

willkommen bei kodschul.com  
Schulung: “Angular-Testing ”

Tag 1/2



## Clients Coached

200+

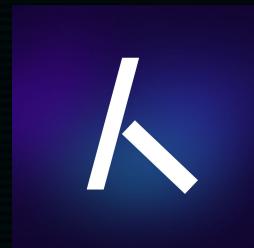
## Team Size

50+ members (including Rigle GmbH)

## Experience in IT Coaching

10 years

Agility, Continuous Learning, and Open Communication, Client Satisfaction through quality over quantity, High quality at affordable prices delivered on time, whether it's a seminar, staffing, or consulting project.



**Who is Who?**



# Kursinhalte

# Inhaltsverzeichnis: Tag 1

## **Modul 1: Grundlagen der Fehlerbehandlung und Angular-Syntax verstehen**

- Lab 1.1. Einführung in Fehlerbehandlung und Fehlermeldungen in Angular
- Lab 1.2. Hinzufügen einer neuen Datei mit Angular CLI & Fehlerbehandlung
- Lab 1.3. Coding Practise & Solutions

## **Modul 2: Arbeit mit Pipes und Dekoratoren in Angular**

- Lab 2.1. Häufige Syntax- und Pipe-Fehler
- Lab 2.2. Erstellen einer benutzerdefinierten Pipe
- Lab 2.3. Coding Practise & Solutions

## **Modul 3: Unit Tests für Services und Komponenten**

Lab 3.1. Überblick über das Testen in Angular

Lab 3.2. Testen eines Angular-Services

Lab 3.3. Coding Practise & Solutions

## **Modul 4: Testen von Pipes**

Lab 4.1. Einführung in das Testen von Pipes

Lab 4.2. Tests für eine benutzerdefinierte Pipe

Lab 4.3. Coding Practise & Solutions

# Inhaltsverzeichnis: Tag 2

## **Modul 5: Einführung in End-to-End Tests mit Protractor**

- Lab 5.1. Grundlegende Protractor-Konfiguration
- Lab 5.2. Einfache End-to-End-Tests schreiben
- Lab 5.3. Coding Practise & Solutions

## **Modul 6: Arbeiten mit Seitenobjekten und erweiterten Protractor-Tests**

- Lab 6.1. Seitenobjekte in Protractor
- Lab 6.2. Schreiben eines End-to-End-Tests mit Seitenobjekten
- Lab 6.3. Coding Practise & Solutions

## **Modul 7: Statisches Code-Linting und Fehlerbehebung**

Lab 7.1. Einführung in ESLint und Code-Qualität

Lab 7.2. Linting eines Angular-Projekts

Lab 7.3. Coding Practise & Solutions

## **Modul 8: Integration von Tests und Linting in CI/CD-Pipelines**

Lab 8.1. Einführung in CI/CD für Angular-Projekte

Lab 8.2. Erstellen einer einfachen CI/CD-Pipeline

Lab 8.3. Coding Practise & Solutions



# **Modul 1: Grundlagen der Fehlerbehandlung und Angular-Syntax verstehen**

# Modul 1 Aufbau





# **Lab 1: Einführung in Fehlerbehandlung und Fehlermeldungen in Angular**

## **Ziel der Fehlerbehandlung**

- Proaktives Erfassen und Analysieren von Fehlern, um diese schnell zu beheben.
- Gewährleistung der Stabilität und Zuverlässigkeit der Anwendung.
- Bessere Nutzererfahrung durch saubere Fehlermeldungen und effiziente Fehlerbehandlung.

## Typen von Fehlern in Angular

- **Syntaxfehler:** Fehler in TypeScript oder HTML.
- **Laufzeitfehler:** Fehler, die zur Laufzeit auftreten, z.B. nicht verfügbare Abhängigkeiten.
- **Asynchrone Fehler:** Fehler, die durch Observable oder Promises verursacht werden.

# Tools und Mechanismen zur Fehlerbehandlung in Angular

- Verwende den ErrorHandler, um globale Fehler in der Anwendung abzufangen und zu verarbeiten.
- Custom Error Handling:
  - Override des Standard-Error-Handlers für spezifische Anforderungen.

```
import { ErrorHandler, Injectable } from '@angular/core';

@Injectable()
export class GlobalErrorHandler implements ErrorHandler {
  handleError(error: any): void {
    // Log the error
    console.error('An error occurred:', error.message);
    // Send the error to a logging service
  }
}
```

# Common Error Types



## **ExpressionChangedAfterItHasBeenCheckedError**

- Ursachen: Änderungen am Datenmodell nach dem Rendering des View.
- Lösung: Verwende ChangeDetectorRef oder setTimeout für asynchrone Updates.



## **NullInjectorError**

- Ursachen: Eine Abhängigkeit konnte nicht injiziert werden.
- Lösung: Überprüfe, ob die entsprechende Abhängigkeit im Modul deklariert wurde.



## **Unhandled Promise Rejection**

- Ursachen: Eine Promise wurde abgelehnt, aber nicht korrekt behandelt.
- Lösung: Füge catch-Blöcke zu Promises hinzu oder verwende async/await für besseres Error Handling.

## Beispiel für NullInjectorError

```
NullInjectorError: No provider for MyService!
```

➤ Lösung

```
@NgModule({
  providers: [MyService],
})
export class AppModule {}
```

# Fehler bei der Injektion von Abhängigkeiten

## ➤ Was ist Dependency Injection (DI)?

- Ein Designprinzip, das Angular verwendet, um Abhängigkeiten automatisch bereitzustellen.

## Typische DI-Fehler

- **NullInjectorError:** Keine passende Abhängigkeit gefunden.
- **Cyclic Dependency Error:** Zirkuläre Abhängigkeiten zwischen Services oder Komponenten.
- **Provider Fehler:** Fehlerhafte Konfiguration oder das Fehlen eines Providers.

## Lösungsansätze für DI-Fehler

- **NullInjectorError:** Überprüfe, ob der Dienst im richtigen Modul oder auf Komponentenniveau bereitgestellt wurde.
- **Cyclic Dependency:** Stelle sicher, dass es keine zirkulären Abhängigkeiten gibt, indem du Interfaces oder alternative Designmuster verwendest.



## Lab 2: Hinzufügen einer neuen Datei mit Angular CLI & Fehlerbehandlung



## Lab 3: Coding Practise & Solutions





## Modul 2: Arbeit mit Pipes und Dekoratoren in Angular

## Modul 2 Aufbau





# Lab 1: Häufige Syntax- und Pipe-Fehler

## Hinzufügen eines lokalen Speicherdienstes

- Der localStorage ist ein Webspeicher, der Daten im Browser speichert, die auch nach einem Seiten-Neuladen erhalten bleiben.
- Service für lokale Speicherung erstellen:
  - Erstellen eines neuen Services mit Angular CLI:

```
ng generate service local-storage
```

## Hinzufügen eines lokalen Speicherdienstes

- Häufige Fehler:
  - Fehlende Speicherung komplexer Objekte (JSON.stringify/parse nicht verwendet).
  - Zugriff auf den lokalen Speicher, bevor der Service initialisiert wurde.

## Abrufen und Setzen von Daten im lokalen Speicher

- Daten im lokalen Speicher speichern:

- Mit localStorage.setItem() können einfache Werte wie Strings gespeichert werden.

```
this.localStorageService.setItem('user', 'John Doe');
```

- Daten abrufen:

- Werte mit localStorage.getItem() abrufen.

```
const user = this.localStorageService.getItem('user');
console.log(user); // 'John Doe'
```

## Abrufen und Setzen von Daten im lokalen Speicher

- Häufige Fehler:
  - Vergessene JSON-Konvertierung bei komplexen Objekten:

```
this.localStorageService.setItem('user', JSON.stringify(userObject));
const user = JSON.parse(this.localStorageService.getItem('user'));
```

## Hinzufügen einer benutzerdefinierten Angular-Pipe

- Pipes in Angular werden verwendet, um Daten in Templates zu transformieren.
- Erstellen einer benutzerdefinierten Pipe:
  - Generiere eine neue Pipe mit der Angular CLI:

```
ng generate pipe customPipe
```

## Verwenden einer Pipe zum Ändern von HTML-Inhalten

- Pipe, die bestimmte HTML-Tags entfernt oder ersetzt.

```
@Pipe({ name: 'stripHtml' })
export class StripHtmlPipe implements PipeTransform {
  transform(value: string): string {
    return value.replace(/<\/?[^>]+(>|\$/g, '');
  }
}
```

## **Verwenden einer Pipe zum Ändern von HTML-Inhalten**

- Anwendungsfall:
  - Verwende die Pipe, um gefährliche oder unerwünschte HTML-Tags aus einer Benutzereingabe zu entfernen.
- Häufige Fehler:
  - Sicherheitsrisiken durch ungenügende Sanitization von HTML-Inhalten.
  - Nicht deklarierte Pipe im AppModule.
  - Typkonflikte bei der Rückgabe von Werten (z.B. Rückgabe von Objekten, wenn Strings erwartet werden).

## Häufige Pipe-Fehler

- **Nicht deklarierte Pipes:** Vergessene Deklaration im AppModule oder in einem spezifischen Modul.
- **Falsche Rückgabewerte:** Pipes sollten stets die korrekten Typen zurückgeben, die in der Template-Bindung erwartet werden.
- **Fehlerhafte Parameterübergabe:** Falsche oder nicht übergebene Argumente an Pipes.

```
<!-- Falsche Argumente -->  
<div>{{ value | customPipe:'incorrectArg' }}</div>
```

## Verstehen von Angular-Dekoratoren

- Was sind Dekoratoren?: Spezielle TypeScript-Annotationen, die Klassen und Eigenschaften mit zusätzlichem Verhalten versehen.
- Wichtige Angular-Dekoratoren:
  - @Component(): Markiert eine Klasse als Angular-Komponente.
  - @Injectable(): Kennzeichnet eine Klasse als injizierbaren Service.
  - @Input() / @Output(): Ermöglichen die Kommunikation zwischen Komponenten.

## Verstehen von Angular-Dekoratoren

- Häufige Fehler:
  - Vergessene oder falsch platzierte Dekoratoren.
  - Falsche Deklaration von Abhängigkeiten im Konstruktor (Dependency Injection).



## Lab 2: Erstellen einer benutzerdefinierten Pipe



## Lab 3: Coding Practise & Solutions





## Modul 3: Unit Tests für Services und Komponenten

# Modul 3 Aufbau

Überblick über das  
Testen in Angular



Testen eines  
Angular-Services



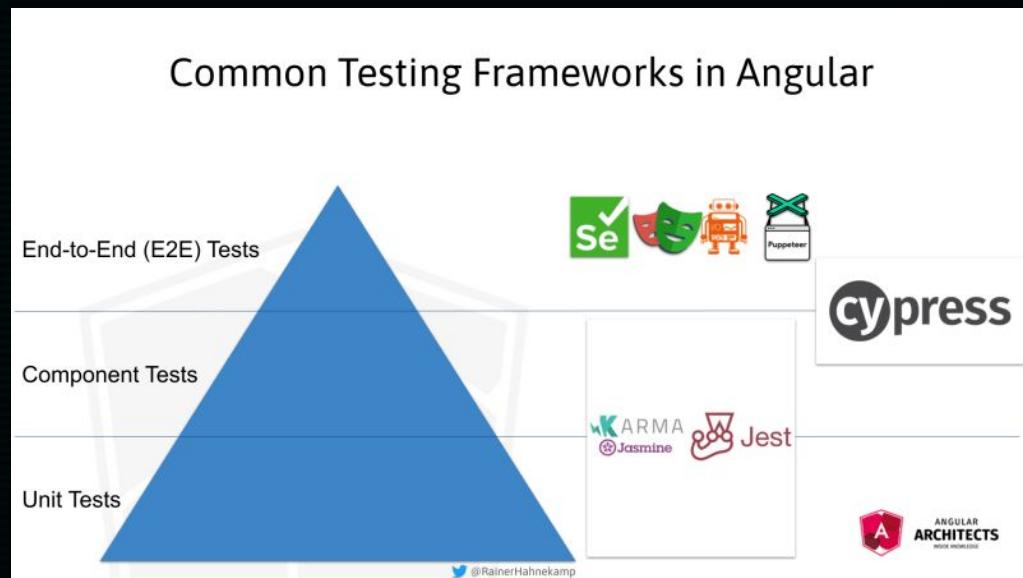
Coding Practise &  
Solutions



# **Lab 1: Überblick über das Testen in Angular**

## Arten von Tests

- **Unit Tests:** Testen von isolierten Einheiten wie Funktionen oder Klassen.
- **Integration Tests:** Überprüfen das Zusammenspiel verschiedener Komponenten und Module.
- **End-to-End (E2E) Tests:** Simulieren Benutzeraktionen, um das gesamte System zu testen.



# Testen in Angular

<https://www.rainerhahnekamp.com/en/angular-testing-in-2023-past-present-and-future/>

	Official	Community
Framework	 Jasmine	 Jest
Web API	 KARMA	 jsdom
Integration	 A	just-jeb/ angular-builders
Build Process	 A	jest-preset-angular

## Testen in Angular

- **Angular Testing Frameworks:** Angular nutzt Jasmine als Testframework und Karma als Test Runner, um Unit Tests auszuführen.

## Einführung in Karma

- Ein Test-Runner für JavaScript-Anwendungen, der speziell für Angular verwendet wird, um Unit Tests in einer realen Browserumgebung auszuführen.
- **Funktion von Karma:**
  - Startet einen Browser: Karma führt die Tests direkt in einem oder mehreren realen Browsern aus.
  - Läuft im Hintergrund: Beobachtet den Code auf Änderungen und führt die Tests bei jeder Änderung automatisch erneut aus.

# Einführung in Jasmine

- Ein Testframework für JavaScript, das speziell für das Schreiben von Unit Tests in Angular-Projekten verwendet wird.
- **Schlüsselkonzepte in Jasmine:**
  - **Describe:** Definiert eine Test-Suite (Sammlung von Tests).
  - **It:** Definiert einen einzelnen Testfall.
  - **Expect:** Stellt Erwartungen oder Bedingungen an das Ergebnis eines Tests.
  - **Matchers:** Vergleichsfunktionen wie toBe, toEqual,toContain, die zum Überprüfen der Testergebnisse verwendet werden.

## Beispiel für einen Jasmine-Test

```
describe('Test-Suite für eine Funktion', () => {
  it('sollte true zurückgeben, wenn die Funktion korrekt ausgeführt wird', () => {
    const result = true;
    expect(result).toBe(true);
  });
});
```

## Schritte zum Testen einer Komponente

- **Setup:** Erstelle eine Testumgebung mithilfe des TestBed-Moduls.
- **Instanziierung:** Erstelle eine Instanz der Komponente in der Testumgebung.
- **Ausführung von Tests:** Überprüfe Methoden und das DOM der Komponente.

# Schritte zum Testen einer Komponente

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { MyComponent } from './my.component';

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

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [MyComponent]
    });
    fixture = TestBed.createComponent(MyComponent);
    component = fixture.componentInstance;
  });

  it('sollte erstellt werden', () => {
    expect(component).toBeTruthy();
  });
});
```

# Teststrategien für Angular-Komponenten

-  **Testen von DOM-Interaktionen:** Überprüfe, ob Benutzerinteraktionen wie Klicks oder Tastatureingaben das gewünschte Verhalten der Komponente auslösen.
-  **Testen von Datenbindung:** Stelle sicher, dass Daten korrekt zwischen der Komponente und dem Template gebunden werden.
-  **Testen von Event-Emittern:** Überprüfe, ob die richtigen Ereignisse durch die Komponente ausgelöst werden.

## Schritte zum Testen eines Services

- **Setup:** Nutze das TestBed, um den Service zu initialisieren.
- **Mocks:** Verwende Mock-Objekte oder spyOn, um Abhängigkeiten zu isolieren (z.B. HTTP-Requests mocken).
- **Tests schreiben:** Überprüfe Methoden und Rückgabewerte des Services.

# Schritte zum Testen eines Services

```
import { TestBed } from '@angular/core/testing';
import { MyService } from './my.service';

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

  beforeEach(() => {
    TestBed.configureTestingModule({}); // ConfigureTestingModule
    service = TestBed.inject(MyService);
  });

  it('sollte den Dienst erstellen', () => {
    expect(service).toBeTruthy();
  });

  it('sollte den korrekten Wert zurückgeben', () => {
    const value = service.getValue();
    expect(value).toBe('expectedValue');
  });
});
```

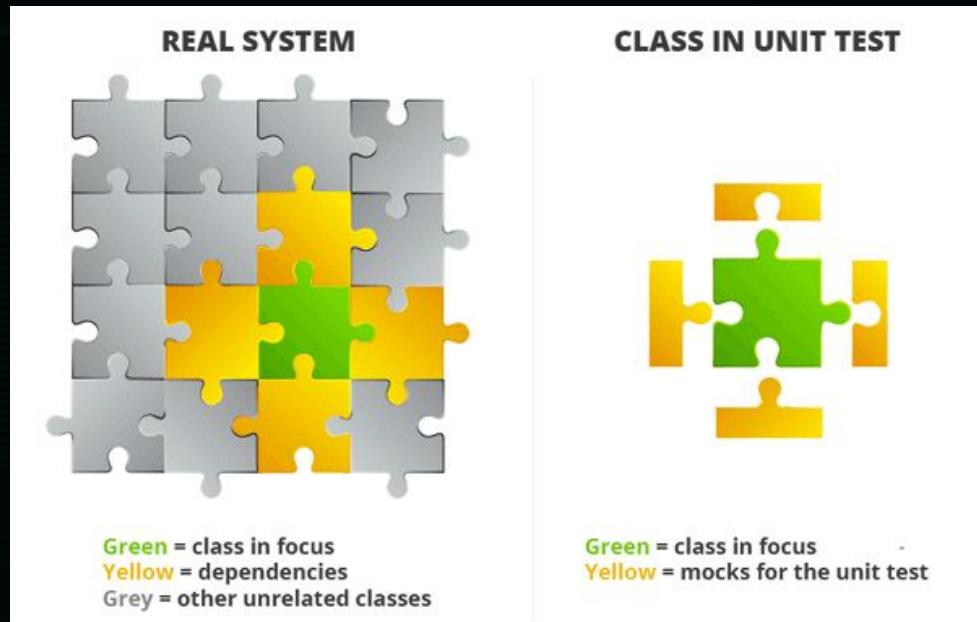
# Mocks

Was sind Mocks?

- Mocks sind Objekte, die das Verhalten von echten Objekten simulieren.
- Sie werden verwendet, um Abhängigkeiten in Unit-Tests zu isolieren.

Arten von Mocks

- Spy
- Dummy
- Stub
- Fake



# Spy Mocks

## Definition

- Ein Spy ist ein Mock-Objekt, das das Verhalten eines echten Objekts überwacht.
- Er speichert Informationen über die Aufrufe, die an ihn gerichtet wurden (z. B. Argumente, Rückgabewerte).

## Verwendung

- Wenn du wissen möchtest, ob eine Methode aufgerufen wurde und mit welchen Argumenten.
- 

```
import { HttpClient } from '@angular/common/http';
import { TestBed } from '@angular/core/testing';

describe('MyService', () => {
  let httpClientSpy: jasmine.SpyObj<HttpClient>;

  beforeEach(() => {
    httpClientSpy = jasmine.createSpyObj('HttpClient', ['get']);
  });

  it('should call get method', () => {
    httpClientSpy.get.and.returnValue(of([]));
    // Test, der den Aufruf überprüft
    expect(httpClientSpy.get).toHaveBeenCalled();
  });
});
```

# Dummy Mocks

## Definition

- Dummy-Objekte sind Platzhalter, die oft keine Funktionalität bieten.
- Sie werden hauptsächlich verwendet, um die Anforderungen eines Konstrukteurs zu erfüllen.

## Verwendung

- Wenn eine Methode oder ein Parameter benötigt wird, aber keine Logik benötigt wird.

```
import { TestBed } from '@angular/core/testing';
import { MyComponent } from './my.component';

describe('MyComponent', () => {
  let component: MyComponent;

  beforeEach(() => {
    component = new MyComponent(null); // Dummy-Service übergeben
  });

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

# Stub Mocks

## Definition

- Stubs sind Objekte, die vordefinierte Antworten auf bestimmte Aufrufe zurückgeben.
- Sie sind nützlich, um die Logik eines Tests zu kontrollieren.

## Verwendung

- Wenn du spezifische Rückgabewerte für Methoden benötigst, um Tests vorhersehbar zu machen.

```
import { DataService } from './data.service';
import { of } from 'rxjs';
import { TestBed } from '@angular/core/testing';

describe('MyComponent', () => {
  let dataServiceStub: Partial<DataService>;

  beforeEach(() => {
    dataServiceStub = {
      loadData: () => of(['Data A', 'Data B'])
    };
  });

  TestBed.configureTestingModule({
    providers: [{ provide: DataService, useValue: dataServiceStub }]
  });
});

it('should load data', () => {
  // Test-Logik, die den Stub verwendet
});
});
```

# Fake Mocks

## Definition

- Fakes sind voll funktionsfähige Implementierungen, die jedoch vereinfacht sind.
- Sie können echte Logik enthalten, aber sind oft einfacher und schneller.

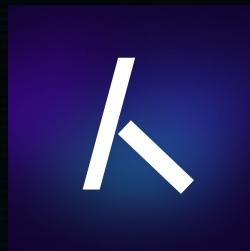
## Verwendung

- Wenn du eine einfache Version einer Klasse benötigst, die echte Logik hat, aber nicht so komplex ist wie das echte Objekt.

```
class FakeDataService {  
    loadData() {  
        return ['Data A', 'Data B']; // Fake-Implementierung  
    }  
  
    describe('MyComponent', () => {  
        let component: MyComponent;  
  
        beforeEach(() => {  
            component = new MyComponent(new FakeDataService());  
        });  
  
        it('should load fake data', () => {  
            expect(component.data).toEqual(['Data A', 'Data B']);  
        });  
    });  
}
```

## Aufgabe

- Finde ein geeignetes Beispiel aus deinem Arbeits- oder Privat-Kontext (oder frei gewählt) und implementiere einen der Mock Varianten für einen Service oder einen Komponente (Spy, Dummy, Stub, Fake)
- Stelle das Beispiel vor und erkläre die Entscheidung, warum du diese Mock Variante für passend hältst.



## **Lab 3: Testgetriebene Entwicklung (TDD)**

## **Was ist Testgetriebene Entwicklung (TDD)**

- In TDD beginnt die Entwicklung immer mit dem Schreiben von Tests. Diese Tests definieren, wie sich der Code verhalten soll.

## Warum Tests vor dem Code schreiben?

- **Klare Anforderungen:** Tests helfen, die Anforderungen zu definieren und Missverständnisse zu vermeiden.
- **Besseres Design:** Tests erzwingen modulare und testbare Code-Strukturen.
- **Frühzeitige Fehlersuche:** Probleme werden entdeckt, bevor der eigentliche Code geschrieben wird.

# Die drei Phasen des TDD-Zyklus

- 
- **Schreiben eines fehlschlagenden Tests:** Der erste Schritt besteht darin, einen Test zu schreiben, der basierend auf den Anforderungen fehlschlägt.
  - **Implementieren des Codes:** Genug Code schreiben, damit der Test erfolgreich besteht.
  - **Refaktorisieren:** Den Code verbessern, ohne dass der Test fehlschlägt.

## TDD Kreislauf in Java

- Phase 1: Test schreiben:

```
@Test  
public void testMultiplication() {  
    Calculator calculator = new Calculator();  
    int result = calculator.multiply(2, 3);  
    assertEquals(6, result);  
}
```

## TDD Kreislauf in Java

- Phase 2: Code implementieren:

```
public int multiply(int a, int b) {  
    return a * b;  
}
```

- Phase 3: Refaktorisieren:

- Den Code optimieren und redundante Elemente entfernen, z.B. durch bessere Namensgebung oder Reduzierung von Duplikationen.

## Wichtige Überlegungen

- **Was soll getestet werden?**: Überprüfe funktionale Anforderungen.
- **Grenzfälle und Fehlerbehandlung**: Tests helfen, auch Randbedingungen zu bedenken.

## Ziele der Refaktorisierung

- **Lesbarkeit:** Bessere Lesbarkeit und Verständlichkeit des Codes.
- **Reduzierung von Komplexität:** Vereinfachung der Code-Struktur.
- **Wartbarkeit:** Erleichterung zukünftiger Änderungen.

## Beispiel - Vor der Refaktorisierung

```
public int add(int a, int b) {  
    if (a == 0) return b;  
    return a + b;  
}
```

## Beispiel - Nach der Refaktorisierung

```
public int add(int a, int b) {  
    return a + b;  
}
```



## **Lab 2: Testen eines Angular-Dienstes**



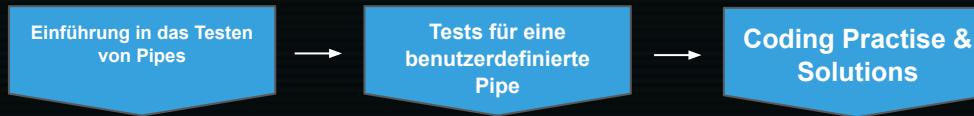
## Lab 3: Coding Practise & Solutions





## Modul 4: Testen von Pipes

# Modul 4 Aufbau





# Lab 1: Einführung in das Testen von Pipes

## Arten von Pipes

- **Reine Pipes:** Funktioniert nur auf Basis ihrer Eingabewerte und hat keine Abhängigkeiten.
- **Unreine Pipes:** Kann durch äußere Faktoren beeinflusst werden (z.B. Benutzerinteraktionen, Datenströme).

## **Was wird beim Testen von Pipes geprüft?**

- Die Korrektheit der Transformation.
- Die Handhabung von Grenzfällen.
- Die Performance der Pipe (bei unreinen Pipes wichtig).

## Grundlegender Ansatz für das Testen von Pipes

- **Unit-Tests für Pipes:** Pipes sollten als reine Funktionen betrachtet und unabhängig getestet werden.

# Teststrategie

- 
- **Initialisierung der Pipe:** Verwende die Pipe-Klasse direkt, um die gewünschte Transformation zu testen.
  - **Validierung der Transformation:** Verifiziere, dass die Pipe das richtige Ergebnis für verschiedene Eingaben liefert.
  - **Testen von Grenzfällen:** Prüfe, wie die Pipe auf ungültige oder unerwartete Eingabewerte reagiert.

## Beispiel einer benutzerdefinierten Pipe

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'capitalize'
})
export class CapitalizePipe implements PipeTransform {
  transform(value: string): string {
    return value.charAt(0).toUpperCase() + value.slice(1);
  }
}
```

# Testfall-Beispiel für die CapitalizePipe

```
import { CapitalizePipe } from './capitalize.pipe';

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

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

  it('should capitalize the first letter', () => {
    expect(pipe.transform('angular')).toBe('Angular');
  });

  it('should return an empty string if input is empty', () => {
    expect(pipe.transform('')).toBe('');
  });

  it('should handle undefined or null values', () => {
    expect(pipe.transform(null)).toBeNull();
  });
});
```

## **Best Practices beim Testen von Pipes**

### ➤ **Testing von reinen Pipes:**

- Reine Pipes hängen nur von den Eingabewerten ab und sollten daher schnell und deterministisch sein.
- Beispiel: Wenn ein Eingabewert sich nicht ändert, sollte das Ausgabeergebnis ebenfalls stabil bleiben.

### ➤ **Testing von unreinen Pipes:**

- Bei unreinen Pipes sollten verschiedene Szenarien berücksichtigt werden, z.B. wenn sich die Abhängigkeiten ändern.

## Was ist das Angular TestBed?

- TestBed ist die primäre API, um Angular-Komponenten, -Pipes, -Services und -Direktiven in einer Testumgebung zu initialisieren und zu testen.
- Ermöglicht das Erstellen und Testen von Angular-Komponenten in einer isolierten Umgebung.

# Struktur eines TestBed-Tests

- 
- **Konfiguration:** Definiere die zu testende Komponente oder Pipe.
  - **Initialisierung:** Setze die Umgebung und initialisiere die Komponente oder Pipe.
  - **Assertions:** Führe die Tests durch und verifiziere die erwarteten Ergebnisse

# Schritte zum Erstellen eines Testbett-Wrappers

- 
- **Deklaration der Pipe in einer Test-Komponente:** Füge die Pipe in eine Test-Komponente ein.
  - **Verwendung von TestBed:** Initialisiere die Komponente und prüfe, ob die Pipe korrekt funktioniert.
  - **Testumgebung aufsetzen:** Simuliere verschiedene Eingaben und prüfe die Ausgaben.

# Beispiel für einen TestBed-Wrapper

```
import { Component } from '@angular/core';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { CapitalizePipe } from './capitalize.pipe';

@Component({
  template: `{{ 'angular' | capitalize }}`
})
class TestComponent {}

describe('CapitalizePipe in TestBed', () => {
  let fixture: ComponentFixture<TestComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [TestComponent, CapitalizePipe]
    });
    fixture = TestBed.createComponent(TestComponent);
    fixture.detectChanges();
  });

  it('should capitalize "angular" to "Angular"', () => {
    const compiled = fixture.nativeElement;
    expect(compiled.textContent).toBe('Angular');
  });
});
```



## Lab 2: Tests für eine benutzerdefinierte Pipe



## Lab 3: Coding Practise & Solutions

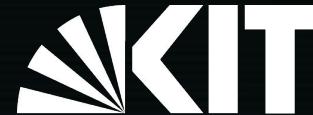




Trusted by great Companies



Hochschule Fulda  
University of Applied Sciences



# Vielen Dank für Ihre Teilnahme!

Ihr Referent & und das Team von  
Kodschul.com

**Kodschul.com**

Nextise GmbH  
Am Klingenbach 33  
70188 Stuttgart  
Web. <https://kodschul.com/>  
Mail. [franz@nextise.de](mailto:franz@nextise.de)