

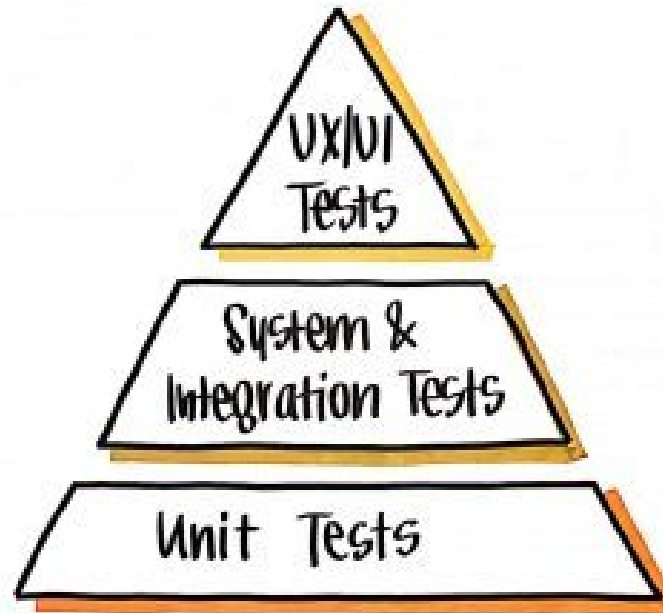
RICH CLIENT TESTING

GUIDELINES TESTING

- es geht um Qualität, nicht Quantität
- Test Coverage zeigt das Fehlen von Tests, nicht die Qualität
- Softwareentwickler sind selbst für Qualität verantwortlich
- es geht um gute Tests, nicht um 100% Coverage

TEST PYRAMIDE

TEST PYRAMIDE



UNIT TESTS

- testen einer Unit
- Units sind die kleinsten Einheiten in unserem System
- auf sehr detaillierter Ebene

INTEGRATION TESTS

- testen von zusammenhängenden Units
- weniger Detailtiefe
- Fokus liegt auf
 - wichtigen Szenarien
 - interessanten Edge-Cases
 - Fehlern die aufgetreten sind

UI TESTS

- testen über das richtige UI
- manuell oder automatisiert
- Styling erfordert manuelle Tests

UNIT TESTS

UNIT TESTS

- in unserem Fall sind Units
 - Components
 - Services

WIESO UNIT TESTS?

- divide et impera
 - Wir teilen Komplexität mit Components
 - Wieso nicht auch bei Tests?
- Testen auf sehr detaillierter Ebene
 - viele Kombinationen möglich
 - Übersichtlichkeit?

TEST DRIVEN

- wir entwickeln Components
- also müssen wir auch Components testen
- es reicht nicht ein ganzes Feature zu testen
 - Components können in anderen Features eingesetzt werden
 - funktioniert die Component in einem anderen Szenario?
 - schnelles Feedback durch Tests

ISOLIERTES TESTEN

- Unit-Testing heißt isoliertes Testing einer Unit
- Wie isolieren wir die Unit?
- Dependency Injection
 - eine Component kennt nur die Schnittstelle eines Services
 - sie kennt nicht seine Implementierung
 - die Implementierung tauschen wir im Tests aus

MOCKS

- sind die Implementierungen im Testfall
- sind nur Attrappen für Objekte (Services)
- über sie können wir
 - variables Verhalten von Services eliminieren
 - Mock-Daten an die Units geben
 - prüfen ob Methoden richtig aufgerufen wurden

FRONTEND UNIT TESTS

FRONTEND UNIT TESTS

- Service Tests
 - Services sind einfache Typescript Klassen
 - testen von Programmlogik
- Was ist mit Component Tests?
- Component enthält
 - Logik im Typescript File
 - Struktur und Styling im HTML & CSS

COMPONENT TEST

- Was testen wir?
 - Logik
 - dynamische Struktur?
 - dynamisches Styling?
- dynamische Struktur/Styling
 - *ngIf
 - *ngFor
 - [class]

COMPONENT TESTS

- Nutzerinteraktionen
- ein Nutzer
 - ruft nicht die Typescript Methoden auf
 - klickt auf die Buttons/Links
 - interagiert mit den Inputs
- wir sollten daher
 - das HTML testen
 - programmatisch die Buttons klicken
 - Input Felder direkt bearbeiten

TESTING MIT JAVASCRIPT (JASMINE)

```
1 describe('ich bin eine Beschreibung', () => {
2     beforeAll(() => {});
3
4     beforeEach(() => {});
5
6     it('ich bin ein Test', () => {
7         expect(actual).toEqual(expect) // quasi ein assert
8     });
9
10    afterEach(() => {});
11
12    afterAll(() => {});
13 });
```

SAUBERER AUFBAU VON JAVASCRIPT TESTS

TESTBESCHREIBUNG

- sollte einen Satz ergeben
- folgt einem Schema
 - z.B. "Object ... should ... when"
 - gerne auch andere Schema's
 - viele gehen in ähnliche Richtung
- Testbeschreibung als lebende Doku

```
1 it('ComponentUnderTest should show element-card  
2     when element-data is not empty', () => {});
```

TESTBESCHREIBUNG

- sollte wenig Duplizierungen enthalten
- damit entsteht eine saubere Struktur

```
1 describe('ComponentUnderTest', () => {  
2   describe('updateData()', () => {  
3     it('should update data when data is not empty',  
4       () => {});  
5  
6     it('should not update data when data is empty',  
7       () => {});  
8  
9     ...  
10  });  
11  
12  ...  
13 });
```

SINGLE RESPONSIBILITY PRINCIPLE

- jeder Test sollte nur eine Sache testen
 - am besten ein "expect" pro Test
 - macht es einfacher eine Testbeschreibung zu finden
- im Fehlerfall ist das Problem schneller erkannt

CODEDUPLIZIERUNG

- soll vermieden werden
- setup Code kann in "beforeEach"/"beforeAll"
- parametrisierte Tests
 - gleicher Test mit unterschiedlichen Parametern
 - weniger Codeduplizierung

```
1 it.each([
2   {input: 'input1', expected: 'expected1'},
3   {input: 'input2', expected: 'expected2'},
4   ...
5 ])( 'should to something', ({input, expected}) => {
6   // code for testing
7 });
```

REPRODUZIERBAR

- Tests müssen reproduzierbar sein
- "date.now()"?
 - Produktivcode ist abhängig vom aktuellen Datum
 - Testcode muss damit auch vom aktuellen Datum abhängig sein

TESTING IN ANGULAR

TESTING MODULE

- ein Angular Module
- um die Component isoliert zu testen
- für jeden Test ein neues Module

```
1 beforeEach(waitForAsync(() => {
2   TestBed.configureTestingModule({
3     imports: [
4       RouterTestingModule
5     ],
6     declarations: [
7       AppComponent
8     ],
9   }).compileComponents();
10 }));
```

FIXTURE

- bekommt man beim Erstellen der Component
- lässt uns auf die Bestandteile der Component zugreifen
- kennt außerdem den Zustand der Component

```
1 const fixture = TestBed.createComponent(AppComponent);
```

FIXTURE

- gibt Zugriff auf das JavaScript Objekt einer Component
- ermöglicht Zugriff auf das HTML

```
1 const component = fixture.debugElement.componentInstance;
```

```
1 fixture.debugElement.query(By.css( '.some-class' ));  
2 fixture.debugElement.query(By.css( '#some-id' ));  
3 const nativeElement = fixture.debugElement.query(By.css(  
4     '[data-test-id=some-data-test-id]'  
5 )).nativeElement;  
6 const componentInstance = fixture.debugElement.query(  
7     By.directive(SomeComponent)  
8 ).componentInstance;
```

FIXTURE

- kennt den Zustand einer Component
- kann die Change Detection auslösen
 - Change Detection wird im Test nicht automatisch ausgelöst

```
1 fixture.detectChanges();
```

```
1 await fixture.whenStable();
```

TS-MOCKITO

- bekannt aus Java
- Library fürs Mocking
- Syntax ist für andere Sprachen ähnlich

Zero Setup:

```
"ts-mockito": "^2.6.1"
```

TS-MOCKITO

- `mock()` liefert ein Rekorder Objekt
 - interaktionen mit dem Mock Objekt werden erkannt
 - Rückgabewerte für Methoden werden hier definiert
- `instance()` liefert das Mock Objekt
 - ein Objekt der Klasse
 - mockt öffentliche Schnittstellen

```
1 class SomeService {
2     someFunction(): boolean {
3         return true;
4     }
5
6     someOtherFunction(): string {
7         return 'foo';
8     }
9 }
10
11 const someServiceMock = mock(SomeService);
12 const someServiceInstance = instance(someServiceMock);
```

WHEN

- when() mockt Rückgabewerte des Mocks
- ein Mock kann verschiedene Rückgabewerte haben

```
1 when(mock.someFunction('test')).thenReturn(false);  
2 when(mock.someFunction('testi')).thenReturn(true);  
3 when(mock.someFunction(anything())).thenReturn(true);  
4 when(mock.someOtherFunction()).thenReturn('bar');
```


VERIFY

- mit `verify()` prüfen wir Interaktionen mit dem Mock
- wir können testen wie viele Interaktionen stattgefunden haben
- Parameter werden "strict-equal" verglichen
 - bei Objekten heißt das per Referenz
- für Objekte nutzt man meistens `deepEqual()`
 - schließlich will man die Werte prüfen

```
1 verify(mock.someFunction()).once();
2 verify(mock.someOtherFunction('test123')).once();
3 verify(mock.someOtherFunction(deepEqual({}))).once();
4 verify(mock.someOtherFunction(anything())).never();
5 verify(mock.someOtherFunction(anyString())).never();
```

ALTERNATIVEN ZU MOCKITO

- Jasmine/Jest
 - liefert Testing mit
 - nicht so komfortabel
- manuelles Mocking
 - sehr aufwendig
 - ungeschützt bei Renaming

```
1 const someServiceMock = {  
2     someFunction(): boolean {  
3         return true;  
4     },  
5     someOtherFunction(): string {  
6         return 'lala';  
7     }  
8 } as SomeService;
```

NG-MOCKS

- Library zum Mocken von Components
- Component kann andere Components enthalten
- Subcomponents sollen nicht getestet werden

NG-MOCKS

Zero Setup:

```
"ng-mocks": "^12.5.0"
```

Mocking von Components:

```
1 TestBed.configureTestingModule({  
2   declarations: [  
3     MockComponent(ButtonComponent),  
4     ComponentUnderTest  
5   ],  
6 }).compileComponents();
```

NG-MOCKS

Mock Component ist über das Fixture erreichbar

```
1 const componentInstance = fixture.debugElement.query(  
2   By.directive(SomeComponent)  
3 ).componentInstance;
```

Input und Output Parameter sind verwendbar

```
1 componentInstance.someOutput.emit('someOutput');  
2 componentInstance.someInput = 'someInput';
```

TESTING LIBRARIES

- Karma
 - Standard für Angular
 - nutzt echten Browser für Testausführung
 - nah an der Realität
- Jest
 - nutzt Headless Browser
 - viel schneller als Karma

PRAXIS

TESTEN

- testen einer Component eurer Wahl
- testen eines Services eurer Wahl
- Branch: testing_with_jest