

Spring Boot Advanced Testing

ObjektForum Karlsruhe

Florian Ulrich, andrena objects ag

David Burkhart, andrena objects ag

Inhalt

- Unit- vs. Integration-Tests
- Rest-Clients
- WebMVC
- Cacheable
- Async



Unit- vs. Integration-Tests



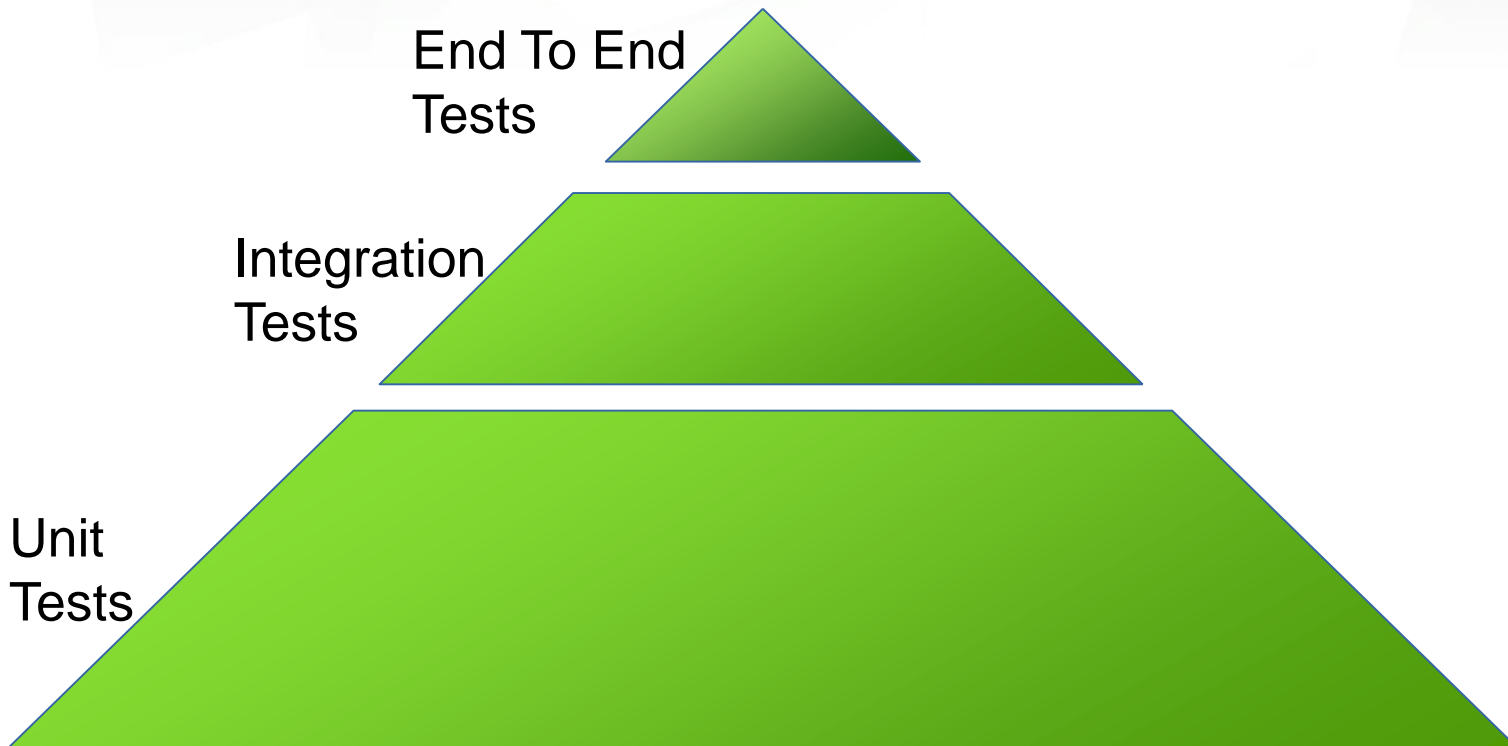
Motivation

“Code without tests is bad code. It doesn't matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse.”

— Michael Feathers, *Working Effectively with Legacy Code*[1]



Test Pyramide



Bevorzuge Unit Tests

- Schnellere Tests
 - Schnelleres Feedback
 - Schnellere CI
 - InfiniTest, Mutation Testing, ...
- Billiger
- Robuster
- Verlässlicher
- Weniger Technologie
 - einfachere Fehlersuche



HowTo: Erkennen von Integration Tests

- Spring Runner bzw. Extension

```
@ExtendWith(SpringExtension.class)
class SwaggerTest {
    ...
}
```

```
@RunWith(SpringRunner.class)
public class SwaggerTest {
    ...
}
```

```
@SpringBootTest
class SwaggerTest {
    ...
}
```



Best Practices für Injection in Spring Boot

- Spring bevorzugt Constructor Injection im Code. Benutze kein `@Autowired` auf Feldern im Anwendungs-Code. Felder im Testcode zu autowiren ist okay

```
@RestController
public class StarshipBookingController {

    private final StarshipBookingService bookingService;

    @Autowired
    public StarshipBookingController(StarshipBookingService bookingService) {
        this.bookingService = bookingService;
    }
}
```

- Junit5 Feature: Konstruktoren für Tests
- Junit5 Feature: Methoden mit Parametern



Wann sind Integration Tests sinnvoll?

Benutze Spring Tests für technische Aspekte, wie z. B.:

- Http Calls
 - Web Controller, Security
 - JSON Mapping
 - Database layer
- Konsequenz: Controller, Daos, ... beinhalten keine Business Logik



Rest-Clients



Kommunikation mit REST-Services über RestTemplate

- Erinnerung: Trennung von Business Logik und Rest Kommunikation
 - *Reiner Unit-Test vs. Spring-Integration-Test*
 - Technische Aspekte
 - Was wird übermittelt (Daten -> **Mapping**)
 - i.d.R. JSON, Mapping per default mit Jackson
 - Wie wird es übermittelt (**HTTP**)
- Beides kann und sollte getestet werden



Mapping testen - Demo



Mapping testen mit @JsonTest

- Lege mindestens ein Beispiel aller Arten von Requests und Responses ab
- Teste gemappte POJOs dagegen (read -> write -> assertEquals)
 - Alle Felder müssen gemappt sein
 - In der Anwendung ignorierte Felder können Object sein
 - Falls nicht möglich, mit Map testen oder nur read
 - Refactoring wird möglich!
- Verwende JsonPath-Definitionen sparsam
 - Brechen bei Änderungen der Datenstrukturen
- Aktualisiere abgelegte Beispiele automatisiert
 - z.B. gegen fremdes Testsystem (evt. instabil!)
 - Benachrichtigung über Änderungen könnte ausreichen



HTTP Kommunikation testen - Demo



HTTP Kommunikation testen mit MockRestServiceServer

- Teste HTTP-Kommunikation
 - HTTP Status, Header, Parameter, Authentifizierung, ...
- Verwende ObjectMapper für Request-Body
 - Schreibe Builder für Daten-Objekte frühzeitig
 - Alternativ: eingecheckte JSON Beispiele, falls z.B. einfach gültige Antwort benötigt
- Asserte nur relevante Teile der Antwort



Goodies

- Wiremock oder einfacher HTTP Mock für Fremdsystem für manuelle Tests
 - Antworten liegen schon vor
 - Samples evt. um fachliche Varianten ergänzen
- Änderungen am Fremdsystem werden erkannt
- Man kann Beispiele schnell und einfach anschauen, da eingeecheckt
- Diff über verschiedene Versionen der eigenen Anwendung
 - Einfach auf JSON Dateien einschränkbar
 - `git diff 1.0..1.1 -- test/resources/*.json`



WebMVC

Eigene Schnittstellen mit WebMVC anbieten

- Erinnerung: Trennung von Business Logik und Schnittstellen-Logik
 - *Reiner Unit-Test vs. Spring-Integration-Test*
- Technische Aspekte
 - Was wird übermittelt (Daten -> **Mapping**)
 - Auch hier lohnt es sich Beispiele abzulegen und dagegen zu testen
 - Wie wird es übermittelt (**HTTP**)



WebMVC Controller testen - Demo



WebMVC Controller testen mit MockMvc

- Achtung bei MockBean: Kein strikter Mock
 - Falls Parameter o.ä. nicht passen -> null
 - Bei seltsamen Fehlern als erstes prüfen
- Tipp (Eclipse): Import Favorites auf MockMvc Factories
- Duplikation im Test entfernen
- Für Rückgabe-Objekte lohnt es sich, Builder anzulegen
- Details anschauen: `.andDo(MockMvcResultHandlers.print())`
- `org.springframework.mock.web`
 - Hilfsklassen, nützlich z.B. um Filter oder Scoped Beans zu testen
 - `MockHttpServletRequest`, `MockHttpServletResponse`, `MockHttpSession`, ...



WebMVC Security testen mit MockMvc – Demo



WebMVC Security testen mit MockMvc

- Security kann und sollte getestet werden
 - Properties im Test passend wählen
- Weitere Themen, wie csrf, jwt:
 - SecurityMockMvcRequestPostProcessors



Swagger-Definition testen - Demo



Swagger-Definition testen mit MockMvc

- Swagger-Info kann im MockMvc-Test erzeugt werden!
- Test gegen eingetragene Definition erkennt Änderungen
 - Macht Änderungen von Schnittstellen explizit
 - Bei false positives einfach die Datei aktualisieren
 - Bessere Meldung im Fehlerfall mit SwaggerDiff
 - Ermöglicht diff über verschiedene Releases aus dem Repo heraus
 - `git diff 1.0..1.1 - test/resources/swagger-definition.json`



Cacheable



Performance-Boost mit Caching

- Technische Aspekte
 - Funktioniert das Caching?
 - Wie lange wird gecached?
 - Wie viele Requests werden gecached?
 - Welche Parameter werden berücksichtigt?
- Funktionsfähigkeit testen
- Anderes nur testen, falls es sinnvoll ist



Caching Demo



Performance-Boost mit Caching

- Caching Tests lohnen sich unter anderem wegen equals (und hashCode) bei Objekten als Parameter
- Benutze (fehlende) Mock-Interaktionen, um Caching zu testen
- Aufräumen des Caches nicht vergessen (sonst Seiteneffekte!)

```
@Autowired
private CacheManager cacheManager;

@AfterEach
void clearCache() {
    cacheManager.getCache("bookable-starships").clear();
}
```

- Für Funktionsfähigkeitstest Time To Live niedrig setzen (FIRST)



Async



Async Testing

- Wird der Call wirklich async ausgeführt?



Async Testing - Demo



Fazit

- Spring bietet Features für Tests mit Context
 - Profile, Properties, MockBeans, etc.
 - Module wie WebMVC, Security, ... bieten eigenen Support
 - Spring Dokumentation zu Testing lohnt sich!
- Testen technischer Aspekte wird sehr einfach und lohnt sich
- Fehler bei der Konfiguration und im Setup werden vor der Laufzeit aufgedeckt
- Synergien möglich
 - Mock-Server als Fremdsystemersatz (-> JSON von Tests)
 - Schnittstellen-Doku im Repository (-> Swagger im Projekt eingecheckt)



Links

- Spring Framework Reference: Testing
<https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html>
<https://docs.spring.io/spring-security/site/docs/current/reference/html/test.html>
- Testing Spring Boot Applications - Phil Webb
<https://www.youtube.com/watch?v=QjaoAWLIGGs>
- Star Wars API: <https://swapi.co/>

Copyright and stuff?

Star Wars and all associated names are copyright Lucasfilm Ltd.

This project is open source and carries a BSD licence.

All data has been freely collected from open sources such as Wookieepedia.

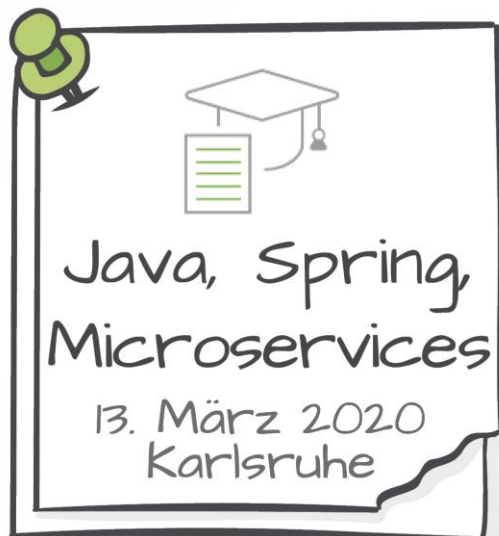


Quellen

[1] Feathers, Michael. Working Effectively with Legacy Code: WORK EFFECT LEG CODE _p1.
Prentice Hall Professional, 2004.



Kostenloser Workshop für Studierende



Anmeldung und
Infos unter:



<https://www.andrena.de/fuer-studierende>