

Web Engineering

Integration-Testing & Persistenz (Teil 2)

Adrian Herzog

(basierend auf der Arbeit von Michael Faes)

Integration Testing

Klassen mit Abhängigkeiten

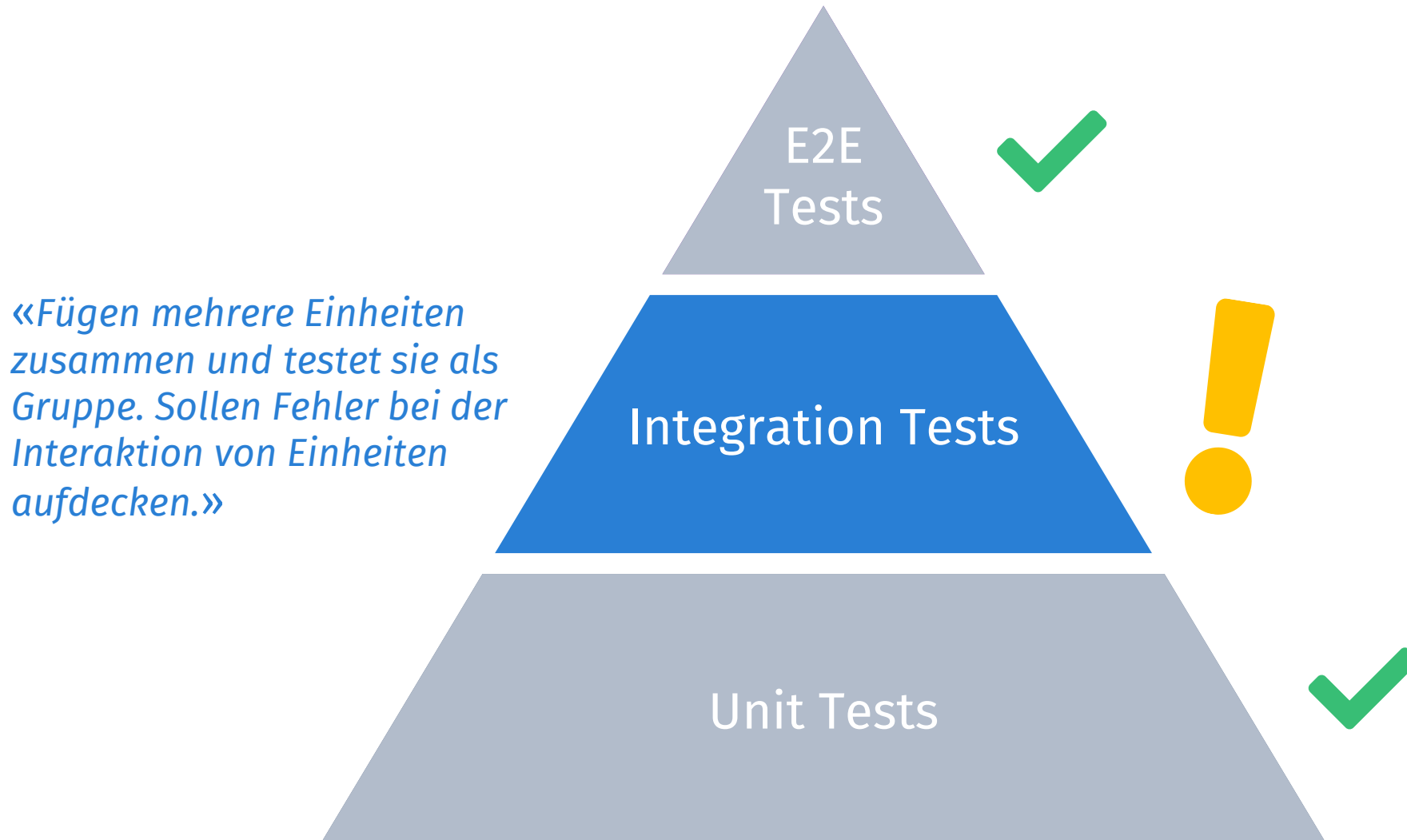
Letzte Woche: `ContactService` ist neu abhängig von `ContactRepository`. Wie testen?

```
public class ContactService {  
    public ContactService(ContactRepository repo) {  
        ...  
    }  
}
```

Zwei Ansätze:

1. Zusammen mit allen Abhängigkeiten testen → *Integration-Testing*
2. Abhängigkeiten durch Test Doubles ersetzen → *Unit-Test*

Test-Pyramide



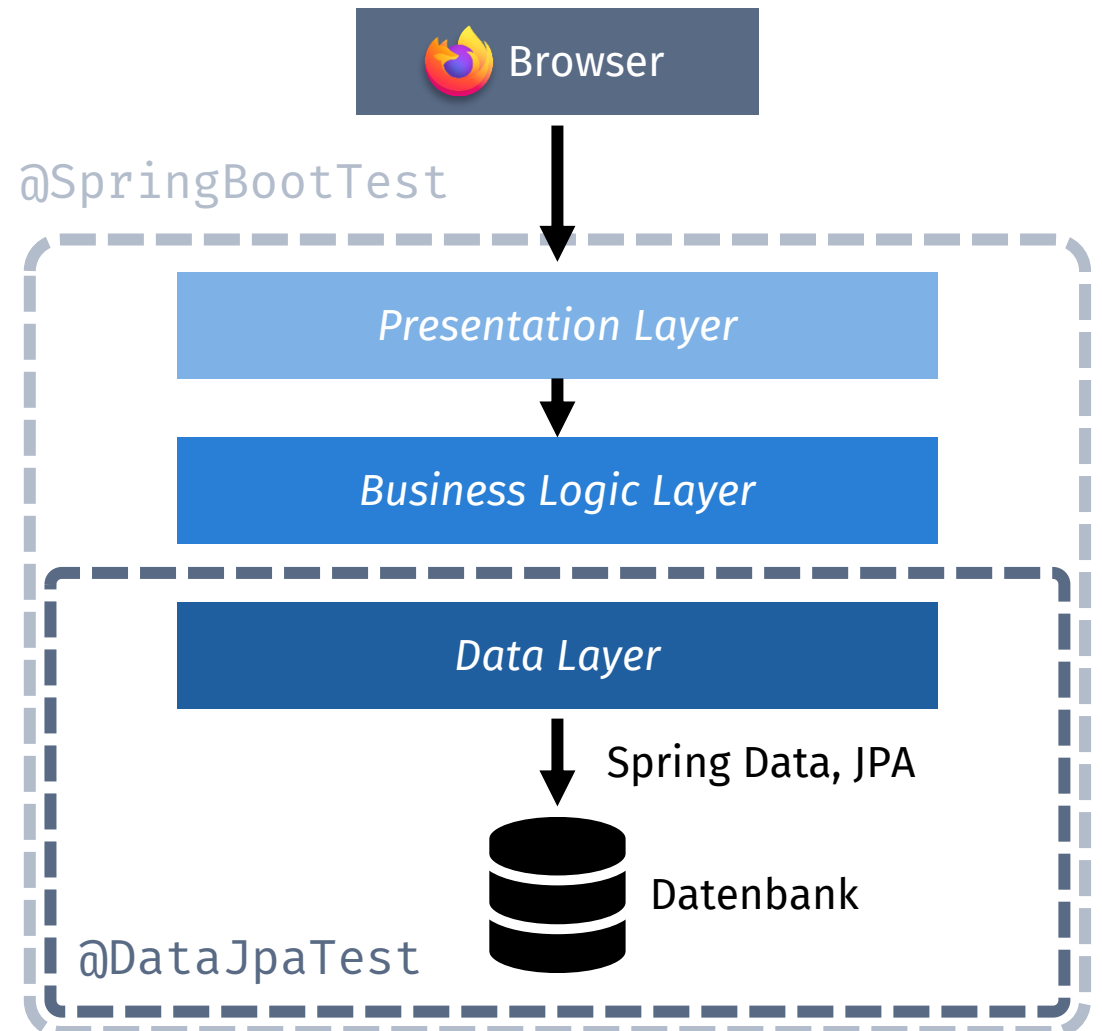
Integration-Tests in Spring Boot

Im Prinzip einfach: Mehrere Klassen gemeinsam testen.

Spring: Wie z. B. Integration mit JPA testen?

`@SpringBootTest`:
Komplette Applikation, mit oder ohne Web-Server
(für E2E-Tests schon verwendet)

`@DataJpaTest`:
Nur *Data Layer*, d.h. Repositories



Integration-Test mit @DataJpaTest

ContactServiceTest zu einem Integration-Test machen:

```
@DataJpaTest
public class ContactServiceIT {

    ContactService service;

    @Autowired
    ContactServiceIT(ContactRepository repo) {
        service = new ContactService(repo);
        ...
    }
    ...
}
```

Spring Boot konfiguriert Test-Datenbank und Repositories, aber nicht gesamten Spring-Kontext und auch keinen Web-Server.

Test-DB für E2E-Tests: @AutoConfigureTestDatabase auf Testklasse

If we don't want to test the real dependency ...



We use a Test Double (Dummy, Fake, Stub, Spy, Mock)

Test Double selber schreiben

Können wir `ContactService` auch ohne JPA-Maschinerie testen?

Ja, mit Fake-Implementation von `ContactRepository`!

Ist ja ein Interface; können für Unit-Tests eigene Implementation schreiben und verwenden:

```
class ContactRepositoryStub implements ContactRepository {  
    public List<Contact> findAll() {  
        return List.of(...);  
    }  
  
    public Optional<Contact> findById(Integer id) {  
        return ...;  
    }  
}
```



...

kleines Problem:
28 weitere Methoden...

Test Double mit Mock-Framework

Dafür gibt es Mock-Frameworks!

Beispiel: Mockito

```
public class ContactServiceTest {  
    ContactService service;  
  
    ContactServiceIT() {  
        var repositoryStub = Mockito.mock(ContactRepository.class);  
        Mockito.when(repositoryStub.findAll()).thenReturn(List.of(...));  
        service = new ContactService(repositoryStub);  
    }  
    ...  
}
```



Müssen nur Methoden spezifizieren, die verwendet werden!

(Stubbing vs. Mocking: martinfowler.com/articles/mocksArentStubs.html)

Übung 1: Integration-Testing vs. Stubbing



Demo

- a) Kommentiere den `ContactServiceTest` im Projekt von letzter Woche ein und ergänze ihn um einen Stub für `ContactRepository`, sodass die Tests wieder funktionieren.
- b) Kopiere die Testklasse und mache einen Integration-Test mit `@DataJpaTest` daraus.

Persistenz: Assoziationen

1:1-Assoziation

```
@Entity
public class Contact {

    @OneToOne
    private Address address;

    ...
}
```

```
@Entity
public class Address {

    ...
}
```

CONTACT		
ID	NAME	ADDRESS_ID
1	Sarah	13
2	Mike	17

ADDRESS		
ID	STREET_NO	CITY
13	Hauptstrasse 0	Gümlingen
17	Elchweg 33	Oberdorf

1:N-Assoziation (default)

```
@Entity
public class Contact {

    @OneToMany
    pr... List<Address> addresses;

    ...
}
```

```
@Entity
public class Address {

    ...
}
```

CONTACT

ID	NAME
1	Sarah
2	Mike

«Join Table»

CONTACT_ADDRESSES

CONTACT_ID	ADDRESS_ID
1	13
2	17
2	18

ADDRESS

ID	STREET_NO	CITY
13	Hauptstrasse 0	Gümlingen
17	Elchweg 33	Oberdorf
18	Mattenweg 4	Unterdorf

1:N-Assoziation (optimiert)

```
@Entity
public class Contact {

    @OneToMany
    @JoinColumn(name="CONTACT")
    pr... List<Address> addresses;

    ...
}
```

```
@Entity
public class Address {

    ...
}
```

CONTACT	
ID	NAME
1	Sarah
2	Mike

ADDRESS			
ID	CONTACT	STREET_NO	...
13	1	Hauptstrasse 0	...
17	2	Elchweg 33	...
18	2	Mattenweg 4	...

N:1-Assoziation

```
@Entity
public class Contact {
    ...
}
```

```
@Entity
public class Address {
    @ManyToOne
    private Contact contact;
    ...
}
```

CONTACT

ID	NAME
1	Sarah
2	Mike

ADDRESS

ID	CONTACT_ID	STREET_NO	...
13	1	Hauptstrasse 0	...
17	2	Elchweg 33	...
18	2	Mattenweg 4	...

1:N/N:1 Bidirektional

```
@Entity
public class Contact {

    @OneToMany(mappedBy="contact")
    pr... List<Address> addresses;

    ...
}
```

```
@Entity
public class Address {

    @ManyToOne
    private Contact contact;

    ...
}
```

CONTACT	
ID	NAME
1	Sarah
2	Mike

ADDRESS			
ID	CONTACT_ID	STREET_NO	...
13	1	Hauptstrasse 0	...
17	2	Elchweg 33	...
18	2	Mattenweg 4	...

M:N-Assoziation

```
@Entity
public class Contact {

    @ManyToMany
    pr... List<Address> addresses;

    ...
}
```

```
@Entity
public class Address {


    ...
}
```

CONTACT

ID	NAME
1	Sarah
2	Mike

CONTACT_ADDRESSES

CONTACT_ID	ADDRESS_ID
1	17
2	17
2	18



ADDRESS

ID	STREET_NO	CITY
13	Hauptstrasse 0	Gümlingen
17	Elchweg 33	Oberdorf
18	Mattenweg 4	Unterdorf

M:N Bidirektional: falsch

```
@Entity
public class Contact {

    @ManyToMany
    pr... List<Address> addresses;

    ...
}
```

```
@Entity
public class Address {

    @ManyToMany
    pr... Set<Contact> contacts;

    ...
}
```

CONTACT

ID	NAME
1	Sarah
2	Mike

CONTACT_ADDRESSES

CONTACT_ID	ADDRESS_ID
1	17
2	17
2	18

ADDRESS_CONTACTS

ADDRESS_ID	CONTACT_ID
17	1
17	2
18	2

ADDRESS

ID	STREET_NO	CITY
17	Elchweg 33	Oberdorf
18	Mattenweg 4	Unterdorf

!

M:N Bidirektional: korrekt

```
@Entity
public class Contact {

    @ManyToMany
    pr... List<Address> addresses;

    ...
}
```

```
@Entity
public class Address {

    @ManyToMany(mappedBy="addresses")
    pr... Set<Contact> contacts;

    ...
}
```

CONTACT

ID	NAME
1	Sarah
2	Mike

CONTACT_ADDRESSES

CONTACT_ID	ADDRESS_ID
1	17
2	17
2	18

ADDRESS

ID	STREET_NO	CITY
13	Hauptstrasse 0	Gümlingen
17	Elchweg 33	Oberdorf
18	Mattenweg 4	Unterdorf

Übung 2: Entities mit Beziehungen

- a) Lade das neue Projekt *wishlist-persistence* in deine IDE, starte die Web-App und mache dich mit der Funktionalität und dem Code vertraut.
- b) Mache die Model-Klassen zu JPA-Entities. Welche Beziehungen (*@OneToMany*, *@ManyToOne*, *@ManyToMany*) machen wo Sinn? Denke auch an die benötigten Maven-Dependencies (siehe letzte Woche).

JPA & Spring Data: Details

Echte Persistenz

Bisher: In-Memory-Datenbank. Um Daten effektiv persistent zu speichern, muss Datei-basierte DB konfiguriert werden:

```
spring.datasource.url=jdbc:h2:file:./contactlist
```

relativ zu
«working directory»

Zusätzlich: Wie umgehen mit vorhandenen Daten in DB?

```
spring.jpa.hibernate.ddl-auto=update
```

Mögliche Werte:

- `none` : nimmt an, vorhandene Tabellen passen
- `validate`: *überprüft* beim Start, dass Tabellen passen
- `create` : löscht vorhandene Daten und erstellt Tabellen neu
- `create-drop`: erstellt Tabellen, löscht Daten *vorher und nachher*
- `update` : versucht, Tabellen zu erweitern, wenn nötig

} Production-
tauglich

Cascade-Typen

Bei `@OneToOne`, `@OneToMany`, usw. muss man entscheiden, was passieren soll, wenn Entity mit Referenzen auf andere Entities gespeichert oder gelöscht wird.

- Betroffene Operationen: `PERSIST`, `REMOVE`, `MERGE`, ..., `ALL`

Beispiel:

```
@Entity
public class Wishlist {

    @OneToMany(cascade = ALL)
    private List<Wish> wishes;
}
```

```
@Entity
public class Wish {

    ...
}
```



Annahme: `Wish` ist ausserhalb von `Wishlist` sinnlos. Soll also an Lebensdauer von `Wishlist` gebunden sein.

Weiteres Beispiel:

```
@Entity
public class Wish {

    @ManyToMany (cascade = ALL)
    private Set<Category> categories;
}
```

```
@Entity
public class Category {

    ...
}
```

Bedeutet: Wenn Wunsch gelöscht wird, werden alle dazugehörigen Kategorien gelöscht!

Und wenn noch von anderen Wünschen verwendet?

- Entweder: `JdbcSQLIntegrityConstraintViolationException`



- Oder sogar Kettenreaktion:

```
public class Category {
    @ManyToMany(cascade = ALL)
    private Set<Wishes> wishes;
}
```



Flushing und Transaktionen

Wann werden Änderungen eigentlich in DB geschrieben (*geflusht*)?

Am Ende einer *Transaktion*: Sammlung von Änderungen, welche *atomar*, d.h. *vollständig oder gar nicht*, in die DB übertragen werden.

```
contact.setFirstName("Mabel");  
contact.setLastName("Guppy");  
contactRepo.save(contact);
```

auch relevant bei
Exceptions, usw.

```
contact.setFirstName("Gerda");  
contact.setLastName("Nurden");  
contactRepo.save(contact);
```

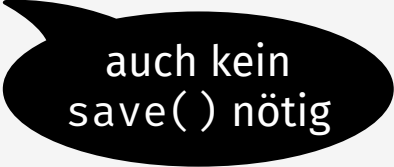
Default für Spring Data: Repo-Methoden *save*, *saveAll*, *delete*, ... werden als (einzelne) Transaktionen ausgeführt.

@Transactional

Default-Verhalten anpassen, indem man eigene Methoden mit `@Transactional` deklariert, z. B. in Service-Klassen:

```
@Service
public class BankingService {

    @Transactional
    public void transfer(Account from, Account to, int amount) {
        from.withdraw(amount);
        to.deposit(amount);
    }
}
```



(Könnte hier auch mit `saveAll(List.of(from, to))` gelöst werden.)

Fallstrick: Aufrufe von Methoden *in gleicher Klasse* nicht transaktional!

Fragen?

