Table of Contents

# Bab 18: Testing di TypeScript

## Penjelasan Materi

Testing di TypeScript adalah proses memverifikasi bahwa kode berfungsi sesuai yang diharapkan. TypeScript menyediakan fitur type-checking yang membantu mencegah bug, namun testing tetap diperlukan untuk memastikan logika bisnis dan fungsionalitas aplikasi berjalan dengan benar.

## Analogi yang Mudah Dipahami

Bayangkan testing seperti quality control di pabrik: - Unit Testing seperti memeriksa setiap komponen secara individual - Integration Testing seperti memastikan komponen bekerja sama dengan baik - End-to-End Testing seperti menguji produk akhir secara keseluruhan - Test Coverage seperti daftar periksa yang memastikan semua bagian sudah diperiksa - Mocking seperti menggunakan replika komponen untuk testing

## Point Penting

1. **Unit Testing**
   * Test isolation
   * Test cases
   * Assertions
   * Test suites
2. **Integration Testing**
   * Component interaction
   * API testing
   * Database testing
   * Service integration
3. **End-to-End Testing**
   * User flow
   * UI testing
   * Performance testing
   * Cross-browser testing
4. **Test Tools**
   * Jest
   * Testing Library
   * Cypress
   * Supertest
5. **Best Practices**
   * Test organization
   * Test naming
   * Test coverage
   * Continuous testing

## Contoh Kode dan Penjelasan

```typescript // 1. Unit Test Basic // calculator.ts export class Calculator { 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('Division by zero');  
 }  
 return a / b;  
}

}

// calculator.test.ts import { Calculator } from ‘./calculator’;

describe(‘Calculator’, () => { let calculator: Calculator;

beforeEach(() => {  
 calculator = new Calculator();  
});  
  
test('should add two numbers correctly', () => {  
 expect(calculator.add(2, 3)).toBe(5);  
});  
  
test('should throw error when dividing by zero', () => {  
 expect(() => calculator.divide(1, 0)).toThrow('Division by zero');  
});

});

// 2. Integration Test // userService.ts interface User { id: number; name: string; email: string; }

class UserService { private db: Database;

constructor(db: Database) {  
 this.db = db;  
}  
  
async createUser(user: Omit<User, 'id'>): Promise<User> {  
 const newUser = await this.db.insert('users', user);  
 return newUser;  
}  
  
async getUser(id: number): Promise<User | null> {  
 return this.db.findOne('users', { id });  
}

}

// userService.test.ts describe(‘UserService Integration’, () => { let userService: UserService; let db: Database;

beforeAll(async () => {  
 db = await createTestDatabase();  
 userService = new UserService(db);  
});  
  
afterAll(async () => {  
 await db.close();  
});  
  
test('should create and retrieve user', async () => {  
 const user = await userService.createUser({  
 name: 'John',  
 email: 'john@example.com'  
 });  
  
 const retrieved = await userService.getUser(user.id);  
 expect(retrieved).toEqual(user);  
});

});

// 3. E2E Test with Cypress // cypress/integration/login.spec.ts describe(‘Login Flow’, () => { beforeEach(() => { cy.visit(‘/login’); });

it('should login successfully', () => {  
 cy.get('[data-testid="username"]')  
 .type('testuser');  
   
 cy.get('[data-testid="password"]')  
 .type('password123');  
   
 cy.get('[data-testid="login-button"]')  
 .click();  
   
 cy.url()  
 .should('include', '/dashboard');  
   
 cy.get('[data-testid="welcome-message"]')  
 .should('contain', 'Welcome, testuser');  
});

});

// 4. API Testing // api.test.ts import request from ‘supertest’; import app from ‘./app’;

describe(‘API Endpoints’, () => { test(‘GET /api/users should return users list’, async () => { const response = await request(app) .get(‘/api/users’) .expect(‘Content-Type’, /json/) .expect(200);

expect(Array.isArray(response.body)).toBe(true);  
});  
  
test('POST /api/users should create new user', async () => {  
 const userData = {  
 name: 'John Doe',  
 email: 'john@example.com'  
 };  
  
 const response = await request(app)  
 .post('/api/users')  
 .send(userData)  
 .expect('Content-Type', /json/)  
 .expect(201);  
  
 expect(response.body).toMatchObject(userData);  
});

});

// 5. Mock Testing // userRepository.ts interface UserRepository { findById(id: number): Promise<User | null>; save(user: User): Promise; }

// userService.test.ts with mocks describe(‘UserService with Mocks’, () => { let userService: UserService; let mockRepository: jest.Mocked;

beforeEach(() => {  
 mockRepository = {  
 findById: jest.fn(),  
 save: jest.fn()  
 };  
 userService = new UserService(mockRepository);  
});  
  
test('should update user email', async () => {  
 const user: User = {  
 id: 1,  
 name: 'John',  
 email: 'old@example.com'  
 };  
  
 mockRepository.findById.mockResolvedValue(user);  
 mockRepository.save.mockResolvedValue({  
 ...user,  
 email: 'new@example.com'  
 });  
  
 const updated = await userService.updateEmail(1, 'new@example.com');  
 expect(updated.email).toBe('new@example.com');  
});

});

// 6. Custom Test Matchers // matchers/toBeValidEmail.ts expect.extend({ toBeValidEmail(received: string) { const pass = /[[1]](#footnote-23)+@[^\s@]+.[^\s@]+$/.test(received); return { pass, message: () => pass ? Expected ${received} not to be a valid email : Expected ${received} to be a valid email }; } });

// Using custom matcher test(‘should validate email’, () => { expect(‘valid@email.com’).toBeValidEmail(); expect(‘invalid-email’).not.toBeValidEmail(); });

// 7. Snapshot Testing // component.test.tsx import { render } from ‘@testing-library/react’; import { UserProfile } from ‘./UserProfile’;

test(‘UserProfile renders correctly’, () => { const user = { name: ‘John Doe’, email: ‘john@example.com’, role: ‘admin’ };

const { container } = render(<UserProfile user={user} />);  
expect(container).toMatchSnapshot();

});

// 8. Performance Testing class PerformanceTest { private startTime: number = 0; private results: Array<{ operation: string; duration: number }> = [];

start(): void {  
 this.startTime = performance.now();  
}  
  
end(operation: string): void {  
 const duration = performance.now() - this.startTime;  
 this.results.push({ operation, duration });  
}  
  
getResults(): Array<{ operation: string; duration: number }> {  
 return this.results;  
}

}

// 9. Test Coverage // jest.config.js module.exports = { preset: ‘ts-jest’, testEnvironment: ‘node’, collectCoverage: true, coverageThreshold: { global: { branches: 80, functions: 80, lines: 80, statements: 80 } } };

// 10. Test Utilities // testUtils.ts export class TestUtils { static createMockUser(override: Partial = {}): User { return { id: Math.floor(Math.random() \* 1000), name: ‘Test User’, email: ‘test@example.com’, …override }; }

static async createTestData(count: number): Promise<User[]> {  
 const users: User[] = [];  
 for (let i = 0; i < count; i++) {  
 users.push(this.createMockUser({  
 name: \`User \${i}\`,  
 email: \`user\${i}@example.com\`  
 }));  
 }  
 return users;  
}

} ```

## Cara Kerja Testing

1. **Test Setup**:
   * Environment configuration
   * Test data preparation
   * Mock setup
   * Test runner configuration
2. **Test Execution**:
   * Test discovery
   * Test running
   * Assertion evaluation
   * Result collection
3. **Test Reporting**:
   * Coverage reporting
   * Test results
   * Performance metrics
   * Error logging

## Tips dan Trik

1. **Test Organization**

* // ✅ Organize tests logically  
  describe('UserService', () => {  
   describe('create', () => {  
   test('should create valid user', () => {  
   // Test implementation  
   });  
    
   test('should validate input', () => {  
   // Test implementation  
   });  
   });  
    
   describe('update', () => {  
   test('should update existing user', () => {  
   // Test implementation  
   });  
    
   test('should handle non-existent user', () => {  
   // Test implementation  
   });  
   });  
  });

1. **Test Data Factory**

* // ✅ Use factory functions  
  function createTestUser(override: Partial<User> = {}): User {  
   return {  
   id: 1,  
   name: 'Test User',  
   email: 'test@example.com',  
   ...override  
   };  
  }  
    
  test('should update user', () => {  
   const user = createTestUser({ name: 'John' });  
   // Test implementation  
  });

1. **Async Testing**

* // ✅ Handle async operations properly  
  test('should fetch user data', async () => {  
   await expect(userService.getUser(1)).resolves.toMatchObject({  
   id: 1,  
   name: expect.any(String)  
   });  
  });

## Kesalahan yang Sering Dilakukan Pemula

1. **Testing Implementation Details**

* // ❌ Buruk: Testing implementation  
  test('should call database', () => {  
   const spy = jest.spyOn(db, 'query');  
   service.getUser(1);  
   expect(spy).toHaveBeenCalled();  
  });  
    
  // ✅ Baik: Testing behavior  
  test('should return user data', async () => {  
   const user = await service.getUser(1);  
   expect(user).toHaveProperty('id', 1);  
  });

1. **Tidak Menangani Async**

* // ❌ Buruk: Missing await  
  test('should save user', () => {  
   service.saveUser(user);  
   expect(db.users.length).toBe(1);  
  });  
    
  // ✅ Baik: Proper async handling  
  test('should save user', async () => {  
   await service.saveUser(user);  
   expect(db.users.length).toBe(1);  
  });

1. **Test yang Tidak Terisolasi**

* // ❌ Buruk: Shared state  
  let users: User[] = [];  
    
  test('should add user', () => {  
   users.push(newUser);  
   expect(users.length).toBe(1);  
  });  
    
  // ✅ Baik: Isolated tests  
  describe('UserManagement', () => {  
   let users: User[];  
    
   beforeEach(() => {  
   users = [];  
   });  
    
   test('should add user', () => {  
   users.push(newUser);  
   expect(users.length).toBe(1);  
   });  
  });

### Solusi:

1. Fokus pada testing behavior
2. Gunakan proper async/await
3. Isolasi test state
4. Implementasi test factories
5. Gunakan test coverage

1. ^@ [↑](#footnote-ref-23)