Spotitube
Opleverdocument

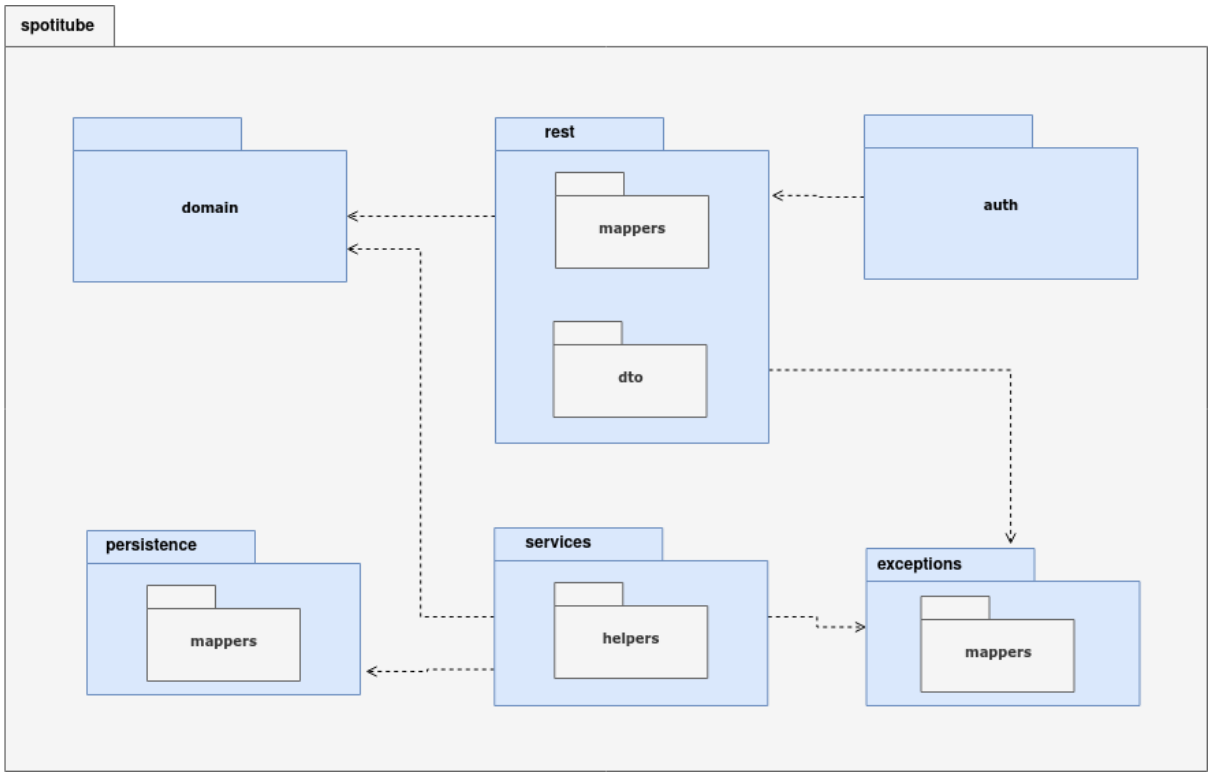
Naam	Kaddouri, Iliass, El
Klas	ITA-2AF
Course	OOSE DEA
Datum	25-03-2021
Versie	1.0

Inhoudsopgave

Inleiding

In het kader van het OOSE semester heb ik de opdracht gekregen om een Spotify/Youtube clone te ontwikkelen. Het is de bedoeling om de technieken en bevindingen, die ik heb opgedaan bij het vak DEA, te implementeren. In dit verslag lees je mijn bevindingen en conclusies. Tevens leg ik uit waarom ik bepaalde keuzes heb gemaakt.

1. Package Diagram



Figuur 1: Package diagram

Het bovenstaande UML-packagediagram presenteert de indeling van de spotitube applicatie op package-niveau. Het package diagram is ook apart te vinden in de bijlagen.

De applicatie voldoet aan alle, in de opdracht opgestelde requirements. Er is gebruikt gemaakt van JAX-RS v2.0, CDI en JDBC API die communiceert met een lokaal draaiende MySQL server. Tevens is zijn alle genoemde patterns geïmplementeerd.

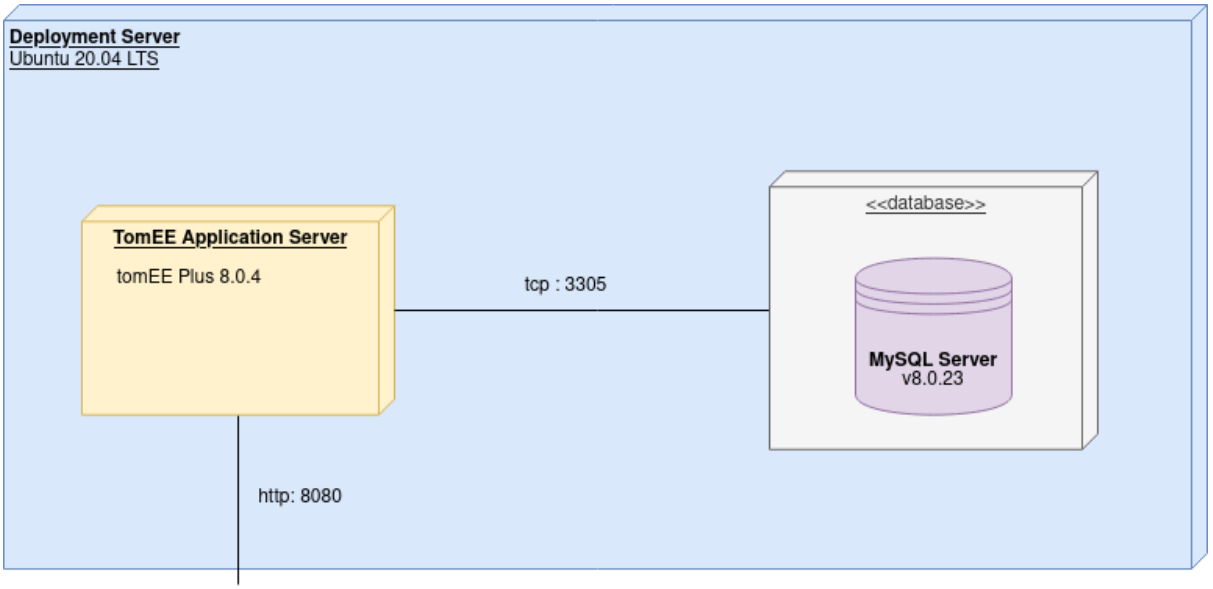
Aan de indeling van de packages is te zien dat ik alle patterns die in de opdracht genoemd zijn geimplementeerd: data access layer, domain layer, service layer en remote facade layer pattern. Tevens heb ik, indien nodig was, per layer ook mappers geimplementeerd.

De indeling van de packages was achteraf gezien niet handig. Naarmate de applicatie groeide werd het steeds lastiger om met deze indeling te werken. De oorzaak daarvan is dat de applicatie ingedeeld is op basis van abstractie layers. Als er bijvoorbeeld iets aangepast moet worden aan het onderdeel *Playlists*, moet men in 6 verschillende mappen zoeken naar de klassen die bij deze feature horen, wat niet erg handig is. Ik heb echter voor deze indeling gekozen, omdat ik niet beter wist en dacht dat het logisch zou zijn om alles per laag op te delen. De praktijk heeft mij anders laten zien.

Echter is er een alternatieve oplossing die dit probleem zou kunnen voorkomen: het opdelen van de packages op basis van *features*. Zo zijn alle klassen die te maken hebben met, bijvoorbeeld, de feature *playlists* in een package verzameld. Daarin kun je de Rest Service, Controllers (in deze applicatie ook wel services genoemd), exception, dao's en alle andere klassen vinden. Dat maakt het heel makkelijk om de applicatie aan te passen.

2 Deployment diagram

Het onderstaande diagram illustreert hoe de applicatie draait op een server. Het diagram is tevens ook in de bijlagen te vinden.



Figuur 2: Deployment diagram

Bij het deployment heb ik gekozen om alles, “bare metal, te laten draaien op mijn machine. In dit project wilde ik mij vooral verdiepen in JavaEE en wilde een weinig tijd besteden aan de infrastructuur. Doordat ik Linux draai was het erg makkelijk om alle benodigde software te installeren en te configureren. Alles werkte “out of the box” en ik er was hooguit een half uur bezig om alles op te zetten. Echter is deze opzet niet (makkelijk) schaalbaar en is er geen onderscheid tussen development- en productieserver. Er is gelukkig een alternatieve oplossing die deze problemen voorkomt, hier wordt later in dit hoofdstuk op in gegaan.

Alle requirements die opgesteld zijn in de opdracht zijn aan voldaan. Er is gebruik gemaakt van een TomEE applicatie server en een SQL server, in dit geval MySQL. Tevens is ook de configuratie van de JDBC API via CDI verlopen.

Een alternatieve oplossing voor de deployment van deze applicatie zou Docker zijn geweest. Door de applicatie in een Docker Container te deployen zou het erg gemakkelijk zijn geweest om twee verschillende servers te draaien: development en productie, dit komt omdat beiden containers exact dezelfde omgeving en configuratie hebben. De enige dependency die een server dan heeft is docker. Alle overige dependencies, java, TomEE, JDBC, MySQL, etc. hoeven niet meer op de server geïnstalleerd te worden aangezien deze al in de Docker Container staan. Dit maakt de applicatie automatisch schaalbaar: er kunnen makkelijk nieuwe containers worden aangemaakt.

Ontwerp keuzes

Helper klassen

Om de scheiding tussen packages en features te waarborgen heb ik ervoor gekozen om gebruik te maken van Helper klassen. Deze helpers bevinden zich in de package waar ze ook gebruikt worden. Een voorbeeld van een helper klasse wordt in onderstaande afbeelding weergegeven:

```java
package han.oose.dea.services.helpers;

import ...

public class PlaylistHelper {
    @Inject
    private TrackService trackService;

    public void populatePlaylistsWithTracks(List<Playlist> playlists) throws PersistenceException {
        // Populate playlists with tracks
        for (Playlist playlist : playlists) {
            playlist.setTracks(trackService.getInPlaylist(playlist.getId()));
        }
    }
}
```

Achteraf gezien waren deze klassen overbodig. Ik was in de veronderstelling dat het niet de bedoeling was om een service (controller) in een andere service te gebruiken, maar dit bleek niet zo te zijn. Helaas had ik geen tijd meer over om dit te refactoren, maar ik zal dit zeker meenemen in een volgend project.

Rest en Data Mappers

Om ervoor te zorgen dat alle data vanaf de remote facade laag correct naar de database laag wordt gestuurd heb ik ervoor gekozen om twee mappers te introduceren:

1. Restmappers: die data van Rest objecten (DTO's) naar Domein objecten omzetten en vice versa. Deze domein objecten worden gebruikt in de business logic.
2. Datamappers: die data vanuit de database, in dit geval MySQL omzetten naar domein objecten

Beiden mappers zijn geïmplementeerd door middel van interfaces:

```java
package han.oose.dea.persistence.mappers;

public interface IDataMapper <ResultSet, DO> {
    DO toDomainObject (ResultSet resultSet);
}
```

```java
package han.oose.dea.rest.mappers;

public interface IRestMapper<DTO, DO> {
    DTO toDTO(DO domainObject);
}
```

Omdat de meerderheid van de objecten niet naar twee richtingen gemapt hoeft te worden heb ik ervoor gekozen om ze uit de interface te halen. Objecten waarbij dit wel nodig was heb ik geimplementeerd:

```java
package han.oose.dea.rest.mappers;

import ...

public class UserRestMapper implements IRestMapper<UserDTO, User> {
    @Override
    public UserDTO toDTO(User user) {

        return new UserDTO(user.getUsername(), user.getPassword());
    }

    public User toDomainObject(UserDTO userDTO) {
        return new User(userDTO.user, userDTO.password);
    }
}
```

Hier volgt nog een voorbeeld van een datamapper.

```java
package han.oose.dea.persistence.mappers;

import ...

public class TokenMySQLDataMapper implements IDataMapper <ResultSet, Token> {
    @Override
    public Token toDomainObject(ResultSet resultSet) {
        try {
            return new Token(
                    resultSet.getString( s: "username"),
                    resultSet.getString( s: "token"),
                    resultSet.getDate( s: "expiry_date")
                    );
        } catch (SQLException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

Exceptions

Om fouten zo eenvoudig mogelijk af te handelen heb ik ervoor gekozen om Exceptions en daarbij behorende exception mappers te implementeren. JavaEE of TomEE? Heeft het namelijk zo geregeld dat wanneer je een Exception “throwt” deze automatisch kan worden omgezet naar een Response door middel van een mapper. Hieronder een voorbeeld van een dergelijke exception en mapper:

```java
package han.oose.dea.exceptions;

public class MissingTokenException extends Exception {
    public MissingTokenException(String message) { super(message); }
}
```

```java
package han.oose.dea.exceptions.mappers;

import ...

public class MissingTokenExceptionMapper implements ExceptionMapper<MissingTokenException> {
    @Override
    public Response toResponse(MissingTokenException e) {
        ErrorDTO errorDTO = new ErrorDTO(e.getMessage());
        return Response.status(Response.Status.BAD_REQUEST).entity(errorDTO).build();
    }
}
```

Slot

Al met al vond ik dit project erg leerzaam. Ik heb heel veel nieuwe technieken geleerd die ik ook zeker ga toepassen in de toekomst. Mijn doel is om vanaf nu Spring Boot te gebruiken in mijn vrije tijd. Tot nu toe had ik alleen ervaring met JavaScript backends (NodeJS), heel veel principes komen overeen maar er zijn ook een aantal dingen anders bij JavaEE.