Documentatie Spotitube

Naam: Ciyan Çöcelli

Studentnummer: **548189**

Klas: **herkanser**

Groepsnummer: **n.v.t.**

Docent: **T.B.A.**

Course: **DEA**

Datum: **19 maart 2020**

Versie: **2.0**

Semester: **OOSE**

*Hogeschool van Arnhem en Nijmegen*

Inhoudsopgave

[1. Inleiding 3](#_Toc35698920)

[2. Requirements 4](#_Toc35698921)

[3. Package Diagram 6](#_Toc35698922)

[3.1. Diagram 6](#_Toc35698923)

[3.2. Requirements 7](#_Toc35698924)

[3.3. Design Patterns & Ontwerpkeuzes 7](#_Toc35698925)

[3.3.1 Controllers los getrokken van services 7](#_Toc35698926)

[4. Deployment Diagram 9](#_Toc35698927)

[5. Craftsmanship & Ontwerpkeuzes 10](#_Toc35698928)

[4.1. ORM 10](#_Toc35698929)

[4.2. HashMap voor het bijhouden van gebruiken 10](#_Toc35698930)

[4.3. Naamgeving interfaces 10](#_Toc35698931)

[6. Conclusie 11](#_Toc35698932)

[Literatuurlijst 12](#_Toc35698933)

# 1. Inleiding

Dit document beschrijft de verschillende ontwerpkeuzes tijdens het ontwikkelen van de spotitube applicatie.

Spotify en Youtube slaan de handen ineen en werken gezamenlijk aan een applicatie genaamd Spotitube. Een klant kan hiermee een overzicht krijgen van afspeellijsten met audio- en videostreams. De opdrachtgever wil eerst een gedeeltelijke back-end applicatie ontwikkelen. Zij willen de applicatie testen via een bestaande webapplicatie.

Als eerste breng ik het package diagram in beeld. Dit diagram geeft de structuur weer van de verschillende packages. Aan de hand van het diagram laat ik zien welke requirements het raakt. Daarna geef ik het deployment diagram weer. Het deployment diagram laat zien hoe de applicatie draait op een server. Hierna benoem ik de gemaakte ontwerpkeuzes tijdens het ontwikkelen van de applicatie. Als laatste concludeer ik het verslag.

# 

# 2. Requirements

Dit hoofdstuk beschrijft de functionele en non functionele requirements van het Spotitube project. De opdrachtgever heeft verschillende eisen. Ik kies ervoor om de requirements met behulp van FURPS+ in te delen. Een systematische aanpak helpt bij het indelen van requirements. Tabel 1 geeft de verscheidene requirements weer.

|  |  |
| --- | --- |
| **Functional**  **Requirements** | **Beschrijving** |
| Fr#1 | Het systeem moet Restful kunnen communiceren volgens de REST API specificatie. |
| **Non-functional**  **Requirements** | |
| **Supportability** | |
| Sr#1 | Het systeem moet meerdere verschillende relationele databases ondersteunen. |
| Sr#2 | Het wisselen van database moet mogelijk zijn zonder het systeem opnieuw te moeten compileren. |
| Sr#3 | Het systeem bevat minimaal 80% code coverage. |
| Sr#4 | Het systeem moet gedeployed kunnen worden op Apache TomEE Plus. |
| **+ (plus)** | |
| **Implementation** | |
| Impl#1 | Het systeem moet gebruik maken van een data access layer. |
| Impl#2 | Het systeem moet gebruik maken van een domain layer. |
| Impl#3 | Het systeem moet gebruik maken van het Remote Facade pattern door middel van REST Services. |
| Impl#4 | Het systeem moet gebruik maken van het Data Mapper Pattern. |
| Impl#5 | Het systeem moet gebruik maken van het Separated Interface Pattern. |
| Impl#6 | Het systeem moet gebruik maken van dependency injection om de afhankelijkheid te verlagen. |
| Impl#7 | Het systeem moet gebruik maken van JAX-RS v2.0. |
| Impl#8 | Het systeem moet gebruik maken van Context & Dependency Injection (CDI). |

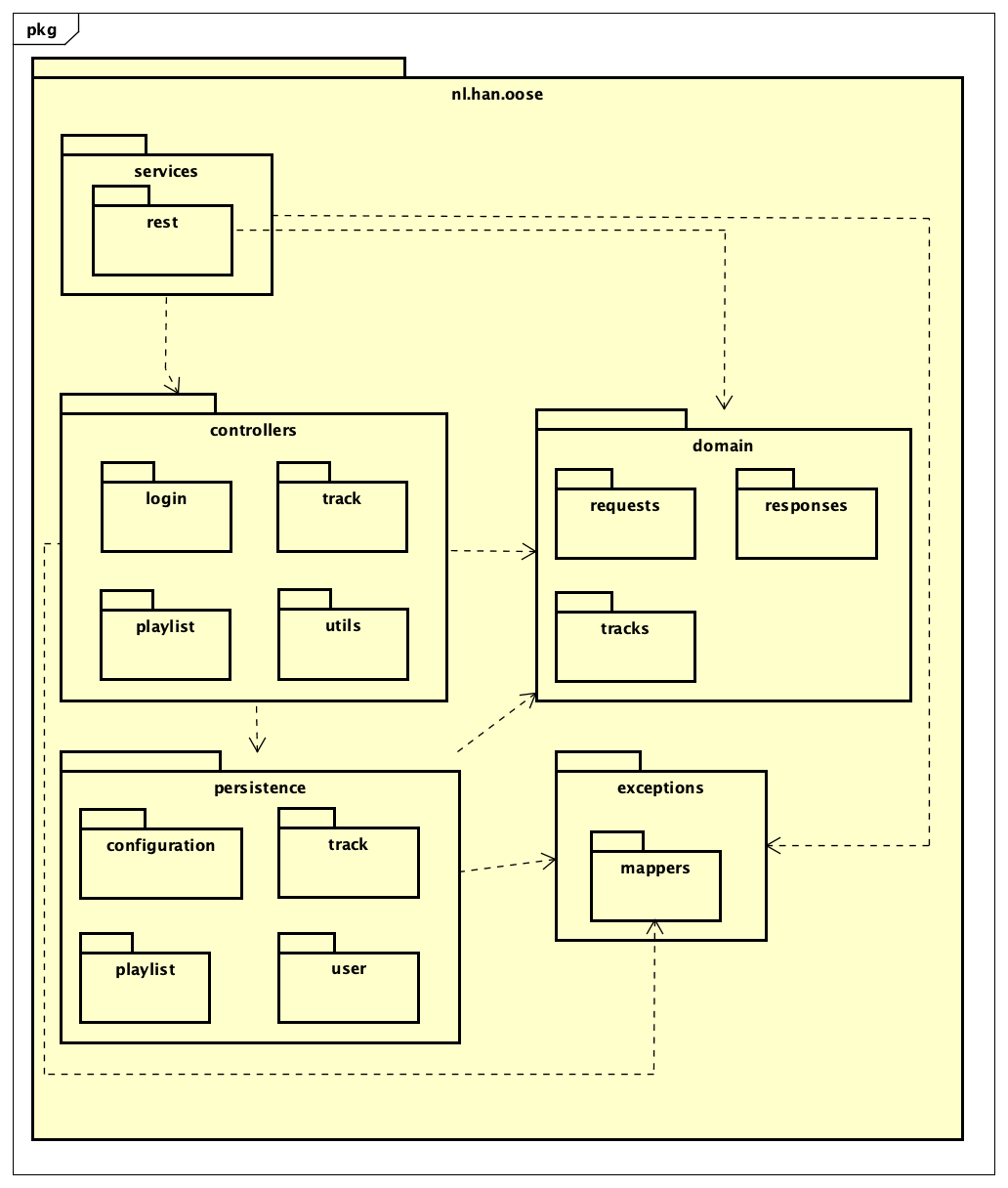
*Tabel 2.1: requirements ingedeeld met behulp van FURPS+*

# 3. Package Diagram

In het vorige hoofdstuk benoemde ik de requirements. Dit hoofdstuk gaat over het package diagram. Als eerste geef ik het package diagram weer. Daarna benoem ik welke requirements ik ermee raak. Daaropvolgend benoem ik welke design patterns ik toepaste. Ik benoem wat de alternatieve oplossingen waren en waarom ik voor de huidige oplossing koos.

## 3.1. Diagram

Figuur 1 geeft het huidige diagram weer. Ik stel vijf verschillende hoofd packages op. Deze packages bevatten verschillende sub packages. Tabel 1 legt uit wat elke package inhoudt.



Figuur 1. Huidige iteratie van het package diagram.

|  |  |
| --- | --- |
| **Package** | **Uitleg** |
| Services | De services package bevat een REST package. In de REST package zijn de services beschikbaar. |
| Controllers | In deze package zijn de controllers beschikbaar. De controllers zijn opgedeeld in vier packages: login, track, playlist en utils. De controllers voeren de business logica uit. |
| Persistence | De persistence package regelt de communicatie tussen de database en de applicatie. De package bevat vier verschillende sub packages: configuration, track, playlist en user. |
| Domain | De domain package bevat alle domeinobjecten. De package bevat drie subpackages: requests, responses en tracks. |
| Exceptions | De exception package bevat alle zelfgemaakte exceptions van de applicatie. De package bevat ook een subpackage waar de exception mappers van de applicatie in zitten. |

Tabel 1. Uitleg over packages in Spotitube.

## 3.2. Requirements

Dit deelhoofdstuk licht toe welke requirements (zie hoofdstuk 2) ik raak en oplos aan de hand van het package diagram.

Op basis van afbeelding 3.1 is te zien dat de package diagram de volgende requirements raakt:

* Impl#1: het systeem maakt gebruik van een data access layer. Ik noem dit de persistence package.
* Impl#2: het systeem maakt gebruik van een domain layer. Alle domeinobjecten zitten in de domain package.

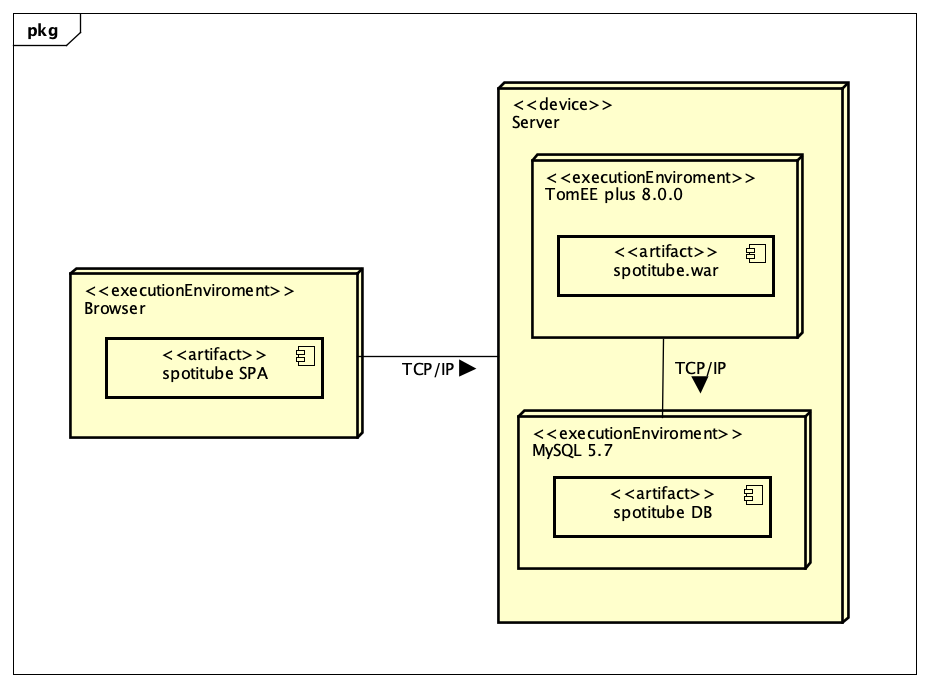
## 3.3. Design Patterns & Ontwerpkeuzes

### 3.3.1 Controllers los getrokken van services

De eerste iteratie van een applicatie maakte gebruik van een service layer. In deze laag werden de rest aanvragen afgehandeld. Hierin werd het overgrote deel van het werk gedaan. Dit is ook conform het **remote facade pattern** (Fowler, z.d.). Een rest call haalt overal de correcte data op. Echter zorgt dit ervoor dat ik de code niet kon unit testen. Het JaxRS valt niet te mocken waardoor de tests continu falen. Hierdoor besloot ik om een controllerlaag toe te voegen. In deze controllerlaag definieer ik interfaces waarin ik meerdere controllers heb. Deze controllers handelen de logica van de applicatie af. De services hoeven dit alleen te gebruiken via DI. Omdat de controllers los staan van het JaxRS framework valt het te testen. Deze keuze maakt de dependencies tussen packages overzichtelijker.

# 

# 4. Deployment Diagram

Het vorige hoofdstuk ging over het package diagram. In dit hoofdstuk geef ik het deployment diagram weer. Ik leg uit op wat voor manier de applicatie draait. Ik koppel dit aan de requirements. Als laatste benoem de ontwerpkeuzes. 

*Afbeelding 2. Deployment diagram*

Afbeelding twee geeft het deployment diagram weer. Het is een simpele deployment diagram. De Spotitube.war draait op een TomEE server. Deze back-end communiceert met de client via TCP/IP. Dit gebeurt via het REST formaat. Daarnaast bevat de server een connectie met de database. Door de JDBC api kunnen deze twee nodes met elkaar communiceren. Dit gebeurt via TCP/IP.

# 

# 5. Craftsmanship & Ontwerpkeuzes

Dit hoofdstuk legt mijn ontwerpkeuzes uit. Ik maakte dit project een aantal keuzes. Hieronder som ik de keuzes op. Daarna leg ik ze uit.

## 4.1. ORM

Ik koos ervoor om een ORM te gebruiken. Hiervoor maak ik gebruik van Hibernate. Sasidharan (2018) benoemt een aantal voor- en nadelen om een ORM te gebruiken. De voordelen wegen de nadelen af. Het faciliteert het implementeren van het **domain model pattern.** Het zorgt voor een reductie in code. Het doet aan cache management. De nadelen zijn: een langere startup tijd, een grote leercurve, moeilijk om fine te tunen door gegenereerde SQL queries. Aangezien ik ervaring heb met Hibernate en weet hoe ik dit moet implementeren, koos ik ervoor om hier gebruik van te maken.

## 4.2. HashMap voor het bijhouden van gebruiken

Ik kies ervoor om een HashMap te gebruiken voor het bijhouden van gebruikers op basis van token. Bij het inloggen wordt een token gegenereerd. De HashMap maakt gebruik van deze token als key. De HashMap slaat bij deze token de desbetreffende gebruiker op. Een HashMap sorteert geen items op basis van wanneer het object is toegevoegd. Het gebruik van een HashMap is sneller dan een List door de manier van data ophalen. Ik kies ervoor om een singleton HashMap te gebruiken i.c.m. het keyword synchronized. Dit zorgt ervoor dat de HashMap hetzelfde is voor alle afnemers.

## 4.3. Naamgeving interfaces

Ik preface interfaces niet met een *I*. C. Martin (2009) prefereert het weglaten van de I. Ik zie vaak het verkeerde voorbij komen. Meerdere developers gebruiken IObject i.p.v Object. De implementaties van de interfaces geef ik aan met *Impl*. Dus *ObjectImpl implements Object* i.p.v. *Object implements IObject*. Dit bevordert een heldere naamgeving en dus clean code.

# 

# 6. Conclusie

# Literatuurlijst

C. Martin, R. C. (2009). *Clean Code: A Handbook of Agile Software Craftsmanship*. Geraadpleegd van <https://www.investigatii.md/uploads/resurse/Clean_Code.pdf>

Fowler, M. (z.d.). *P of EAA: Remote Facade*. Geraadpleegd op 29 oktober 2019, van <https://martinfowler.com/eaaCatalog/remoteFacade.html>

Sasidharan, M. (2018, 26 mei). *Should I Or Should I Not Use ORM ?* Geraadpleegd op 29 oktober

2019, van <https://medium.com/@mithunsasidharan/should-i-or-should-i-not-use-orm-4c3742a639ce>