29/05/2020

Opleverdocument spotitube

Door: Jelmer van Vugt

Inhoudsopgave

[1. Inleiding 2](#_Toc41654177)

[2. Packagediagram 3](#_Toc41654178)

[3. Deployment diagram 4](#_Toc41654179)

[4. Ontwerp-/Craftmanship -keuzes 5](#_Toc41654180)

[4.1 Lagen 5](#_Toc41654181)

[4.2 Datasource 5](#_Toc41654182)

[4.3 Data Transfer Objects 5](#_Toc41654183)

[4.4 Interfaces 6](#_Toc41654184)

[4.5 Information hiding 6](#_Toc41654185)

[4.6 Feedback implementatie 6](#_Toc41654186)

[4.6.1 Servicelaag 6](#_Toc41654187)

[4.6.2 Errorhandling Database 6](#_Toc41654188)

[4.6.3 Util Package 6](#_Toc41654189)

[5. Bronnenlijst 8](#_Toc41654190)

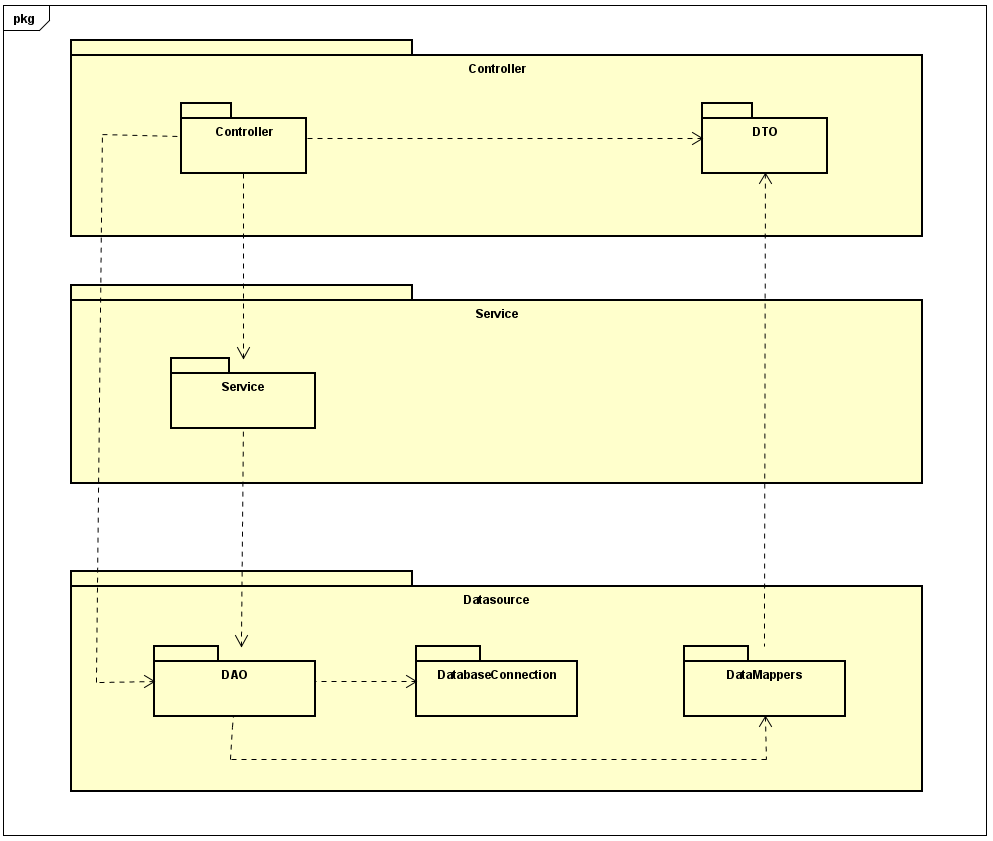
# 1. Inleiding

In dit document zal de opgeleverde Java Enterprise Edition applicatie ‘Spotitube’ besproken worden. Deze maakt deel uit van de HBO-ICT – Software Engineering studie uit zoals gegeven op de Hogeschool van Arnhem en Nijmegen. Deze desbetreffende eisen die verbonden zijn aan deze opdracht zijn te vinden in [de bronnenlijst](#_5._Bronnenlijst).

Eerst zal er worden verteld over de package structuur van de applicatie. Daarna zal gesproken worden hoe de applicatie op de juiste manier in gebruik kan worden genomen inclusief het deployment diagram. Tot slot worden er in het laatste hoofdstuk verschillende ontwerpkeuzes toegelicht.

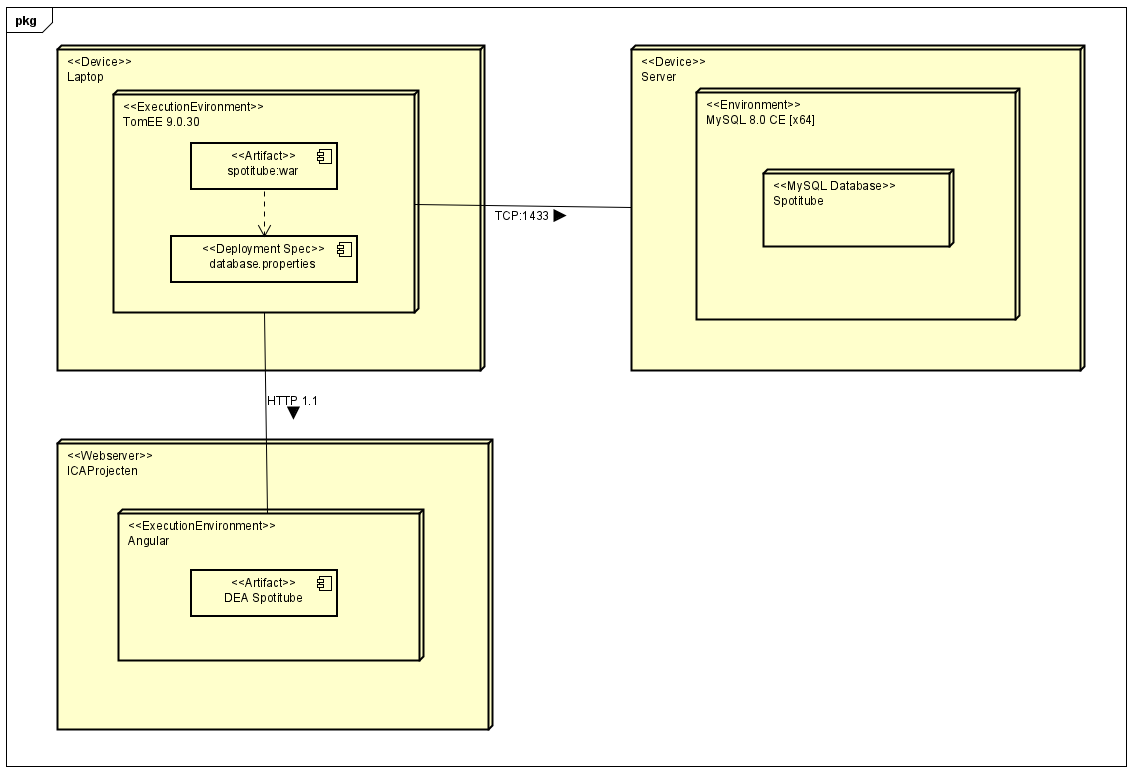
# 2. Packagediagram

Door middel van een package diagram te maken wordt overzichtelijk gemaakt hoe verschillende onderdelen van de applicatie interactie met elkaar aangaan. Ook wordt de coupling en cohesie van de applicatie hierdoor helder.



Figuur 1: Package diagram

# 3. Deployment diagram

  
In het deployment diagram moet de omgeving duidelijk worden waarin de applicatie draait. Hieronder zijn de vereisten die hieraan verbonden zijn weergeven in een deployment diagram.

Figuur 2: Deployment diagram

# 4. Ontwerp-/Craftmanship -keuzes

In dit hoofdstuk zal ik verschillende ontwerpkeuzen documenteren en aangeven dat ik heb gewerkt volgens de clean code principes.

## 4.1 Lagen

Ten eerste is de applicatie gescheiden in drie lagen, namelijk: de controller laag, de service laag en de datasource laag. Deze verschillende lagen zijn op zo’n manier gebouwd dat deze zo min mogelijk van elkaar afhankelijk zijn. Dit heeft als gevolg dat de verschillende delen van de applicatie makkelijker onafhankelijk van elkaar te testen zijn. Ook creëert dit de mogelijkheid om de applicatie in de toekomst makkelijker uit te breiden.

Hert moet echter natuurlijk wel mogelijk zijn voor deze lagen om samen een geheel te vormen en met elkaar samen te kunnen werken. Dit wordt gedaan door het aanmaken van instanties die klassen van elkaar hebben te laten afhandelen door de inject functie van CDI.

## 4.2 Datasource

De verantwoordelijkheid van de datasource ligt in het feit dat hij moet praten met de database. In deze laag wordt constant data heen en weer gestuurd tussen de applicatie en de database door middel van prepared statements. Echter vond ik het een slechte implementatie om hier query’s hard in te zetten. Door dit te doen hoeft er maar een verandering plaats te vinden aan de database kant en je hebt een foutmelding te pakken. Ik heb dit opgelost met stored procedures aan de database kant. Als er nu een wijziging plaatsvindt binnen de database zal dit niks uitmaken voor de applicatie. Zolang de database ervoor zorgt dat de stored procedures beschikbaar en actueel blijven kan de applicatie constant blijven.

Tevens heeft dit het bijkomende effect dat er business logica kan worden verplaatst naar de database, wat naar mijn mening mooier is. De applicatie hoeft hierdoor amper iets te weten van de database wat op zijn beurt twee lagen van elkaar scheid en de tight-coupling wordt verlaagd.

Het laatste effect dat dit heeft is het verhogen van de veiligheid van de database. Omdat de applicatie alleen maar met de database praat via stored procedures is er geen mogelijkheid om query’s te doen die wijzigingen doen in de structuur van de database. De database kiest zelf wat hij publiek maakt aan de gebruiker.

## 4.3 Data Transfer Objects

Als er vanuit de datasource laag een stored procedure wordt aangeroepen en deze resultaten teruggeeft wordt dit gedaan in de vorm van een resultset. Een resultset is alleenstaand binnen Java niet heel vriendelijk om mee te werken, dit is waarom er gebruik wordt gemaakt van DTO’s (data transfer objects). Data transfer objects zijn Java objecten die precies dezelfde structuur hebben als de binnenkomende resultset. Door hun getters en setters zijn deze makkelijk in gebruik te nemen door functies. Ook kan hij makkelijk als argument worden doorgegeven naar een andere functie.  
Om het nog makkelijker te maken om hiermee te werken zijn datamappers gerealiseerd. Datamappers zijn klassen die door de functies uit de datasource laag worden aangeroepen om inkomende resultsets vanuit de database naar het gewilde DTO te converteren.

## 4.4 Interfaces

Een van de dingen die je tijdens het programmeren het liefst zoveel mogelijk wilt vermijden is herhalende code. Hier heb ik onder andere rekening mee gehouden tijdens het schrijven van de data mappers en custom exceptions. Ondanks dat zowel de mappers als de exceptions verschillende implementaties kunnen hebben nadat ze worden uitgewerkt is hun essentie hetzelfde. Dit is waarom ik heb gekozen om voor zowel de datamappers en de exceptions een interface te maken die door de verschillende instanties geïmplementeerd dient te worden.

## 4.5 Information hiding

Tijdens het schrijven van klassen is er nagedacht over hun essentie en op basis hiervan variabelen en/of methoden private gemaakt. Dit is vooral terug te vinden in de Playlist- en LoginDAO klassen. Binnen deze klassen roepen methoden vaak andere interne functies aan waar de buitenkant niks vanaf hoeft te weten. Om deze reden zijn deze functies prive gemaakt.   
Het afsplitsen van deze functies heeft er ook voor gezorgd dat elke functie zoveel mogelijk een enkele verantwoordelijkheid hebben.

## 4.6 Feedback implementatie

Na de eerste beoordeling van deze opdracht zijn er door de beoordelende docent verschillende verbeterpunten aangekaart die ik in de nieuwe versie heb verwerkt. Deze zullen hier benoemd worden.

### 4.6.1 Servicelaag

Origineel is de servicelaag bedoelt om de business logica binnen de applicatie te implementeren. Als er echter geen business logica valt te implementeren dient deze laag enkel als een extra klasse die informatie doorspeelt tussen de controller – en datasource laag. Als dit het geval is heeft de desbetreffende service laag geen nut en mag deze verwaarloost worden.   
Om deze reden is zijn de service lagen tussen de Playlist- en TrackController en Playlist- en TrackJDBC uitgehaald. Deze communiceren nu direct met elkaar. Omdat de LoginService wel logica implementeert is deze behouden.

### 4.6.2 Errorhandling Database

In mijn eerste versie van Spotitube werden foutmeldingen vanuit de database wel opgevangen maar werd hier niks mee gedaan. Als er in de huidige versie iets fout gaat in de database properties of de ConnectionService wordt een custom exceptie en bijbehorende statuscode naar de controller laag geretourneerd.

### 4.6.3 Util Package

Origineel had ik een ‘util’ package gebruikt om de klassen te waarborgen die zorgden voor de connectie met de database. Deze benaming is echter te algemeen wat in een later traject kan zorgen voor een rommelige package structuur. Dit is waarom hij nu de nieuwe naam ‘DatabaseConnection’ heeft gekregen.  
Ook is hier de klasse ConnectionService hierin toegevoegd. Deze stond origineel in de service package maar aangezien deze hier niet thuishoort is hij verplaatst.

# 5. Bronnenlijst

University of Applies Sciences, HAN. (27 juli, 2017) *Spotitube.* Via: <https://github.com/HANICA-DEA/spotitube>. Geraadpleegd op: 29 mei, 2020.