Dokumentasjon for bruk av JPA implementasjon

For IDATT1005 applikasjon

Oppsett av database og innstillinger for Spring JPA:

Innstillingene for å sette opp forbindelsen til deres lokale MySQL database finnes i **application.properties** i java/resources mappen i src.

Databaseoppsett:

1. Bytt ut idatt1005_db med navnet på databasen din.

```
spring.datasource.url=jdbc:mysql://localhost:3306/idatt1005_db
```

2. Skriv inn brukernavn og passord (default brukernavn er root).

```
spring.datasource.username=root
spring.datasource.password=Password
```

Viktige properties:

spring.jpa.hibernate.dll-auto avgjør hva Hibernate gjør med databasen under oppstart. Sånn det er nå vil Hibernate automatisk generere tabeller basert på data.sql filen som er i java/resources og slette databasen når applikasjonen stopper. Dette er fint ettersom du ofte ikke trenger å lagre endringer du har gjort på databasen etter du er ferdig med å kjøre applikasjonen. Hvis du derimot ikke vil genrere databasen med sql skriptet kan du sette spring.sql.init.mode til never og sette dll.auto propertien til enten create-drop (Hibernate lager databasen når applikasjonen begynner og sletter den når den er ferdig, men vil ikke inneholde «dummy» data), eller update (lager databasen og oppdaterer den dersom det skjer endringer, men sletter den ikke når applikasjonen stopper).

Spring.jpa.show-sql: Hvis denne er satt til true vil Hibernate skrive ut alle sql queriene den genererer når vi kaller metoder i service klassen. Kjekt å ha når man vil se hva som egentlig skjer i databasen når vi lagrer og henter objekter).

Hvordan fungerer Spring JPA:

Entities og Annotations:

JPA eller Jakarta Persistence API er en modell som brukes for å konvertere Javaobjekter til og fra relasjonsdatabaser. Det fine med å benytte denne modellen er at JPA lar oss enkelt beskrive hvordan vi vil mappe informasjonen i en klasse til en tabell, ved å definere fieldene til klassen som kolloner i en tabell, der feltene korresponderer til kolonner i «klasse»tabellen.

Eksempel:

```java
@Entity
@Table(name = "ingredient")
public class Ingredient {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(
            name = "id",
            updatable = false
    )
    private Long id;

    4 usages
    @Column(
            name = "ingredient_name",
            nullable = false,
            columnDefinition = "TEXT",
            unique = true
    )
    private String ingredientName;
```

Måten man definerer denne mappingen er ved å bruke @Annotations som fungerer på samme måte som når man bruker **CREATE TABLE** i SQL. Objekter som kan konverteres av JPA til tabeller kallen en **Entity** og må derfor ha annoteringen over klassedefinisjonen. Her har jeg definert at **id** er primærnøkkelen til ingrediens-entitien ved å bruke **@id** og bruker **@GeneratedValue** til å automatisk gi hver ny ingrediens en unik id når de puttes i databasen. Det som også er fint er at mappingen av en klasse er helt adskildt fra selve objektet så du kan bruke objektet på akkurat samme måte som du ville gjort uten alle annoteringene, det at klassen er entity endrer ingenting om hvordan klassen fungerer.

I Recipe klassen er det også definert et Relationship mellom ingredient og recipe ettersom Recipeklassen må kunne assosieres med ingredienser:

```
4 usages
@ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
@JoinTable(name = "recipe_ingredient",
        joinColumns = @JoinColumn(name = "recipe_id"),
        inverseJoinColumns = @JoinColumn(name = "ingredient_id"))
private List<Ingredient> ingredients;
```

Det er unødvendig å lagre ingrediensene til oppskriften i selve recipe tabellen når vi har ingredienstabellen, vi bruker **@joinTable** til å lage en egen recipe_ingredient tabell som lagrer assosiasjoner mellom ingredienser og oppskrifter slik man ville gjort i SQL. Cascade betyr at når vi f.eks. sletter en ingrediens fra en oppskrift vil Hibernate automatisk ta vekk assosiasjonen til Ingredienstabellen (ikke selve ingrediensen!).

Det viktig å merke seg at JPA er en standard ikke et rammeverk, og jeg har brukt Spring og Hibernate rammeverket som faktisk implementerer måten man lagrer og henter ut data fra en database.

Service and repository:

Når man driver med ORM (Object Relational Mapping, et samlebegrep på kommunikasjon mellom objektorienterte programmeringsspåk og relasjonsdatabaser) er det vanlig å benytte noe som heter Repository design pattern. Det går ut på å definere et interface for hver entity vi vil manipulere (som vi kaller et repository, f.eks. IngredentRepository), der vi definerer hvilke operasjoner vi kan utføre på databasen. Hvis vi bare ønsker å gjøre de mest grunnleggende operasjonen (CREATE, READ UPDATE, DELETE) kan vi bare extende repositoriet vårt med spring sitt innebygde CrudRepository:

```
public interface RecipeRepository extends CrudRepository<Recipe, Long> {
```

Som lar oss lagre, hente ut, oppdatere og slette entities fra databasen uten å måtte skrive noe kode. Dersom man ønsker å gjennomføre mer avanserte SQL queries er det her man definerer disse metodene.

Men dette er jo bare et interface uten noen implementasjon, så vi må lage en annen klasse som kan bruke CRUD operasjonene som blir definert her og faktisk manipulere databasen. Vi lager derfor en såkalt Service klasse som er den klassen man kaller på når man skal samhandle med databasen. Øverst i denne klassen ser du at det over konstruktøren står **@Autowired.** Dette er en funksjon i Spring rammeverket som lar deg automatisk instansere og injekte objekter inn i Service klassen. Når du har

deklarert at konstruktører er autowired, vil Spring automatisk lage de forespurte objektene og sette dem når objektet blir kalt på:

```
7 usages
private final RecipeRepository repository;
4 usages
private final IngredientService ingredientService;

/**
 * Constructor for RecipeService
 * The recipeRepository is autowired into the class to handle the CRUD operations by Spring.
 * @param repository the repository for handling CRUD operations on Recipe objects
 * @see RecipeRepository
 */
Victor Udnæs
@Autowired
public RecipeService(RecipeRepository repository, IngredientService ingredientService) {
    this.repository = repository;
    this.ingredientService = ingredientService;
}
```

Det som kan virke merkelig er at den tilsynelatende istanserer et interface i stedet for å implementere det, men dette er også en Spring funksjonalitet der Spring injecter konstruktøren med en «implementasjon» av RecipeRepository interfacet slik at variablen **repository** har CRUD metoder som du kan kalle på slik som repository.findById(id). (Dette virker nok litt merkelig, men det påvirker ikke hvordan man bruker Service klassen i noen grad så det er ikke så nøye.)

Vi kan nå definere metoder i service klassen som kan samhandle med databasen gjennom repository variablen, slik som dette:

```
2 usages   Victor Udnæs
public Optional<Recipe> findById(Long aLong) {
    try {
        return repository.findById(aLong);
    } catch (DataAccessException e) {
        throw new ServiceException(CrudOperation.FIND_BY_ID, aLong, e);
    }
}
```

Når du da f.eks. kaller på **recipeService.findById(entity.getId());** vil Service funksjonen bare igjen kalle på repository funksjonen med det gitte parameteret, hente ut all dataen og konvertere det til et Recipe objekt som kan brukes som et vanlig objekt i applikasjons-laget.

Når man skal lagre en Recipe er det litt mer komplisert ettersom du må passe på at Ingrediensene matcher Ingredienstabellen, men dette løser vi med å bare autowire inn et IngrediensService objekt som

kan hente ut alle ingrediensene fra ingredienstabellen og matche det med ingrediensobjektene i recipe-objektet.

Et veldig viktig prinsipp i JPA er database sessions. Bare fordi vi har lagret et objekt i en database betyr ikke det at det er en automatisk link mellom objektet og den informasjonen som ligger i databasen. Vi sier at et objekt er **peristed** når Hibernate har etablert denne linken, og Hibernate vil kun opprettholde linken så lenge database sessionen er aktiv. Hver metode i RecipeService er derfor annotert med **@Transaction** som definerer at alt som skjer under kjøringen av metoden tilhører samme session. Grunnen til at dette er viktig er fordi hvis du lagrer et ingrediensobjekt i databasen, og deretter putter det samme objektet inn i en ny recipe vil ikke Hibernate skjønne at det objektet du lagret er det samme som du puttet inn i recipe objektet. Dermed når du prøver å lagre recipe objektet vil du få et DuplicationException som sier at du prøver å lagre en ingrediens som allerede eksister. Hvis du derimot oppretter en session, henter ingrediensene du ønsker at oppskriften skal ha FRA DATABASEN, og putter dem i recipe objektet vil Hibernate skjønne at de er det samme objektet og bare opprette en assosiasjon, ikke et nytt objekt.

Databasemetoder du kan bruke:

1. **Service.findAll()** – Henter ut alle objekter den finner i databasen til service objektet.
2. **Service.findByid(id)** – Henter ut objektet med matchende id, null hvis det ikke eksisterer.
3. **Service.find(entity)** – Lar deg sjekke om et objekt finnes i databasen.
4. **Service.save(entity)** – lagrer objektet i databasen (Recipe objekter kan inneholde ingredienser som ikke finnes i Ingrediensdatabasen, de blir lagt til automatisk.) PS litt usikker på hva som skjer hvis du prøver å lagre en recipe som allerede eksister tbh.
5. **Service.deleteById(id) / delete(entity)** – Sletter objekter med matchende id / objekt.

Hvordan bruke i applikasjonslaget:

Klassen som skal bruke et Service objekt må være annotert med **@SpringBootApplication** eller så klarer ikke Spring å bruke Autowired og da fungerer ingenting. (**NB!** Klassen som skal bruke en Service klasse må være i en mappe minst ET LAG HØYERE enn der Service klassene ligger i mappestrukturen, ellers klarer ikke Spring å finne klassene.)

I selve metoden må du først deklarere at spring skal kjøre applikasjonen slik:

```
ApplicationContext context = SpringApplication.run(Main.class, args);
RecipeService recipeService = context.getBean(RecipeService.class);
IngredientService ingredientService = context.getBean(IngredientService.class);
```

Der alle service-klasser du skal bruke må appendes på context variablen ellers vil de ikke kjøre. Nå er du endelig klar til å manipulere databasen og du kjører metoden ved å bruke Service variablen slik:

```
Iterable<Ingredient> allIngredients = ingredientService.findAll();
```

Her blir for eksempel alle ingrediensene hentet ut fra databasen og lagret i en Iterable collection som du kan bruke akkurat som du vil. Her er det igjen veldig viktig å være klar over at Hibernate holder ikke styr på at det du tok ut er det som er i databasen, og vil ikke lagre eventuelle endringer du gjør på objektet. Har ikke testet hvordan den gjør det med å oppdatere et objekt og det er mulig jeg må implementere en metode for å behandle dette.

Good luck motherfuckers.