Niels Borkes (student)

Studentnummer: 616825

Course: OOSE-DEA

Docenten: Meron Brouwer & Rody Middelkoop

Spotitube

Oplever documentatie

Contents

[1. Inleiding 1](#_Toc85131356)

[2. UML-diagrammen 2](#_Toc85131357)

[Package diagram 2](#_Toc85131358)

[Deployment diagram 3](#_Toc85131359)

[3. Ontwerpkeuzes 4](#_Toc85131360)

[Gebruik van Data Access Objects (DAOs) 4](#_Toc85131361)

[Gebruik van Datamappers 4](#_Toc85131362)

[Gebruik van Data Transfer Objects (DTOs) 4](#_Toc85131363)

[Gebruik van Dependency Injection 4](#_Toc85131364)

[Gebruik van Dependecy Inversion Principle 5](#_Toc85131365)

[Gebruik JAX-RS foutafhandeling 5](#_Toc85131366)

[Gebruik van de Service Layer 5](#_Toc85131367)

[4. Kwaliteit 6](#_Toc85131368)

[Unittest 6](#_Toc85131369)

[API functionaliteit 6](#_Toc85131370)

[Conclusie 6](#_Toc85131371)

# Inleiding

Voor het vak Distributed Enterprise Applications (DEA) uit het semester OOSE is de casusopdracht om een RESTfull-API te ontwikkelen voor de al bestaande aangeleverde front-end applicatie Spotitube. De opdracht wordt beschreven in het bestand “OpdrachtSpotitube.pdf” van onderwijsonline, ook is er een GitHub-repo waar de eisen over de functionaliteit van de API beschreven worden. Deze is te vinden door de volgende [link](https://github.com/HANICA-DEA/spotitube).

In dit document zullen de gemaakte keuzes die tijdens de ontwikkeling van de API tot stand zijn gekomen toe gelicht worden. Dit zal gebeuren aan de hand van een package diagram die inzicht geeft in de structuur van de API, verder zal er een deployment diagram beschreven worden die duidelijk maakt hoe de onderdelen van de API met elkaar samen hangen. Tot slot zal er een conclusie volgen over de kwaliteit van het product.

# UML-diagrammen

In dit hoofdstuk zullen de package en deployment diagrammen besproken worden.

## Package diagram

Diagram

Description automatically generated

Figuur Package diagram

In het bovenstaande diagram zijn de drie verschillende lagen weergegeven door middel van de toegepaste package structuur. Uit het diagram wordt duidelijk dat een laag alleen afhankelijk is van een abstractie van de laag onder zich. Dit zorgt voor een “”High [Cohesion](https://en.wikipedia.org/wiki/Cohesion_(computer_science)), Low [Coupling](https://en.wikipedia.org/wiki/Coupling_(computer_programming))” systeem wat betekend dat de lagen zo onafhankelijk mogelijk zijn en dus alleen afhankelijkheid hebben naar hetgene wat daadwerkelijk nodig is. Dit heeft als voordeel dat het systeem goed onderhoudbaar en aanpasbaar is.

De client zal zoals beschreven over het [Facade Design Pattern](https://en.wikipedia.org/wiki/Facade_pattern) alleen kennis te hoeven hebben van de resources laag en niet de onderliggende lagen waar de logica in afgehandeld wordt.

## Deployment diagram

Diagram

Description automatically generated

Figuur Deployment diagram

IN het bovenstaande diagram is te zien dat de subsystemen de API draaiende op TomEE en een MySQL database zijn die door middel van JDBC met elkaar communiceren.

# Ontwerpkeuzes

In het ontwikkelproces van de API zijn er keuzes gemaakt op het gebied van implementatie die hieronder worden toegelicht.

## Gebruik van Data Access Objects (DAOs)

Voor het ophalen van data uit de database is gekozen voor het gebruik van het [Row data gateway Pattern](https://en.wikipedia.org/wiki/Row_data_gateway). Dit betekent dat elke rij in een databasetabel een Java Object variant heeft waar de waarde naartoe gemapped kan worden. De DOAs dienen als een directe link naar de databasetabellen.

## Gebruik van Datamappers

De [datamappers](https://en.wikipedia.org/wiki/Data_mapper_pattern) hebben de rol van het omzetten van ruwe data uit de database naar Java-objecten. De data mappers zijn door middel van een [generic](https://en.wikipedia.org/wiki/Generics_in_Java#:~:text=Generics%20are%20a%20facility%20of,compile%2Dtime%20type%20safety%22.) interface geïmplementeerd aangezien de implementatie van de enige methode afhankelijk is van het type DTO dat geproduceerd moet worden. Het gebruik van de generic inteface zorgt ervoor dat de implementaties makkelijk vervangbaar zijn, waardoor de subsystemen niet veranderd hoeven te worden bij aanpassingen aan de data mappers.

De data mappers dienen ook om de S van [SOLID](https://en.wikipedia.org/wiki/SOLID), het “Single responsebility principle toe te passen. De verantwoordelijkheid van het omzetten van ruwe data naar Java-objecten ligt nu bij de data mappers en niet bij de DAO die zorgt voor het ophalen van de data.

## Gebruik van Data Transfer Objects (DTOs)

De DTOs zijn het middel van transportatie van de data binnen de applicatie. In het geval van Client naar API word de binnen gekomen data naar DTOs vertaald en voor het geval van data uit de database zal de datamapper DTOs aanleveren gevuld met de ruwe opgehaalde data.

Het voordeel van DTOs is de mogelijkheid om een grote hoeveelheden data in object vorm binnen de classes van de applicatie door te geven, dit beperkt het totaal aantal calls voor data wat voor een positief effect zal zorgen op de snelheid van de gehele applicatie.

## Gebruik van Dependency Injection

Om dependecy injection toe te passen in de applicatie is er gebruik gemaakt van de CDI API zoals beschreven als eis in de casus opdracht. De toepassing van dependecy injection heeft als voordeel dat de coupling omlaag gaat doordat de clients niet de kennis hebben over hoe de dependecies geïmplementeerd zijn. Dit heeft een positief effect op de herbruikbaarheid, testbaar- en onderhoudbaarheid.

## Gebruik van Dependecy Inversion Principle

Om de D van SOLID toe te passen is er gebruikgemaakt van het [Dependecy Inversion Principle (DIP)](https://en.wikipedia.org/wiki/Dependency_inversion_principle). De bedoeling van DIP is om de afhankelijkheid binnen de klassen van de applicatie te verlagen, in plaats dat een klasse afhankelijk is van een concrete implementatie is er nu een afhankelijkheid van een abstractie in de vorm van een Interface. Deze interfaces worden geïmplementeerd door andere klassen die dan de details bevatten.

Het gebruik is onder andere terug te zien in het bovenstaande Package diagram, zo staan de dependecies naar de interfaces van de laag eronder i.p.v. naar de directe implementaties. Een voorbeeld uit de applicatie is dat de login resource een interface van de login service aanspreekt, die op zij beurt weer een interface van de user DAO aanspreekt, die dit ook doet voor de gebruikte data mapper.

## Gebruik JAX-RS foutafhandeling

Er wordt op het gebied van fout afhandeling handig gebruikgemaakt van de al aanwezige JAX-RS API. Zo worden de JAX-RS excepties gebruikt in de inlog functionaliteit, waar als het een foutieve login poging is, er een NotAuthorizedException gegooid word. Deze zal door middel van de exceptiemappers van JAX-RS omgezet worden naar de juiste HTTP reponse en doorgestuurd worden naar de client.

## Gebruik van de Service Layer

In de eisen van de casus opdracht is gedefinieerd dat er gebruik gemaakt moet worden van een service layer, deze laag is er dus ook in de implementatie aanwezig. Echter is er in deze opdracht niet veel logica die in deze laag hoort, wel zou er een kans zijn dat er in de toekomst uitbreidingen gedaan worden aan de opdracht die meer logica vereisen waardoor het geen slechte keuze is om de laag alsnog opgenomen te hebben.

Het implementeren van de service laag puur om de toekomst mogelijkheden gaat wel in tegen het principe van clean code, waarbij er alleen functionaliteit wordt gerealiseerd als het nodig is.

# Kwaliteit

## Unittest

De applicatie is op alle lagen geunittest, dit heeft een totale code coverage van 86% van de klassen, 92% van alle methodes en 93% van alle lines. Hierbij is de DTO-package uitgesloten in de berekening omdat de Getters en Setters van de DTOs niet getest hoeven worden. Hier onder afbeeldingen van het test coverage report:

Figuur Totale coverage

Graphical user interface, application

Description automatically generated

Figuur Totale coverage per laag

De werking van de SQL queries zijn niet afgevangen door middel van Unittest, dit zou wel kunnen door bijvoorbeeld gebruik te maken van een in-memory database zoals DBUnit en hiermee integratie tests te schrijven die daadwerkelijk aan de hand van de werkende code en echte test data de calls test.

De werking van de SQL queries zijn in dit geval van ontbrekende integratietests, door de applicatie end-to-end te testen bewezen.

## API functionaliteit

Alle functionaliteiten beschreven in de READ.ME van de casus GitHub zijn geïmplementeerd, ook wordt er voldaan aan de beschreven eisen uit de casus opdracht.

## Conclusie

Kijkend naar de eis van minimaal 80% testcoverage, de eisen over functionaliteit en eisen over de wijze van implementatie is deze opdracht naar mijn mening zeer voldoende afgerond.