Solvas Fleet - Testhandleiding

Steven Bastiaens
Benjamin Cousaert
Nils Mak
Niko Strijbol
Karel Vandenbussche
David Vandorpe
Domien Van Steendam

Contents

Testhandleiding

Inleiding

Omdat we voor deze milestone eerst een goede testomgeving wouden opstellen voor de backend wordt de frontend nog niet getest. De infrastructuur bij het testen van de frontend krijgt een grotere focus bij de tweede milestone.

De tests kunnen we in twee grote categoriën opdelen: 1. Integration tests 2. Unit tests

De reden voor de opsplitsing is dat er bij de integration tests aparte resources nodig zijn om de tests uit te voeren (MockMVC en een testdatabank)

Integration tests: Bij de integration tests gaan we zowel de DAO-laag als de RestController laag uitgebreid testen.

Unit tests: Bij de Unit Tests is het de bedoeling dat elke klasse met complexe interne logica getest wordt. De unit tests omvatten dus voornamelijk de laag met de business logica. Het is belangrijk dat we klassen testen met zo weinig mogelijk invloed van andere klassen. Dit kunnen we gemakkelijk doen aan de hand van Mockito.

Gebruikte technologiën

- JUnit
- Harmcrest (eventueel FEST in de toekomst)
- Mockito als mocking framework
- MockMVC voor het mocken van de Spring webserver
- random-beans voor het creëren van random objecten
- JaCoCo als code coverage tool
- H2 als test databank

Uitvoeren van de tests

Via gradle kunnen we de tests apart uitvoeren. Met de volgende commando's:

```
$ gradle test
...
...
$ gradle itest
```

Het verslag van de unit tests vinden we in `/backend/build/reports/tests/test/index.html`

Het verslag van de integration tests vinden we in `/backend/build/reports/tests/itest/index.html`

Code Coverage

Natuurlijk is code coverage ook van belang. Het genereren van de code coverage **na** de tests doen we aan de hand van het volgend commando: `$ gradle jacocoReport`

Het verslag van de Code Coverage vinden we in

`/backend/build/reports/jacoco/jacocoReport/html/index.html`

Mockito

Mockito is het mocking framework dat we in al onze tests zullen gebruiken. Het is dan ook vanzelfsprekend dat we hier een kleine toelichting geven.

De bedoeling van mocking is om een klasse te kunnen testen, zonder afhankelijk te zijn van de correcte werking van een andere klasse.

Hoe gaat we nu te werk?

De gemakkelijkste manier om een mock te initialiseren is aan de hand van de `@Mock` annotation. Om dit dan te kunnen laten werken moeten we in onze Setup methode van de tests ervoor zorgen dat de volgende lijn werd toegevoegd.

```
@Mock
private CompanyDao companyDaoMock;
...
@Before
public void setUp(){
MockitoAnnotations.initMocks(this); //Deze lijn
...
}
```

Deze zal dan kijken naar al de velden met mockito annotaties, deze zullen daarna gemocked worden.

Nu is het ook belangrijk dat we de mocks kunnen invoeren in de klasse die we willen testen. Dit kan aan de hand van 2 manieren: - Gebruik maken van de `@InjectMocks` annotatie, deze zorgt ervoor dat dit automatisch gebeurt. Het nadeel hiervan is dat debuggen moeilijk wordt als het niet mogelijk zou zijn om te injecteren. - De mock meegeven met de constructor of via andere methodes. Dit is gemakkelijker te debuggen.

Default geven methodes van mocks die een object van een klasse teruggeven `null` terug. Boolean methodes false en int methodes 0. Meestal is dit niet het gewenste gedrag. Gelukkig is het via mockito mogelijk om methodes aan te passen. Een voorbeeld hiervan vind je in de volgende code:

```
when(roleDaoMock.find(anyInt()))
 .thenThrow(new EntityNotFoundException());
```

Deze gaat dus ervoor zorgen dat de roleDao een EntityNotFoundException zal throwen bij oproep van eender welke id bij de `find(int id)` methode. Dit zou dus gebruikt kunnen worden om te testen dat de RestController een correcte HTTP response geeft als de entiteit niet gevonden kan worden in de databank.

Meer info over Mockito kan je hier vinden.

Integration tests: RestController layer

Om het gedrag van de RestControllers te testen maken we gebruik van `MockMvc`. Dit is een onderdeel van *spring-test* en wordt globaal gebruikt om restcontrollers te testen.

We kunnen een object van de klasse MockMvc creëren door aan een constructor de restcontroller mee tegeven die we willen testen, zoals hieronder gedaan wordt.

```
@Before
    public void setUp() throws JsonProcessingException {
        RoleRestController roleRestController=
        new RoleRestController(daoContext,roleMapper,roleValidator);
        mockMvc=
         MockMvcBuilders.standaloneSetup(roleRestController).build();
      }
```

Vervolgens kunnen we testen uitvoeren op HTTP requests zoals get,put,post,delete en kijken of er aan bepaalde verwachtingen voldaan wordt.

```
mockMvc.perform(get("/roles/1"))
  .andExpect(status().isOk())
  .andExpect(content().contentType(MediaType.APPLICATION_JSON_UTF8));
```

In het bovenstaand voorbeeld wordt er dus gekeken of er bij een get request op een role met id 1 een HTTP 200 status wordt teruggegeven en er wordt ook gecontroleerd dat het contenttype in JSON UTF8 staat.

Specifieke tests op de json doen we aan de hand van `jsonpath`, maar dit bespreken we niet in de handleiding.

Workflow bij het testen van de RestControllers

- Test elke controller in een aparte klasse
- Kijk naar de specificaties van de API waar de RestControllers aan moeten voldoen
- Test elke specificatie apart, dus bvb "*test dat HTTP status 404 wordt teruggegeven wanneer een component niet aanwezig is in de databank*".
- Maak zoveel mogelijk gebruik van mocks, de RestController mag niet afhankelijk zijn van de werking van implementaties van andere klassen

Integration tests: Data access layer (DAO)

Bij de integratietests van de data access layer maken we gebruik van een test databank. De databank die we gebruiken is h2. Dit is een embedded databank, wat het dus mogelijk maakt om een databank op te stellen zonder enige configuratie. De databank wordt automatisch aan de start van de tests aangemaakt.

Aan de databank wordt er testdata toegevoegd. Deze kan gevonden worden in: `/backend/src/itest/resources/schema.sql`

Elke testklasse van de DAO laag moet starten met volgende annotaties:

```
@RunWith(SpringJUnit4ClassRunner.class) 1
@ActiveProfiles("test") 2
@ContextConfiguration(
classes = {HibernateConfig.class,HibernateTestConfig.class},
loader = AnnotationConfigContextLoader.class) 3
@Transactional
```

1. zorgt ervoor dat de de configuraties worden uitgevoerd (zie 3).
2. geeft ons het spring profile "test", hierdoor worden bepaalde configuraties aangepast
3. De testconfig en de gewone config worden als configuraties toegevoegd. Dit is belangrijk voor het autowiren van de dao's en het opstellen van de h2 databank.

Workflow bij het testen van de DAO-laag

- Test elke dao in een aparte klasse
- Maak gebruik van de testdata (van elk model zijn er 100 voorbeelden te vinden op de databank, buiten vehicletype (5 types))
- Test ook specifieke methodes die de dao voorzien

Unit tests

De unit tests zijn een pak eenvoudiger. Ze hebben geen enkele opstelling nodig en moeten maar 1 iets doen: het *gedrag* van **één** klasse testen.

We gebruiken bij de unit tests enorm veel gebruik van mockito, zodat we enkel afhankelijk zijn van de klasse die we testen.

Workflow bij unit testing

- Kijk naar het gewenst gedrag (in de documentatie)
- Moduleer verschillende testen zodat bijna elke situatie getest wordt.