Testing in Spring Boot

Michael Inden

Freiberuflicher Consultant und Trainer

Speaker Intro

- Michael Inden, Jahrgang 1971
- Diplom-Informatiker, C.v.O. Uni Oldenburg

- ~8 ¼ Jahre **SSE** bei Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Jahre **TPL, SA** bei IVU Traffic Technologies AG in Aachen
- ~4 ¼ Jahre **LSA / Trainer** bei Zühlke Engineering AG in Zürich
- ~3 Jahre **TL / CTO** bei Direct Mail Informatics / ASMIQ in Zürich
- **Freiberuflicher Consultant, Trainer und Konferenz-Speaker**
- Autor und Gutachter beim dpunkt.verlag

E-Mail: michael_inden@hotmail.com
Blog: https://jaxenter.de/author/minden

Kurse: Bitte sprecht mich an!

https://github.com/Michaeli71/Spring_Intro_3d

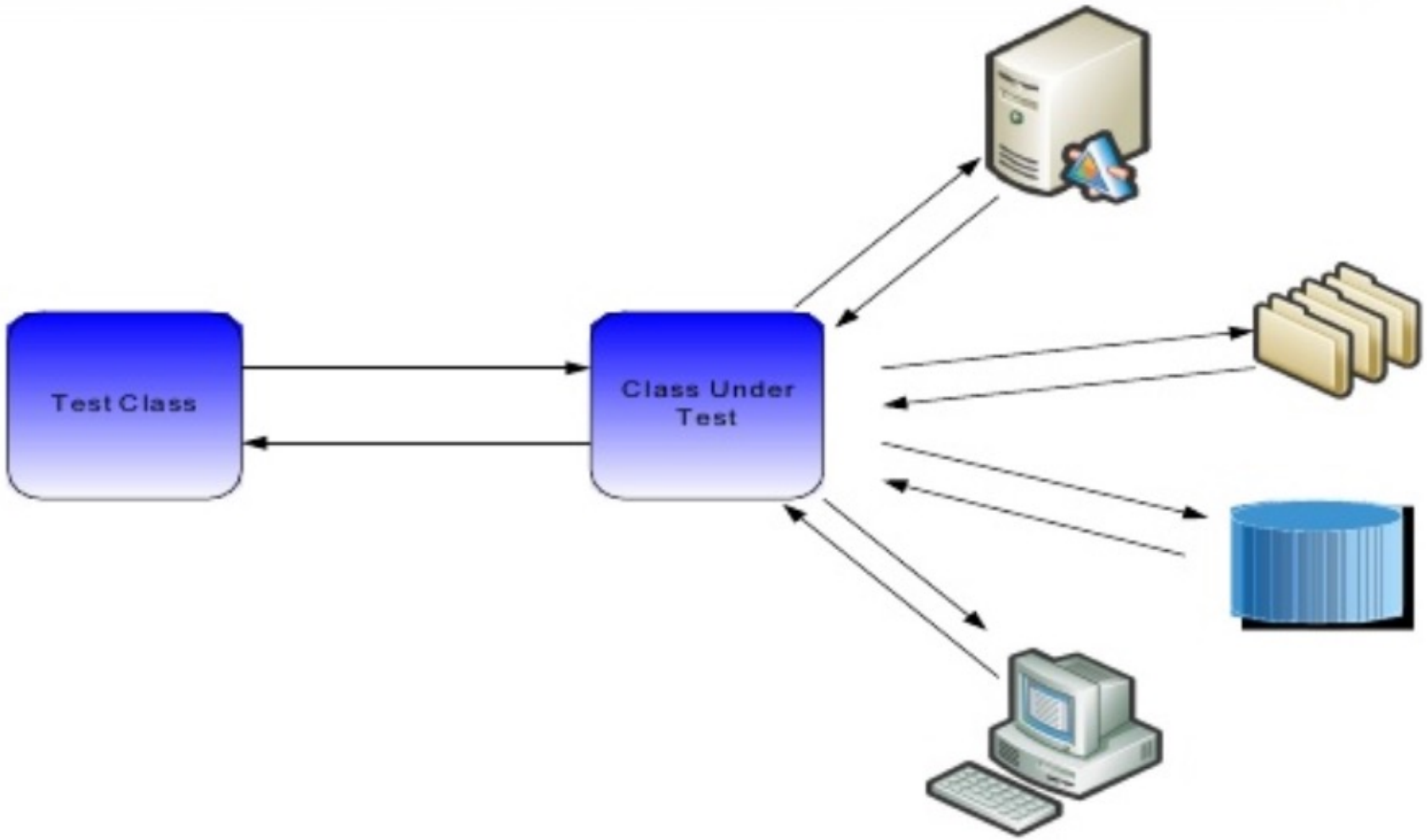
Agenda

Workshop Contents

- RECAP: Motivation für Mocking & Intro REST Assured
- PART 1: Testen mit Spring Boot
- PART 2: Slice Based Testing

RECAP
Motivation für Mocking

Zu testende Klasse mit vielen Abhängigkeiten



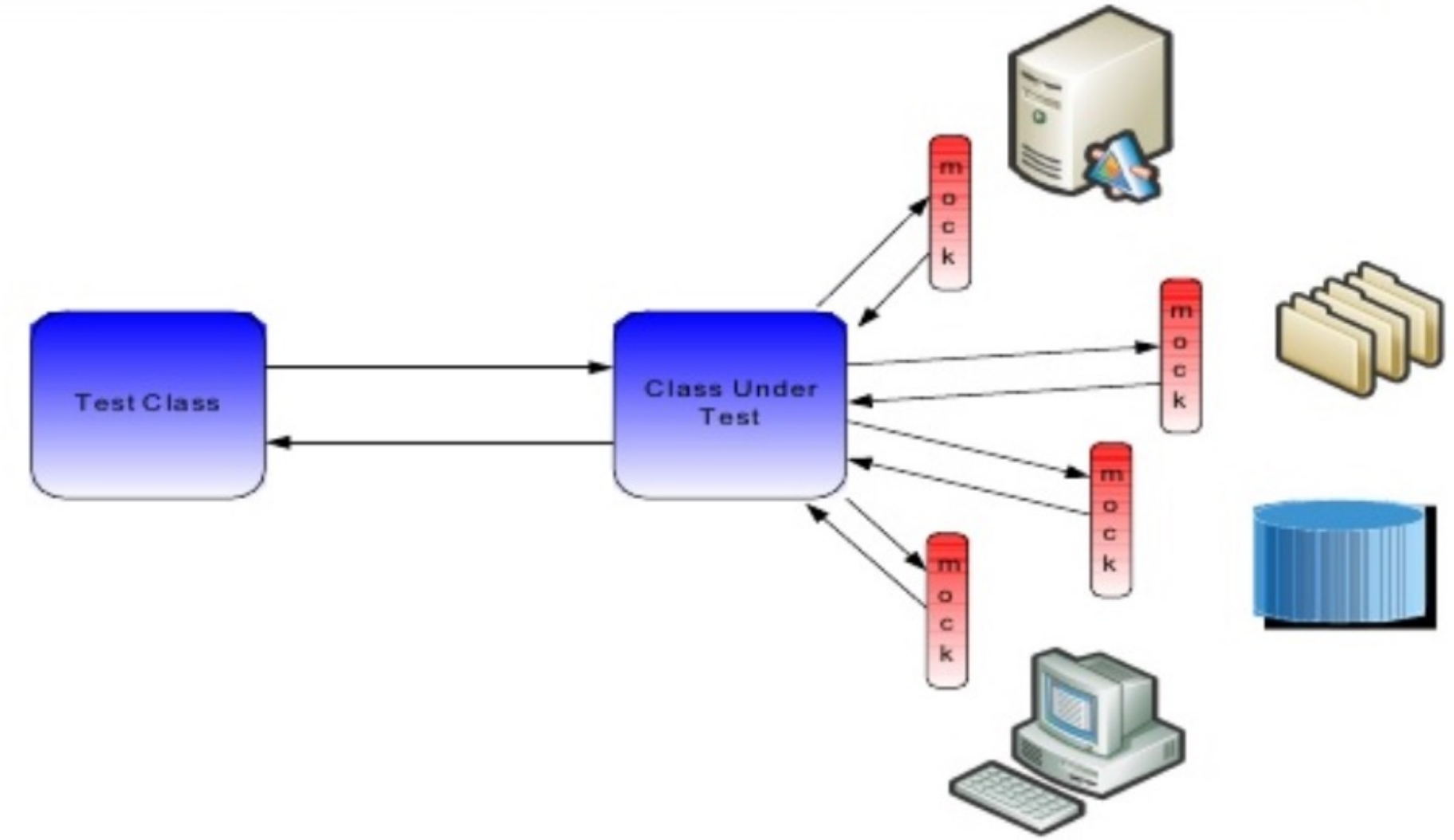
https://www.slideshare.net/Gemsbond/easymock-for-java

Beispiel

```
public class IncomeCalculator
{
    private final SalaryService salaryService;

    public IncomeCalculator(final SalaryService salaryService)
    {
        this.salaryService = salaryService;
    }

    public int getIncomePerMonth(final Position position)
    {
        return salaryService.salaryForPosition(position) / 12;
    }
}
```

Wie schreiben wir einen Test ohne auf den externen Salary Service angewiesen zu sein?

Abhängigkeiten mit Mocks im Griff



https://www.slideshare.net/Gemsbond/easymock-for-java

Einschub REST Assured und Demo-Webserver «Req Res» & «Ergast» & «http://api.zippopotam.us»

REST Assured

- REST Assured als DSL für lesbare REST-API-Tests
- Folgt dem AAA / GWT-Stil (BDD-Stil) mit Given / When / Then
- Gute lesbar und verständlich
- Fluent API, dadurch in einem Rutsch die gesamte Verarbeitung spezifizierbar

```
@Test
public void makeSureThatGoogleIsUp()
{
    given().
    when().get("http://www.google.com").
    then().statusCode(200);
}
```

REST Assured Maven Dependency

```xml
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <version>4.4.0</version>
    <scope>test</scope>
</dependency>


<!-- Ab Java 9 -->
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured-all</artifactId>
    <version>4.4.0</version>
    <scope>test</scope>
</dependency>
```

https://reqres.in/

REQ | RES

Test your front-end against a real API

Fake data

No more tedious sample data creation, we've got it covered.

Real responses

Develop with real response codes. GET, POST, PUT & DELETE supported.

Always-on

24/7 *free* access in your development phases. Go nuts.

A hosted REST-API ready to respond to your AJAX requests.

Beispiel für POST und DELETE

```
@Test
public void test_post()
{
    JSONObject request = new JSONObject();
    request.put("name", "Michael");
    request.put("job", "Trainer");

    given().body(request.toJSONString()).
    when().post("https://reqres.in/api/users").
    then().statusCode(201);
}

@Test
public void test_delete()
{
    given().
    when().delete("https://reqres.in/api/users/5").
    then().statusCode(204);
}
```

http://ergast.com/mrd/

Ergast Developer API

API Documentation

The Ergast Developer API is an experimental web service which provides a historical record of motor racing data for non-commercial purposes. Please read the terms and conditions of use. The API provides data for the Formula One series, from the beginning of the world championships in 1950.

Non-programmers can query the database using the manual interface or download the database tables in CSV format for import into spreadsheets or analysis software.

If you have any comments or suggestions please post them on the Feedback page. If you find any bugs or errors in the data please report them on the Bug Reports page. Any enhancements to the API will be reported on the News page. Example applications are shown in the Application Gallery.

Overview

All API queries require a GET request using a URL of the form:

```
http[s]://ergast.com/api/<series>/<season>/<round>/...
```

where:

`<series>`	should be set to "f1"
`<season>`	is a 4 digit integer
`<round>`	is a 1 or 2 digit integer

Lesbare Abfragen ohne Parametrierung

```
@Test
public void test_NumberOfCircuitsFor2019Season_ShouldBe21() {

    given().
    when().
        get("http://ergast.com/api/f1/2019/circuits.json").
    then().
        assertThat().
        body("MRData.CircuitTable.Circuits.circuitId", hasSize(21));
}
```

Lesbare Abfragen mit Parametrierung und als ParameterizedTest

```
@ParameterizedTest
@CsvSource({"2017,20", "2019, 21"})
public void test_NumberOfCircuits_Parameterized(String season, int numberOfRaces)
{
    given().
        pathParam("raceSeason", season).
    when().
        get("http://ergast.com/api/f1/{raceSeason}/circuits.json").
    then().
        assertThat().
        body("MRData.CircuitTable.Circuits.circuitId", hasSize(numberOfRaces));
}
```

http://api.zippopotam.us

Zippopotam.us Samples Countries Who Code Uptime

New Management The Zippopotam.us project is under new management. View the Github Repository for details. ×

Zip Code Galore!

Zip·po·pot·amus /ˈzipōpätəməs/

Postal Codes and Zip Codes made easy

- Free API with JSON Response Format
- Over 60 Countries Supported
- Perfect for Form Autocompletion
- Open for Crowdsourcing and Contribution

Try It Out »

Structure: api.zippopotam.us/country/postal-code
Example: api.zippopotam.us/us/90210
NEW! City->Zip: api.zippopotam.us/country/state/city
Example: api.zippopotam.us/us/ma/belmont

Autocomplete Examples: USA Germany Spain France

Lesbare Abfragen mit Verknüpfung und Zugriff auf Body

```
import static io.restassured.RestAssured.*;
import static io.restassured.matcher.RestAssuredMatchers.*;
import static org.hamcrest.Matchers.*;


@Test
public void checkContentTypeAndLog()
{
    given().
        pathParam("country","ch").
        pathParam("zipcode","8047").
    when().
        get("http://api.zippopotam.us/{country}/{zipcode}").
    then().
        assertThat().statusCode(200).
        body("places.'place name'[0]", equalTo("Zürich")).
     and().
        contentType(ContentType.JSON).
        log().all();
}
```

Ergebnis der ZIP-Abfrage


```
{
    "post code": "8047",
    "country": "Switzerland",
    "country abbreviation": "CH",
    "places": [
        {
            "place name": "Zürich",
            "longitude": "8.487",
            "state": "Kanton Zürich",
            "state abbreviation": "ZH",
            "latitude": "47.3739"
        }
    ]
}
```

Prof of Concept
HTTP/2-API statt REST Assured

WireMock Prüfen mit JUnit und HTTP/2-API

```
@Test
public void testAnEndpoint_ShouldReturn_StatusCode_200_Http2_API() throws Exception
{
    setupStub();

    HttpResponse<String> result = performGetCall("/an/endpoint", "");

    assertEquals(200, result.statusCode());
}
```

HTTP/2-API aus Java 11 zum Request senden (statt REST Assured)

```
static HttpResponse<String> performGetCall(String path, String contentType)
                            throws IOException, InterruptedException
{
    if (contentType == null || contentType.isEmpty())
        contentType = "text/plain";

    HttpClient client = HttpClient.newHttpClient();
    HttpRequest request = HttpRequest.newBuilder()
                          .uri(URI.create("http://localhost:8089" + path))
                          .header("Content-Type", contentType)
                          .build();

    return client.send(request, BodyHandlers.ofString());
}
```

HTTP/2-API aus Java 11 für POST-Request (statt REST Assured)

```
static HttpResponse<String> performPostWithXmlBody(String path, String content)
                            throws IOException, InterruptedException
{
    HttpClient client = HttpClient.newHttpClient();
    HttpRequest request = HttpRequest.newBuilder()
                          .uri(URI.create("http://localhost:8089" + path))
                          .POST(HttpRequest.BodyPublishers.ofString(content))
                          .header("Content-Type", "text/xml")
                          .header("Accept", "text/xml")
                          .build();

    return client.send(request, BodyHandlers.ofString());
}
```

PART 1:
Testen mit Spring Boot

Testen und Spring

- **Das Spring Framework ermöglicht es, lose gekoppelte Komponenten zu erstellen.**
- **Spring bietet zudem verschiedene Tools, um Unit- und Integrationstests zu erleichtern**
- **Idealerweise sollte dabei Plain JUnit möglich sein**
- **Spring ermöglicht es, auf einfache Weise den `ApplicationContext` in Tests zu nutzen**
  - JUnit 4: `@RunWith(SpringRunner.class)`
  - JUnit 5: `@ExtendWith(SpringExtension.class)`, oft nicht nötig, da Bestandteil von `@SpringBootTest`

Testen und Spring

- **Hinweis aus der Spring Documentation**

> Dependency injection should make your code less dependent on the container than it would be with traditional Java EE development. The POJOs that make up your application should be testable in JUnit or TestNG tests, with objects instantiated by using the new operator, without Spring or any other container.

- **Auswirkungen auf die Laufzeit von Tests**
  - Integration Test dauern alleine durch das Hochfahren des Containers mehrere Sekunden
  - In Unit Tests verwenden wir Mocks statt der tatsächlichen Objekte, etwa Repositories. Die Tests laufen dadurch oftmals lediglich in Millisekunden statt in Sekunden.

RECAP: Typischer REST Controller

- **Schauen wir zunächst nochmal auf einen typischen REST Controller**

```
@RestController
class RegisterRestController {
  private final RegisterUseCase registerUseCase;

  @PostMapping("/forums/{forumId}/register")
  UserResource register(
          @PathVariable("forumId") Long forumId,
          @Valid @RequestBody UserResource userResource,
          @RequestParam("sendWelcomeMail") boolean sendWelcomeMail) {

    User user = new User(
            userResource.getName(),
            userResource.getEmail());
    Long userId = registerUseCase.registerUser(user, sendWelcomeMail);

    return new UserResource(userId, user.getName(), user.getEmail());
  }
}
```

RECAP: Typischer REST Controller

- Die Methoden im Controller sind annotiert mit etwa `@PostMapping`, um die URL, HTTP Verb usw. zu mappen.
- Eingaben können über normale Parameter entgegengenommen werden, auch wieder über Annotation in deren Ausprägung gesteuert `@PathVariable`, `@RequestBody` und `@RequestParam` und automatisch aus dem HTTP-Request befüllt
- Es kann sogar Validierung kann mit @Valid aktiviert werden
- Der Controller nutzt dann in der Regel Services oder andere Businesslogik und als Rückgabe entsteht ein POJO, welches automatisch in JSON gewandelt und dann in den Body der Response geschrieben wird

Was passiert an Magie unter der Motorhaube?

#	Responsibility	Description
1.	**Listen to HTTP Requests**	The controller should respond to certain URLs, HTTP methods and content types.
2.	**Deserialize Input**	The controller should parse the incoming HTTP request and create Java objects from variables in the URL, HTTP request parameters and the request body so that we can work with them in the code.
3.	**Validate Input**	The controller is the first line of defense against bad input, so it's a place where we can validate the input.
4.	**Call the Business Logic**	Having parsed the input, the controller must transform the input into the model expected by the business logic and pass it on to the business logic.
5.	**Serialize the Output**	The controller takes the output of the business logic and serializes it into an HTTP response.
6.	**Translate Exceptions**	If an exception occurs somewhere on the way, the controller should translate it into a meaningful error message and HTTP status for the user.

https://reflectoring.io/spring-boot-web-controller-test/

Was folgt eigentlich zwangsläufig daraus?

Ein Controller hat ohnehin schon offensichtlich eine Menge zu tun!

1. Wir sollten darauf achten, dass wir nicht noch mehr Aufgaben hinzufügen! Manchmal sieht man leider trotzdem dort die Ausführung von Geschäftslogik. Das lässt sich oftmals sehr gut refactorisieren => später in einer Übung (Stichwort: Separation of Concerns)

2. Beachtet man dies nicht, wird der Controller immer schwieriger und aufwendiger zu testen.

Was bedeutet das alles für das Testing und welche Form sollten wir wählen?

Unit vs. Integration Testing in Spring

1. Endpoints sind mit @PostMapping und @GetMapping usw. annotiert. In Unit-Tests werden diese Annotationen nicht verarbeitet. Woher wissen wir, dass eine HTTP-Anfrage auf die richtigen Endpunkte abgebildet wird oder dass Pfadvariablen mit @PathVariable abgebildet werden?
2. Was ist mit @RequestBody und @ResponseBody? Ein Unit-Test verarbeitet auch diese Annotationen nicht. Wird die Eingabe korrekt deserialisiert?
3. Sobald die Eingabe deserialisiert wurde, werden mit @Valid annotiere Parameter validiert. Woher wissen wir, dass die Validierung tatsächlich stattgefunden hat?
4. Sobald der Controller Businesslogik aufgerufen hat, wird er einige Antworten an den Aufrufer zurückgeben. Woher wissen wir, dass unsere Ausgabe korrekt in JSON serialisiert wurde?
5. Exception-Handler können ausgelöste Ausnahmen behandeln und in einen HTTP-Statuscode umwandeln. Die Umwandlung könnten wir testen, aber woher wissen wir, dass die Methode aufgerufen wird, wenn eine Ausnahme ausgelöst wird?

Unit vs. Integration Testing in Spring

In einem Unit-Test wird der Controller isoliert getestet, indem ein Controller instanziiert wird, die Geschäftslogik weggemockt wird und dann die Methoden des Controllers aufgerufen und die Ergebnisse überprüft werden.

#	Responsibility	Covered in a Unit Test?
1.	**Listen to HTTP Requests**	✖ No, because the unit test would not evaluate the `@PostMapping` annotation and similar annotations specifying the properties of a HTTP request.
2.	**Deserialize Input**	✖ No, because annotations like `@RequestParam` and `@PathVariable` would not be evaluated. Instead we would provide the input as Java objects, effectively skipping deserialization from an HTTP request.
3.	**Validate Input**	✖ Not when depending on bean validation, because the `@Valid` annotation would not be evaluated.
4.	**Call the Business Logic**	✔ Yes, because we can verify if the mocked business logic has been called with the expected arguments.
5.	**Serialize the Output**	✖ No, because we can only verify the Java version of the output, and not the HTTP response that would be generated.
6.	**Translate Exceptions**	✖ No. We could check if a certain exception was raised, but not that it was translated to a certain JSON response or HTTP status code.

https://reflectoring.io/spring-boot-web-controller-test/

Integration Testing in Spring

- **Demnach gibt es mindestens fünf verschiedene Dinge, die von den Unit-Tests nicht überprüft werden können:**

    1. Zuordnung der HTTP-Requests
    2. Deserialisierung
    3. Validierung von Eingabefeldern
    4. Serialisierung
    5. Fehlerbehandlung

- **In einer Spring-Anwendung kümmert sich das Framework um die oben genannten Belange, indem Spring alle erforderlichen Beans in den Anwendungskontext erstellt.**
- **Um auch diese Fälle zu testen, müssen wir Spring in unsere Tests nutzen. Das bedeutet, dass wir Integrationstests schreiben.**

Unit Testen mit Spring Boot

Unit Testen in Spring Boot

- **Wie wir Klassen isoliert testen, haben wir zuvor mit JUnit gesehen**
- **Hier nun im Kontext von Spring Test und Spring Boot**
- **`'org.springframework.boot:spring-boot-starter-test'` bindet automatisch unter anderem folgende Test-Tools/-Bibliotheken ein:**
  - AssertJ
  - Mockito
- **ABER: Don’t Use Spring in Unit Tests**

Unit Testen in Spring Boot

`spring-boot-starter-test` inkludiert tatsächlich insgesamt Folgendes:

- JUnit
- JSON PATH
- AssertJ
- Mockito
- Hamcrest
- JSONAssert
- SpringTest
- SpringBootTest

Bei Bedarf folgende Add-ons:

- HTMUnit
- Selenium
- H2
- Flapdoodle (Embedded MongoDB)
- ...

Unit Testen in Spring Boot

- **Creating a Testable Spring Bean: Field Injection vermeiden**

```
@Service
public class RegisterUseCase {

    @Autowired
    private ToDoRepository todoRepository;

    public ToDo createTodo(ToDo newTodo) {
        return todoRepository.save(newTodo);
    }
}
```

- **Diese Klasse kann nicht ohne Spring getestet werden, da sie keine Möglichkeit bietet, eine `ToDoRepository`-Instanz zu übergeben.**

Unit Testen in Spring Boot

- **Creating a Testable Spring Bean: Bevorzuge Constructor / Setter Injection**

```
@Service
public class NewToDoUseCaseV2 {

    private final ToDoRepository todoRepository;

    public NewToDoUseCaseV2(ToDoRepository todoRepository) {
        this.todoRepository = todoRepository;
    }

    public ToDo createTodo(ToDo newTodo) {
        return todoRepository.save(newTodo);
    }
}
```

- **Mit dieser einfachen Änderung wird es möglich, zugehörige Tests unabhängig von Spring zu machen … schauen wir uns das schrittweise an!**

Unit Testen in Spring Boot

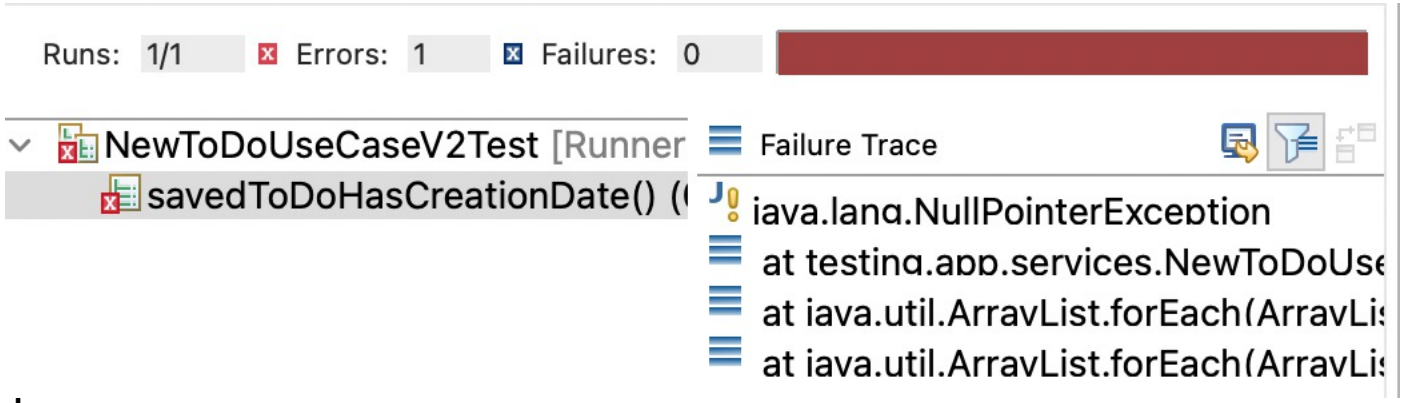
- **Unit Test ohne Spring**

```
public class NewToDoUseCaseV2Test {

    private ToDoRepository todoRepository = /*...*/;

    private NewToDoUseCaseV2 newToDoUseCaseV2;

    @BeforeEach
    void initUseCase() {
        newToDoUseCaseV2 = new NewToDoUseCaseV2(todoRepository);
    }

    @Test
    void savedUserHasRegistrationDate() {
        ToDo toDo = new ToDo("Solve Exercises");
        ToDo savedToDo = newToDoUseCaseV2.createTodo(toDo);
        assertThat(savedToDo.getCreatedAt()).isNotNull();
    }
}
```

Nun fehlt nur noch eine Sache: Wie werden wir unabhängig vom konkreten Repository und der DB?

Unit Testen in Spring Boot

```
public class NewToDoUseCaseV2Test {

    private ToDoRepository todoRepository = Mockito.mock(ToDoRepository.class);

    private NewToDoUseCaseV2 newToDoUseCaseV2;

    @BeforeEach
    void initUseCase() {
        newToDoUseCaseV2 = new NewTo
    }

    @Test
    void savedToDoHasCreationDate() {
        ToDo toDo = new ToDo("Solve Exercises");

        ToDo savedToDo = newToDoUseCaseV2.createTodo(toDo);

        assertThat(savedToDo.getCreatedAt()).isNotNull();
    }
}
```



Unit Testen in Spring Boot

```
public class NewToDoUseCaseV2Test {

    private ToDoRepository todoRepository = Mockito.mock(ToDoRepository.class);
    private NewToDoUseCaseV2 newToDoUseCaseV2;

    @BeforeEach
    void initUseCase() {
        newToDoUseCaseV2 = new NewToDoUseCaseV2(todoRepository);
        when(todoRepository.save(Mockito.any(ToDo.class))).
        then(AdditionalAnswers.returnsFirstArg());
    }

    @Test
    void savedToDoHasCreationDate() {
        ToDo toDo = new ToDo("Solve Exercises");

        ToDo savedToDo = newToDoUseCaseV2.createTodo(toDo);

        assertThat(savedToDo.getCreatedAt()).isNotNull();
    }
}
```

Unit Testen in Spring Boot

```java
@ExtendWith(MockitoExtension.class)
public class NewToDoUseCaseV2AlternativeTest {

    @Mock
    private ToDoRepository todoRepository;
    @InjectMocks
    private NewToDoUseCaseV2 newToDoUseCaseV2;

    @BeforeEach
    void initUseCase() {
        newToDoUseCaseV2 = new NewToDoUseCaseV2(todoRepository);
        when(todoRepository.save(Mockito.any(ToDo.class))).
        then(AdditionalAnswers.returnsFirstArg());
    }

    @Test
    void savedToDoHasCreationDate() {
        ToDo toDo = new ToDo("Solve Exercises");

        ToDo savedToDo = newToDoUseCaseV2.createTodo(toDo);

        assertThat(savedToDo.getCreatedAt()).isNotNull();
    }
}
```

Kann man die Assertions noch lesbarer machen?

Ausflug: Eigene Asserts

```
public class NewToDoUseCaseV2ImprovedTest {

    private ToDoRepository todoRepository = Mockito.mock(ToDoRepository.class);
    private NewToDoUseCaseV2 newToDoUseCaseV2;

    @BeforeEach
    void initUseCase() {
        newToDoUseCaseV2 = new NewToDoUseCaseV2(todoRepository);
        when(todoRepository.save(Mockito.any(ToDo.class))).
        then(AdditionalAnswers.returnsFirstArg());
    }

    @Test
    void savedToDoHasCreationDate() {
        ToDo toDo = new ToDo("Solve Exercises");

        ToDo savedToDo = newToDoUseCaseV2.createTodo(toDo);

        ToDoAssert.assertThat(savedToDo).hasCreationDate();
    }
}
```

Ausflug: Eigene Asserts

```
class ToDoAssert extends AbstractAssert<ToDoAssert, ToDo> {

    ToDoAssert(ToDo toDo) {
        super(toDo, ToDoAssert.class);
    }

    static ToDoAssert assertThat(ToDo actual) {
        return new ToDoAssert(actual);
    }

    public ToDoAssert hasCreationDate() {
        isNotNull();
        if (actual.getCreatedAt() == null) {
            failWithMessage("Expected ToDo to have a creation date, but it was
null");
        }
        return this;
    }
}
```

RECAP: Unit Testen in Spring Boot

- **Durch Constructor / Setter Injection ist es möglich, zugehörige Tests unabhängig von Spring zu machen und PROBLEMLOS ohne Spring zu testen**

```
@Service
public class NewToDoUseCaseV2 {

    private final ToDoRepository todoRepository;

    public NewToDoUseCaseV2(ToDoRepository todoRepository) {
        this.todoRepository = todoRepository;
    }

    public ToDo createTodo(ToDo newTodo) {
        return todoRepository.save(newTodo);
    }
}
```

- Tests lassen sich durch Injektion von Mocks jetzt schnell und isoliert ausführen.

D E M O

Integration Testing mit Spring Boot

Beispiel: `https://spring.io/guides/gs/testing-web/`

- Einfache Web-Applikation basierend auf Getting Started Guide
- Wie wir Klassen insoliert testen, haben wir zuvor gesehen
- Hier nun wirklich mit Spring Test und Spring Boot
- Zunächst ein einfacher Test, dass der `ApplicationContext` erfolgreich geladen wird
- Danach nur die Webebene testen, indem man Spring's MockMvc und andere verwendet.

Beispiel: Simple Hello World Controller

```java
@SpringBootApplication
public class SpringBootFirstTestWebApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBootFirstTestWebApplication.class, args);
    }
}
```

```java
@RestController
public class SimpleHelloWorldController {

    @GetMapping("/")
    public String greeting() {
        return "Hello, World";
    }
}
```

Beispiel: SmokeTest

- **Wir wollen nur testen, ob die Applikation hochfährt (ohne dass wir es im Browser prüfen)**

```
import static org.assertj.core.api.Assertions.assertThat;
…
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
public class SmokeTest {

    @Autowired
    private SimpleHelloWorldController controller;

    @Test
    public void appContextCorrectlyLoaded() {
        assertThat(controller).isNotNull();
    }
}
```

Beispiel: Rest Controller Test V1

- `TestRestTemplate` mit zufälligem Port und Injektion mit `@LocalServerPort`

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class HttpRequestTest {

    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void greetingShouldReturnDefaultMessage() {
        String contextPath = "http://localhost:" + port + "/";
        String resultOfHttpGet = restTemplate.getForObject(contextPath,
                                                           String.class);

        assertThat(resultOfHttpGet).contains("Hello, World");
    }
}
```

Beispiel: Rest Controller Test V2

- RestAssured mit zufälligem Port und Injektion mit @LocalServerPort

```
import io.restassured.RestAssured;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class HttpRequestTestV2 {

    @LocalServerPort
    private int port;

    @Test
    public void greetingShouldReturnDefaultMessage() {
        RestAssured.given().
        when().get("http://localhost:" + port + "/").
        then().assertThat().statusCode(200).and().
                              toString().contains("Hello, World");
    }
}
```

Beispiel: Rest Controller Test V2

- **RestAssured notwendige Dependencies**

```
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured-all</artifactId>
    <version>4.4.0</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>spring-mock-mvc</artifactId>
    <version>4.4.0</version>
    <scope>test</scope>
</dependency>
```

Beispiel: Rest Controller Test V3

- **MockMvc versteckt Port …**

```
@SpringBootTest
@AutoConfigureMockMvc
public class HttpRequestTestV3 {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void greetingShouldReturnDefaultMessage() throws Exception {
        mockMvc.perform(get("/")).
                // andDo(print()).
                andExpect(status().isOk()).
                andExpect(content().string(containsString("Hello, World")));
    }
}
```

Beispiel: Rest Controller Test V3

- **`MockMvc` bietet durch `@AutoConfigureMockMvc` leichtgewichtigen Ansatz:**
  - Server wird ***nicht*** gestartet, sondern nur die darunter liegende Schicht, in der Spring die eingehende HTTP-Anfrage verarbeitet und an Controller weitergibt.
  - So wird fast der gesamte Stack verwendet, und der Code wird genauso aufgerufen, als würde er eine echte HTTP-Anfrage verarbeiten, ohne dass der Server gestartet werden muss.
- **RECAP: Hierbei wird der gesamte Spring-Anwendungskontext gestartet, jedoch ohne den Server.**
- **Später Slice-based Testing: Mithilfe von `@WebMvcTest` können wir die Tests sogar nur auf die Webebene beschränken!**

D E M O

Beispiel: Controller und Service in Kombination

```java
@RestController
public class GreetingController {

    private final GreetingService service;

    public GreetingController(GreetingService service) {
        this.service = service;
    }

    @GetMapping("/greeting")
    public String greeting() {
        return service.generateGreeting();
    }
}

@Service
public class GreetingService {
    public String generateGreeting() {
        return "Hello, World";
    }
}
```

Spring injiziert die Service-Abhängigkeit automatisch in den Controller (aufgrund der Konstruktorsignatur).

Was ist daran für das Testen unschön?

Beispiel: Controller und Service in Kombination

```
@SpringBootTest
@AutoConfigureMockMvc
public class GreetingControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private GreetingService service;

    @Test
    public void greetingShouldReturnDefaultMessage() throws Exception {
        when(service.generateGreeting()).thenReturn("CHANGED FOR TESTING");

        mockMvc.perform(get("/greeting")).
                // andDo(print()).
                andExpect(status().isOk()).
                andExpect(content().string(containsString("CHANGED FOR TESTING")));
    }
}
```

Spring injiziert den Service-MOCK in den ApplicationContext

PART 2:
Slice Based Testing

Slice Based Testing

- **Slice Based Testing ermöglicht das Testen ohne die Instanziierung diverser Infrastruktur-Komponenten:**
  - @WebMvcTest – Tests der Controller ohne kompletten Server, Beans mit @Component werden nicht gescanned => @MockBean als Abhilfe
  - @DataJpaTest – Tests der DAO-Schicht mit in-Memory-DB und JPA inklusive TestEntityManager und scanned vor allem @Entity
  - @JdbcTest -- Tests der DAO-Schicht mit in-Memory-DB und JdbcTemplate
  - @JsonTest – verwendet nur die entsprechenden Mapper nichts sonst
  - @RestClientTest – Tests von Rest Clients
  - …

Testen mit WebMvcTest

Slice Based Testing: `WebMvcTest`

- **`WebMvcTest` ermöglicht Slice Based Testing**

```
@WebMvcTest
public class HttpRequestTestV3 {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void greetingShouldReturnDefaultMessage() throws Exception {
        mockMvc.perform(get("/")).
                // andDo(print()).
                andExpect(status().isOk()).
                andExpect(content().string(containsString("Hello, World")));
    }
}
```

Slice Based Testing

- **Bei mehreren Controllern kann man sogar nur einen (bzw. genauer: den gewünschten) instanziieren**

```
@WebMvcTest(SimpleHelloWorldController.class)
public class HttpRequestTestV3 {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void greetingShouldReturnDefaultMessage() throws Exception {
        mockMvc.perform(get("/")).
                // andDo(print()).
                andExpect(status().isOk()).
                andExpect(content().string(containsString("Hello, World")));
    }
}
```

Beispiel: Rest Controller Test V3

- **MockMvc ermöglicht REST Calls**

```
@WebMvcTest(SimpleHelloWorldController.class)
public class HttpRequestTestV3 {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void greetingShouldReturnDefaultMessage() throws Exception {
        mockMvc.perform(get("/")).
                // andDo(print()).
                andExpect(status().isOk()).
                andExpect(content().string(containsString("Hello, World")));
    }
}
```

Testen mit DataJpaTest

Slice Based Testing: `DataJpaTest`

- **`DataJpaTest` ermöglicht Slice Based Testing zwischen unserem Code und der Datenbank.**

```
@DataJpaTest
class DbAcessComponentsInitTest {

    @Autowired
    private DataSource dataSource;
    @Autowired
    private JdbcTemplate jdbcTemplate;
    @Autowired
    private EntityManager entityManager;

    @Test
    void injectedComponentsAreNotNull() {
        assertAll(() -> assertNotNull(dataSource),
                  () -> assertNotNull(jdbcTemplate),
                  () -> assertNotNull(entityManager);
    }
}
```

Slice Based Testing

- `DataJpaTest` ermöglicht Slice Based Testing zwischen unserem Code und der Datenbank.

- Der so erstellte Anwendungskontext enthält nicht den gesamten Kontext, der für unsere Spring Boot-Anwendung benötigt wird, sondern nur einen "Ausschnitt" davon!

- Dieser «Slice» enthält die Komponenten, die für die Initialisierung aller JPA-bezogenen Komponenten als auch für unsere Spring Data-Repositories benötigt werden.

Slice Based Testing

- `DataJpaTest` ermöglicht Slice Based Testing zwischen unserem Code und der Datenbank.

```
@DataJpaTest
public class SliceBasedDataJpaTest {

    @Autowired
    private TestEntityManager em;

    @Autowired
    private ToDoRepository repository;

    @Test
    public void todoDataTest() throws Exception {
        em.persist(new ToDo("Mail @ Anne"));

        Collection<ToDo> todos = repository.findByDescription("Mail @ Anne");
        assertThat(todos).contains(new ToDo("Mail @ Anne"));
    }
}
```

Slice Based Testing: `DataJpaTest`

- **`DataJpaTest` ermöglicht Slice Based Testing zwischen unserem Code und der Datenbank.**

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

```
<dependency>
<groupId>com.h2database</groupId>
<artifactId>h2</artifactId>
<scope>runtime</scope>
</dependency>
```

Testen mit JsonTest

Slice Based Testing: `JsonTest`

- **`JsonTest` ermöglicht Slice Based Testing des JSON (Un)-Marshallings**

```
@JsonNaming(PropertyNamingStrategies.LowerCaseStrategy.class)
public class UserDetails {

    private Long id;
    private String firstName;
    private String lastName;

    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd.MM.yyyy")
    private LocalDate dateOfBirth;

    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private boolean enabled;

    public UserDetails(Long id, String firstName, String lastName,
                       LocalDate dateOfBirth, boolean enabled) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.dateOfBirth = dateOfBirth;
        this.enabled = enabled;
    }
```

Slice Based Testing: `JsonTest`

- **`JsonTest` ermöglicht Slice Based Testing des JSON (Un)-Marshallings**

```
@JsonTest
public class UserDetailsJsonTest {

    @Autowired
    private JacksonTester<UserDetails> jsonSerializer;

    @Test
    public void testSerialize() throws Exception {

        UserDetails userDetails = new UserDetails(1L, "Duke", "Java",
                                                  LocalDate.of(1995, 1, 1), true);

        JsonContent<UserDetails> result = jsonSerializer.write(userDetails);

        assertThat(result).hasJsonPathStringValue("$.firstname");
        assertThat(result).extractingJsonPathStringValue("$.firstname").isEqualTo("Duke");
        assertThat(result).extractingJsonPathStringValue("$.lastname").isEqualTo("Java");
        assertThat(result).extractingJsonPathStringValue("$.dateofbirth").
                           isEqualTo("01.01.1995");
        assertThat(result).doesNotHaveJsonPath("$.enabled");
    }
```

Slice Based Testing: `JsonTest`

- **`JsonTest` ermöglicht Slice Based Testing des JSON (Un)-Marshallings**

```
@Test
public void testDeserialize() throws Exception {

    String jsonContent = "{\"firstname\":\"Mike\", \"lastname\": \"Inden\","
                       + " \"dateofbirth\":\"07.02.1971\","
                       + " \"id\": 7271, \"enabled\": true}";

    UserDetails result = jsonSerializer.parse(jsonContent).getObject();
    System.out.println("parse: " + result);

    assertThat(result.getFirstName()).isEqualTo("Mike");
    assertThat(result.getLastName()).contains("den");
    assertThat(result.getDateOfBirth()).isAfter(LocalDate.of(1971, 02, 07));
    assertThat(result.getId()).isBetween(7000L, 8000L);
    assertThat(result.isEnabled()).isTrue();
}
```

Testen mit RestClientTest

Slice Based Testing: `RestClientTest`

- **`RestClientTest` ermöglicht Slice Based Testing für REST Clients**

```
@Service
public class UserDetailsClient {

    private final RestTemplate restTemplate;

    public UserDetailsClient(RestTemplateBuilder restTemplateBuilder) {
        restTemplate = restTemplateBuilder.build();
    }

    public UserDetails getUserDetails(int id) {
        String detailsPath = "http://localhost:8080/{id}/details";
        return restTemplate.getForObject(detailsPath, UserDetails.class, id);
    }
}
```

Slice Based Testing: `RestClientTest`

```
@RestClientTest(UserDetailsClient.class)
public class UserDetailsClientTest {

    @Autowired
    private UserDetailsClient userDetailsClient;
    @Autowired
    private MockRestServiceServer mockRestServiceServer;

    @Test
    public void shouldReturnEmployeeDetailsFromHttpRequest() {
        String detailsPath = "http://localhost:8080/1/details";
        mockRestServiceServer.expect(requestTo(detailsPath)).andRespond(
        withSuccess(new ClassPathResource("userDetails.json"),
MediaType.APPLICATION_JSON));

        UserDetails userDetails = userDetailsClient.getUserDetails(1);

        assertThat(userDetails.getAddress()).isEqualTo("Zürich");
        assertThat(userDetails.getSalary()).isEqualTo(100_000);
    }
}
```

Exercises

https://github.com/Michaeli71/Spring_Intro_3d

Hilfe


HELP
IS ON THE
WAY


5.
Auflage
Michael Inden
Der Weg zum
Java-Profi
Konzepte und Techniken für die
professionelle Java-Entwicklung
dpunkt.verlag


Java
Michael Inden
Challenge
Fit für das Job-Interview und die Praxis –
mit mehr als 100 Aufgaben
und Musterlösungen
dpunkt.verlag

Weitere Infos / Quellen

- **Spring Testing**
  - https://docs.spring.io/spring-boot/docs/1.5.2.RELEASE/reference/html/boot-features-testing.html
  - https://www.baeldung.com/spring-tests
  - https://medium.com/swlh/https-medium-com-jet-cabral-testing-spring-boot-restful-apis-b84ea031973d

- **@DataJpaTest**
  - https://www.bezkoder.com/spring-boot-unit-test-jpa-repo-datajpatest/
  - https://reflectoring.io/spring-boot-data-jpa-test/
  - https://zetcode.com/springboot/datajpatest/

- **AssertJ**
  - https://assertj.github.io/doc/
  - https://joel-costigliola.github.io/assertj/assertj-core-quick-start.html
  - https://www.vogella.com/tutorials/AssertJ/article.html
  - https://dzone.com/articles/assertj-and-collections-introduction
  - https://de.slideshare.net/tsveronese/assert-j-techtalk (Hamcrest vs. AssertJ)

Thank You