



SPOTITUBE DOCUMENTATIE

OOSE DEA
27 maart 2020

Alex Post
531671

Docenten: Michel Portier / Meron Brouwer

Inhoud

Inleiding	2
REST-api	3
Database	4
Deployment	5
Lagenarchitectuur	6
Package diagram	6
Testen	7
Testen lagen	7
Gemaakte keuzes	9
Dependency Injection.....	9
Testen van DAO's	9
Single Responsibility Principle	9
Databaseconnectie in TomEE configuratie	9
Moeilijkheden.....	10

Inleiding

In dit document zal ik de applicatie Spotitube beschrijven. Voor de applicatie is gebruik gemaakt van de door school aangeleverde frontend, te vinden op <http://ci.icaprojecten.nl/spotitube/>.

Ik heb de backend en REST-api geïmplementeerd zodat de frontend naar behoren te gebruiken is.

Dit document zal inzicht geven in de structuur van de backend, de database en de deployment van de backend applicatie. Hierbij zal ik ook toelichten welke keuzes ik heb gemaakt om tot het behaalde resultaat te komen.

De backend van de applicatie is gemaakt met Java EE. Hier bovenop is Apache TomEE Plus gebruikt als Java Application server.

REST-api

Vanuit school is een lijst met endpoints opgesteld, deze endpoints zijn te vinden op <https://github.com/HANICA-DEA/spotitube>). De endpoints moesten geïmplementeerd worden in de backend. Voor deze REST api is gebruik gemaakt van JAX-RS, dat gebruikt wordt om RESTful webservices te maken.

Hieronder volgt een lijst met endpoints die geïmplementeerd zijn:

Spotitube REST endpoints	
url:	/login
method:	POST
url:	/playlists
method:	GET
query parameter:	token
url:	/playlists/:id
method:	DELETE
query parameter:	token
url:	/playlists
method:	POST
query parameter:	token
url:	/playlists/:id
method:	PUT
query parameter:	token
url:	/tracks
method:	GET
query parameter:	forPlaylist
query parameter:	token
url:	/playlists/:id/tracks
method:	GET
query parameter:	token
url:	/playlists/:id/tracks/:id
method:	DELETE
query parameter:	token
url:	/playlists/:id/tracks
method:	POST
query parameter:	token

Figuur 1. Geïmplementeerde endpoints

Database

Spotitube maakt gebruik van een MySQL database, waar de data voor de applicatie in is opgeslagen. De applicatie communiceert met de database door middel van de MySQL JDBC driver. De verbinding met de database is gerealiseerd in de configuratie files van Apache TomEE Plus. Dit voorkomt dat er configuratiezaken van de database in de code terecht komen.

De database bestaat uit vier tabellen: user, playlist, track en playlisttracks.

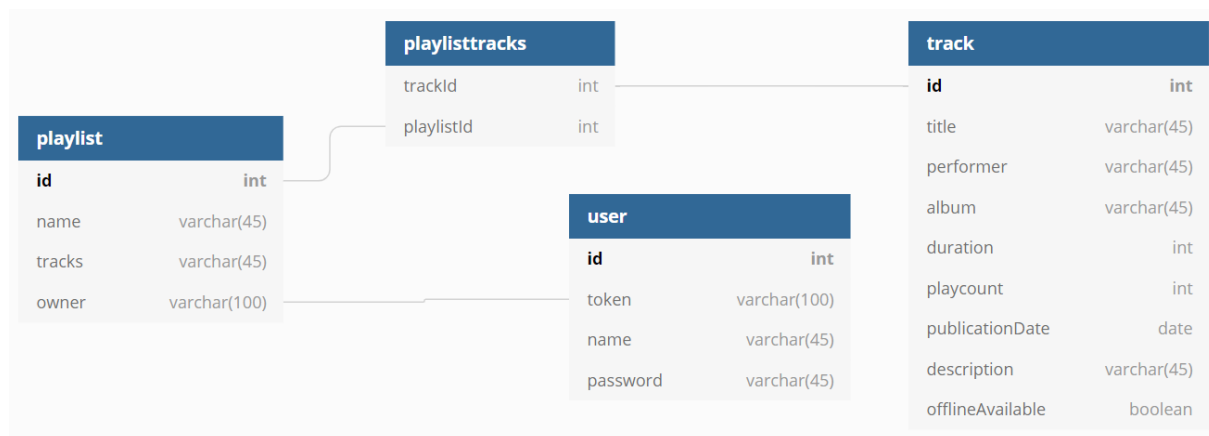
In de tabel user wordt alles opgeslagen wat met useraccounts te maken heeft, in dit geval is dat een unieke id en de token die gegenereerd wordt bij het aanmaken van een user account.

De tabel 'playlist' bestaat uit de id, naam, owner en tracks. Het veld 'owner' is gekoppeld aan het veld 'token' uit de tabel 'user', deze velden zijn aan elkaar gekoppeld door middel van een foreign key.

In 'track' is alles opgeslagen wat met een specifieke track te maken heeft.

De tabel 'playlisttracks' koppelt tracks aan playlists. Dit wordt gedaan door middel van de twee velden, die op hun beurt met foreign keys corresponderen met de id's van 'track' en 'playlist'. Op deze manier is het mogelijk om bij te houden welke tracks bij welke playlists horen.

Het databaseontwerp is te zien in Figuur 2. Hierin zijn ook alle relaties(foreign keys) tussen tabellen weergegeven.



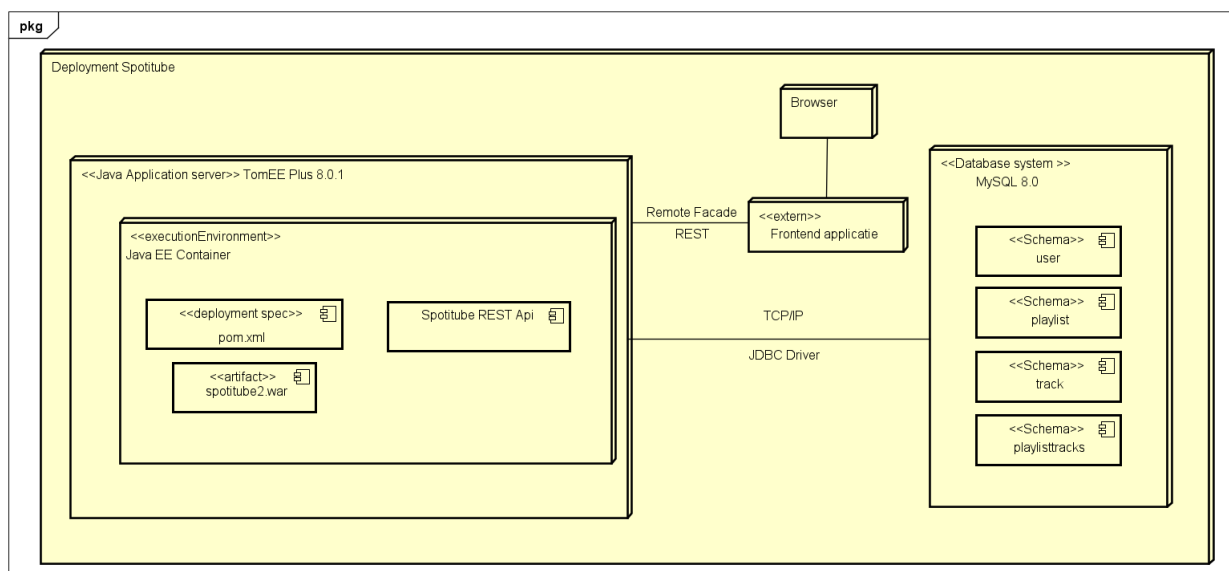
Figuur 2. Database diagram Spotitube

Deployment

De deployment van Spotitube laat zien uit welke componenten de applicatie bestaat en op welke manier deze componenten met elkaar communiceren. De drie belangrijkste componenten zijn de Java EE omgeving(TomEE Plus), de database(MySQL) en de browser waar de frontend in draait.

De manier waarop de applicatie met de browser communiceert is via REST. De REST api is geïmplementeerd met behulp van JAX-RS. Doordat de communicatie via REST verloopt, zijn voor de buitenwereld alleen maar de endpoints zichtbaar die geïmplementeerd zijn in de REST api, de rest van de code is hiermee afgeschermd en niet zichtbaar voor de buitenwereld. Dit is de Remote Facade, omdat de REST api eigenlijk een soort masker is die voor de buitenwereld zichtbaar is op een bepaalde manier, maar waar onder water nog veel meer aanwezig is. Het gebruik van de Remote Facade zorgt voor een stukje extra veiligheid.

De applicatie haalt zijn data uit een MySQL database die bestaat uit de vier schema's die te zien zijn in Figuur 3.

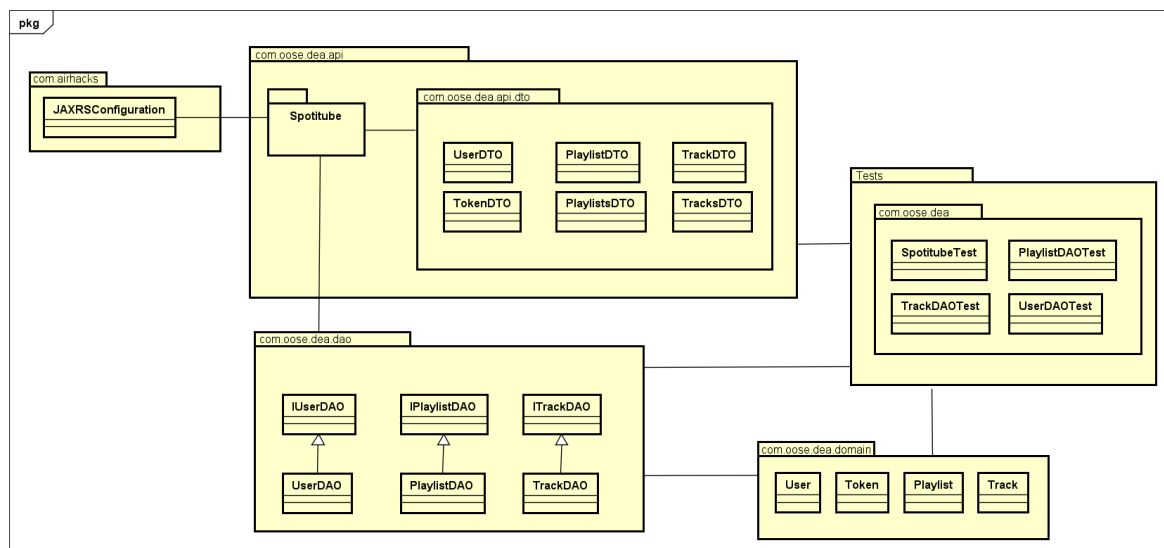


Figuur 3. Deployment diagram Spotitube

Een ander voorbeeld is de domainlaag, waarin de REST api zich bevindt. Volgens het schema in Figuur 3 mogen lagen alleen communiceren met de laag die zich direct onder de laag bevindt. In het voorbeeld van de domain- en datasourcelaat is dit het geval. De domainlaag(waar in de REST api zich bevindt) is de enige laag die communiceert met de datasourcelaat.

De package `com.airhacks` is standaard gegenereerd, hierin bevindt zich de JAX-RS configuratie.

Package diagram

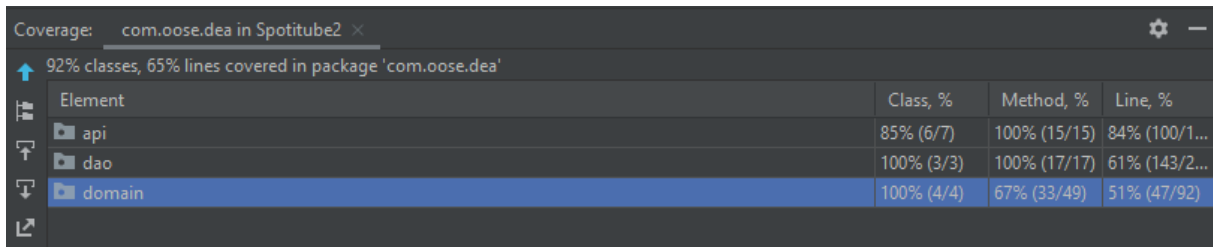


Figuur 4. Package diagram Spotitube

Testen

Om de kwaliteit van de applicatie te waarborgen en om ervoor te zorgen dat de applicatie niet stuk gaat als er aanpassingen gemaakt worden, is de applicatie getest door middel van unittests. Als testframework is gebruik gemaakt van JUnit, en binnen JUnit is gebruik gemaakt van Mockito om objecten te mocken.

Er moest een minimum van 80% code coverage zijn voor de unittests. De behaalde coverage is te zien in Figuur 5. Zie hoofdstuk Moeilijkheden voor meer uitleg over de tests.



Element	Class, %	Method, %	Line, %
api	85% (6/7)	100% (15/15)	84% (100/1...
dao	100% (3/3)	100% (17/17)	61% (143/2...
domain	100% (4/4)	67% (33/49)	51% (47/92)

Figuur 5. Test coverage

Testen lagen

Ik heb alle lagen afzonderlijk van elkaar getest. Dat was op zich niet een heel groot probleem, alleen de datasourcelaad (waar de DAO's zich in bevinden) is moeilijk te testen omdat deze altijd afhankelijk is van de database die je gebruikt. En bij unittests is het juist de bedoeling om de lagen afzonderlijk van elkaar te testen zonder dat er afhankelijkheden zijn naar andere lagen, of in dit geval externe resources.

Om deze laag toch te kunnen testen heb ik hiervoor de sql queries vergeleken. Door de verwachte query en de daadwerkelijke query met elkaar te vergelijken kan je weten of de applicatie het juiste commando naar de database stuurt.

Om dit te kunnen realiseren heb ik een aantal zaken moeten mocken, namelijk alles wat te maken heeft met het tot opzetten van de connectie met de database en het uitvoeren van de queries.

```
TrackDAO trackDAO;
DataSource dataSource;
Connection connection;
PreparedStatement preparedStatement;
ResultSet resultSet;

@BeforeEach
public void setup(){
    trackDAO = new TrackDAO();
    dataSource = mock(DataSource.class);
    connection = mock(Connection.class);
    preparedStatement = mock(PreparedStatement.class);
    resultSet = mock(ResultSet.class);
}
```

Figuur 6. Mocks om datasource laag te testen

Zoals in Figuur 6 te zien is worden er diverse mocks gemaakt om de connectie naar de database te simuleren. Door deze mocks voor de gehele testklasse aan te maken en vervolgens in de `@BeforeEach` te instantiëren hoef ik niet voor elke testmethode opnieuw de mocks aan te maken.

In Figuur 7 is een voorbeeld te zien van hoe een DAO daadwerkelijk getest wordt. Hierbij worden de eerder gemaakte mocks teruggegeven als erom gevraagd wordt, en uiteindelijk wordt de verwachte query vergeleken met de uiteindelijke query om te controleren of deze twee gelijk zijn. Op deze manier is de afhankelijkheid van de database omzeild en is het dus nog steeds een volledige unittest.

```
String expectedSQL = "select * from track where id not in (select t.id from track t inner join playlisttracks pt on t.id = pt.trackId where pt.playlistId = ?)";

when(dataSource.getConnection()).thenReturn(connection);
when(connection.prepareStatement(expectedSQL)).thenReturn(preparedStatement);
when(preparedStatement.executeQuery()).thenReturn(resultSet);
when(resultSet.next()).thenReturn(false);

trackDAO.setDataSource(dataSource);
int playlistId = 1;
ArrayList<Track> tracks = trackDAO.getTracks(playlistId);

verify(dataSource).getConnection();
verify(connection).prepareStatement(expectedSQL);
verify(preparedStatement).setInt(1, playlistId);
verify(preparedStatement).executeQuery();
```

Figuur 7. Voorbeeld DAO test

Gemaakte keuzes

Tijdens deze opdracht heb ik een aantal keuzes gemaakt, een aantal hiervan zal ik kort beschrijven.

Dependency Injection

```
@Inject
public void setPlaylistDAO(IPlaylistDAO iPlaylistDAO) {
    this.iPlaylistDAO = iPlaylistDAO;
}
```

Figuur 8. Dependency Injection

Een van de voordelen van dependency injection is dat het de koppeling tussen objecten verlaagt. Dit zorgt ervoor dat er makkelijk van implementatie te wisselen is, maar de code wordt ook beter testbaar door het gebruik van dependency injection.

In Figuur 8 is te zien hoe een setter gebruikt wordt om de injection te realiseren.

Testen van DAO's

Bij het testen van de DAO's heb ik ervoor gekozen om de verwachte en uiteindelijke queries met elkaar te vergelijken, om zo de afhankelijkheid van de database weg te nemen. Als er wel met de database getest zou worden, zou het geen unittest meer zijn, maar een integratietest.

Single Responsibility Principle

Ik heb het single responsibility principle toegepast. Dit betekent dat elke klasse of methode één specifieke functie heeft, en niet meerdere. Dit voorkomt lange methodes.

Databaseconnectie in TomEE configuratie

Ik heb ervoor gekozen om de connectie van de database te maken in de configuratie van TomEE. Dit voorkomt dat er in de code zaken als connectionstrings zitten. Een voorbeeld hiervan is te zien in Figuur 9. Door voor deze oplossing te kiezen is de databaseconnectie beschikbaar op application container level, waardoor alle componenten van TomEE er gebruik van kunnen maken. In de DAO's ziet dit er als volgt uit:

```
@Resource(name = "jdbc/spotitube")
```

```
48 <Resource
49   name="jdbc/spotitube"
50   auth="Container"
51   type="javax.sql.DataSource"
52   maxActive="100"
53   maxIdle="30"
54   maxWait="10000"
55   driverClassName="com.mysql.cj.jdbc.Driver"
56   url="jdbc:mysql://localhost:3306/spotitube?serverTimezone=UTC"
57   username="
58   password="
59 />
```

Figuur 9. Databaseconnectie

Moeilijkheden

Ik heb tijdens het programmeren een fout gemaakt die er uiteindelijk voor heeft gezorgd dat een deel van mijn code niet testbaar was.

Situatie: voeg een nieuwe playlist toe, en het REST endpoint verwacht dat er een nieuwe lijst met alle playlists gereturned wordt.

De fout die ik begaan had is dat ik in de DAO een gecombineerde methode had gemaakt, die eerst een playlist toevoegt en vervolgens alle playlists ophaalt(of de methode aanroept die dit doet), zie Figuur 10. Dit werkte perfect, maar het bleek niet testbaar. In mijn DAO tests kreeg ik het niet voor elkaar om deze gecombineerde methodes te testen. Ik weet niet of dit aan mij heeft gelegen of dat het echt niet mogelijk is.

```
210     @Override
211     public ArrayList<Playlist> addPlaylist(String name, String owner) {
212
213         try (Connection connection = dataSource.getConnection()) {
214             String sql = "insert into playlist(name, owner) values(?, ?)";
215             PreparedStatement preparedStatement = connection.prepareStatement(sql);
216             preparedStatement.setString(1, name);
217             preparedStatement.setString(2, owner);
218
219             preparedStatement.executeUpdate();
220
221         } catch (SQLException e) {
222             e.printStackTrace();
223         }
224     }
```

Figuur 10. Oorspronkelijke situatie DAO

```
@Override
public void addPlaylist(String name, String owner) {

    try (Connection connection = dataSource.getConnection()) {
        String sql = "insert into playlist(name, owner) values(?, ?)";
        PreparedStatement preparedStatement = connection.prepareStatement(sql);
        preparedStatement.setString(1, name);
        preparedStatement.setString(2, owner);

        preparedStatement.executeUpdate();

    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Figuur 11. Uiteindelijke situatie DAO

Uiteindelijk heb ik alle methodes omgebouwd naar dat ze alleen de actie uitvoeren die ze zouden moeten doen(Figuur 11), en vervolgens roep ik in Spotitube(waar de REST api wordt opgesteld) de individuele methodes aan, zoals ik Figuur 12 te zien is.

```

/**
 * Add a new playlist
 * @param owner the user token that comes with every request
 * @param playlist the playlist that is attached to the response body
 * @return a response with a list of all playlists
 */
@POST
@Path("playlists")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public Response addPlaylist(@QueryParam("token") String owner, Playlist2 playlist) {

    iPlaylistDAO.addPlaylist(playlist.name, owner);
    ArrayList<Playlist> playlists = iPlaylistDAO.getPlaylists(owner);

    if (owner == null) {
        return Response.status(403).build();
    }
    if (playlists == null) {
        return Response.status(400).build();
    }

    PlaylistsDTO playlistsDTO = new PlaylistsDTO();
    playlistsDTO.playlists = playlists;
    playlistsDTO.length = iPlaylistDAO.getTotalDuration(owner);

    return Response.status(201).entity(playlistsDTO).build();
}

```

Figuur 12. Aanroep verschillende DAO's

Helaas kwam ik er heel laat achter dat ik de methodes uit elkaar moest halen, wat overigens sowieso moet volgens het Single Responsibility Principle. Ik had nog net genoeg tijd om al mijn DAO tests te fixen, maar voor het testen van de api had ik helaas geen tijd meer. Ik zal ervoor zorgen dat dit alsnog gedaan is voor het assessment.