Workshop 'Clean Persistency'

First8, 23-05-2017

Inhoudsopgave

Cheatsheets

Lombok

De opdrachten van de workshop maken gebruik van Project Lombok, een simpel framework waarmee je met behulp van annotaties een hoop standaard code kan genereren, zoals Setters/Getters en Constructors.
Je hoeft geen gebruik te maken van de features van Lombok, maar het kan je wel wat tijd/moeite besparen.
Voor meer informatie, zie https://projectlombok.org/features/
Om je IDE te laten snappen wat Lombok doet, moet je wel een plugin installeren. Zie hiervoor: https://projectlombok.org/download.html

Git

De workshop opdrachten staan in de volgende github:
https://github.com/First8/Workshop-JPA

Enkele handige git commando's:

- Repository clonen: *git clone https://github.com/First8/Workshop-JPA.git*
- Switchen van branch: *git checkout assignment1_registration_crud*
- Veranderingen tijdelijk wegzetten: *git stash*
- Weggezette veranderingen terughalen: *git stash pop*
- Alle lokale wijzigingen ongedaan maken: *git reset --hard*

Reset Docker Database

Om de database snel te resetten, kan je de volgende commando's gebruiken in de directory van de Workshop

- *docker-compose down*
- *docker-compose up -d*

Handige Links

- Referentie van de JPA annotaties: http://www.objectdb.com/api/java/jpa/annotations
- EntityManager JavaDoc: http://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html

Opdrachten

Met de opdrachten gaan we een simpele applicatie opzetten waarmee gebruikers concert tickets kunnen kopen.
Elke opdracht staat in zijn eigen branch en bevat de antwoorden van de vorige opdracht. Dit kan je ook gebruiken om te spieken naar de oplossingen indien je vast komt te zitten.

Bij elke opdracht is een integratietest opgezet met daarin de tests die moeten gaan werken, maar momenteel nog falen. Aan jullie de opdracht om de gevraagde functionaliteit te implementeren om deze tests werkend te krijgen.

De integratietests testen tegen een lokale database en zijn zo opgezet dat de testdata van de laatste test laten staan. Als je moeite hebt om een test werkend te krijgen, kan het helpen om met MySQL Workbench te kijken wat er precies in de database staat.

Assignment 1: Registratie gebruiker

- branch: *assignment1_registration_crud*

Voor de eerste opdracht gaan we een simpele gebruiker registeren in onze database. We hebben een Spring Boot applicatie genaamd TicketSale. Deze applicatie maakt met behulp van Flyway een tabel “account” aan dmv de scripts in ***src/resources/db/migration***.
Daarnaast bevat deze applicatie rest services in RegistrationResource om gebruikersaccounts aan te maken en op te vragen. Helaas mist deze kleine applicatie nog de koppeling met de database, dit is aan jullie om te bouwen.

In ***RegistrationRepositoryIntegrationTest*** staan een serie integratietests die de rest services aanroepen en het resultaat controleren. Probeer elk deze tests werkend te krijgen door de missende functionaliteit te implementeren.

Kijk hiervoor naar:

- Maak een correcte entity *Account* die aansluit op de “account” tabel
- Implementeer de missende functionaliteit in RegistrationRepository mbv de EntityManager

Assignment 2: Account bewerkbaar maken

- branch: *assignment2_autogenerated_ids*

In de vorige opdracht hebben we een gebruikersaccount gemaakt, waarbij het E-mailadres gebruikt werd als de Identifier. Dit lijkt functioneel logisch, maar maakt het onmogelijk voor een gebruiker om het E-mailadres van zijn account te wijzigen..

Daarom willen we liever een losse gegenereerde identifier gebruiken voor de *Account* entity.

Probeer de integratietests in ***RegistrationIntegrationTest*** werkend te krijgen en kijk hierbij naar:

- Aanpassen *Account* om een gegeneerde Id te gebruiken
- Implementeer functionaliteit in RegistrationRepository om een bestaande entity te kunnen wijzigen

Assignment 3: Toevoegen NAW gegevens

- branch: *assignment3_onetoone_relation*

We hebben nu een simpele *Account*, maar we zouden graag ook wat meer gebruikersinformatie willen vastleggen, zoals naam en adres. Dit zouden we kunnen toevoegen als velden in de *Account*, maar we houden dit liever gescheiden aangezien je niet altijd deze gegevens nodig hebt bij het ophalen van een *Account*.

We gaan een nieuw entity *AccountInfo* maken met hierin de persoonsgegevens. Vervolgens willen we deze koppelen aan de account met een One to One relatie.
Probeer de integratietests in ***RegistrationRepositoryIntegrationTest*** werkend te krijgen en kijk hierbij naar:

- Aanmaken entity *AccountInfo* (je kan zien hoe de tabel wordt aangemaakt in het migratiescript)
- Opzetten relatie tussen *Account* en *AccountInfo*
- Uitbreiden functionaliteit om NAW gegevens te kunnen opslaan voor een specifieke account

Assignment 4: Tickets!

- branch: *assignment4_manytoone_relation*

We hebben nu wat basisfunctionaliteit voor gebruikers, dus laten we eindelijk wat tickets in onze applicatie bouwen. We willen dat elke *Account* meerdere tickets kan hebben, waarin concertinformatie staat.

Probeer de integratietests in ***TicketRepositoryIntegrationTest*** werkend te krijgen en kijk hierbij naar:

- Aanmaken tabel *'Ticket'* met hierin een kolom voor id, artist, genre en location. Voeg ook een kolom Account_Id toe met een foreign key naar de *Account* tabel.
- Aanmaken entity *Ticket* (met een generated Id)
- Relatie tussen *Ticket* en *Account*
- Functionaliteit voor het aanmaken van een ticket voor een account
- Functionaliteit voor het ophalen van tickets voor een account

Assignment 5: Concerning Concerts

- branch: *assignment5_compound_key*

Laten we al die concertinformatie uit het ticket halen en dit netjes scheiden in losse *Concert* en *Location* tabellen neerzetten, zodat we deze informatie los van de tickets kunnen gebruiken.
Daarnaast gaan we de primary key van de *Ticket* tabel veranderen van een Generated Id naar een Compound Key. Deze zal bestaan uit de combinatie van het Concert Id en de Account Id, oftewel een ticket kan geïdentificeerd worden door het concert en de eigenaar van het ticket.

Voor deze opdracht moet je de tests in de integratietest ***ConcertIntegrationTest*** laten slagen. Let hierbij op de volgende punten:

- Aanmaken entity *Concert* met hierin de artiest en het genre
- Aanmaken entity *Location* met hierin de locatienaam
- Toevoegen correcte OneToMany & ManyToOne mappings in *Concert, Location* en *Ticket*
- Wijzigen van generated Id in *Ticket* naar een Compound Identifier, waarbij je de velden “concert” en “account” beide markeert als identifiers.
- Functionaliteit voor het aanmaken van een ticket aanpassen

Assignment 6: Houd de database netjes

- branch: *assignment6_cascades*

Voor nu zijn we blij genoeg met het netjes maken van de ticket- & concertdata. Laten we ook kijken of we de accountdata netjes kunnen houden. Als we een *Account* verwijderen, zouden we automatisch de *AccountInfo* willen verwijderen.

Kijk voor deze opdracht naar de tests in ***AccountInfoIntegrationTest*** en let hierbij op:

- Cascading instellen tussen *Account* en *AccountInfo*
- Als een account verwijderd wordt, ook de bijbehorende *AccountInfo* verwijderen

Assignment 7: Rapportage

- branch: *assigment7_reporting*

Tot zover hebben we direct met entities gewerkt, maar soms zal het nodig zijn om informatie uit je database te rapporteren waar je niet toevallig een entity voor hebt.

Stel dat we via een rest service het volgende rapport willen kunnen teruggeven:

    - Gegeven een genre
    - Geef voor elk ticket een rapport
    - Met hierin de naam van de artiest, de locatie van het concert en de woonplaats van de tickethouder

Om dit werkend te krijgen, kijk of je de test van ***ReportingIntegrationTest*** kunt laten slagen. Hierbij kan je letten op:

- Opzetten JPQL query voor ophalen gegevens

Assignment 8: Sales Transactions

- branch: *assignment8_transactions*

Een belangrijk onderdeel van database interacties zijn natuurlijk transacties, waarbij we garanderen dat een specifieke unit of work altijd gezamenlijk zal slagen of falen.

Hiervoor gaan we de functionaliteit voor de verkoop van een *Ticket* inbouwen. De rest service kan aangeroepen worden met een Account Id, Concert Id en Prijs. Deze zal de ticket aanmaken en vervolgens de verkoop vastleggen in een *Sale* tabel met de ticket Id en prijs. Echter, als het vastleggen van de verkoop in de Sales tabel faalt, willen we uiteraard het *Ticket* ook niet vastleggen.

Deze functionaliteit wordt getest met de integratietest ***SalesIntegrationTest***. Probeer hiervoor te letten op de volgende onderdelen:

- Aanmaken Tabel & Entity *Sale* met hierin:
    - Link naar Ticket
    - Link naar Account
    - Verkoopdatum met type Date
    - Prijs met type Integer
    - Voeg een constraint toe aan de tabel: “CONSTRAINT CHK_PRICE CHECK (Price > 0)”
- Implementeer functionaliteit om Ticket & Sale aan te maken en let hierbij op de transaction scope.

Assignment 9: Houd de code netjes

- branch: *assignment11_clean_up*

Nu we een soort van werkende applicatie hebben opgezet, is het tijd om het op te poetsen. We willen enkele de volgende verbeteringen doorvoeren, welke (deels) getest worden in de ***ValidationIntegrationTest***:

- Voeg validatie toe die controleert of het E-mailadres uit een account uniek is
- Voeg validatie toe aan de Sale entity die controleert of de Prijs groter is dan 0 vóórdat we naar de database gaan.

- Verander het Verkoopdatum veld in de Sale naar een Instant en voeg een Converter toe die deze automatisch vertaald naar een Date.
- Verplaats de gemaakte JPQL queries naar voorgedefinieerde Named Queries

Huiswerk

We hebben drie korte huiswerkopdrachten voor jullie gemaakt. Deze kan je baseren op het resultaat van de workshop welke staat in branch *assignments_final*
Deze kunnen jullie sturen naar ons (tobias.poll@first8.nl en frank.dejong@first8.nl) in het formaat dat jullie willen. Het mag een zipfile zijn, een GitHub repo, een DropBox, etc, zolang het volgende maar geldt:

- Het moet te bouwen zijn met "*mvn clean install*"
- Het moet werken als de Spring Boot applicatie handmatig wordt gestart. Je hoeft geen tests te leveren (maar dat kan het ontwikkelen wel makkelijker maken)
- Het moet natuurlijk leesbare code hebben

Huiswerk 1: Toevoegen Artist als first-class citizen

Als eerste huiswerkopdracht willen we de artiesteninformatie lostrekken van de Concert entiteit.

- Maak een entiteit *Artist* met velden naam (string) en genre (enum)
- *Artist* moet in een relatie staan met de entiteit concerten
- Denk aan het aanmaken van de tabellen in een Flyway-script voor migratie

Huiswerk 2: Auditing

Als tweede huiswerkopdracht willen we een Audit Trail bijhouden van de gemaakte Sales. Wanneer een ticket wordt verkocht, willen we de Account Id en Sale Id vastleggen in de Audit Trail tabel. Als de verkoop echter faalt, willen we *niet* dat het vastleggen van de Audit Trail wordt teruggedraaid. Deze moet altijd weggeschreven bij een verkoop poging.

- Maak een entiteit voor een *AuditTrail* op de *Sale* Tabel (OneToOne) met velden Sale Id en Account Id
- Voor elke *Sale* wordt een AuditTrail entity vastgelegd in de *AuditTrail* tabel
- Bij het terugdraaien van een *Sale* (door invalide prijs), moet de *AuditTrail* insert wel blijven staan.

Huiswerk 3: Zoekfunctionaliteit

We willen een simpele rest service toevoegen waarmee gezocht kan worden naar concerten. Hiervoor kan gebruik worden gemaakt van de Criteria API (maar hoeft niet).

- Voeg een service toe voor de zoekfunctionaliteit die Concerten teruggeeft
- Er moet op de volgende attributen gezocht kunnen worden:
    - Naam artiest
    - Genre artiest
    - Minimale datum concert
    - Locatie concert