return empty string for null/undefined', () =
    expect(pipe.transform(null as any)).toBe('');
    expect(pipe.transform(undefined as any)).toBe('');
  });

  it('should return original string if shorter than limit
    const text = 'Short text';
    expect(pipe.transform(text, 50)).toBe(text);
  });

  it('should truncate long text with default ellipsis', (
    const text = 'This is a very long text that should be
    const result = pipe.transform(text, 20);
    expect(result).toBe('This is a very long ...');
  });

  it('should truncate with custom ellipsis', () => {
    const text = 'This is a very long text that should be
    const result = pipe.transform(text, 20, '---');
    expect(result).toBe('This is a very long ---');
  });

  it('should use default limit of 50', () => {
    const text = 'a'.repeat(60);
```

```
    const result = pipe.transform(text);
    expect(result.length).toBe(53); // 50 + '...'
  });
});
```

Impure Pipe (Testing in Component)

```
// filter.pipe.ts
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'filter',
  pure: false // Impure pipe
})
export class FilterPipe implements PipeTransform {
  transform<T>(items: T[], callback: (item: T) => boolean
    if (!items || !callback) {
      return items;
    }
    return items.filter(callback);
  }
}
```

```
// filter.pipe.spec.ts
import { Component } from '@angular/core';
import { ComponentFixture, TestBed } from '@angular/core/
import { FilterPipe } from './filter.pipe';

interface Item {
  id: number;
  name: string;
  active: boolean;
```

```
}

@Component({
  imports: [FilterPipe],
  template: `
    @for (item of items | filter:filterActive; track item
      <div>{{ item.name }}</div>
    }
  `
})
class TestComponent {
  items: Item[] = [
    { id: 1, name: 'Item 1', active: true },
    { id: 2, name: 'Item 2', active: false },
    { id: 3, name: 'Item 3', active: true }
  ];

  filterActive = (item: Item) => item.active;
}

describe('FilterPipe', () => {
  let fixture: ComponentFixture<TestComponent>;
  let component: TestComponent;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [TestComponent]
    }).compileComponents();

    fixture = TestBed.createComponent(TestComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should filter active items', () => {
    const divs = fixture.nativeElement.querySelectorAll('
```

```
    expect(divs.length).toBe(2);
    expect(divs[0].textContent).toContain('Item 1');
    expect(divs[1].textContent).toContain('Item 3');
  });

  it('should update when array changes', () => {
    component.items.push({ id: 4, name: 'Item 4', active:
    fixture.detectChanges();

    const divs = fixture.nativeElement.querySelectorAll('
    expect(divs.length).toBe(3);
  });
});
```

↑ Back to Top

Testing with Signals

Component with Signals

```
// counter.component.ts
import { Component, signal, computed } from '@angular/cor

@Component({
  selector: 'app-counter',
  template: `
    <div class="counter">
      <p>Count: {{ count() }}</p>
      <p>Double: {{ doubled() }}</p>
      <p>Status: {{ status() }}</p>
      <button (click)="increment()">Increment</button>
      <button (click)="decrement()">Decrement</button>
```

```
      <button (click)="reset()">Reset</button>
    </div>
  `
})
export class CounterComponent {
  count = signal(0);
  doubled = computed(() => this.count() * 2);
  status = computed(() => this.count() > 0 ? 'positive' :

  increment(): void {
    this.count.update(c => c + 1);
  }

  decrement(): void {
    this.count.update(c => c - 1);
  }

  reset(): void {
    this.count.set(0);
  }
}
```

```
// counter.component.spec.ts
import { ComponentFixture, TestBed } from '@angular/core/
import { CounterComponent } from './counter.component';

describe('CounterComponent', () => {
  let component: CounterComponent;
  let fixture: ComponentFixture<CounterComponent>;
  let compiled: HTMLElement;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [CounterComponent]
```

```
    }).compileComponents();

    fixture = TestBed.createComponent(CounterComponent);
    component = fixture.componentInstance;
    compiled = fixture.nativeElement;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('should initialize with count of 0', () => {
    expect(component.count()).toBe(0);
    expect(component.doubled()).toBe(0);
    expect(component.status()).toBe('zero');
  });

  it('should increment count', () => {
    component.increment();
    expect(component.count()).toBe(1);
  });

  it('should decrement count', () => {
    component.decrement();
    expect(component.count()).toBe(-1);
  });

  it('should reset count', () => {
    component.count.set(10);
    component.reset();
    expect(component.count()).toBe(0);
  });

  it('should compute doubled value', () => {
    component.count.set(5);
```

```
    expect(component.doubled()).toBe(10);
  });

  it('should compute status correctly', () => {
    component.count.set(5);
    expect(component.status()).toBe('positive');

    component.count.set(-5);
    expect(component.status()).toBe('negative');

    component.count.set(0);
    expect(component.status()).toBe('zero');
  });

  it('should update view when count changes', () => {
    component.increment();
    fixture.detectChanges();

    const countP = compiled.querySelectorAll('p')[0];
    const doubledP = compiled.querySelectorAll('p')[1];
    const statusP = compiled.querySelectorAll('p')[2];

    expect(countP.textContent).toContain('Count: 1');
    expect(doubledP.textContent).toContain('Double: 2');
    expect(statusP.textContent).toContain('Status: positi
  });

  it('should respond to button clicks', () => {
    const buttons = compiled.querySelectorAll('button');
    const incrementBtn = buttons[0];
    const decrementBtn = buttons[1];
    const resetBtn = buttons[2];

    incrementBtn.click();
    fixture.detectChanges();
    expect(component.count()).toBe(1);
```

```
    incrementBtn.click();
    fixture.detectChanges();
    expect(component.count()).toBe(2);

    decrementBtn.click();
    fixture.detectChanges();
    expect(component.count()).toBe(1);

    resetBtn.click();
    fixture.detectChanges();
    expect(component.count()).toBe(0);
  });
});
```

Service with Signals

```
// todo.service.ts
import { Injectable, signal, computed } from '@angular/co

export interface Todo {
  id: number;
  title: string;
  completed: boolean;
}

@Injectable({
  providedIn: 'root'
})
export class TodoService {
  private todos = signal<Todo[]>([]);

  allTodos = this.todos.asReadonly();
  completedTodos = computed(() => this.todos().filter(t =
```

```
  activeTodos = computed(() => this.todos().filter(t => !
  completedCount = computed(() => this.completedTodos().l
  activeCount = computed(() => this.activeTodos().length)

  addTodo(title: string): void {
    const newTodo: Todo = {
      id: Date.now(),
      title,
      completed: false
    };
    this.todos.update(todos => [...todos, newTodo]);
  }

  toggleTodo(id: number): void {
    this.todos.update(todos =>
      todos.map(todo =>
        todo.id === id ? { ...todo, completed: !todo.comp
      )
    );
  }

  removeTodo(id: number): void {
    this.todos.update(todos => todos.filter(todo => todo.
  }

  clearCompleted(): void {
    this.todos.update(todos => todos.filter(todo => !todo
  }
}
```

```
// todo.service.spec.ts
import { TestBed } from '@angular/core/testing';
import { TodoService } from './todo.service';
```

```
describe('TodoService', () => {
  let service: TodoService;

  beforeEach(() => {
    TestBed.configureTestingModule({});
    service = TestBed.inject(TodoService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });

  it('should initialize with empty todos', () => {
    expect(service.allTodos()).toEqual([]);
    expect(service.completedCount()).toBe(0);
    expect(service.activeCount()).toBe(0);
  });

  it('should add a todo', () => {
    service.addTodo('Test Todo');
    expect(service.allTodos().length).toBe(1);
    expect(service.allTodos()[0].title).toBe('Test Todo')
    expect(service.allTodos()[0].completed).toBe(false);
  });

  it('should toggle todo completion', () => {
    service.addTodo('Test Todo');
    const todoId = service.allTodos()[0].id;

    service.toggleTodo(todoId);
    expect(service.allTodos()[0].completed).toBe(true);

    service.toggleTodo(todoId);
    expect(service.allTodos()[0].completed).toBe(false);
  });
```

```
  it('should remove a todo', () => {
    service.addTodo('Test Todo');
    const todoId = service.allTodos()[0].id;

    service.removeTodo(todoId);
    expect(service.allTodos().length).toBe(0);
  });

  it('should compute completed todos correctly', () => {
    service.addTodo('Todo 1');
    service.addTodo('Todo 2');
    service.addTodo('Todo 3');

    const todoId = service.allTodos()[0].id;
    service.toggleTodo(todoId);

    expect(service.completedTodos().length).toBe(1);
    expect(service.completedCount()).toBe(1);
    expect(service.activeTodos().length).toBe(2);
    expect(service.activeCount()).toBe(2);
  });

  it('should clear completed todos', () => {
    service.addTodo('Todo 1');
    service.addTodo('Todo 2');
    service.addTodo('Todo 3');

    service.toggleTodo(service.allTodos()[0].id);
    service.toggleTodo(service.allTodos()[1].id);

    service.clearCompleted();

    expect(service.allTodos().length).toBe(1);
    expect(service.allTodos()[0].title).toBe('Todo 3');
```

```
  });
});
```

↑ Back to Top

Asynchronous Testing

Testing with `fakeAsync` and `tick`

```
import { Component } from '@angular/core';
import { ComponentFixture, TestBed, fakeAsync, tick } fro

@Component({
  selector: 'app-delayed-message',
  template: `
    <button (click)="showMessage()">Show Message</button>
    @if (message) {
      <p>{{ message }}</p>
    }
  `
})
class DelayedMessageComponent {
  message = '';

  showMessage(): void {
    setTimeout(() => {
      this.message = 'Hello after delay!';
    }, 1000);
  }
}

describe('DelayedMessageComponent with fakeAsync', () =>
```

```
  let component: DelayedMessageComponent;
  let fixture: ComponentFixture<DelayedMessageComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [DelayedMessageComponent]
    }).compileComponents();

    fixture = TestBed.createComponent(DelayedMessageCompo
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should display message after delay', fakeAsync(() =
    const button = fixture.nativeElement.querySelector('b
    button.click();

    // Message should not be displayed yet
    expect(component.message).toBe('');

    // Advance time by 1000ms
    tick(1000);
    fixture.detectChanges();

    // Now message should be displayed
    expect(component.message).toBe('Hello after delay!');
    const p = fixture.nativeElement.querySelector('p');
    expect(p.textContent).toContain('Hello after delay!')
  }));

  it('should not display message before delay', fakeAsync
    component.showMessage();

    // Advance time by only 500ms
    tick(500);
```

```
    expect(component.message).toBe('');
  }));
});
```

Testing with `waitForAsync`

```
import { Component, OnInit, signal } from '@angular/core'
import { ComponentFixture, TestBed, waitForAsync } from '

@Component({
  selector: 'app-async-data',
  template: `
    <div>
      @if (data()) {
        <p>{{ data() }}</p>
      }
    </div>
  `
})
class AsyncDataComponent implements OnInit {
  data = signal<string>('');

  ngOnInit(): void {
    this.loadData();
  }

  async loadData(): Promise<void> {
    const result = await this.fetchData();
    this.data.set(result);
  }

  private async fetchData(): Promise<string> {
    return new Promise((resolve) => {
      setTimeout(() => resolve('Async data loaded'), 100)
```

```
    });
  }
}

describe('AsyncDataComponent with waitForAsync', () => {
  let component: AsyncDataComponent;
  let fixture: ComponentFixture<AsyncDataComponent>;

  beforeEach(waitForAsync(() => {
    TestBed.configureTestingModule({
      imports: [AsyncDataComponent]
    }).compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(AsyncDataComponent)
    component = fixture.componentInstance;
  });

  it('should load data asynchronously', waitForAsync(() =>
    fixture.detectChanges();

    fixture.whenStable().then(() => {
      fixture.detectChanges();
      expect(component.data()).toBe('Async data loaded');

      const p = fixture.nativeElement.querySelector('p');
      expect(p.textContent).toContain('Async data loaded'
    });
  }));
});
```

Testing Observables with `done`

```
import { Injectable } from '@angular/core';
import { Observable, of, delay } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
class DataService {
  getData(): Observable<string> {
    return of('Observable data').pipe(delay(100));
  }
}

describe('DataService with done callback', () => {
  let service: DataService;

  beforeEach(() => {
    service = new DataService();
  });

  it('should return data from observable', (done: DoneFn)
    service.getData().subscribe(data => {
      expect(data).toBe('Observable data');
      done();
    });
  });
});
```

Testing with `flush` and `flushMicrotasks`

```
import { Component } from '@angular/core';
import { ComponentFixture, TestBed, fakeAsync, flush, flu

@Component({
```

```
  selector: 'app-promise-test',
  template: '<p>{{ result }}</p>'
})
class PromiseTestComponent {
  result = '';

  executePromise(): void {
    Promise.resolve('Promise result').then(value => {
      this.result = value;
    });
  }

  executeTimeout(): void {
    setTimeout(() => {
      this.result = 'Timeout result';
    }, 5000);
  }
}

describe('PromiseTestComponent', () => {
  let component: PromiseTestComponent;
  let fixture: ComponentFixture<PromiseTestComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [PromiseTestComponent]
    }).compileComponents();

    fixture = TestBed.createComponent(PromiseTestComponen
    component = fixture.componentInstance;
  });

  it('should handle promises with flushMicrotasks', fakeA
    component.executePromise();
    expect(component.result).toBe('');
```

```
    flushMicrotasks(); // Flush all pending microtasks (p
    expect(component.result).toBe('Promise result');
  }));

  it('should handle timeouts with flush', fakeAsync(() =>
    component.executeTimeout();
    expect(component.result).toBe('');

    flush(); // Flush all pending macrotasks (timeouts)
    expect(component.result).toBe('Timeout result');
  }));
});
```

See Also:

- Testing HTTP Requests - Async HTTP testing with HttpTestingController
- Testing Observables - Working with RxJS streams
- Troubleshooting - Debugging async test issues

↑ Back to Top

Mocking and Spies

Jasmine Spies

```
describe('Jasmine Spies', () => {
  it('should spy on a method', () => {
    const obj = {
      getValue: () => 'original value'
    };

    spyOn(obj, 'getValue').and.returnValue('mocked value'
```

```
  expect(obj.getValue()).toBe('mocked value');
  expect(obj.getValue).toHaveBeenCalled();
});

it('should spy and call through', () => {
  const obj = {
    getValue: () => 'original value'
  };

  spyOn(obj, 'getValue').and.callThrough();

  expect(obj.getValue()).toBe('original value');
  expect(obj.getValue).toHaveBeenCalled();
});

it('should spy and call fake', () => {
  const obj = {
    getValue: () => 'original value'
  };

  spyOn(obj, 'getValue').and.callFake(() => 'fake value

  expect(obj.getValue()).toBe('fake value');
});

it('should spy and throw error', () => {
  const obj = {
    getValue: () => 'original value'
  };

  spyOn(obj, 'getValue').and.throwError('Error occurred

  expect(() => obj.getValue()).toThrowError('Error occu
});
```

```
  it('should track spy calls', () => {
    const obj = {
      add: (a: number, b: number) => a + b
    };

    spyOn(obj, 'add');

    obj.add(1, 2);
    obj.add(3, 4);

    expect(obj.add).toHaveBeenCalledTimes(2);
    expect(obj.add).toHaveBeenCalledWith(1, 2);
    expect(obj.add).toHaveBeenCalledWith(3, 4);
  });
});
```

Creating Spy Objects

```
interface UserRepository {
  getUser(id: number): Observable<User>;
  createUser(user: User): Observable<User>;
  deleteUser(id: number): Observable<void>;
}

describe('Spy Objects', () => {
  it('should create a spy object', () => {
    const userRepoSpy = jasmine.createSpyObj<UserReposito
      'UserRepository',
      ['getUser', 'createUser', 'deleteUser']
    );

    const mockUser = { id: 1, name: 'John', email: 'john@
    userRepoSpy.getUser.and.returnValue(of(mockUser));
```

```
    userRepoSpy.getUser(1).subscribe(user => {
      expect(user).toEqual(mockUser);
    });

    expect(userRepoSpy.getUser).toHaveBeenCalledWith(1);
  });
});
```

Mocking Services

```
// auth.service.ts
@Injectable({
  providedIn: 'root'
})
export class AuthService {
  private http = inject(HttpClient);

  login(credentials: { email: string; password: string })
    return this.http.post<{ token: string }>('/api/login'
  }

  isAuthenticated(): boolean {
    return !!localStorage.getItem('token');
  }
}

// Component using AuthService
@Component({
  selector: 'app-login',
  template: `
    <form (ngSubmit)="onSubmit()">
      <input [(ngModel)]="email" type="email" />
      <input [(ngModel)]="password" type="password" />
      <button type="submit">Login</button>
```

```
    </form>
    @if (error) {
      <p class="error">{{ error }}</p>
    }
  `
})
export class LoginComponent {
  private authService = inject(AuthService);

  email = '';
  password = '';
  error = '';

  onSubmit(): void {
    this.authService.login({ email: this.email, password:
      .subscribe({
        next: (response) => {
          localStorage.setItem('token', response.token);
        },
        error: () => {
          this.error = 'Login failed';
        }
      });
  }
}

// login.component.spec.ts
describe('LoginComponent with Mocked Service', () => {
  let component: LoginComponent;
  let fixture: ComponentFixture<LoginComponent>;
  let authService: jasmine.SpyObj<AuthService>;

  beforeEach(async () => {
    const authServiceSpy = jasmine.createSpyObj('AuthServ

    await TestBed.configureTestingModule({
```

```
      imports: [LoginComponent, FormsModule],
      providers: [
        { provide: AuthService, useValue: authServiceSpy
      ]
    }).compileComponents();

    fixture = TestBed.createComponent(LoginComponent);
    component = fixture.componentInstance;
    authService = TestBed.inject(AuthService) as jasmine.
  });

  it('should login successfully', () => {
    const mockResponse = { token: 'fake-token' };
    authService.login.and.returnValue(of(mockResponse));

    spyOn(localStorage, 'setItem');

    component.email = 'test@example.com';
    component.password = 'password123';
    component.onSubmit();

    expect(authService.login).toHaveBeenCalledWith({
      email: 'test@example.com',
      password: 'password123'
    });
    expect(localStorage.setItem).toHaveBeenCalledWith('to
  });

  it('should handle login error', () => {
    authService.login.and.returnValue(throwError(() => ne

    component.onSubmit();

    expect(component.error).toBe('Login failed');
```

```
  });
});
```

↑ Back to Top

Testing Forms

Reactive Forms

```
// user-form.component.ts
import { Component, OnInit, inject, output } from '@angul
import { FormBuilder, FormGroup, Validators, ReactiveForm
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-user-form',
  imports: [CommonModule, ReactiveFormsModule],
  template: `
    <form [formGroup]="userForm" (ngSubmit)="onSubmit()">
      <div>
        <label for="name">Name</label>
        <input id="name" formControlName="name" />
        @if (userForm.get('name')?.invalid && userForm.ge
          <span class="error">Name is required</span>
        }
      </div>

      <div>
        <label for="email">Email</label>
        <input id="email" type="email" formControlName="e
        @if (userForm.get('email')?.invalid && userForm.g
          <span class="error">Valid email is required</sp
```

```
          }
        </div>

        <div>
          <label for="age">Age</label>
          <input id="age" type="number" formControlName="ag
          @if (userForm.get('age')?.errors?.['min']) {
            <span class="error">Age must be at least 18</sp
          }
        </div>

        <button type="submit" [disabled]="userForm.invalid"
      </form>
    `
})
export class UserFormComponent implements OnInit {
  private fb = inject(FormBuilder);

  userForm!: FormGroup;
  formSubmitted = output<any>();

  ngOnInit(): void {
    this.userForm = this.fb.group({
      name: ['', Validators.required],
      email: ['', [Validators.required, Validators.email]
      age: ['', [Validators.required, Validators.min(18)]
    });
  }

  onSubmit(): void {
    if (this.userForm.valid) {
      this.formSubmitted.emit(this.userForm.value);
    }
  }
}
```

```
// user-form.component.spec.ts
import { ComponentFixture, TestBed } from '@angular/core/
import { ReactiveFormsModule } from '@angular/forms';
import { UserFormComponent } from './user-form.component'

describe('UserFormComponent', () => {
  let component: UserFormComponent;
  let fixture: ComponentFixture<UserFormComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [UserFormComponent]
    }).compileComponents();

    fixture = TestBed.createComponent(UserFormComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create form with 3 controls', () => {
    expect(component.userForm.contains('name')).toBeTruth
    expect(component.userForm.contains('email')).toBeTrut
    expect(component.userForm.contains('age')).toBeTruthy
  });

  it('should make name control required', () => {
    const control = component.userForm.get('name');
    control?.setValue('');
    expect(control?.valid).toBeFalsy();
    expect(control?.hasError('required')).toBeTruthy();
  });

  it('should validate email format', () => {
    const control = component.userForm.get('email');
```

```
    control?.setValue('invalid-email');
    expect(control?.valid).toBeFalsy();
    expect(control?.hasError('email')).toBeTruthy();

    control?.setValue('valid@email.com');
    expect(control?.valid).toBeTruthy();
  });

  it('should validate minimum age', () => {
    const control = component.userForm.get('age');

    control?.setValue(15);
    expect(control?.valid).toBeFalsy();
    expect(control?.hasError('min')).toBeTruthy();

    control?.setValue(18);
    expect(control?.valid).toBeTruthy();
  });

  it('should disable submit button when form is invalid',
    const button = fixture.nativeElement.querySelector('b
    expect(button.disabled).toBeTruthy();
  });

  it('should enable submit button when form is valid', ()
    component.userForm.patchValue({
      name: 'John Doe',
      email: 'john@example.com',
      age: 25
    });
    fixture.detectChanges();

    const button = fixture.nativeElement.querySelector('b
    expect(button.disabled).toBeFalsy();
  });
```

```
  it('should display validation errors', () => {
    const nameControl = component.userForm.get('name');
    nameControl?.markAsTouched();
    fixture.detectChanges();

    const errorSpan = fixture.nativeElement.querySelector
    expect(errorSpan?.textContent).toContain('Name is req
  });

  it('should emit form data on valid submission', () => {
    const formData = {
      name: 'John Doe',
      email: 'john@example.com',
      age: 25
    };

    let emittedData: any;
    component.formSubmitted.subscribe(data => {
      emittedData = data;
    });

    component.userForm.patchValue(formData);
    component.onSubmit();

    expect(emittedData).toEqual(formData);
  });

  it('should not emit on invalid submission', () => {
    let emittedData: any;
    component.formSubmitted.subscribe(data => {
      emittedData = data;
    });

    component.onSubmit(); // Form is invalid

    expect(emittedData).toBeUndefined();
```

```
  });
});
```

Custom Validators

```
// password-validator.ts
import { AbstractControl, ValidationErrors, ValidatorFn }

export function passwordStrengthValidator(): ValidatorFn
  return (control: AbstractControl): ValidationErrors | n
    const value = control.value;

    if (!value) {
      return null;
    }

    const hasUpperCase = /[A-Z]/.test(value);
    const hasLowerCase = /[a-z]/.test(value);
    const hasNumeric = /[0-9]/.test(value);
    const hasSpecialChar = /[!@#$%^&*(),.?":{}|<>]/.test(
    const isLengthValid = value.length >= 8;

    const passwordValid = hasUpperCase && hasLowerCase &&

    return passwordValid ? null : {
      passwordStrength: {
        hasUpperCase,
        hasLowerCase,
        hasNumeric,
        hasSpecialChar,
        isLengthValid
      }
    };
```

```
  };
}
```

```
// password-validator.spec.ts
import { FormControl } from '@angular/forms';
import { passwordStrengthValidator } from './password-val

describe('passwordStrengthValidator', () => {
  it('should return null for valid password', () => {
    const control = new FormControl('StrongP@ss1');
    const validator = passwordStrengthValidator();
    expect(validator(control)).toBeNull();
  });

  it('should return error for password without uppercase'
    const control = new FormControl('weakp@ss1');
    const validator = passwordStrengthValidator();
    const result = validator(control);
    expect(result?.['passwordStrength'].hasUpperCase).toB
  });

  it('should return error for password without lowercase'
    const control = new FormControl('WEAKP@SS1');
    const validator = passwordStrengthValidator();
    const result = validator(control);
    expect(result?.['passwordStrength'].hasLowerCase).toB
  });

  it('should return error for password without numbers',
    const control = new FormControl('WeakP@ss');
    const validator = passwordStrengthValidator();
    const result = validator(control);
    expect(result?.['passwordStrength'].hasNumeric).toBeF
  });
```

```
  it('should return error for password without special cha
    const control = new FormControl('WeakPass1');
    const validator = passwordStrengthValidator();
    const result = validator(control);
    expect(result?.['passwordStrength'].hasSpecialChar).to
  });

  it('should return error for short password', () => {
    const control = new FormControl('Wp@1');
    const validator = passwordStrengthValidator();
    const result = validator(control);
    expect(result?.['passwordStrength'].isLengthValid).to
  });

  it('should return null for empty value', () => {
    const control = new FormControl('');
    const validator = passwordStrengthValidator();
    expect(validator(control)).toBeNull();
  });
});
```

See Also:

- Testing Components - Testing form components
- Custom Validators - Writing and testing validators
- Common Patterns - Reusable form testing patterns

↑ Back to Top

Testing HTTP Requests

HttpClient Testing

```
// user.service.ts (already shown above)

// user.service.spec.ts (comprehensive version)
import { TestBed } from '@angular/core/testing';
import { HttpClientTestingModule, HttpTestingController }
import { UserService, User } from './user.service';

describe('UserService HTTP Testing', () => {
  let service: UserService;
  let httpMock: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [UserService]
    });

    service = TestBed.inject(UserService);
    httpMock = TestBed.inject(HttpTestingController);
  });

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

  it('should handle multiple simultaneous requests', () =
    const mockUsers: User[] = [
      { id: 1, name: 'User 1', email: 'user1@example.com'
    ];
    const mockUser: User = { id: 2, name: 'User 2', email

    service.getUsers().subscribe();
    service.getUserById(2).subscribe();
```

```
    const requests = httpMock.match(req => req.url.includ
    expect(requests.length).toBe(2);

    requests[0].flush(mockUsers);
    requests[1].flush(mockUser);
  });

  it('should handle HTTP error responses', () => {
    service.getUsers().subscribe({
      next: () => fail('should have failed'),
      error: (error) => {
        expect(error.message).toBe('Something went wrong'
      }
    });

    const req = httpMock.expectOne(`${environment.apiUrl}
    req.flush('Error occurred', {
      status: 500,
      statusText: 'Internal Server Error'
    });
  });

  it('should set correct headers', () => {
    service.getUsers().subscribe();

    const req = httpMock.expectOne(`${environment.apiUrl}
    // You can check headers if your service sets them
    // expect(req.request.headers.get('Authorization')).t
    req.flush([]);
  });

  it('should handle network errors', () => {
    service.getUsers().subscribe({
      next: () => fail('should have failed'),
      error: (error) => {
```

```
        expect(error.message).toBe('Something went wrong'
      }
    });

    const req = httpMock.expectOne(`${environment.apiUrl}
    req.error(new ProgressEvent('Network error'), {
      status: 0,
      statusText: 'Unknown Error'
    });
  });
});
```

↑ Back to Top

Testing Routing

Router Testing

```
// app.routes.ts
import { Routes } from '@angular/router';
import { HomeComponent } from './home.component';
import { AboutComponent } from './about.component';
import { UserDetailComponent } from './user-detail.compon

export const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'users/:id', component: UserDetailComponent }
];
```

```
// navigation.component.ts
import { Component, inject } from '@angular/core';
import { Router, RouterLink } from '@angular/router';

@Component({
  selector: 'app-navigation',
  imports: [RouterLink],
  template: `
    <nav>
      <a routerLink="/">Home</a>
      <a routerLink="/about">About</a>
      <button (click)="goToUser(1)">User 1</button>
    </nav>
  `
})
export class NavigationComponent {
  private router = inject(Router);

  goToUser(id: number): void {
    this.router.navigate(['/users', id]);
  }
}
```

```
// navigation.component.spec.ts
import { ComponentFixture, TestBed } from '@angular/core/
import { Router } from '@angular/router';
import { NavigationComponent } from './navigation.compone
import { provideRouter } from '@angular/router';
import { routes } from './app.routes';

describe('NavigationComponent', () => {
  let component: NavigationComponent;
  let fixture: ComponentFixture<NavigationComponent>;
  let router: Router;
```

```
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [NavigationComponent],
      providers: [provideRouter(routes)]
    }).compileComponents();

    fixture = TestBed.createComponent(NavigationComponent
    component = fixture.componentInstance;
    router = TestBed.inject(Router);
    fixture.detectChanges();
  });

  it('should navigate to user detail', () => {
    spyOn(router, 'navigate');

    component.goToUser(1);

    expect(router.navigate).toHaveBeenCalledWith(['/users
  });

  it('should have correct router links', () => {
    const links = fixture.nativeElement.querySelectorAll(
    expect(links[0].getAttribute('href')).toBe('/');
    expect(links[1].getAttribute('href')).toBe('/about');
  });
});
```

Testing Route Parameters

```
// user-detail.component.ts
import { Component, OnInit, inject, signal } from '@angul
import { ActivatedRoute } from '@angular/router';
import { UserService } from './user.service';
```

```
import { User } from './user.model';

@Component({
  selector: 'app-user-detail',
  template: `
    @if (user()) {
      <div>
        <h2>{{ user()!.name }}</h2>
        <p>{{ user()!.email }}</p>
      </div>
    }
  `
})
export class UserDetailComponent implements OnInit {
  private route = inject(ActivatedRoute);
  private userService = inject(UserService);

  user = signal<User | null>(null);

  ngOnInit(): void {
    const id = Number(this.route.snapshot.paramMap.get('i
    this.userService.getUserById(id).subscribe(user => {
      this.user.set(user);
    });
  }
}
```

```
// user-detail.component.spec.ts
import { ComponentFixture, TestBed } from '@angular/core/
import { ActivatedRoute } from '@angular/router';
import { of } from 'rxjs';
import { UserDetailComponent } from './user-detail.compon
import { UserService } from './user.service';
import { User } from './user.model';
```

```
describe('UserDetailComponent', () => {
  let component: UserDetailComponent;
  let fixture: ComponentFixture<UserDetailComponent>;
  let userService: jasmine.SpyObj<UserService>;

  const mockUser: User = {
    id: 1,
    name: 'John Doe',
    email: 'john@example.com'
  };

  beforeEach(async () => {
    const userServiceSpy = jasmine.createSpyObj('UserServ

    await TestBed.configureTestingModule({
      imports: [UserDetailComponent],
      providers: [
        { provide: UserService, useValue: userServiceSpy
        {
          provide: ActivatedRoute,
          useValue: {
            snapshot: {
              paramMap: {
                get: (key: string) => '1'
              }
            }
          }
        }
      ]
    }).compileComponents();

    fixture = TestBed.createComponent(UserDetailComponent
    component = fixture.componentInstance;
    userService = TestBed.inject(UserService) as jasmine.
  });
```

```
  it('should load user based on route parameter', () => {
    userService.getUserById.and.returnValue(of(mockUser))

    fixture.detectChanges();

    expect(userService.getUserById).toHaveBeenCalledWith(
    expect(component.user()).toEqual(mockUser);
  });

  it('should display user information', () => {
    userService.getUserById.and.returnValue(of(mockUser))
    fixture.detectChanges();

    const h2 = fixture.nativeElement.querySelector('h2');
    const p = fixture.nativeElement.querySelector('p');

    expect(h2.textContent).toContain('John Doe');
    expect(p.textContent).toContain('john@example.com');
  });
});
```

↑ Back to Top

Best Practices

1. Follow AAA Pattern

```
it('should add two numbers', () => {
  // Arrange
  const a = 5;
  const b = 3;
```

```
  const calculator = new Calculator();

  // Act
  const result = calculator.add(a, b);

  // Assert
  expect(result).toBe(8);
});
```

2. Test One Thing at a Time

```
// Bad - testing multiple things
it('should handle user creation', () => {
  service.createUser(user);
  expect(service.users.length).toBe(1);
  expect(service.users[0].name).toBe('John');
  expect(service.lastCreatedId).toBe(1);
});

// Good - separate tests
it('should add user to list', () => {
  service.createUser(user);
  expect(service.users.length).toBe(1);
});

it('should set user name correctly', () => {
  service.createUser(user);
  expect(service.users[0].name).toBe('John');
});

it('should track last created ID', () => {
  service.createUser(user);
```

```
  expect(service.lastCreatedId).toBe(1);
});
```

3. Use Descriptive Test Names

```
// Bad
it('should work', () => {});

// Good
it('should calculate total price including tax', () => {}
it('should throw error when dividing by zero', () => {});
it('should display loading spinner while fetching data',
```

4. Keep Tests Independent

```
// Bad - tests depend on each other
describe('TodoService', () => {
  let service: TodoService;

  it('should add todo', () => {
    service.addTodo('Task 1');
    expect(service.todos.length).toBe(1);
  });

  it('should remove todo', () => {
    // This depends on previous test!
    service.removeTodo(0);
    expect(service.todos.length).toBe(0);
  });
});

// Good - each test is independent
```

```
describe('TodoService', () => {
  let service: TodoService;

  beforeEach(() => {
    service = new TodoService();
  });

  it('should add todo', () => {
    service.addTodo('Task 1');
    expect(service.todos.length).toBe(1);
  });

  it('should remove todo', () => {
    service.addTodo('Task 1');
    service.removeTodo(0);
    expect(service.todos.length).toBe(0);
  });
});
```

5. Don't Test Implementation Details

```
// Bad - testing private implementation
it('should call private method', () => {
  spyOn<any>(component, 'privateMethod');
  component.publicMethod();
  expect(component['privateMethod']).toHaveBeenCalled();
});

// Good - test public behavior
it('should update result when public method is called', (
  component.publicMethod();
  expect(component.result).toBe(expected);
});
```

6. Use Test Doubles Appropriately

```
// Use real objects when possible
it('should format date', () => {
  const formatter = new DateFormatter();
  expect(formatter.format(new Date('2024-01-01'))).toBe('
});

// Use mocks for external dependencies
it('should fetch data from API', () => {
  const httpMock = jasmine.createSpyObj('HttpClient', ['g
  const service = new DataService(httpMock);

  httpMock.get.and.returnValue(of({ data: 'test' }));
  service.getData().subscribe(result => {
    expect(result).toEqual({ data: 'test' });
  });
});
```

7. Test Edge Cases

```
describe('divide', () => {
  it('should divide positive numbers', () => {
    expect(divide(10, 2)).toBe(5);
  });

  it('should divide negative numbers', () => {
    expect(divide(-10, 2)).toBe(-5);
  });

  it('should handle division by zero', () => {
    expect(() => divide(10, 0)).toThrow();
  });
```

```
  it('should handle zero dividend', () => {
    expect(divide(0, 5)).toBe(0);
  });

  it('should handle decimal results', () => {
    expect(divide(5, 2)).toBe(2.5);
  });
});
```

8. Maintain Test Code Quality

```
// Extract common setup
describe('UserService', () => {
  let service: UserService;
  let httpMock: HttpTestingController;

  function setupTestBed(): void {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [UserService]
    });
    service = TestBed.inject(UserService);
    httpMock = TestBed.inject(HttpTestingController);
  }

  function createMockUser(id: number): User {
    return {
      id,
      name: `User ${id}`,
      email: `user${id}@example.com`
    };
  }
```

```
  beforeEach(() => {
    setupTestBed();
  });

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

  // Tests...
});
```

↑ Back to Top

Common Patterns

Testing Component Inputs and Outputs

```
it('should handle input changes', () => {
  fixture.componentRef.setInput('user', mockUser);
  fixture.detectChanges();
  expect(component.user()).toEqual(mockUser);

  const newUser = { ...mockUser, name: 'Jane' };
  fixture.componentRef.setInput('user', newUser);
  fixture.detectChanges();
  expect(component.user()).toEqual(newUser);
});

it('should emit output events', () => {
  let emittedValue: any;
  component.dataChanged.subscribe((value: any) => {
    emittedValue = value;
```

```
  });

  component.triggerChange('test');
  expect(emittedValue).toBe('test');
});
```

Testing with NgFor and NgIf

```
it('should render list of items', () => {
  component.items.set([
    { id: 1, name: 'Item 1' },
    { id: 2, name: 'Item 2' }
  ]);
  fixture.detectChanges();

  const listItems = fixture.nativeElement.querySelectorAl
  expect(listItems.length).toBe(2);
});

it('should show message when list is empty', () => {
  component.items.set([]);
  fixture.detectChanges();

  const message = fixture.nativeElement.querySelector('.e
  expect(message).toBeTruthy();
});
```

Testing Error Handling

```
it('should handle and display errors', () => {
  service.getData.and.returnValue(throwError(() => new Er
```

```
  component.loadData();
  fixture.detectChanges();

  expect(component.error()).toBe('Failed to load data');

  const errorElement = fixture.nativeElement.querySelecto
  expect(errorElement.textContent).toContain('Failed to l
});
```

Testing Loading States

```
it('should show loading indicator', fakeAsync(() => {
  service.getData.and.returnValue(of(data).pipe(delay(100

  component.loadData();
  expect(component.loading()).toBe(true);

  tick(1000);
  fixture.detectChanges();

  expect(component.loading()).toBe(false);
}));
```

↑ Back to Top

Advanced Techniques

Custom Test Harness

```
// user-profile.harness.ts
import { ComponentHarness } from '@angular/cdk/testing';

export class UserProfileHarness extends ComponentHarness
  static hostSelector = 'app-user-profile';

  private getNameElement = this.locatorFor('h2');
  private getEmailElement = this.locatorFor('p');
  private getDeleteButton = this.locatorFor('button');

  async getName(): Promise<string> {
    const element = await this.getNameElement();
    return (await element.text()).trim();
  }

  async getEmail(): Promise<string> {
    const element = await this.getEmailElement();
    return (await element.text()).trim();
  }

  async clickDelete(): Promise<void> {
    const button = await this.getDeleteButton();
    return button.click();
  }
}

// Usage in tests
it('should display user information using harness', async
  const harness = await loader.getHarness(UserProfileHarn

  expect(await harness.getName()).toBe('John Doe');
  expect(await harness.getEmail()).toBe('john@example.com

  await harness.clickDelete();
```

```
  expect(component.deleteClicked).toBe(true);
});
```

Testing with RxJS Operators

```
import { TestScheduler } from 'rxjs/testing';

describe('Observable Testing with TestScheduler', () => {
  let scheduler: TestScheduler;

  beforeEach(() => {
    scheduler = new TestScheduler((actual, expected) => {
      expect(actual).toEqual(expected);
    });
  });

  it('should map values', () => {
    scheduler.run(({ cold, expectObservable }) => {
      const source$ = cold('a-b-c|', { a: 1, b: 2, c: 3 }
      const expected = '     a-b-c|';
      const expectedValues = { a: 2, b: 4, c: 6 };

      const result$ = source$.pipe(map(x => x * 2));

      expectObservable(result$).toBe(expected, expectedVa
    });
  });

  it('should debounce values', () => {
    scheduler.run(({ cold, expectObservable }) => {
      const source$ = cold('a-b-c----|');
      const expected = '     ------c----|';

      const result$ = source$.pipe(debounceTime(30, sched
```

```
      expectObservable(result$).toBe(expected);
    });
  });
});
```

Page Object Pattern

```
// login.page.ts
export class LoginPage {
  constructor(private fixture: ComponentFixture<LoginComp

  get emailInput(): HTMLInputElement {
    return this.fixture.nativeElement.querySelector('#ema
  }

  get passwordInput(): HTMLInputElement {
    return this.fixture.nativeElement.querySelector('#pas
  }

  get submitButton(): HTMLButtonElement {
    return this.fixture.nativeElement.querySelector('butt
  }

  get errorMessage(): HTMLElement | null {
    return this.fixture.nativeElement.querySelector('.err
  }

  setEmail(email: string): void {
    this.emailInput.value = email;
    this.emailInput.dispatchEvent(new Event('input'));
  }

  setPassword(password: string): void {
```

```
    this.passwordInput.value = password;
    this.passwordInput.dispatchEvent(new Event('input'));
  }

  submit(): void {
    this.submitButton.click();
  }

  login(email: string, password: string): void {
    this.setEmail(email);
    this.setPassword(password);
    this.submit();
  }
}

// Usage in tests
describe('LoginComponent with Page Object', () => {
  let page: LoginPage;
  let fixture: ComponentFixture<LoginComponent>;

  beforeEach(() => {
    fixture = TestBed.createComponent(LoginComponent);
    page = new LoginPage(fixture);
    fixture.detectChanges();
  });

  it('should login successfully', () => {
    page.login('test@example.com', 'password123');
    fixture.detectChanges();

    expect(page.errorMessage).toBeNull();
  });
});
```

Testing Memory Leaks

```
it('should not have memory leaks', () => {
  const subscription = component.data$.subscribe();

  // Trigger component destruction
  fixture.destroy();

  expect(subscription.closed).toBe(true);
});

it('should clean up subscriptions on destroy', () => {
  spyOn(component['subscription'], 'unsubscribe');

  fixture.destroy();

  expect(component['subscription'].unsubscribe).toHaveBee
});
```

↑ Back to Top

Troubleshooting

Common Issues

1. "Cannot read property of undefined"

```
// Problem: Trying to access property before initializati
it('should work', () => {
  expect(component.user.name).toBe('John'); // Error if u
});
```

```
// Solution: Check for existence or initialize
it('should work', () => {
  component.user = mockUser;
  expect(component.user?.name).toBe('John');
});
```

2. "ExpressionChangedAfterItHasBeenCheckedError"

```
// Problem: Changing values during change detection
ngAfterViewInit() {
  this.value = 'changed'; // Causes error
}

// Solution: Use setTimeout or ChangeDetectorRef
ngAfterViewInit() {
  setTimeout(() => {
    this.value = 'changed';
  });
}
```

3. Tests Failing Randomly

```
// Problem: Asynchronous operations not completed
it('should load data', () => {
  component.loadData();
  expect(component.data).toBeDefined(); // May fail
});

// Solution: Use async testing utilities
it('should load data', fakeAsync(() => {
  component.loadData();
  tick();
```

```
  expect(component.data).toBeDefined();
}));
```

4. "No provider for Service"

```
// Problem: Service not provided in TestBed
TestBed.configureTestingModule({
  imports: [MyComponent]
  // Missing providers
});

// Solution: Add providers
TestBed.configureTestingModule({
  imports: [MyComponent],
  providers: [MyService]
});
```

5. HTTP Requests Not Mocked

```
// Problem: Real HTTP requests being made
it('should fetch data', () => {
  service.getData().subscribe(); // Makes real HTTP reque
});

// Solution: Use HttpClientTestingModule
beforeEach(() => {
  TestBed.configureTestingModule({
    imports: [HttpClientTestingModule]
  });
  httpMock = TestBed.inject(HttpTestingController);
});
```

Debugging Tests

Enable Detailed Error Messages

```
// karma.conf.js
client: {
  jasmine: {
    random: false,
    seed: 42,
    stopSpecOnExpectationFailure: true, // Stop on first
    stopOnSpecFailure: true
  }
}
```

Log Values for Debugging

```
it('should debug values', () => {
  console.log('Component state:', component);
  console.log('Service data:', service.data);

  // Use fail() to stop and see output
  // fail('Stopping here to debug');

  expect(component.value).toBe(expected);
});
```

Run Single Test

```
// Use fit() to run only this test
fit('should run only this test', () => {
  // Test code
```

```
});

// Use fdescribe() to run only this suite
fdescribe('Only this suite', () => {
  // Tests
});
```

↑ Back to Top

Coverage Reports

Generate Coverage

```
# Run tests with coverage
ng test --code-coverage --watch=false

# Open coverage report
open coverage/index.html
```

Configure Coverage Thresholds

```
// karma.conf.js
coverageReporter: {
  check: {
    global: {
      statements: 80,
      branches: 80,
      functions: 80,
      lines: 80
    },
```

```
    each: {
      statements: 70,
      branches: 70,
      functions: 70,
      lines: 70
    }
  }
}
```

Exclude Files from Coverage

```
// karma.conf.js
preprocessors: {
  'src/**/*.ts': ['coverage']
},
coverageReporter: {
  exclude: [
    'src/**/*.spec.ts',
    'src/**/test/**',
    'src/main.ts',
    'src/environments/**'
  ]
}
```

↑ Back to Top

Continuous Integration

GitHub Actions Example

```
# .github/workflows/test.yml
name: Run Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3

      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '20'

      - name: Install dependencies
        run: npm ci

      - name: Run tests
        run: npm run test:ci

      - name: Upload coverage
        uses: codecov/codecov-action@v3
        with:
          files: ./coverage/lcov.info
```

package.json Scripts

```
{
  "scripts": {
    "test": "ng test",
```

```
    "test:ci": "ng test --watch=false --browsers=ChromeHe
    "test:coverage": "ng test --code-coverage --watch=fal
  }
}
```

↑ Back to Top

Quick Reference Q&A

General Questions

Q: What's the difference between Jasmine and Karma?

A: Jasmine is a testing framework that provides the syntax and assertions for writing tests. Karma is a test runner that executes those tests in real browsers. Think of Jasmine as the "what" and Karma as the "how."

Q: How do I run tests in Angular?

A: Use `ng test` to run tests in watch mode, or `ng test --watch=false` for a single run. For CI/CD, use `ng test --browsers=ChromeHeadless --watch=false`. See Initial Setup for more options.

Q: How do I generate code coverage reports?

A: Run `ng test --code-coverage --watch=false`. Coverage reports will be generated in the `coverage/` directory. See Coverage Reports for configuration details.

Component Testing

Q: How do I test a component with inputs?

A: Use `fixture.componentRef.setInput('inputName', value)` in modern Angular. See Basic Component Testing for examples.

Q: How do I test component outputs?

A: Subscribe to the output signal: `component.outputName.subscribe(value => { ... })`. See Testing Component Inputs and Outputs.

Q: How do I test a component that uses a service?

A: Create a spy object for the service and provide it in `TestBed.configureTestingModule`. See Testing Component with Dependencies.

Q: How do I trigger change detection?

A: Call `fixture.detectChanges()` after making changes to component state or inputs. This updates the view.

Q: Should I use nativeElement or DebugElement?

A: Use `nativeElement` for simple DOM queries. Use `DebugElement` with `By.css()` for more powerful queries and better testability across platforms.

Service Testing

Q: How do I test a service with HTTP calls?

A: Import `HttpClientTestingModule` and inject `HttpTestingController` to mock HTTP requests. See Service with HTTP Dependencies.

Q: How do I test a service that depends on another service?

A: Create spy objects for dependencies and provide them in `TestBed`. See Mocking Services.

Q: Do I need to use TestBed for simple services?

A: No, you can instantiate simple services directly with `new ServiceName()`. Use TestBed when you need dependency injection. See Basic Service Testing.

Asynchronous Testing

Q: When should I use fakeAsync vs waitForAsync?

A: Use `fakeAsync` with `tick()` for most async operations (setTimeout, setInterval). Use `waitForAsync` when testing promises that don't resolve with tick() or when dealing with external async operations.

Q: What's the difference between tick(), flush(), and flushMicrotasks()?

A:

- `tick(ms)`: Advances time by milliseconds (for setTimeout/setInterval)
- `flush()`: Flushes all pending macrotasks (all timeouts)
- `flushMicrotasks()`: Flushes all pending microtasks (promises)

See Testing with flush and flushMicrotasks.

Q: How do I test observables?

A: For simple cases, subscribe and use the `done` callback. For complex cases, use RxJS TestScheduler with marble diagrams.

Q: How do I wait for async operations to complete?

A: Use `fixture.whenStable()` which returns a promise that resolves when all async operations are done.

Signals Testing

Q: How do I test signals in components?

A: Access signal values with `component.signalName()` and set values with `component.signalName.set(value)`. See Component with Signals.

Q: How do I test computed signals?

A: Computed signals automatically update when their dependencies change. Just verify their value: `expect(component.computedSignal()).toBe(expected)`.

Q: Do signals trigger change detection automatically?

A: Yes, signals integrate with Angular's change detection. You still need to call `fixture.detectChanges()` in tests to update the view.

Mocking and Spies

Q: How do I create a spy for a service method?

A: Use `jasmine.createSpyObj('ServiceName', ['method1', 'method2'])` to create a spy object. See Creating Spy Objects.

Q: How do I make a spy return a value?

A: Use `spy.method.and.returnValue(value)` for synchronous values or `spy.method.and.returnValue(of(value))` for observables.

Q: How do I verify a method was called?

A: Use `expect(spy.method).toHaveBeenCalled()` or `expect(spy.method).toHaveBeenCalledWith(arg1, arg2)`.

Q: What's the difference between spyOn and jasmine.createSpyObj?

A: `spyOn` spies on an existing object's method. `jasmine.createSpyObj` creates a new mock object. See Jasmine Spies.

Forms Testing

Q: How do I test form validation?

A: Get the form control and check its `valid` property and `errors` object. See Reactive Forms.

Q: How do I test custom validators?

A: Create a FormControl, apply the validator, and test the returned ValidationErrors object. See Custom Validators.

Q: How do I set form values in tests?

A: Use `form.patchValue({...})` or `form.get('controlName')?.setValue(value)`.

Q: How do I test form submission?

A: Set valid form values, call the submit method, and verify the emitted output or service call.

HTTP Testing

Q: How do I mock HTTP requests?

A: Import `HttpClientTestingModule` and use `HttpTestingController` to intercept and respond to requests. See HttpClient Testing.

Q: How do I verify HTTP request details?

A: Use `req.request.method`, `req.request.url`, `req.request.body` to inspect the request.

Q: How do I simulate HTTP errors?

A: Use `req.error(new ProgressEvent('error'))` or `req.flush(errorData, { status: 500, statusText: 'Error' })`.

Q: Why is my test failing with "Expected no open requests"?

A: You have HTTP requests that weren't flushed. Make sure to call `req.flush()` for every request. Also, call `httpMock.verify()` in `afterEach()`.

Routing Testing

Q: How do I test navigation?

A: Spy on `router.navigate` and verify it was called with the correct parameters. See Router Testing.

Q: How do I test route parameters?

A: Mock `ActivatedRoute` with snapshot.paramMap or observable params. See Testing Route Parameters.

Q: How do I test guards?

A: Instantiate the guard, call its method (canActivate, canDeactivate, etc.), and verify the return value.

Best Practices

Q: What is the AAA pattern?

A: Arrange (setup), Act (execute), Assert (verify). It's a standard pattern for organizing tests. See Follow AAA Pattern.

Q: How do I make tests run faster?

A: Use `fakeAsync` instead of real timers, mock HTTP calls, avoid unnecessary TestBed configurations, and keep tests focused. See Best Practices.

Q: Should I test private methods?

A: No, test public behavior instead. Private methods are implementation details. See Don't Test Implementation Details.

Q: How do I skip a test temporarily?

A: Use `xit()` or `xdescribe()` to skip tests, or `pending('reason')` inside the test. See Test Helpers.

Q: How do I run only one test?

A: Use `fit()` or `fdescribe()` to focus on specific tests. See Debugging Tests.

Troubleshooting

Q: Why is my test failing with "Cannot read property of undefined"?

A: You're likely accessing a property before initialization. Check if the object exists or initialize it in `beforeEach()`. See Common Issues.

Q: What is "ExpressionChangedAfterItHasBeenCheckedError"?

A: Angular detected that you changed a value after change detection ran. Use `setTimeout()` or `ChangeDetectorRef` to fix it. See Common Issues.

Q: Why are my tests failing randomly?

A: Tests are probably dependent on each other or have async operations that aren't properly handled. Ensure tests are independent and use proper async testing utilities.

Q: How do I debug tests?

A: Use `console.log()`, browser DevTools (run `ng test` and open the debug window), or `fit()` to run single tests. See Debugging Tests.

Q: Why is my coverage lower than expected?

A: Some code paths aren't being tested. Review the coverage report (coverage/index.html) to see which lines are missed. See Coverage Reports.

Advanced Topics

Q: What are test harnesses?

A: Test harnesses provide a high-level API for interacting with components in tests, making tests more maintainable. See Custom Test Harness.

Q: What is the Page Object Pattern?

A: A pattern that encapsulates DOM queries and interactions in a reusable class. See Page Object Pattern.

Q: How do I test RxJS operators?

A: Use RxJS TestScheduler with marble diagrams for precise timing control. See Testing with RxJS Operators.

Q: How do I test for memory leaks?

A: Verify that subscriptions are closed when components are destroyed. See Testing Memory Leaks.

CI/CD

Q: How do I run tests in CI/CD?

A: Use headless Chrome: `ng test --browsers=ChromeHeadless --watch=false --code-coverage`. See Continuous Integration.

Q: How do I fail the build if coverage is too low?

A: Configure coverage thresholds in `karma.conf.js`. See Configure Coverage Thresholds.

Q: Can I run tests in parallel?

A: Not natively with Karma, but you can split test suites and run them separately in parallel CI jobs.

↑ Back to Top

Summary

This comprehensive guide covers all aspects of testing Angular applications with Jasmine and Karma:

1. **Setup**: Configure Jasmine and Karma
2. **Fundamentals**: Master test structure and matchers
3. **Components**: Test component logic and templates
4. **Services**: Test business logic and HTTP calls
5. **Directives & Pipes**: Test custom directives and pipes
6. **Signals**: Test reactive state with signals

7. **Async**: Handle asynchronous operations
8. **Mocking**: Use spies and mocks effectively
9. **Forms**: Test reactive and template-driven forms
10. **HTTP**: Mock HTTP requests with HttpTestingController
11. **Routing**: Test navigation and route parameters
12. **Best Practices**: Follow testing principles
13. **Advanced**: Use test harnesses and page objects
14. **Troubleshooting**: Solve common issues

Key Takeaways

- Write tests following the AAA pattern (Arrange, Act, Assert)
- Keep tests focused, independent, and descriptive
- Use appropriate test utilities (`fakeAsync`, `waitForAsync`)
- Mock external dependencies effectively
- Test behavior, not implementation
- Maintain high code coverage
- Run tests in CI/CD pipelines

Resources

- Angular Testing Guide
- Jasmine Documentation
- Karma Documentation
- Angular CDK Testing
- RxJS Testing

↑ Back to Top

Happy Testing! 🧪 ✨