Documentatie Spotitube

Titel	Documentatie Spotitube
Student	Ciyan Çöcelli
Studentnummer	548189
Docent	Meron Brouwer
Datum	27 maart 2020
Semester	OOSE
Course	DEA
Versie	1.1
School	Hogeschool van Arnhem en Nijmegen

Inhoudsopgave

1. Inleiding

Dit document beschrijft de verschillende ontwerpkeuzes tijdens het ontwikkelen van de Spotitube applicatie.

Spotify en Youtube slaan de handen ineen en werken gezamenlijk aan een applicatie genaamd Spotitube. Een klant kan hiermee een overzicht krijgen van afspeellijsten met audio- en videostreams. De opdrachtgever wil eerst een gedeeltelijke bacXk-end applicatie ontwikkelen. Zij willen de applicatie testen via een bestaande webapplicatie.

Als eerste breng ik requirements in beeld in hoofdstuk 2. Hoofdstuk 3 gaat over het package diagram. Dit diagram geeft de structuur weer van de verschillende packages. Aan de hand van het diagram laat ik zien welke requirements het raakt. Daarna geef ik het deployment diagram weer in hoofdstuk 4. Het deployment diagram laat zien hoe de applicatie draait op een server. In hoofdstuk 5 benoem ik de gemaakte ontwerpkeuzes tijdens het ontwikkelen van de applicatie. Als laatste concludeer ik het verslag in hoofdstuk 6.

2. Requirements

Dit hoofdstuk beschrijft de functionele en non functionele requirements van het Spotitube project. De opdrachtgever heeft verschillende eisen. Ik kies ervoor om de requirements met behulp van FURPS+ in te delen. Een systematische aanpak helpt bij het indelen van requirements. Tabel 1 geeft de verscheidene requirements weer.

Functional Requirements	**Beschrijving**	**Hoofdstuk**
Fr#1	Het systeem moet Restful kunnen communiceren volgens de REST API specificatie.	Zie hoofdstuk 4.2.
Non-functional Requirements		
Supportability		
Sr#1	Het systeem moet meerdere verschillende relationele databases ondersteunen.	Zie hoofdstuk 5.1.
Sr#2	Het wisselen van database moet mogelijk zijn zonder het systeem opnieuw te moeten compileren.	Zie hoofdstuk 5.1.
Sr#3	Het systeem bevat minimaal 80% code coverage.	Zie hoofdstuk 5.3.
Sr#4	Het systeem moet gedeployed kunnen worden op Apache TomEE Plus.	Zie hoofdstuk 4.2.
+ (plus)		
Implementation		
Impl#1	Het systeem moet gebruik maken van een data access layer.	Zie hoofdstuk 3.2.
Impl#2	Het systeem moet gebruik maken van een domain layer.	Zie hoofdstuk 3.2.
Impl#3	Het systeem moet gebruik maken van het Remote Façade pattern door middel van REST Services.	Zie hoofdstuk 3.2.
Impl#4	Het systeem moet gebruik maken van het Data Mapper Pattern.	Zie hoofdstuk 5.1.
Impl#5	Het systeem moet gebruik maken van het Separated Interface Pattern.	Zie hoofdstuk 5.2.

Impl#6	Het systeem moet gebruik maken van dependency injection om de afhankelijkheid te verlagen.	Zie hoofdstuk 5.2.
Impl#7	Het systeem moet gebruik maken van JAX-RS v2.0.	Zie hoofdstuk 3.2.
Impl#8	Het systeem moet gebruik maken van Context & Dependency Injection (CDI).	Zie hoofdstuk 5.2.

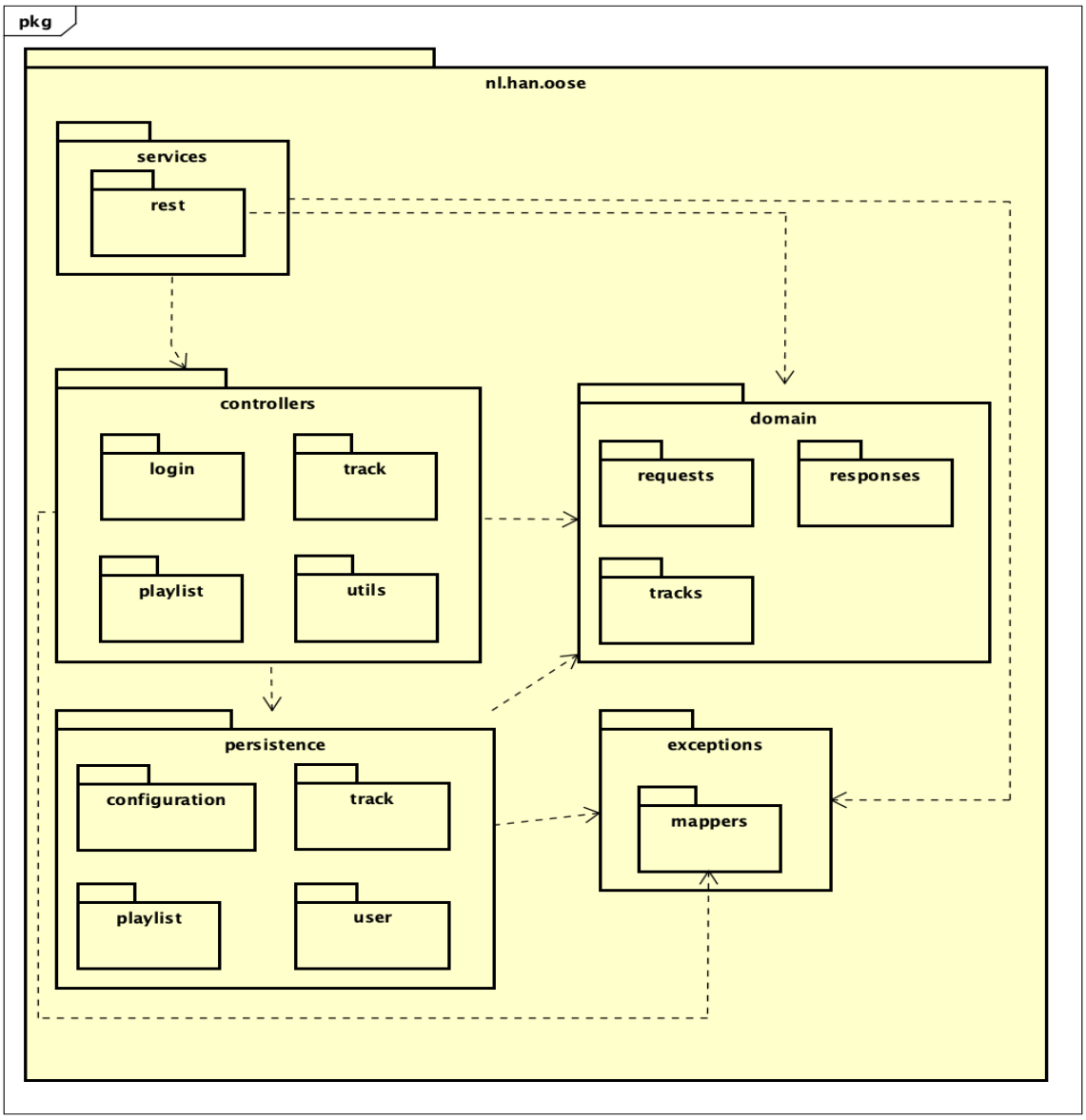
Tabel 1. Requirements ingedeeld met behulp van FURPS+

3. Package Diagram

In het vorige hoofdstuk benoemde ik de requirements. Dit hoofdstuk gaat over het package diagram. Als eerste geef ik het package diagram weer. Daarna benoem ik welke requirements ik ermee raak.

3.1. Diagram

Afbeelding 1 geeft het huidige diagram weer. Ik stel vijf verschillende hoofdpackages op. Deze packages bevatten verschillende sub packages. Tabel 2 legt uit wat elke package inhoudt.



Afbeelding 1. Huidige iteratie van het package diagram.

Package	*Uitleg*
Services	De services package bevat een REST package. In de REST package zijn de services beschikbaar conform de REST API specificatie.
Controllers	In deze package zijn de controllers beschikbaar. De controllers zijn opgedeeld in vier packages: login, track, playlist en utils. De controllers voeren de business logica uit.
Persistence	De persistence package regelt de communicatie tussen de database en de applicatie. De package bevat vier verschillende sub packages: configuration, track, playlist en user.
Domain	De domain package bevat alle domeinobjecten. De package bevat drie subpackages: requests, responses en tracks.
Exceptions	De exception package bevat alle zelfgemaakte exceptions van de applicatie. De package bevat ook een subpackage waar de exception mappers van de applicatie in zitten.

Tabel 2. Uitleg over packages in Spotitube.

3.2. Requirements

In hoofdstuk 3.1 leg ik het diagram en de packages uit. Dit ga ik koppelen aan de requirements, zie tabel 3. Ik licht toe welke requirements het diagram raken.

REQUIREMENT	TOELICHTING
IMPL#1	Het systeem maakt gebruik van een data access layer. Ik noem dit de persistence package.
IMPL#2	Het systeem maakt gebruik van een domain layer. Alle domeinobjecten zitten in de domain package.
IMPL#3	Het systeem bevat REST services. De services maken gebruik van het Remote Façade pattern.
IMPL#7	De REST services maken gebruik van JAX-RS v2.0. TomEE levert een implementatie van JAX-RS.

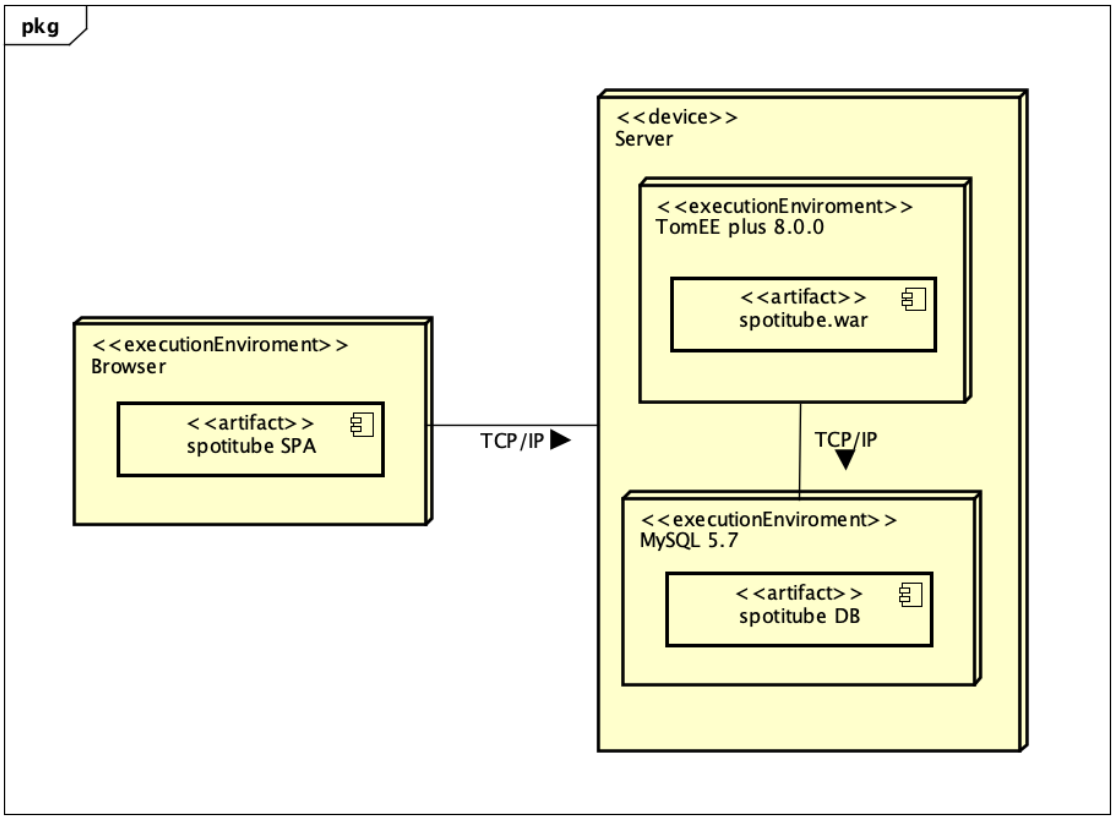
Tabel 3. Toelichting geraakte requirements.

4. Deployment Diagram

Het vorige hoofdstuk ging over het package diagram. In dit hoofdstuk geef ik het deployment diagram weer. Ik leg uit op wat voor manier de applicatie draait. Ik koppel dit aan de requirements.

4.1. Diagram

Afbeelding 2 geeft het deployment diagram weer. De server bevat twee execution environments, TomEE plus 8.0.0 en MySQL 5.7. TomEE draait de spotitube.war. Dit is de ontworpen applicatie. Door middel van TCP/IP communiceert TomEE met MySQL. MySQL draait de Spotitube database. De gebruiker kan op de browser gebruik maken van de Spotitube single page application. De browser communiceert via TCP/IP met de server.



Afbeelding 2. Deployment diagram

4.2. Requirements

In het deelhoofdstuk 4.1 leg ik het deployment diagram uit. Hierdoor kan ik de requirements koppelen aan het diagram. Ik licht toe welke requirements ik raak en oplos, zie tabel 4.

REQUIREMENT	UITLEG
FR#1	Het systeem communiceert restful volgens de REST API specificatie. Dit gebeurt over het TCP/IP protocol. Ik geef dit weer d.m.v. de TCP/IP lijn van browser naar server.
SR#4	Het systeem deploy ik op een TomEE plus 8.0.0 omgeving.

Tabel 4. uitleg geraakte requirements

5. Craftsmanship & Ontwerpkeuzes

Dit hoofdstuk legt mijn ontwerpkeuzes uit. Ik maakte dit project een aantal keuzes. Hieronder leg ik de keuzes uit en koppel ik ze aan de requirements.

5.1. ORM

Ik koos ervoor om een object-relational mapper (ORM) te gebruiken. Hiervoor maak ik gebruik van Hibernate. Sasidharan (2018) benoemt een aantal voor- en nadelen om een ORM te gebruiken. De voordelen wegen de nadelen af. Het faciliteert het implementeren van het domain model pattern**.** Het zorgt voor een reductie in code. Het doet aan cache management. De nadelen zijn: een langere startup tijd, een grote leercurve, moeilijk om fine te tunen door gegenereerde SQL-query's. Aangezien ik ervaring heb met Hibernate en weet hoe ik dit moet implementeren, koos ik ervoor om hier gebruik van te maken.

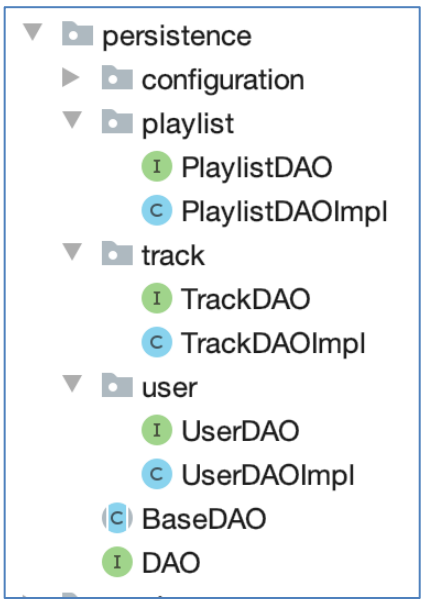
De keuze zorgt ervoor dat ik requirements, sr#1 & sr#2, raak. Hibernate ondersteunt verschillende relationele databases ("Supported Databases2 | JBoss.org Content Archive (Read Only)", 2009). Ik raak ook impl#4. Hibernate is een ORM gebaseerd op het Data Mapper pattern (Hayes, 2018).

5.2. Clean Code

Het systeem moet gebruik maken van het Separated Interface Pattern volgens impl#5. Dit pas ik toe. Alle Data Access Object (DAO) implementaties maken gebruik van een interface. Daarnaast gebruik ik een abstracte BaseDAO die algemene functies implementeert, zie afbeelding 3. Hiervoor maak ik gebruik van generics. Dit vermindert de hoeveelheid code. Afbeelding 4 geeft de implementaties en interfaces weer in de persistence laag.

```
public interface DAO<T> {
    void save(T entity) throws DatabaseException;
    void update(T entity) throws DatabaseException;
    void delete(int id, String hql) throws DatabaseException;
    List<T> getListByIdAndHql(int id, String hql) throws DatabaseException, EntityNotFoundException;
    T getByIdAndHql(int id, String hql) throws DatabaseException, EntityNotFoundException;
}
```

Afbeelding 3. Generic DAO interface.



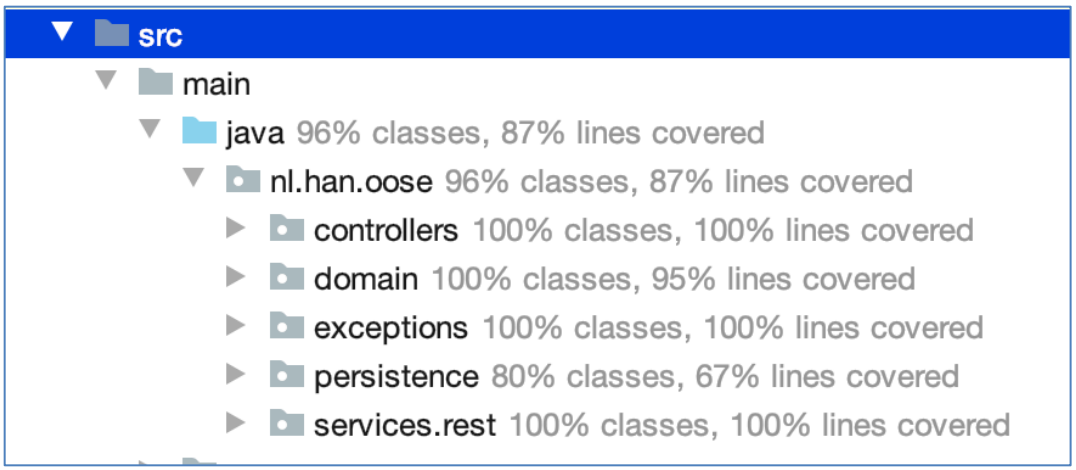
Afbeelding 4. Interfaces en implementaties in de persistence laag.

Martin (2009) raadt aan om dependency injection (DI) te gebruiken. Het is ook een requirement vanuit de opdrachtgever: impl#6 & impl#8. In het project maak ik gebruik van het Contexts and Dependency Injection (CDI) framework van JavaEE. De DAO's en controllers maken gebruik van interfaces. Hierdoor zijn klassen afhankelijk van abstracties en niet van concrete implementaties.

Ik preface interfaces niet met een *I*. C. Martin (2009) prefereert het weglaten van de I. Ik zie vaak het verkeerde voorbij komen. Meerdere developers gebruiken IObject i.p.v. Object. De implementaties van de interfaces geef ik aan met *Impl*. Dus *ObjectImpl implements Object* i.p.v. *Object implements IObject*. Dit zorgt voor een heldere naamgeving, zie afbeelding 4.

5.3. Testen

Volgens requirement sr#3 moet de applicatie minimaal 80% code coverage bevatten. Afbeelding 5 geeft aan dat de applicatie hieraan voldoet. Ik test alle lagen afhankelijk van elkaar, behalve de persistence laag. Om de lagen afhankelijk van elkaar te testen, gebruik ik Mockito.



Afbeelding 5. Code coverage van de applicatie.

Om de persistence laag te testen maak ik gebruik van integratietests. Ik kies hiervoor omdat ik gebruik maak van Hibernate. Om te testen of de functies werken, heb ik een connectie met een database nodig. Ik configureer een sessie met een in-memory database (H2), zie afbeelding 6. De sessie vul ik in de setup van de tests met testdata. Het gebruik van integratietests voor de persistence laag zorgt ervoor dat ik kom aan de minimale eis van 80% code coverage.

```
public abstract class BasePersistenceIntegrationTest {
    protected SessionFactory newSessionFactory() {
        Properties properties = new Properties();
        properties.put("hibernate.dialect", "org.hibernate.dialect.H2Dialect");
        properties.put("hibernate.hbm2ddl.auto", "create");
        properties.put("hibernate.show_sql", "true");
        properties.put("hibernate.connection.driver_class", "org.h2.Driver");
        properties.put("hibernate.connection.url", "jdbc:h2:mem:test");
        properties.put("hibernate.connection.username", "sa");
        properties.put("hibernate.connection.password", "");

        return new Configuration()
                .addProperties(properties)
                .addAnnotatedClass(User.class)
                .addAnnotatedClass(Playlist.class)
                .addAnnotatedClass(Track.class)
                .addAnnotatedClass(Song.class)
                .addAnnotatedClass(Video.class)
                .buildSessionFactory(
                        new StandardServiceRegistryBuilder()
                                .applySettings(properties)
                                .build());
    }
```

Afbeelding 6. Configuratie van in-memory database.

5.4. HashMap voor het bijhouden van gebruikers

Ik kies ervoor om een HashMap te gebruiken voor het bijhouden van gebruikers op basis van tokens. Bij het inloggen genereert de applicatie een token. De HashMap maakt gebruik van deze token als key. De HashMap slaat bij deze token de desbetreffende gebruiker op. Een HashMap sorteert geen items op basis van wanneer het object is toegevoegd. Het gebruik van een HashMap is sneller dan een List door de manier van data ophalen. Ik kies ervoor om een

singleton HashMap te gebruiken i.c.m. het keyword synchronized. Dit zorgt ervoor dat de HashMap hetzelfde is voor alle afnemers.

5.5 Controllers losgetrokken van services

De eerste iteratie van een applicatie maakte gebruik van een service layer. In deze laag handelde de applicatie de REST calls af. Deze laag voerde ook business logica uit. De laag deed dus een groot deel van het werk. Hierdoor besloot ik om een controllerlaag toe te voegen. In deze controllerlaag definieer ik interfaces waarin ik meerdere controllers heb. Deze controllers handelen de logica van de applicatie af. De services maken gebruik van de controllers via DI. Deze splitsing zorgt ervoor dat de taken gescheiden zijn. De controllers voeren business logica uit en de services handelen REST calls af.

6. Conclusie

In de inleiding geef ik aan dat dit document de verschillende ontwerpkeuzes beschrijft tijdens het ontwikkelen van de Spotitube applicatie. Ik doe dit aan de hand van een package diagram, deployment diagram en ontwerpkeuzes. Daarnaast geef ik aan hoe ik aan verschillende requirements voldoe. Dit document haalt zijn doel.

Literatuurlijst

C. Martin, R. C. (2009). *Clean Code: A Handbook of Agile Software Craftsmanship*. Geraadpleegd van https://www.investigatii.md/uploads/resurse/Clean_Code.pdf

Fowler, M. (z.d.). *P of EAA: Remote Facade*. Geraadpleegd op 29 oktober 2019, van https://martinfowler.com/eaaCatalog/remoteFacade.html

Hayes, D. (2018, 26 september). *ORM Patterns: The Trade-Offs of Active Record and Data Mappers for Object-Relational Mapping.* Geraadpleegd op 26 maart 2020, van https://www.thoughtfulcode.com/orm-active-record-vs-data-mapper/

Sasidharan, M. (2018, 26 mei). *Should I Or Should I Not Use ORM?* Geraadpleegd op 29 oktober 2019, van https://medium.com/@mithunsasidharan/should-i-or-should-i-not-use-orm-4c3742a639ce

Supported Databases2 | JBoss.org Content Archive (Read Only). (2009, 7 augustus). Geraadpleegd op 26 maart 2020, van https://developer.jboss.org/wiki/SupportedDatabases2