

RICH CLIENT TESTING

GUIDELINES TESTING

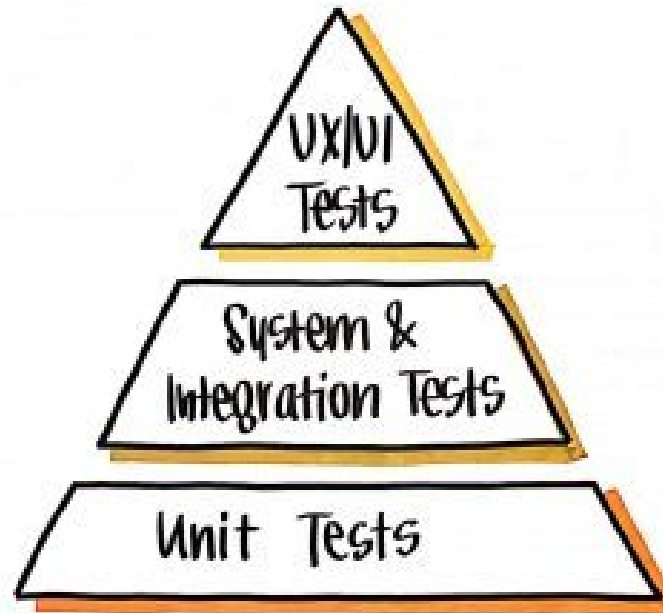
- es geht um Qualität, nicht Quantität
- Test Coverage zeigt das Fehlen von Tests, nicht die Qualität
- Softwareentwickler sind selbst für Qualität verantwortlich
- es geht um gute Tests, nicht um 100% Coverage

Speaker notes

- Es geht beim Testing nicht darum 100% Test-Coverage zu erreichen und jeden erdenklichen Fall zu testen
- Eine Menge unnützer Tests gibt uns keinen Garanten für gute Software.
- Softwareentwickler sollten Gehirn investieren, um Tests für sinnvolle Fälle zu schreiben.
- Diese Guidelines kann man auf alle Tests übertragen.

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

Speaker notes

- Meist wird der Code bei einem Integration-Test in einer Testenvironment getestet.
- Das heißt die Schnittstellen werden maschinell angesprochen.
- Da es mehr Fälle geben kann, muss man sich auf wichtige Szenarien beschränken.
- Oft lohnt es sich merkwürdige Fehler die aufgetreten sind über einen Integration-Test zu testen, um den Fehler in Zukunft zu verhindern.

UI TESTS

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

Speaker notes

- Test auf der realen Umgebung.
- Computer oder Mensch klickt sich durch die Anwendung und prüft ob sie richtig funktioniert.
- Ein Computer kann nicht entscheiden ob etwas gut oder schlecht aussieht. Daher kommt es beim Styling immer noch auf den Mensch an.

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?

Speaker notes

- Auch für Tests lohnt es sich sie in kleine Teile aufzuteilen, um die Anwendung übersichtlich zu halten.
- Bei einem Unit-Test können wir auf sehr detaillierter Ebene Testen.
 - Wenn wir mehrere Components zusammen testen, entstehen viele Kombinationen von Eingabeparametern und Zuständen.
 - Wir brauchen also eine Menge Testfälle, die schnell unübersichtlich werden können.

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

Speaker notes

- Components sind nur vollständig, wenn sie auch über einen Test verfügen.
- Schließlich sollen die Component vielleicht in unterschiedlichen Szenarien eingesetzt werden. Es braucht Tests um abzusichern, dass sie sich auch richtig verhält.
- Ein Feature zu entwickeln dauert manchmal ein paar Wochen. Wenn ich erst am Ende einen Test schreibe, bekomme ich sehr spät Feedback. Mit Unit-Tests geht das schneller.

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

Speaker notes

- Wenn wir eine Component testen und diese einen Service nutzt, wollen wir den Service nicht mit testen.
- Wenn wir den Service mittesten würde, müssten wir auch seine Abhängigkeiten bereitstellen.
- Daten die in den Service hineingegeben werden (aus der Component) müssten valide sein, denn er führt darauf ja richtige Operationen aus.
- Wir müssen also richtige Daten nutzen und können nicht simple Testdaten nutzen.

MOCKS

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

Speaker notes

- Mock Frameworks nehmen hier einige Arbeit ab.

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

Speaker notes

- Service Tests sind relativ straight forward. Quasi wie ein Unit Test aus Java.
- Services sind einfache Typescript klassen.
- Components sind da ein wenig komplexer. Schließlich verfügen sie zusätzlich noch über HTML und CSS.

COMPONENT TEST

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

Speaker notes

- Die Logik der Components sollten wir natürlich testen.
- Statische Struktur und Styling zu testen macht in der Regel keinen Sinn. Warum sollte ich testen, dass der Text bold ist, wenn ich grade das Styling hinzugefügt habe.
- Dynamische Struktur und Styling zu testen macht durchaus Sinn. Beispiel: Wenn eine Variable gesetzt ist, soll der Text rot hinterlegt sein.

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

Speaker notes

- Unsere Unit-Tests sind näher an der Realität, wenn wir das HTML mit testen.

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
```

Speaker notes

- JavaScript Test werden normalerweise mit Jasmine geschrieben.
- Jasmine liefert verschiedene Tools, um Tests strukturierter aufzubauen und einfacher zu gestalten.
- "describe" beschreibt einen Test und kann mehrere Tests logisch zusammenfassen.
- "beforeAll" bekommt eine Callback Function, die vor allen Tests in diesem "describe" Block ausgeführt wird.
- "beforeEach" bekommt eine Callback Function, die vor jedem Test in diesem "describe" Block ausgeführt wird.
- "it" ist ein einzelner Test. Er hat eine Beschreibung und eine Callback Function in der Testcode stehen sollte.
- "expect" ist quasi ein assert aus Java. Bietet einige Hilfsmethoden wie .toEqual() um das Testen zu vereinfachen.
- "afterEach" und "afterAll" erklären sich glaube ich selbst.

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' () => {}),
```

Speaker notes

- Testbeschreibung als vollständiger Satz hilft dabei zu verstehen, was im Test passieren soll.
- ein Schema sorgt dafür, dass sich andere Entwickler besser zurechtfinden. Und auch man selbst hat irgendwann vergessen was hier in diesem Code passiert.
- Über gute Testbeschreibungen kann man sehr gut erkennen wie sich Components verhalten sollen und was ihre Aufgabe ist.

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

Speaker notes

- Wenn der Test nur eine Sache testen, ist im Fehlerfall schneller erkennbar was kaputt ist. Denn es steht ja in der Testbeschreibung.
- Es kann natürlich auch passieren, dass der Test wegen Setup Code fehlschlägt, dieser Setup Code sollte allerdings auch in einem eigenen Test fehlschlagen.

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
```

Speaker notes

- Codeduplizierung soll wie in Produktivcode vermieden werden. Sonst müssen wir in Zukunft Änderungen an 20 Stellen machen.
- Setup Code ist oft dupliziert und sollte daher in entsprechende Blöcke ausgelagert werden.
- Parametrisierte Tests helfen, wenn selbst der Testcode dupliziert wird.

REPRODUZIERBAR

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

Speaker notes

- Tests müssen immer laufen, egal was der Kontext ist in dem sie sich befinden.
- Tests können fehlschlagen, wenn sich das Datum ändert und der Testfall statische Daten anstatt dynamischen verwendet.
- Bei uns sind mal Tests am ersten April gefailed. Hier hat sich ein Entwickler einen Spaß erlaubt.

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   });  
10
```

Speaker notes

- Die Idee dahinter ist, dass die einzelnen Tests so völlig isoliert voneinander sind. Jeder Test hat ja sein eigenes Module.
- Damit können Tests auch nicht aufgrund ihrer Reihenfolge fehlschlagen. Lediglich wenn der Entwickler Daten über die Testfälle teilt.

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 fixture.debugElement.query(By.css( '[data-test-id=some-data-t
```

Speaker notes

- Über das JavaScript Objekt können wir auf den Zustand der Component zugreifen. Wir können Variablen auslesen und Methoden aufrufen.
- Mit dem Zugriff auf das HTML können wir prüfen, ob das HTML richtig gerendert wurde.
- Außerdem können wir programmatisch Events auf den HTML Elementen auslösen. Z.B. ein Klick.

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();
```

Speaker notes

- Normalerweise wird die Change Detection vom Framework immer aufgerufen wenn sich Daten einer Component ändern. Bzw. wenn sich Daten ändern, die auch im HTML relevant sind.
- `detectChanges()` muss aufgerufen werden, wenn man möchte, dass das HTML gerendert wird. Wenn man Daten ändert und sich das HTML verändern soll, muss erst `detectChanges` aufgerufen werden.
- `whenStable()` kann helfen auf asynchrone Events zu warten.

TS-MOCKITO

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

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 }
```

Speaker notes

- Um ein Mock Objekt zu erstellen scannt Mockito die Klassenstruktur und erkennt die öffentlichen Schnittstellen.
- Auf dem Rekorder Objekt können wir dann Rückgabewerte für die Methoden definieren und interaktionen mit dem Mock prüfen.
- Die Mock Instanz geben wir an das zu testende Objekt. In Angular registrieren wir es im Testing Module, damit es der Dependency Injection Container in die Component stecken kann.

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');
```

Speaker notes

- Wenn die Parameter egal sind, die in das Mock Objekt hineingegeben werden, kann `anything()` genutzt werden. Dies gibt es auch in spezielleren Varianten wie `anyString()`.

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();
```

Speaker notes

- `anything()` sollte möglichst nur in Kombination mit `.never()` verwendet werden. Schließlich wollen wir sonst prüfen, ob die Funktion auch mit den richtigen Parametern aufgerufen wurde.
- Statt `once()` gibt es auch `twice()` und `times(number)`
- Wenn man das Objekt nicht matchen kann nutzt lieber `anyString()` als `anything()`. Dann ist es wenigstens der gleiche Typ.

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 }
```

Speaker notes

- Wenn ich Mocking-Frameworks verwende habe eine direkte Referenz auf die Klasse. Nenne ich eine Methode der Klasse um, ändert sich automatisch auch das Mock.
- Mocke ich ein Objekt manuell, dann muss ich die Methoden hier manuell umbenennen.

TESTING LIBRARIES

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

Speaker notes

- Die Frage ist, ob man die Nähe zur Realität braucht. Bzw. ob eigentlich auch ein Headless Browser reicht.
- Headless Browser ist ein Browser ohne UI.

PRAXIS

TESTEN

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