Lernskript: Grundlagen der Integrationstests mit Spring Boot

Integrationstests überprüfen, ob mehrere Komponenten eines Systems korrekt **zusammenarbeiten**. Sie sind der logische nächste Schritt nach Unit-Tests, die Komponenten nur in Isolation prüfen.

Teil 1: Die grundlegende Analogie - Das "Warum"

- **Konzept:** Nachdem man jedes einzelne Zahnrad (Klasse) mit einem Unit-Test geprüft hat, baut man das Uhrwerk (mehrere Klassen) zusammen und testet, ob die **echten Zahnräder** korrekt ineinandergreifen.
- **Zweck:** Integrationstests finden Fehler, die im **Zusammenspiel** der Komponenten entstehen (z.B. eine fehlerhafte SQL-Abfrage, falsche Konfiguration), die von Unit-Tests mit Mocks nicht entdeckt werden können.

Teil 2: Multi-Use-Cases - Die wichtigsten Spring Boot Test-Annotationen

Spring Boot bietet spezielle Annotationen, um verschiedene Ebenen der Integration zu testen ("Slice Tests").

Use Case 1: Testen der Web-Schicht (`@WebMvcTest`)

- **Ziel:** Testet einen Controller isoliert. Die Service-Schicht darunter wird gemockt.
- **Anwendung:** Um zu prüfen, ob HTTP-Endpunkte korrekt auf Anfragen reagieren, die richtigen Statuscodes zurückgeben und JSON-Antworten korrekt formatiert sind.
- **Code-Beispiel:**

```
@WebMvcTest(BenutzerController.class) // Starte nur die Web-Schicht
class BenutzerControllerTest {
    @Autowired
    private MockMvc mockMvc; // Simuliert HTTP-Anfragen

    @MockBean // Erstellt einen Mockito-Mock im Spring-Kontext
    private BenutzerService benutzerService;

    @Test
    void testEndpoint() throws Exception {
        when(benutzerService.findeById(1)).thenReturn(Optional.of(new
Benutzer(1, "Anna")));

        mockMvc.perform(get("/benutzer/1"))
               .andExpect(status().isOk())
               .andExpect(jsonPath("$.name").value("Anna"));
    }
}
```

Use Case 2: Testen der Datenbank-Schicht (@DataJpaTest)

```
Ziel: Testet ein Repository isoliert. Es wird eine echte (aber In-Memory)
Datenbank verwendet.

Anwendung: Um zu prüfen, ob die Datenbank-Abfragen (z.B. in einem
BenutzerRepository) korrekt funktionieren.
```

@DataJpaTest class BenutzerRepositoryTest { @Autowired private BenutzerRepository repository;

```
@Test
void testDatenbankAbfrage() {
    repository.save(new Benutzer(null, "Peter"));
    Optional<Benutzer> user = repository.findByName("Peter");
    assertTrue(user.isPresent());
}
```

}

Use Case 3: Testen der gesamten Anwendung (@SpringBootTest)

```
Ziel: Ein kompletter End-to-End-Test, der die gesamte Anwendung startet.

Anwendung: Um den kompletten Fluss von einer HTTP-Anfrage durch den Controller,
den Service bis in die Datenbank zu testen.
```

@SpringBootTest @AutoConfigureMockMvc class KompletteAnwendungTest { @Autowired private MockMvc mockMvc;

```
// Hier wird KEIN Service gemockt!

@Test
void testKompletterAblauf() throws Exception {
    mockMvc.perform(post("/benutzer")...)
           .andExpect(status().isCreated());
}
```

}

Teil 3: Vertiefung (JavaMasta's Profi-Tipps)

Die Test-Pyramide: Ein gesundes Projekt hat viele, schnelle Unit-Tests (Basis), weniger, mittelschnelle Integrationstests (Mitte) und sehr wenige, langsame End-to-End-Tests (Spitze).

@ActiveProfiles("test"): Eine Annotation, um einem Test zu sagen, dass er eine spezifische Konfigurationsdatei (application-test.properties) laden soll, z.B. um eine Test-Datenbank zu konfigurieren.

@Transactional in Tests: Sorgt dafür, dass alle Datenbankänderungen nach dem Test automatisch zurückgerollt werden. So startet jeder Test mit einem sauberen Zustand.