



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.

Spotitube

Opleverdocument

Lotus ter Haar (579228)

Course: OOSE-DEA

Docent: Uwe van Heesch

**Inhoud**

[1. Package Diagram 1](#_Toc4960897)

[Layers Pattern 2](#_Toc4960898)

[2. Deployment Diagram 3](#_Toc4960899)

[3. Design Keuzes 4](#_Toc4960900)

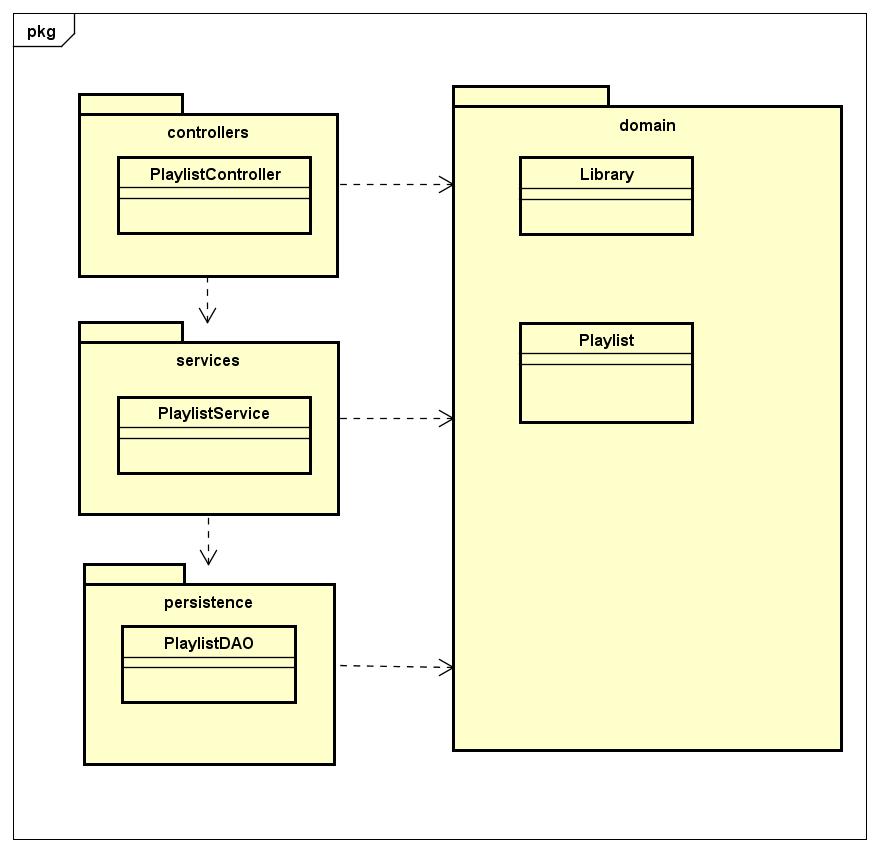
[CDI -Injection: verlagen van afhankelijkheid door dependency injection 4](#_Toc4960901)

[Data Mapper 4](#_Toc4960902)

[Identity Map en Separated Interface werkend met JDBC en MySQL 5](#_Toc4960903)

[Service Layer is een vorm van Application Facade 5](#_Toc4960904)

[Remote Facade pattern door middel van REST services 6](#_Toc4960905)

******1. Package Diagram**  
In deze package diagram zie je de dependencies tussen klassen en tussen packages

Een package is een verzameling van logisch gerelateerde UML elementen. In dit geval bevat elke package meerdere klassen.

In de package diagram is duidelijk te zien dat er het layer pattern wordt gebruikt. Layers zijn een logische manier van het groeperen van componenten. Packages zijn daarentegen een fysieke manier van het groeperen van componenten . In deze package diagram lijkt de package structuur op de layer structuur. De klassen zijn opgedeeld in lagen. Deze lagen bestaan uit de controller laag, service laag en persistence laag. Alle lagen kunnen gebruik maken van de klassen die zich in de domain package bevinden.

Een layer is een groep van klassen die dezelfde verzameling van dependencies heeft naar andere modules. Dus een groep van hergebruikbare componenten die herbruikbaar zijn in andere omstandigheden. Elke layer heeft een specifieke rol en verantwoordelijkheid binnen de applicatie.

Bijvoorbeeld presentation layer hoeft niet te weten hoe de data verkregen moet worden, het hoeft enkel de informatie weer te geven op het scherm.

De layers zijn gesloten. Dat betekent dat een request van laag naar laag beweegt, de request gaat van de ene laag naar de laag er meteen onder totdat het uiteindelijk de database layer raakt.

Er wordt dus gebruik gemaakt van het concept *layers of isolation*. Dit concept houdt in dat veranderingen die in één layer worden gemaakt geen invloed hebben op componenten in andere layers: de verandering is geïsoleerd tot componenten binnen die laag en mogelijk een andere geassocieerde laag.

Bijvoorbeeld: klassen in de controllerlaag handelen alle REST requests af, terwijl de presistencelaag de gegevens ophaalt iot de database.

## **Layers Pattern**

**Context:** Je werkt met een groot complex systeem en wilt die complexiteit beheren door decompositie.

**Probleem:** Het systeem heeft verschillende niveau’s van abstractie. Requests gaan naar beneden en notificatie gaat omhoog. Het is mogelijk om stabiele interfaces te definieren tussen layers.

Het moet mogelijk zijn om layers later nog te veranderen of toe te voegen.

**Oplossing:** Gebruik het layers pattern. Elke layer zou samenhangend en op hetzelfde niveau van abstractie moeten zitten. Elke layer zou los verbonden moeten zijn met de lagen eronder.

Per laag kun je interfaces specificeren. Errors worden behandeld in de laagst mogelijke laag.

**Alternatieven:**

* **CQRS** (Command and Query Responsibility Segregation) is een alternatief voor het layers pattern. Het idee van dit pattern is om lees en schrijf operaties volledig te scheiden.

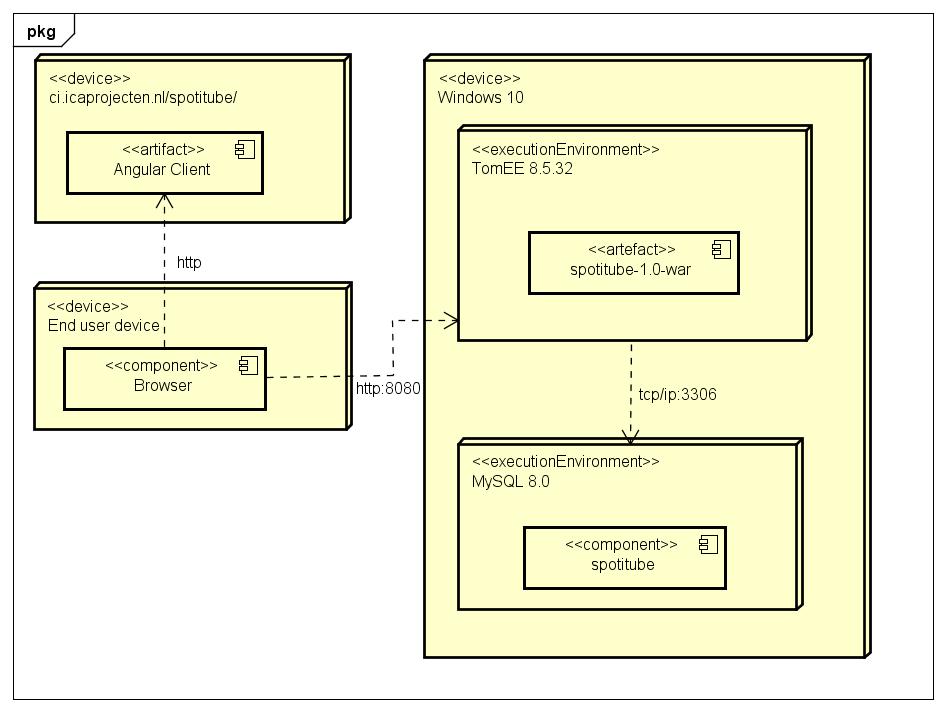
Dat betekent dat het model gebruikt voor schrijf operaties, anders zal zijn van de lees modellen (queries). Daarnaast wordt de data opgeslagen in verschillende locaties.

**Nadeel van CQRS:** het gesynchroniseerd houden van de command en lees modellen kan complex zijn.

* **Event Sourcing** is een pattern waarbij je de huidige staat van je model niet opslaat in de database, maar de veranderingen (events) aan het systeem.

**Nadeel van Event Sourcing:** Er is geen makkelijke manier om te zien wat de ‘persisted state’ is van je applicatie en je kan er ook niet makkelijk over query’en. Alleen run-time zie je wat er gebeurt.

# **2. Deployment Diagram**



Deployment diagrams bestaan uit verschillende UML-vormen: node, associaties en componenten. Nodes ( knooppunten), de drie dimensionale dozen, die representeren de fundamentele software- of hardware-elementen in het systeem. Lijnen van het ene knooppunt naar het andere geven relaties aan en de kleinere vormen in de dozen representeren de software-artefacten die worden ingezet.

Een node is bij deze deployment diagram de doos met ‘device’ of ‘executionEnvironment’ als stereotype. Het ‘device stereotype is hier een server, of een user computer. Het ‘executionEnvironment’ stereotype is de omgeving waarin de component in draait. Dit een webbrowser zijn zoals Google Chrome, of een webserver zoals TomEE.

Naast een node zijn er ook artifacten. Artifacten zijn een collectief aan bestanden die door middel van deployment op de desbtreffende server kunnen draaien. Tussen een aantal componenten bevinden zich associaties. Deze associaties geven aan met welk protocol de nodes communiceren.

Door middel van deze protocollen kan er data uitgewisseld worden tussen de componenten, bijv. in JSON formaat.

De applicatie werking is als volgt: de gebruiker start de applicatie op, er zal dan een HTTP protocol worden verstuurd naar de Angular Client. Deze server geeft de webpagina terug met de informatie.

Door acties uit te voeren op de webpagina zullen er ook HTTP protocollen worden verstuurd naar de back-end server Windows 10. Deze server zal benodigde ionformatie met een TCP/IP protocol ophalen bij de MySQL database server en dan opsturen naar de web browser van de gebruiker.

**Alternatieve oplossing:** Het zou beter zijn om de Windows 10 web server en de MySQL database server gescheiden te houden. Mocht één van de servers kapotgaan dan, dan is de content gescheiden en gaat niet alles verloren. Voor de Spotitube assignment werken we op 1 device, dus is dit niet haalbaar.

# **3. Design Keuzes**

## **CDI -Injection: verlagen van afhankelijkheid door dependency injection**

**Inversion of Control (IOC):** In plaats van objecten die andere objecten aanroepen, worden de afhankelijke objecten toegevoegd door een externe entiteit/container. Inversion of Control is de algemene stijl om Dependency Injection te gebruiken om de objecten in een applicatie te versturen.

Voorkomt het creëren van hardgecode objecten en het opzoeken van object/service. Helpt om effectieve unit testen te schrijven.

Er zijn twee soorten benaderingen:

* Traditionele benadering (de ‘pull’ benadering): Hierbij wordt er gebruik van directe instantiatie gemaakt. Er is gebruik van het Factory pattern nodig voor implementatie.
* *Dependency Injection* benadering (‘Push’ benadering ): iets buiten het object pushed zijn dependencies erin. Het object heeft geen kennis van hoe het zijn dependencies krijgt, het verondersteld dat ze er zijn.

**Dependency injection** is een stijl van object configuratie waarbij de object velden en collaborateurs worden vastgesteld door een externe entiteit.

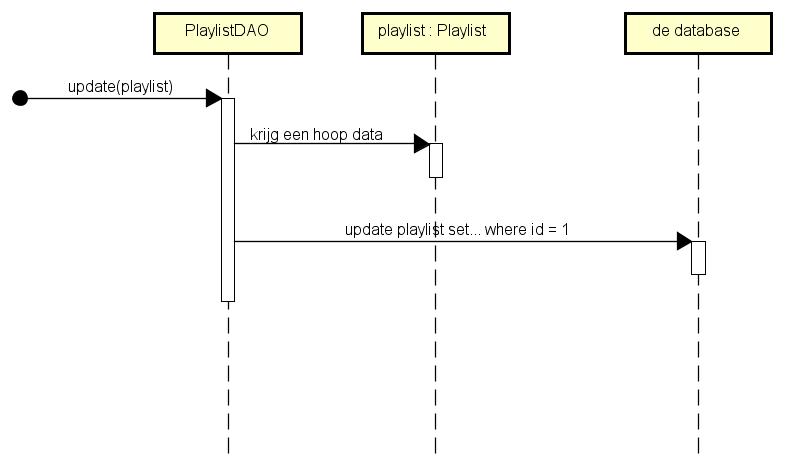
* Dependency injection is de manier van het injecteren van dependencies in een object
* Alle dependency logica wordt runtime uitgevoerd en wordt opgenomen in modules.
* Beans definieren hun depenciies door constructutor argumenten of eigenschappen.

Voordelen Dependency injection: Losse koppeling wordt bereikt doordat je geen dependencies hardcode tussen layers en modules. In plaats daarvan configureer je ze buiten de code. Dit maakt het gemakkelijk om over te stappen naar een nieuwe implementatie van een service. Of een module af te breken en ergens te hergebruiken.

Testbaarheid wordt verbeterd omdat je objecten niet weten wat de omgeving is waarin ze zijn, zolang iemand hun dependencies injecteert. Op die manier kan je objects deployen in een test omgeving en met gemak mocks injecteren voor hun dependencies.

**Alternatief voor CDI:** Een alternatief voor CDI is het object direct aanroepen door een nieuw object aan te maken**.**

## **Data Mapper**

De Data Mapper is een laag van software die de objecten in het geheugen van de database scheidt. Zijn verantwoordelijkheid is om de data tussen beide over te brengen en ze van elkaar te isoleren. Met de Data Mapper hoeven objecten in het geheugen niet eens te weten dat er een database aanwezig is, ze hebben geen SQL interface code nodig, en zeker geen kennis van het database schema. De Data Mapper zelf is onbekend aan de domein laag.

## **Identity Map en Separated Interface werkend met JDBC en MySQL**

Het **identity map pattern** is een database access pattern die wordt gebruikt om alle objecten die zijn ingelezen van de database bij te houden in een enkele business transaction. Wanneer er een object nodig is, wordt eerst de Identity Map gecheckt om te zien of deze er al is.

Database sessions hebben een identity map, die de object identity handhaaft en handelt als een cache. De Java applicatie heeft toegang tot de database door in te loggen tot de database door middel van een JDBC (Java Data Base Connectivity) driver. Een Java-programma kan in SQL communiceren met een database.

Het **Separated Interface pattern** definieert een interface in een afzonderlijk package van zijn implementatie. Het ontkoppelt delen van een systeem. Bijvoorbeeld in een layered systeem hangt de Domein Layer af van de Data Source Layer. Maar de data source layer heeft geen acces van de Domein layer. Het Separated Interface kan dan toegepast worden om deze wel toegang te geven.

## **Screen Shot 2015-06-16 at 12.34.18.pngService Layer is een vorm van Application Facade**

De applicatie facade klasse is verantwoordelijk voor het praten met het domein model en het krijgen van de informatie naar de presentatie klasse in precies de vorm die de presentatie klasse vereist.

De Service Layer defineert de grenzen van de applicatie en zijn verzameling van aanwezige operaties van perspectief van interfacing client layers. Het verdeeld de business logica in domein logica en applicatie logica. De applicatie facade klasse is verantwoordelijk voor het communiceren met het domein model en het krijgen van de informatie naar de presentatie klasse in precies de vorm die de presentatie klasse vereist. Op deze manier hoeft de presentatie klasse niets te weten over wat er omgaat in het model, het handelt alleen de UI af.

## **Remote Facade pattern door middel van REST services**

De GoF definitie van het Remote Facade design pattern is: “*Provides a coarse grained facade on fine grained objects to improve efficiency over a network”*

De applicatie implementeert de REST API Facade. Een API facade is een simpele interface tot een complex probleem. REST (Representational State Transfer) is een software architectuur stijl die een verzameling van constraints definieert die gebruikt worden voor het creëren van Web services.