

Studentnummer: 562975  
klas: ITN-OOSE-A-F  
Course: OOSE-DEA  
Versie 1.0

Docent: Meron brouwers

26 maart 2020

SPOTITUBE

Auteur: Ruben Eppink

Inhoud

[1. Inleiding 2](#_Toc36213208)

[2. Package- en Deployment Diagram 3](#_Toc36213209)

[3. Ontwerpkeuzes 5](#_Toc36213210)

[3.1 Data Access Objects (DAO) 5](#_Toc36213211)

[3.2 Datamappers 5](#_Toc36213212)

[3.3 Dependency Injection (DI) 5](#_Toc36213213)

[3.4 Dependency Inversion Principle (DIP) 5](#_Toc36213214)

[3.5 Foutafhandeling 5](#_Toc36213215)

[3.6 Data Transfer Object (DTO) 6](#_Toc36213216)

[3.7 Clean Code 6](#_Toc36213217)

[3.8 Domein laag 7](#_Toc36213218)

[3.9 Database connection 7](#_Toc36213219)

[4. Zwakke punten en uitbreidingen 8](#_Toc36213220)

[4.1 Unittests 8](#_Toc36213221)

[4.2 Consistentie 8](#_Toc36213222)

[4.3 Statische Implementatie van functionaliteit 8](#_Toc36213223)

[4.4 connectie aanmaken en sluiten 9](#_Toc36213224)

[4.5 Loggen 9](#_Toc36213225)

[5. Conclusie 10](#_Toc36213226)

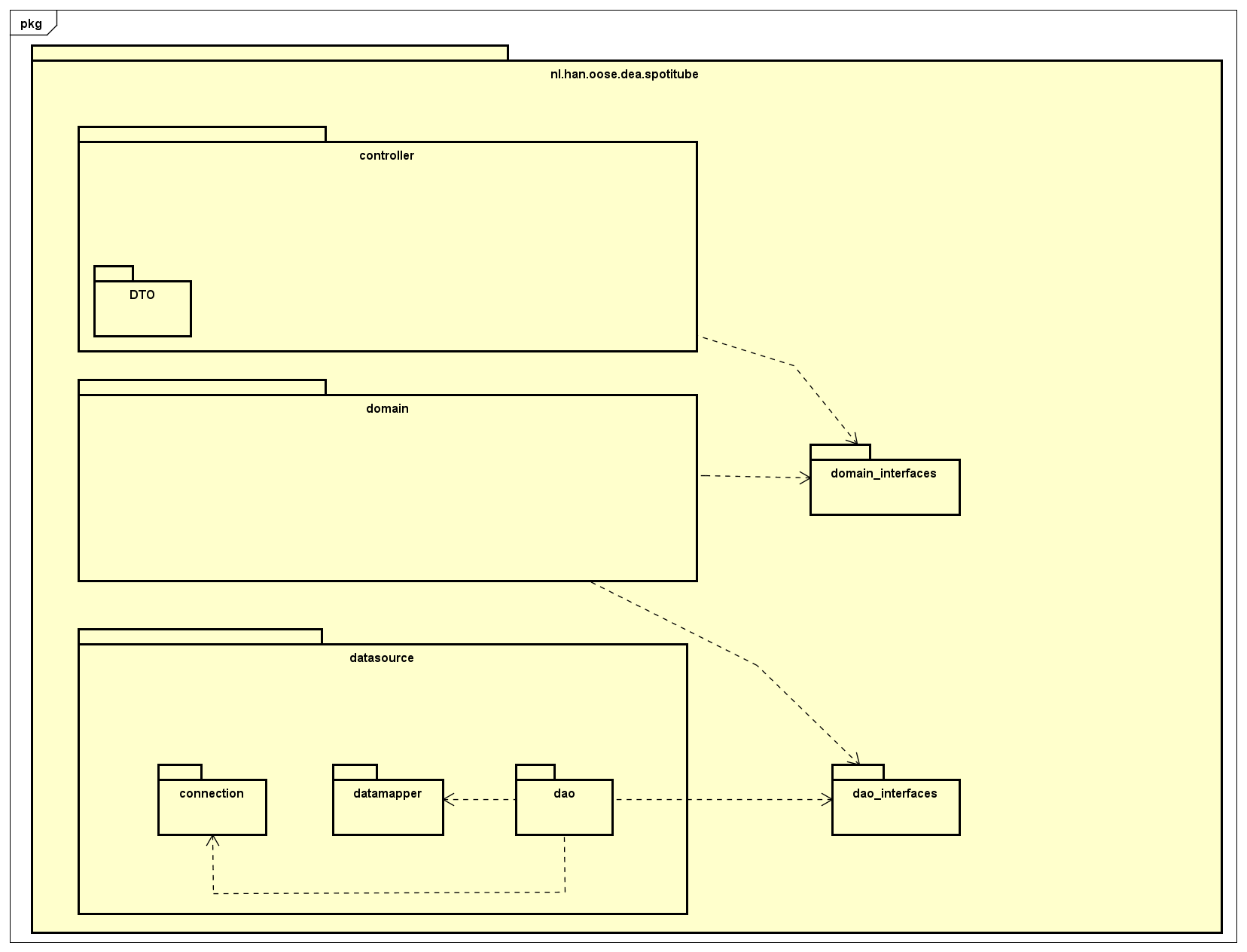
# 1. Inleiding

Vanuit het vak Distributed Enterprise Applications (DEA) uit het semester Object Oriented System Engineering (OOSE) is de opdracht gekomen om een RESTful API te bouwen voor de aangeleverde front-end applicatie: Spotitube.   
  
Tijdens het ontwikkelen van de API zijn er keuzes gemaakt die nadere toelichting verdienen. Deze zal ik onder meer toelichten aan de hand van een tweetal diagrammen:

* Deployment Diagram
* Package Diagram

Vervolgens breng ik de huidige tekortkoming van de applicatie in kaart op het moment van oplevering met eventuele oplossingen. Tot slot een conclusie over de kwaliteit van het product.

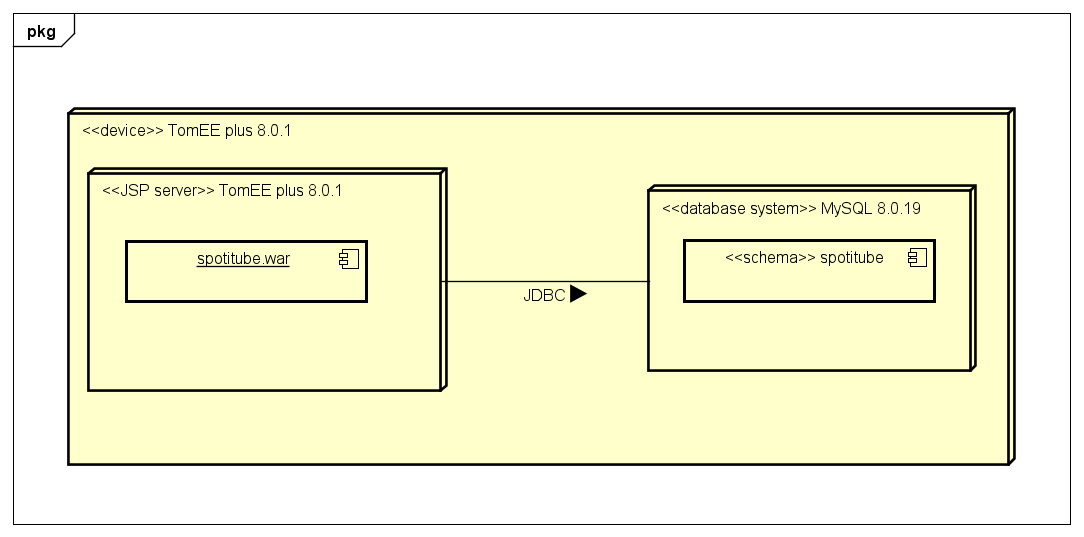
# 2. Package- en Deployment Diagram

Onderstaande package diagram geeft een duidelijke visuele representatie van de logische structuur in de applicatie door middel van afhankelijkheden tussen de verschillende lagen. 

Figuur Package Diagram Spotitube

Hieruit is bijvoorbeeld op te maken dat een laag alleen afhankelijk is van een laag onder zich. Dit zorgt voor een mooi los gekoppeld systeem ([loose coupling](https://en.wikipedia.org/wiki/Loose_coupling)), omdat de lagen alleen afhankelijk zijn van dat gene dat ze ook daadwerkelijk nodig hebben. Dit zorgt voor een onderhoudbaar en aanpasbaar systeem, doordat er geen onnodige aanpassingen aan afhankelijke systemen vereist zijn.

Al deze voordelen kan je als een direct gevolg zien van het [Facade Design Pattern](https://en.wikipedia.org/wiki/Facade_pattern) dat is toegepast tijdens het ontwikkelen van de applicatie. Ook het package diagram geeft hier een mooie visuele representatie, waarbij het duidelijke is dat de client alleen kennis hoeft te hebben van de controller en niet van eventuele ingewikkelde andere subsystemen van de API. Dit vergemakkelijkt de communicatie tussen de client en de API en heeft als gevold dat eventuele implementatie wijzigingen van de API, geen effect hebben op de client.



Figuur Deployment Diagram

# 3. Ontwerpkeuzes

Tijdens het ontwikkelen zijn er ook keuzes gemaakt met betrekking tot meer specifieke implementatie details die hieronder verder worden toegelicht.

## 3.1 Data Access Objects (DAO)

Bij het ophalen van data is gebruik gemaakt van het [Row Data Gateway Pattern](https://martinfowler.com/eaaCatalog/rowDataGateway.html). Dit wil zeggen dat elk rij in een databasetabel praktische gezien synoniem staat met een object in de applicatie. Dit is gerealiseerd door middel van DAO’s, die tevens ook de implementatie van de datasource weg maskeren voor de rest van de applicatie.

## 3.2 Datamappers

[Datamappers](https://martinfowler.com/eaaCatalog/dataMapper.html) hebben de verantwoordelijkheid om de data uit de database om te zetten naar een java-object. Deze functionaliteit is geïmplementeerd door middel van een generic interface, aangezien de implementatie afhankelijk is van het type dat hij moet produceren. Dit heeft als voordeel dat alle datamapper implementaties dezelfde interface kunnen gebruiken en er dus geen onnodige extra code ontstaat die onderhouden moet worden. Verder zorgt de interface voor een makkelijk vervangbare implementatie, die niet zal lijden tot het moeten aanpassen van de subsystemen die afhankelijk zijn van de datamapper.   
Een andere voordeel van deze implementatie is het verdelen van de verantwoordelijkheden. Zonder een datamapper is de DAO verantwoordelijk voor het omzetten van de data naar een java-object, naast het ophalen van data zelf. Dit gaat direct in tegen de ontwerpprincipes van SOLID, met name Single Responsibilty Principle, waarbij het doel is om elke klasse/methoden maar één taak te laten uitvoeren.

## 3.3 Dependency Injection (DI)

Door de gehele applicatie is gebruik gemaakt van de CDI API om [dependency injection](https://nl.wikipedia.org/wiki/Dependency_injection) toe te passen. Dit geeft als meer waarde dat een klasse niet meer verantwoordelijk is voor de creatie van een object, waardoor deze makkelijker vervangbaar wordt. Ook de kans dat je de klasse moet aanpassen wanneer je dit object vervangt, wordt steeds kleiner.

## 3.4 Dependency Inversion Principle (DIP)

Niet te verwarren met DI, is [DIP](https://en.wikipedia.org/wiki/Dependency_inversion_principle) één van de [SOLID-principes](https://en.wikipedia.org/wiki/SOLID). Dit principe schrijft voor dat een high-level module niet afhankelijk mag zijn van de implementatie van een low-level module, maar beide zouden afhankelijk moeten zijn van abstractie.   
Dit is terug te zien in de applicatie bij de implementatie van zowel de DAO- en domeininterfaces. Een implementatie van een domeininterface maakt gebruik van een DAO-interface, die is vervolgens weer gekoppeld aan een implementatie van deze interface.  
Dit draait dus de rollen om waar een domeinimplementatie direct afhankelijk is van één specifieke DAO-implementatie. Hierdoor is het bijzonder moeilijk om aanpassingen te maken in beide modules, omdat ze zo hecht gekoppeld zijn.

## 3.5 Foutafhandeling

In termen van foutafhandeling wordt gebruik gemaakt van de geleverde JAX-RS functionaliteit met betrekking tot het gooien van excepties. Deze implementatie is terug te zien bij de inlogfunctionaliteit waar een authenticatiefout wordt weergegeven en tijdens het gebruik maken van de database. Deze excepties worden door middel van de geleverde exceptiemappers van JAX-RS omgezet naar de correcte HTTP response en vervolgens doorgestuurd naar de client.

## 3.6 Data Transfer Object (DTO)

Vaak heeft een applicatie data uit een database gehaald, die weergegeven moet worden op het scherm of waar bepaalde handelingen op uitgevoerd moeten worden. Hier komt het [DTO](https://martinfowler.com/eaaCatalog/dataTransferObject.html) van pas. Een DTO is in dit geval de Java representatie van de verzameling data uit de database, die door meerdere lagen doorgegeven kan worden. Dit heeft als voordeel dat een grote groep data in één keer uit de database gehaald kan worden en vastgelegd wordt in een object, waardoor het aantal oproepen naar de database tot een minimum beperkt wordt. Dit zorgt voor een drastische optimalisatie van de applicatie in termen van snelheid.

## 3.7 Clean Code

Bij het uitwerken van de applicatie is onder andere rekening gehouden met de principes die komen kijken bij het schrijven van clean code, zoals de eerder benoemde [SOLID-principes](https://en.wikipedia.org/wiki/SOLID), maar ook KISS en DRY spelen hier een belangrijke rol.  
  
Dit kan je terug zien in de naamgeving van zowel de packages, die voldoen aan de conventies van Java, als de klasse en methodes die een beschrijvende consistente naamgeving hebben, waardoor het makkelijk wordt voor andere ontwikkelaars om de code te begrijpen  
  
[Keep It Stupid Simple](https://en.wikipedia.org/wiki/KISS_principle) (KISS) heeft als kern uitgangspunt dat de code zo simpel mogelijk, maar niet te simpel, geschreven moet worden. Dit kan je zien als direct gevolg van de eerder toegelichte applicatiestructuur, waarbij goed is nagedacht over de rol en verantwoordelijkheid op zowel laag- als klasse niveau. Hierdoor ontstaan er geen onnodige afhankelijkheden, die de code moeilijker maken dan dat nodig is. Combineer dit met het SRP uit SOLID, waarbij elke klasse/methode maar één verantwoordelijkheid heeft en dan krijg je simpele doelbewuste code.

Al deze principes zijn moeilijk van elkaar te beschrijven, omdat ze vaak op een goede manier invloed op elkaar hebben. Zo heeft [Don’t Repeat Yourself](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself) (DRY) ook veel invloed gehad op de leesbaarheid van de code, waarbij onnodige code duplicatie voorkomen wordt door het implementeren van een extra methode. Dit creëert korte leesbare methodes, die makkelijk hergebruikt kunnen worden.   
Dit is concreet terug te zien in de DAO’s die voor het uitvoeren van een actie op de database, een methode aanroep doen, die makkelijk herbruikbaar is door een andere methode met een vergelijkbare actie.

## 

## 3.8 Domein laag

Afhankelijk van hoe je tegen de opdracht aankijkt, is het implementeren van een [domein laag](https://martinfowler.com/eaaCatalog/domainModel.html) volstrekt overbodig. Toch is er gekozen voor deze implementatie als gevolg van een aantal overwegingen.  
  
Het biedt een mogelijkheid voor uitbreiding in de toekomst. Momenteel is er nauwelijks tot geen applicatielogica aanwezig, maar een applicatie als Spotitube heeft een grote kans om tot een aanzienlijk ingewikkelde applicatie uit te groeien, als je kijkt naar zijn inspiratiebronnen. Hierdoor zou je kunnen redeneren dat het onvermijdelijk is, om deze laag in je applicatie te implementeren. Echter gaat dit wel in tegen de principes van clean code, waarbij je alleen functionaliteit moet implementeren die je ook daadwerkelijk nodig hebt.   
  
Het zorgt voor een makkelijkere toepassing van SRP, waarbij de controller alleen verantwoordelijk is voor het doorgeven van de informatiebehoefte, het domein is verantwoordelijk voor het bij elkaar zoeken van de juiste informatie op basis van logica en gebruikt hierbij de datasource die verantwoordelijk is voor het aanleveren van de informatie.  
  
Zo is er dus op basis van deze overwegingen toch gekozen om deze laag te implementeren.

## 3.9 Database connection

De DAO’s maken gebruik van een interface DBConnection, waar momenteel een implementatie van een SQLDBConnection van is. Dit maakt het databasesysteem makkelijk vervang- of uitbreidbaar door andere systemen.

# 4. Zwakke punten en uitbreidingen

Vrijwel elke nieuwe ontwikkelaar streeft naar perfectie in zijn/haar producten. Helaas is de perfecte applicatie slechts een illusie aan het licht gebracht door een te klein budget, of in dit geval tijdsdruk. Echter kan een ontwikkelaar zijn applicatie sterker maken door het documenteren van alle zwakke punten en/of mogelijke uitbreidingen. Dit zorgt ervoor dat eventuele opvolgers efficiënter te werk kunnen gaan in het streven naar de perfecte applicatie (of studenten die dit van GitHub jatten en niet dezelfde fouten willen maken).

Verrassend genoeg kent ook mijn uitwerking van Spotitube een aantal pijnpunten die aan het licht gebracht moeten worden.

## 4.1 Unittests

Er wordt helaas niet voldaan aan de verwachtte eis van 80% code coverage. Dit heeft een aantal oorzaken:

* De DTO’s bevatten heel veel getters en setters die te triviaal zijn om te testen, waardoor de line coverage flink omlaag gaat.
* De DAO’s bevatten een aantal afhankelijkheden van buitenaf die moeilijk te mocken zijn, waardoor deze niet getest worden.

De laatste oorzaak kan opgelost worden door een integrale test te maken waarbij je een in-memory database gebruikt. Hiermee zou je alle functionaliteit afdekken met unittests en het gevraagde percentage kunnen behalen.   
  
Ondanks dat de applicatie niet volledig getest wordt door middel van unittests, is er wel een end to end test gedaan om ervoor te zorgen dat alles werkt naar behoren.

## 4.2 Consistentie

Eerder in het document, in hoofdstuk 3.7, spraken we over clean code en bijbehorende principes. Eén van de belangrijkste kernpunten die in elk principe terugkomt, is het consistent zijn in de keuzes die je maakt. Momenteel zijn er nog enkele methodes in de applicatie die niet consistent zijn met de rest qua naamgeving. Dit kan voor verwarring zorgen bij het lezen van de code en kan vrij makkelijk verholpen worden.

## 4.3 Statische Implementatie van functionaliteit

Momenteel zijn er twee functionaliteiten die niet voldoen aan de eisen van de opdracht:

* Generen van een unieke token
* Berekenen van de lengte van alle afspeellijsten

De eerste is simpelweg niet geïmplementeerd en zou een aantal aanpassingen aan de database- en applicatiestructuur nodig hebben. De tweede is momenteel statisch in geprogrammeerd om het effect van een werkende applicatie tot stand te houden voor test doeleinde. Ook deze functionaliteit zou een aantal aanpassingen op applicatieniveau nodig hebben om te implementeren. Denk bijvoorbeeld aan een extra methode die een door middel van een query de totale speelduur ophaalt vanuit de database.

## 4.4 connectie aanmaken en sluiten

Momenteel wordt bij het aanmaken van een DAO een connectie opgezet, die door één van de methodes gebruik kan worden en deze connectie vervolgens weer sluit. Dit is mogelijk omdat de server telkens een nieuwe instantie aanmaakt per verzoek van de client. Het probleem komt kijken wanneer er meerdere keren een oproep wordt gedaan naar een specifieke DAO, waarbij de tweede aanroep geen zinnige connectie heeft, omdat deze al is gesloten door de vorige en er in de tussentijd geen nieuwe instantie is aangemaakt die automatisch een connectie genereert.  
Dit gaat wel goed als je de connectie niet sluit, waardoor hij meerdere malen hergebruikt kan worden en uiteindelijk door de zogenoemde garbage collector van Java opgeruimd wordt. In grote applicaties zal dit voor prestatievermindering zorgen, doordat je een groot aantal ongebruikte connecties in het geheugen hebt zitten.

Een goede oplossing zou dus zijn om aan het begin van elke DAO methode een nieuwe connectie aan te maken en deze ook weer te sluiten.

## 4.5 Loggen

Op een enkele locatie na, is er geen gebruik gemaakt van logger. Implementatie hiervan zou enorm kunnen helpen bij het vastleggen van foutmelding en de toestand van het programma op dat moment. De voordelen in vergelijking met gooien van een exceptie is dat mate van controle over het formaat en hoeveelheid waarmee je fouten vastlegt, ook kan je zo indelen in verschillende categorieën, waardoor er efficiënter gezocht kan worden naar het probleem.

# 5. Conclusie

Achteraf gezien, in mijn ogen, is Spotitube een opdracht waar meer denkwerk in hoort te zitten dan daadwerkelijk implementatiewerk. Dit denkwerk heb ik dan ook zo goed mogelijk proberen te beschrijven door middel van voorbeelden en diagrammen. Koppel dit met de uitwerking van de applicatie zelf en je hebt een degelijk product, waarbij ook de zwakke punten in kaart zijn gebracht.