Beroepsproduct DEA

Opdracht: Spotitube, RESTFUL Applicatie

Datum:	Vrijdag 28 mei, 2021
Auteur:	F.K.A. Soffers
Studentnummer:	567780
Opleiding:	HBO-ICT
Profiel:	Software Development
Onderwijsinstelling:	Hogeschool van Arnhem & Nijmegen
Begeleider:	Dhr. M. Brouwer

HAN_UNIVERSITY
OF APPLIED SCIENCES

Inhoudsopgave

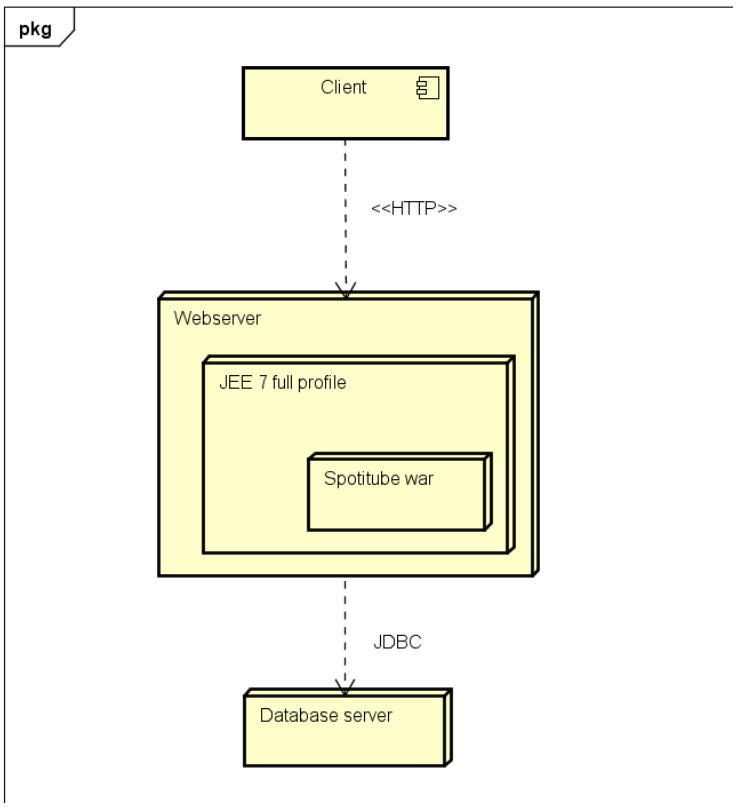
Inleiding

Het doel van dit opleverdocument is een inzicht te geven in hoe de applicatie is opgebouwd door middel van bijvoorbeeld een package- en deployment diagrammen, bij deze diagrammen is er een stuk motivatie meegeleverd over waarom ik gebruik heb gemaakt van bijvoorbeeld een Layered Architecture.

Daarnaast zal in het hoofdstuk Design Choices ook een uitleg gegeven worden over een aantal ontwerpkeuzes die ik heb gemaakt tijdens het maken van de applicatie en waarmee ik aan wil tonen dat ik de applicatie geschreven heb met het boek Clean Code van Robert C. Martin. Dit hoofdstuk omvat alle ontwerpkeuzes die in voorgaande hoofdstukken niet genoemd zijn.

Link naar de Github Repo (up-to-date):
https://github.com/diorcula/OOSE--DEA_spotitube

Deployment Diagram

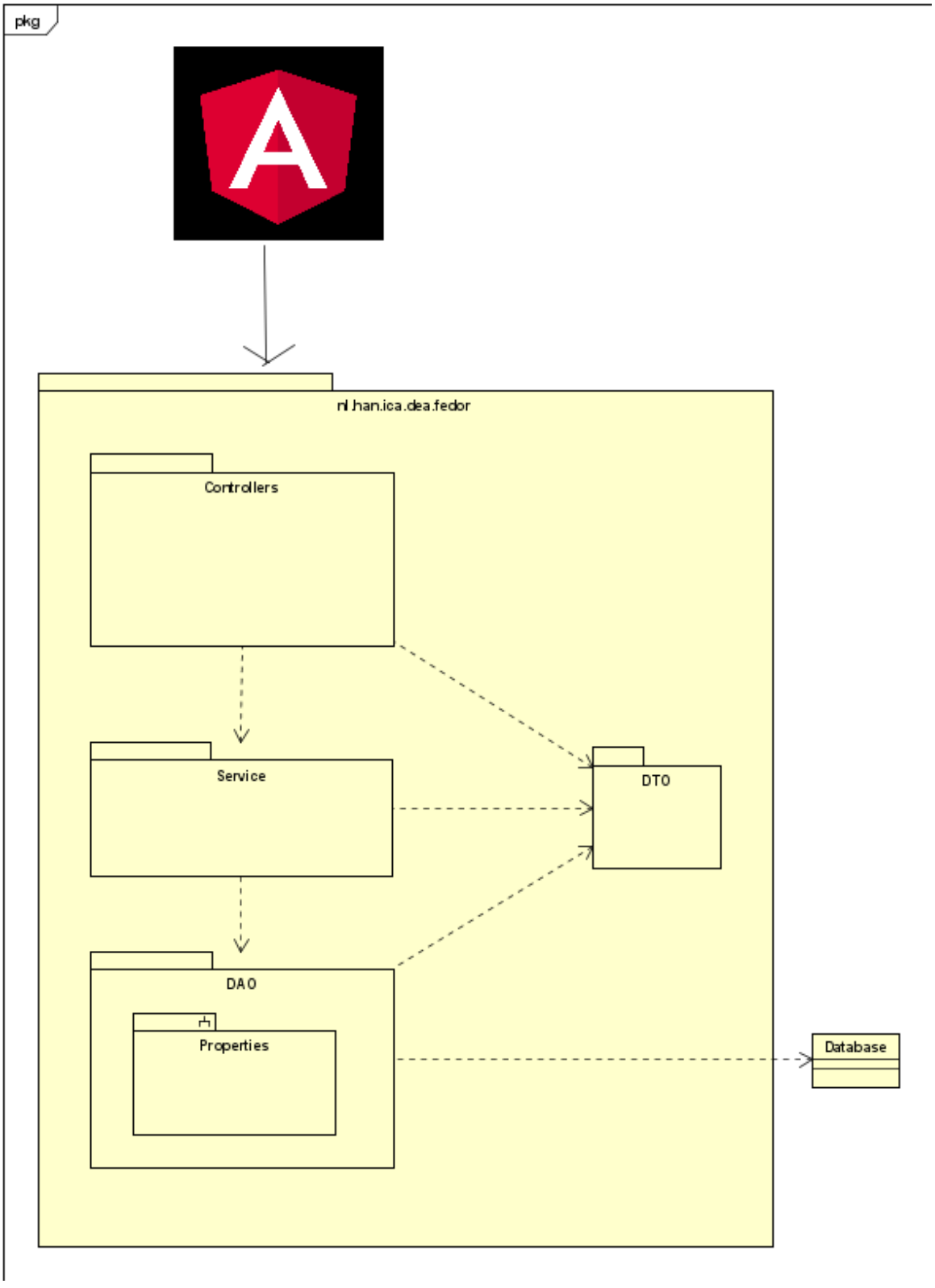


Motivatie

Enkele requirements waren dat er gebruik gemaakt moest worden van een database verbinding, zoals te zien is in bovenstaand diagram wordt er hier gebruik gemaakt van een zogeheten JDBC verbinding om de applicatie met de database te verbinden.
Verder wordt er gebruik gemaakt van HTTP om alle requests af te handelen.

Voor de database verbinding wordt er gebruik gemaakt van een Strategy Pattern, het is namelijk mogelijk om de database verbinding gemakkelijk te vervangen door een andere Database Connection. Op het moment wordt er gebruik gemaakt van SQL Server, maar wanneer bijvoorbeeld een klant een andere database provider gebruikt zoals Postgres, hoeft wanneer de database dezelfde kolommen etc. heeft alleen een andere Database Connection gemaakt te worden, de applicatie werkt nog steeds gewoon.

Package Diagram



Motivatie

Zoals in het Package Diagram hierboven te zien is, wordt er gebruik gemaakt van verschillende layers.
Zo zijn er DTO's (Data Transfer Objects) aanwezig, Controllers, Services en DAO's (Data Access Object).

Bij een Layered Architecture zijn de lagen wel met elkaar verbonden, maar niet afhankelijk van elkaar. Als ineens de DAO laag weg zou vallen valt niet de hele applicatie in duigen, de andere lagen werken nog steeds.

Het Data Mapper Pattern is hier toegepast door gebruik te maken van een DAO, dit is de layer die verbinding maakt met de database. Deze layer stuurt en vraagt data aan de database, en zo hoeven de andere lagen dat niet te doen. De DTO laag 'mapt' deze opgehaalde data daarna naar een voor de applicatie bruikbaar object.

Zo hoeft er maar in één laag van de applicatie verbinding gemaakt te worden met een database, en wanneer er bijvoorbeeld van database verandert wordt, hoeft het niet overal in de database aangepast te worden maar enkel in de DAO laag. Dit draagt weer bij aan de 'onderhoudbaarheid' van de code.

Een alternatieve, doch slechtere, oplossing zou zijn geweest om geen gebruik te maken van deze layers, dat zou dan betekenen dat alle logica en code in een enkel bestand staan. Dat heeft als gevolg dat de code minder goed te lezen is, het is nagenoeg niet te vinden waar wat gebeurt. Dan zou dezelfde laag zorgen voor het communiceren met de database als het daadwerkelijk versturen van de database object als wel andere logica. Een laag heeft dan meerdere taken, en dit kan wel werken, maar een andere developer zou er bijvoorbeeld niet meer aan kunnen werken omdat het niet uit te vogelen is.

Een serieus alternatief voor het Layer Pattern zou het Microservices Pattern zijn, bij het microservices pattern worden er eigenlijk meerdere kleine applicaties gemaakt die samenwerken. Elke microservice heeft zijn eigen specifieke verantwoordelijkheid en kunnen onafhankelijk van elkaar ontwikkeld worden. Het verschil tussen het Layer Pattern en het Microservices Pattern is dat bij het Layer Pattern elke laag een doel heeft, zo is er een database laag waarin alle verbindingen met de database afgehandeld worden maar bijvoorbeeld een aanroep om een playlist te wijzigen komt vanaf de Playlyst Controller door verschillende lagen heen uiteindelijk bij de database layer.
Bij microservices zou er gewoon een service kunnen zijn die "editPlaylistService" heet waarin dit allemaal geregeld wordt, zo is deze hele service verantwoordelijk voor de functionaliteit die bij het Layer Pattern over verschillende lagen verspreid zit.

Voordelen aan het Microservices Pattern zijn dat elke service onafhankelijk onderhouden en verandert kan worden. Het is makkelijk te schalen, zo kan er gewoon voor nieuwe functionaliteit een nieuwe service gemaakt worden.

Nadelen aan het Microservices Patten zijn dat het erg moeilijk is om een goed gestructureerde applicatie te maken wanneer er niet eerst begonnen wordt met een monolith welke vervolgens later opgebroken kan worden in kleinere services.

Het gebruik van deze layers is good practice, om zo de verschillende soorten verantwoordelijkheden te scheiden van elkaar. Daarnaast is wanneer er iets fout gaat goed te traceren waar de fout zit.

Daarnaast is er ook gebruik gemaakt van het Singleton Pattern, waarbij elke methode maar 1 verantwoordelijkheid heeft, het ‘doet/kan maar 1 ding’. Zo kan het bijvoorbeeld in de methode setDuration() enkel en alleen de duration vastleggen, het kan bijvoorbeeld deze niet terugsturen, het kan ook niet nog even de naam aanpassen van een afspeellijst.

Zo is ook duidelijk wanneer er iets fout gaat waar het precies fout gaat.
Als de duration niet goed vastgelegd wordt zal het niet in de functie editPlaylist() fout gaan, maar weten we dat het in de functie setDuration() fout gaat. Code is hierdoor duidelijker en beter te testen.

Design Choices

Enkele ontwerpkeuzes die ik gemaakt heb tijdens het schrijven van de code die voorkomen in het boek Clean Code van Robert C. Martin zijn:

- Class names hebben de voorkeur zelfstandige naamwoorden te zijn zoals Playlist, User etc. en hier geen werkwoorden te gebruiken.
- Methoden bestaan wel uit werkwoorden zoals editPlaylist, deletePlaylist.
- Er wordt geen gebruik gemaakt van ***cute*** namen, dit om te voorkomen dat code duidelijk wordt voor de lezers die dezelfde humor delen als de schrijver i.p.v. iedereen. Namen moeten duidelijk zijn in wat ze betekenen.

- Er wordt gebruik gemaakt van kleine functies en methoden die maar één 'ding' doen, methoden moeten namelijk duidelijk en niet verwarrend zijn. Anders zijn ze moeilijk te begrijpen en moeilijker te onderhouden/veranderen. Een voorbeeld hiervan is:

```
public void setDuration(List<PlaylistDTO> returnList) {
    int finalSom = calculateDuration(returnList);

    playlistsDTO.setLength(finalSom);
}

public int calculateDuration(List<PlaylistDTO> returnList) {
    int som = 0;

    for (PlaylistDTO dto : returnList) {
        som += dto.getDuration();
    }

    return som;
}
```

Hier heb ik ervoor gekozen om de Duration functies op te splitsen, het had net zo goed allemaal in één functie kunnen staan maar dan wordt het onduidelijk.

- Ook heb ik gebruik gemaakt van Dependency Injection, van het SOLID principe, omdat het volgens dit principe belangrijk is dat een class zich focust op de taken die het moet uitvoeren en niet ook nog het aanmaken van andere classes. Hiervoor wordt er gekozen voor Dependency Injection. Bijvoorbeeld hier:

```
public class UserService {
    @Inject
    UserDAO userDAO;

    public boolean isValidLogin(String userName, String password) {
        UserDTO user = userDAO.getUserDTO(userName);
```

- Waar comments geplaatst zijn heb ik er geprobeerd nuttige, waardevolle opmerking van te maken die wat meer uitleggen wat er bedoeld wordt.

- Voor de unit tests heb ik gebruik gemaakt van de @Before annotatie, om er zo voor te zorgen dat er minder duplicate code in de tests komt te staan om ze zo duidelijker leesbaar te maken.

- Ik heb geprobeerd ervoor te zorgen dat er geen uitgecommente code meer staat, om zo te zorgen dat alle code die er is een functie heeft en te zorgen dat het ‘schoon’ blijft. Als iets geen functie heeft hoort het er namelijk niet in.

- Ook heb ik ervoor gekozen om gebruik te maken van een ExceptionMapper. Het handige aan een ExceptionMapper is dat deze een specifieke HTTP response kan terugsturen. Zo maakt mijn Exceptionmapper bij de Unauthorizedexception een response met de HTPP foutmeldingscode 401 aan.

- Zo kan dit erg handig zijn om een beter inzicht te krijgen in wat er foutgaat in het systeem aan de hand van deze foutcodes.

```
@Provider
public class LoginExceptionHandler implements
ExceptionMapper<UnauthorizedLoginException> {

        @Override
        public Response toResponse(UnauthorizedLoginException e) {
        return Response.status(401).entity("Unauthorized Exception Error: " +
e.getMessage()).build();
        }
}

-----------------
public class UnauthorizedLoginException extends RuntimeException {
        public UnauthorizedLoginException(String errorMessage){
        super(errorMessage);
        }
}
------------------------
        public boolean isValidLogin(String userName, String password) {
        if (userDAO.userExists(userName)) {

        UserDTO user = userDAO.getUserDTO(userName);

        return password.equals(user.getPassword());

        } else {
        throw new UnauthorizedLoginException("invalid login");
        }
        }
```