SPOTITUBE

Opleverdocument

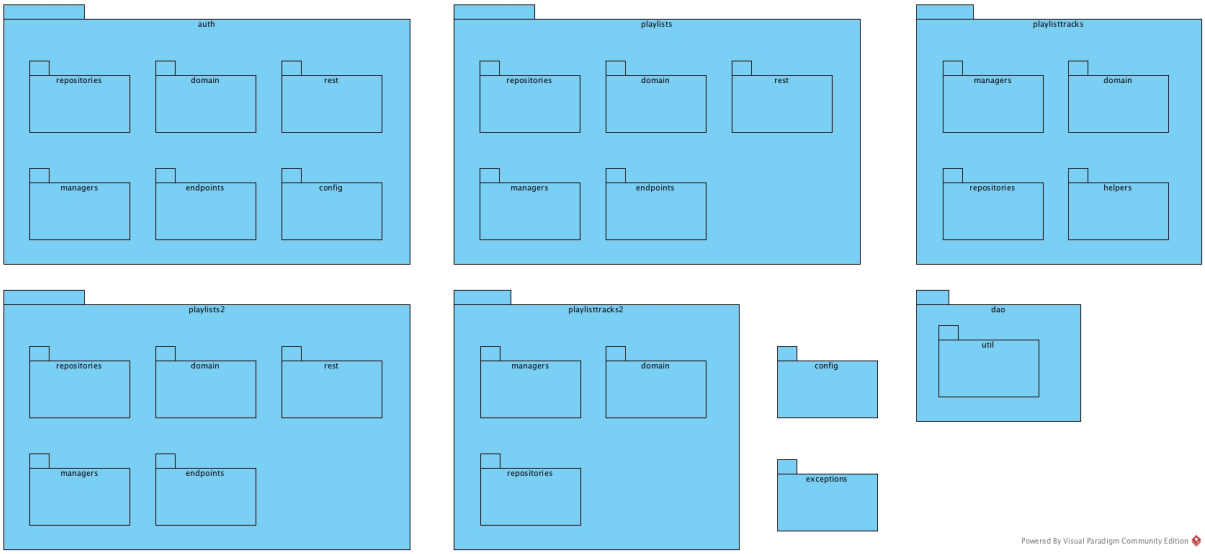
Architectuur- en Infrastructuur beslissingen

6 Nov. 17

Joël Christ
joel@joelchrist.nl

PACKAGE DIAGRAM

Het onderstaande package diagram illustreert de indeling van de applicatie op package-niveau. Dit diagram is ook los in de bijlagen te vinden.

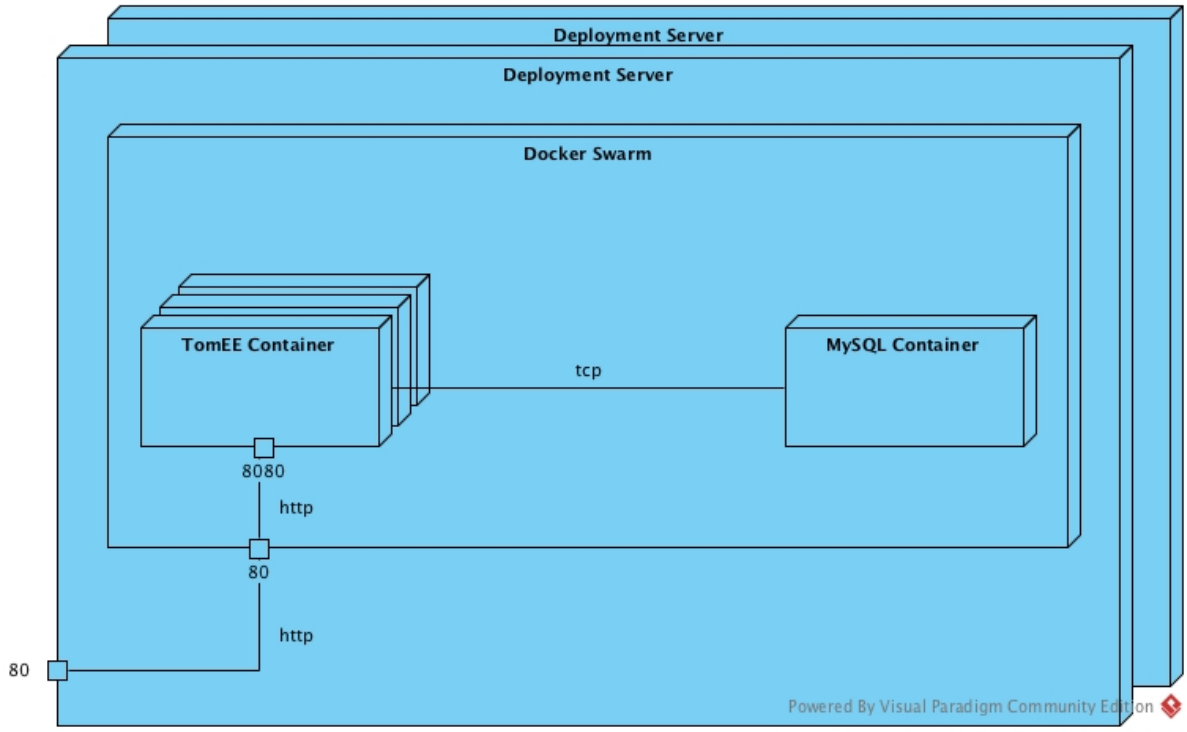


Tijdens het opzetten van de architectuur van de applicatie heb ik besloten om de packages te groeperen op functionaliteit. Dit heeft als grootste voordeel dat de bovenste laag packages heel los gekoppeld is. Als besloten wordt om het domeinobject ‘Track’ aan te passen, zullen deze wijzigingen (bijna) alleen in het ‘tracks’ package moeten worden doorgevoerd.

Een alternatief is om de packages te groeperen op serviceniveau. Alle repositories bij elkaar, alle managers bij elkaar, etc. Een nadeel van deze aanpak is dat er bij een wijziging aan het domein in bijna alle bovenliggende packages aanpassingen gemaakt moeten worden. Je creëert zo dus een hoge cohesie tussen de bovenliggende packages. Dit is niet ideaal.

DEPLOYMENT DIAGRAM

Het onderstaande deployment diagram laat zien hoe de applicatie uiteindelijk draait op een deployment-server. Dit diagram is ook in de bijlagen te vinden.



Figuur 1

Bij de deployment heb ik gekozen voor Docker, om precies te zijn: Docker Swarm. Dit zorgt ervoor dat ik op mijn laptop in precies dezelfde omgeving draai als in productie. Dit minimaliseert de kans op bugs door randzaken. Daarnaast zorgt het voor minimale dependencies, zowel in de ontwikkelomgeving als op de productieserver(s). De enige dependency is Docker.

Docker Swarm zorgt voor een makkelijke orchestratie van de infrastructuur. Zo is het op de productieserver(s) niet nodig om individuele containers op te starten en aan elkaar te linken. Dit gebeurt aan de hand van een stack- of composefile.

Als alternatief zou ik de applicatie ook op ‘bare metal’ kunnen draaien. Een nadeel is dat dit het wel een stuk lastiger maakt om de applicatie te deployen. Zo zou ik bijvoorbeeld Java, TomEE en MySQL moeten installeren, op zowel de productieserver(s) als mijn lokale omgeving. Bij het opzetten van een nieuwe server moet dit opnieuw gedaan worden, én moet er een loadbalancing oplossing worden ingezet. Met de huidige oplossing is dit niet nodig. Het enige dat gedaan moet worden om de applicatie te scalen is een nieuwe server met Docker opzetten en deze toevoegen aan de Swarm. Docker regelt de rest.

OVERIGE ONTWERPKEUZES

In dit hoofdstuk zal ik overige ontwerpkeuzes toelichten

OPTIONALS

Bij het implementeren van de repositories heb ik gekozen om gebruik te maken van java.util.Optional. Zie onderstaande afbeelding.

```
@Override
public Optional<User> getUser(String user) {
    try {
        Connection connection = getConnection();
        PreparedStatement statement = connection.prepareStatement( sql: "SELECT * FROM Users where user = ?");
        statement.setString( parameterIndex: 1, user);
        ResultSet resultSet = statement.executeQuery();
        if (resultSet.next()) {
            User returnUser = buildUserFromResultSet(resultSet);
            return Optional.of(returnUser);
        }
        return Optional.empty();
    } catch (SQLException e) {
        logger.warning(String.format("Failed to get user with name: %s from database", user));
        e.printStackTrace();
        return Optional.empty();
    }
}
```

Het gebruik van een Optional elimineert de mogelijkheid om een null-waarde terug te krijgen uit een repository. Vervolgens heb ik gekozen om de waarde van deze Optional te analyzeren in de managers. Als de Optional een waarde heeft, geeft de manager deze terug. Als de Optional geen waarde heeft, gooit de manager een EntityNotFoundException. Zie onderstaande afbeelding.

```
public User getUser(String user) throws EntityNotFoundException {
    Optional<User> userOptional = userRepository.getUser(user);
    return userOptional.orElseThrow(() -> new EntityNotFoundException(User.class));
}
```

Hiermee dwing je af dat de consumerende Class rekening houdt met niet bestaande waardes op een manier die in Java erg gebruikelijk is; met een try catch block. Zie onderstaande afbeelding.

```
try {
    User user = userManager.getUser(loginCredentials.getUser());

    if (!user.getPassword().equals(loginCredentials.getPassword())) {
        return Response.status(403).build();
    }

    AuthenticationToken authenticationToken = authenticationTokenManager.generateOrUpdateToken(user);
    RestAuthenticationToken restAuthenticationToken = restAuthenticationTokenMapper.toRest(authenticationToken);
    return Response.status(200).entity(restAuthenticationToken).build();

} catch (EntityNotFoundException e) {
    return Response.status(403).build();
}
```

Dit alles zorgt voor een erg overzichtelijke structuur in je code.

HELPER CLASSES

Om te zorgen dat de scheiding tussen packages op functionaliteit wordt gehonoreerd heb ik gekozen om gebruik te maken van Helper classes. Deze helper classes bevinden zich in het package waarom ze betrekking hebben, maar worden eventueel gebruikt in andere packages. Zie onderstaande afbeelding.

```
public class PlaylistTrackHelper {

    @Inject
    private PlaylistTrackManager playlistTrackManager;

    public Boolean isTrackInPlaylist(Track track, Integer playlistId) {
        List<PlaylistTrack> playlistTracks =  playlistTrackManager.getPlaylistTracksByPlaylistId(playlistId);
        return playlistTracks.stream().anyMatch(playlistTrack -> playlistTrack.getTrackId().equals(track.getId()));
    }
}
```

Deze Helper classes kunnen vervolgens weer gebruik maken van evt. Manager classes binnen het betreffende package. Zo houd ik de scheiding tussen packages op functionaliteit in stand.

RESTMAPPERS

Om ervoor te zorgen dat de domein objecten over de lijn naar de front-end kunnen worden gestuurd heb ik gekozen om gebruik te maken van Mappers. Deze Mappers kunnen domein objecten omzetten naar rest objecten, en visa versa. Al deze Mapper classes implementeren dezelfde interface. Zie onderstaande afbeelding.

```
public interface RestMapper<T, K> {
    T toRest(K nonRest);
    K fromRest(T rest);
}
```

Een implementatie van een Mapper kan er als volgt uit zien:

```java
public class RestPlaylistMapper implements RestMapper<nl.joelchrist.spotitube.playlists.rest.RestPlaylist, Playlist>{

    @Inject
    private RestTrackMapper restTrackMapper;

    @Override
    public nl.joelchrist.spotitube.playlists.rest.RestPlaylist toRest(Playlist nonRest) {
        return new nl.joelchrist.spotitube.playlists.rest.RestPlaylist(
                nonRest.getId(),
                nonRest.getName(),
                nonRest.getOwner(),
                nonRest.getTracks().stream().map(restTrackMapper::toRest).collect(Collectors.toList())
        );
    }

    @Override
    public Playlist fromRest(nl.joelchrist.spotitube.playlists.rest.RestPlaylist rest) {
        return new Playlist(
                rest.getId(),
                rest.getName(),
                rest.getOwner(),
                rest.getTracks().stream().map(restTrackMapper::fromRest).collect(Collectors.toList())
        );
    }

}
```

Het gebruik van Mapper classes zorgt ervoor dat het conversieproces dynamisch wordt.

SLOT

Ik wil graag afsluiten door te zeggen dat ik een tijdje uit het OOSE-domein geweest. Ik kwam er dus ook op het laatste moment achter dat ik enkele naming conventions niet helemaal gevolgd heb. Zo heten mijn DAO's bijvoorbeeld Repository, heten mijn Services Managers, en heten mijn Controllers juist Endpoints. De design principes erachter zijn natuurlijk hetzelfde, en ik ga er dan ook vanuit dat dit geen probleem is. Dit is simpelweg hoe ik het in de praktijk heb meegekregen.