|  |
| --- |
|  |
| Opleverdocumentatie |
| OOSE-DEA Spotitube |

|  |
| --- |
| Klaas van der Linden  29-10-2018  Studentnummer: 599644  Klas: ITA-OOSE-A-s  Docent: Michel Koolwaaij |

Inhoud

[1. Opdrachtomschrijving 2](#_Toc528415321)

[2. Packagediagram 3](#_Toc528415322)

[2.1 Algemene indeling 3](#_Toc528415323)

[2.2 Data Mapper 3](#_Toc528415324)

[2.3 Identity Mapper 4](#_Toc528415325)

[2.4 Separated Interface 4](#_Toc528415326)

[2.5 Dependecy Injection 4](#_Toc528415327)

# Opdrachtomschrijving

Voor de course DEA van het semester OOSE moet er de applicatie Spotitube worden gerealiseerd. Spotify en Youtube hebben de handen ineengeslagen en werken gezamenlijk aan een app (Spotitube) waarmee een klant een overzicht kan krijgen van afspeellijsten met daarin audio- en videostreams. Ze willen eerst een deel van de back-end ontwikkelen en deze testen via een bestaande webapplicatie alvorens over te gaan tot de ontwikkeling van de app.

De applicatie moet MySQL 5.1 ondersteunen en minimaal 1 andere database. Daarnaast moet de applicatie gebruik maken van de volgende APIs en frameworks:

* JAX-RS (REST, JSON)
* CDI (Context & Dependency injection)
* JDBC API
* De applicatie wordt ontwikkeld met Java versie 8 en moet draaien in Apache TomEE Plus

# Packagediagram

Figuur 1 Packagediagram

## Algemene indeling

Hierboven bevindt zich het packagediagram wat de indeling van mijn Spotitube applicatie illustreert op packageniveau.

Bij het opzetten van de architectuur het ik besloten om de packages te groeperen op serviceniveau ook wel bekend als het Service Layer pattern. Het voordeel hiervan is dat verschillende operaties binnen de applicatie worden gescheiden. De business logica en het ophalen van de data zit dus in een andere laag. Hierdoor zijn deze operaties gemakkelijk afhankelijk van andere lagen te testen. Bovendien kunnen operaties snel worden aangepast, aangezien elke laag zijn eigen type operaties bevat.

Een alternatief is om de packages te groeperen op domein. Hierbij bevat een package bijvoorbeeld alle functionaliteiten van Playlist. Het nadeel hiervan is dat je niet snel kunt zien wat elk domein doet en is het moeilijk om te onderscheiden wat de verschillende operaties binnen de applicatie zijn.

Dit probleem doet zich niet voor wanneer je het Service Layer Pattern gebruikt. Daar is snel te zien wat de verschillende services zijn. Daarom is het gebruik van het Service Layer Pattern de beste keuze

## Data Mapper

Een van de lagen binnen de applicatie is de DAO laag. Deze laag bevat een super klasse genaamd. DAO. Deze klasse laadt de database instellingen en de JDBC driver in, haalt de verbinding op en kan deze ook sluiten. De data mappers erven over van deze klas. De data mappers halen de data op uit deze database en verplaatsen deze in de vorm van POJO’s naar de andere lagen. Dit zorgt ervoor dat de andere lagen geen weet hebben van de database. Ze hebben geen SQL code nodig en hoeven geen weet te hebben van het database schema. Het voordeel hiervan is dat de andere lagen niet afhankelijk zijn van de database. Bovendien zorgen de data mappers als aparte laag ervoor dat de deze laag gemakkelijk gemockt kan worden bij het afzonderlijk testen van een van de andere lagen.

Een alternatief is het Repository pattern. Hierbij wordt ook data opgehaald uit de database, maar heeft de respository een instantie van een klasse uit het domein nodig om CRUD acties uit te voeren. Hierdoor zijn het domein en de respository aan elkaar gekoppeld terwijl je juist zo min mogelijk koppeling tussen de lagen wil. Het data mapper pattern is daarom de betere optie.

## Identity Mapper

Zoals in het packagediagram is te zien communiceert de service laag met de identity mappers in plaats van direct met de data mappers. De identity map slaat de data die ophaalt op in een map. Zodra de identity map van de service laag een verzoek om data krijgt, kijkt de identity map of hij de data lokaal heeft opgeslagen. Als dit niet geval is haalt de identity map de data op uit de database. Dit zorgt voor betere performance, aangezien de database nu geen verzoek krijgt als de benodigde data al eerder is opgehaald. Bij het identity map pattern heb ik gebruik gemaakt van een Singleton. Zodat het systeem één verzameling van data heeft die wordt opgeslagen en kan worden opgehaald.

Het alternatief is om geen gebruik te maken van het identity map pattern. In dat geval zal er bij elk verzoek om data er een nieuw verzoek naar de database gaat wat minder goed is voor de performance. Om de applicatie soepeler te laten lopen heb ik ervoor gekozen om het identity map pattern te gebruiken. Een snellere applicatie levert een betere ervaring voor de gebruiker, wat het gebruik van het identity map pattern tot de beste oplossing maakt.

## Separated Interface

Om de koppeling tussen de lagen laag te houden, heb ik gebruik gemaakt van interfaces om de lagen met elkaar te laten communiceren. Bovendien zorgt het gebruik van interfaces ervoor dat een klasse alleen methoden gebruik die hij nodig heeft en zorgt het communiceren via interfaces ervoor dat de implementatie van de ene klasse verborgen blijft voor de andere klasse, ook wel bekend als information hiding.

Een alternatief kan het niet gebruiken van interfaces zijn. Het nadeel hiervan is dat de code slechte uitbreid baar is. Als er in de toekomst een klasse bijkomt die een gelijksoortige implementatie vereist als een bestaande klasse, dan kan deze gebruik maken van dezelfde interfaces. Ook zorgen interfaces ervoor dat er gemakkelijk nieuw functionaliteiten kunnen worden toegevoegd aan een bestaande implementatie zonder de bestaande code aan te passen wat in overstemming is met het Open Closed Principle. Het gebruiken van interfaces is dus de beste keuze.

## Dependecy Injection

Om de koppeling tussen de lagen laag te houden en de lagen gemakkelijker testbaar te maken, heb ik gebruik gemaakt van dependecy injection. Dit zorgt ervoor dat higher-level en lower-level software componenten met elkaar kunnen communiceren via interfaces wat de afhankelijkheden tussen de componenten verwijdert.

Het gebruiken van Dependecy injection zorgt ervoor dat code makkelijker testbaar is. In de klasse die worden getest kunnen namelijk mocks worden geïnjecteerd.

@RunWith(MockitoJUnitRunner.class)  
public class LoginControllerTest {  
  
 @InjectMocks  
 private LoginController loginController;  
  
 @Mock  
 private IUserService userService;

Een alternatief voor Dependecy Injection is het gebruik maken van een Service Locator. Hierbij is het idee dat er een object is die weet hoe hij aan alle services kan komen die een applicatie nodig kan hebben. De Service Locator geeft bij een verzoek de informatie terug die nodig is om een taak uit te voeren. Een van de nadelen bij het gebruik van een Service Locator is dat alle test communiceren met een globale Service Locator wat unittesten lastiger maakt.

# Deploymentdiagram