Spring Data

v3.1

Spring Data - konfiguracja

Spring Data

Spring Data - to mechanizm automatycznego tworzenia implementacji repozytorium.

Upraszcza on podstawowe operacje wykonywane na bazie danych.

Dzięki niemu nie będziemy pisać powtarzalnego kodu - tak jak to miało miejsce przy definicji klas **Dao**.

- Udostępnia gotowe do użycia metody.
- Buduje automatycznie zapytania na podstawie nazwy metody.
- Umożliwia przekazanie zdefiniowanego zapytania.

Spring Data

Jest to projekt, który ma na celu ułatwienie pracy z bazami danych, udostępnia on uproszczony sposób wykonywania operacji na danych.

- Wymaga jedynie utworzenia interfejsów
- Nie wymaga dodatkowej implementacji

Spring Data jest mechanizmem automatycznego tworzenia implementacji repozytorium.

Zależności

Pierwszym krokiem do skorzystania z adnotacji przypisanych do encji jest dołączenie odpowiedniej biblioteki do naszego projektu.

Wyszukujemy w repozytorium mavena:

https://mvnrepository.com/artifact/org.springframework.data/spring-data-jpa

Drugim krokiem jest dodanie zależności:

```
<dependency>
   <groupId>org.springframework.data</groupId>
   <artifactId>spring-data-jpa</artifactId>
   <version>1.11.1.RELEASE</version>
</dependency>
```

Konfiguracja

By zacząć korzystać z możliwości Spring Data należy dodać odpowiednią konfigurację

Za pomocą konfiguracji java należy dodać adnotację

```
@EnableJpaRepositories
```

W parametrze adnotacji **basePackageClasses** określamy pakiet, w którym będą się znajdować nasze repozytoria.

Informacje o skanowanych pakietach określamy podobnie jak w przypadku adnotacji ComponentScan, np:

```
@EnableJpaRepositories(basePackages = "pl.coderslab.repository")
```

lub na podstawie klasy z pakietu, który ma być skanowany :

```
@EnableJpaRepositories(basePackageClasses = PersonRepository.class)
```

Konfiguracja - przykład

```
@Configuration
@EnableJpaRepositories(basePackageClasses = PersonRepository.class)
public class JpaConfiguration {
    //nasza konfiguracja z poprzedniego modułu
}
```

jest to konfiguracja tożsama do zapisu xml

```
<jpa:repositories base-package="pl.coderslab.repository" />
```

Pierwsze repozytorium

```
public interface PersonRepository extends JpaRepository<Person, Long> { }
```

Person - to nazwa encji, którego dotyczy nasze repozytorium

Long - to typ identyfikatora

W podstawowej wersji to wszystko czego potrzebujemy by zacząć korzystać z naszego repozytorium.

Zwróć uwagę, że nasze repozytorium jest interfejsem - jednak nie będziemy implementować metod tego interfejsu - zadba o to Spring Data.

Zasada działania

Spring Data, na podstawie parametru określonego w konfiguracji, skanuje określony pakiet wyszukując interfejsy rozszerzające interfejs Repository.

Dla każdego znalezionego interfejsu generuje jego implementację.

Implementacja jest tworzona podczas powstawania kontekstu.

Dokumentacja

Warto zapoznać się ze stroną projektu:

http://projects.spring.io/spring-data-jpa/

oraz dokumentacją:

https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

Interfejs JpaRepository

Utworzone przez nas **PersonRepository** zawiera implementację wszystkich metody zawierających się w:

- JpaRepository
- PagingAndSortingRepository
- CrudRepository

Wynika to z określonej hierarchii dziedziczenia.

Dokumentacja:

http://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html

Podstawowe metody

Standardowo utworzone repozytorium udostępnia nam metody:

- **List findAll()** - zwraca listę wszystkich obiektów danego typu
- **T findOne(ID id)** - zwraca obiekt o zadanym identyfikatorze
- **S save(S entity)** - zapisuje zadany obiekt
- **void delete(T entity)** - usuwa obiekt
- **void delete(ID id)** - usuwa obiekt o zadanym identyfikatorze
- **void deleteAll()** - usuwa wszystkie obiekty
- **long count()** - zwraca ilość dostępnych obiektów

Tworzenie zapytań

Własne metody

Mimo podstawowych metod jakie uzyskujemy dzięki **Spring Data** nie zawsze są one wystarczające, możemy więc definiować własne.

Jako przykład można podać:

- pobieranie z klasy **Person** dla określonego nazwiska.
- pobieranie z klasy **Person** z określoną płacą
- itp.

Własne metody

Aby dodać nową metodę do naszego repozytorium wystarczy zdefiniować jej sygnaturę w interfejsie repozytorium, np:

```
Person findByFirstName(String firstName);
```

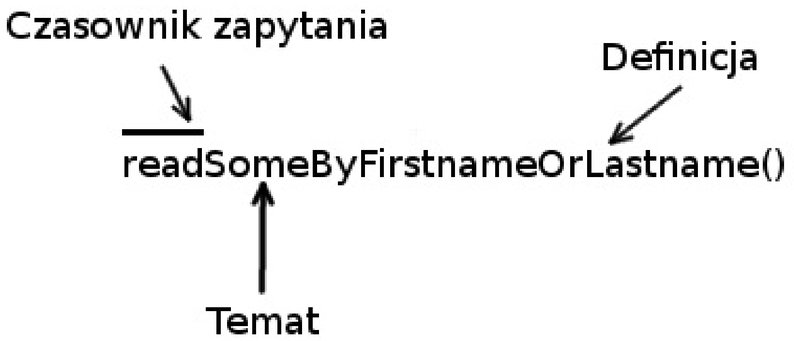
Podczas tworzenia repozytorium Spring Data przetwarza nazwy wszystkich metod i na ich podstawie tworzy odpowiednie metody.

Mogą one przyjmować również bardziej skomplikowane formy:

```
List<Person> findByLastNameIgnoreCase(String lastName);
List<Person> findByLastNameAndFirstNameAllIgnoreCase(String lastName,
                                                     String firstName);
```

Definiowanie nazwy

Sposób tworzenia nazw obrazuje poniższy rysunek:



Zasady tworzenia metod

Tworząc nazwy metod do dyspozycji następujące prefiksy:

- find…By
- read…By
- query…By
- get…By

Możemy je stosować zamiennie.

Są one synonimami.

Za znakami **By** umieszczamy nazwę atrybutu naszej encji:

```
Person findFirstByFirstName(String firstName);
Person getFirstByFirstName(String firstName);
Person queryFirstByFirstName(String firstName);
```

Po tym atrybucie będą wyszukiwane nasze encje.

Limitowanie wyników

Aby określić ilość otrzymanych wyników zapytania możemy dodać odpowiednie znaczniki do naszego zapytania, np.:

- **first** - zwróci pierwszy dopasowany obiekt
- **top** - synonim first
- **firstX** - zwróci pierwsze X elementów

Np:

```
Person findFirstByOrderByLastnameAsc();
Person findTopByOrderByAgeDesc();
List<Person> findFirst10ByLastname(String lastname);
```

Do pobrania wszystkich rekordów wykorzystujemy poznaną wcześniej metodę **`findAll()`**.

Zliczanie elementów

Aby utworzyć metodę służącą do pobrania ilości obiektów dopasowanych do naszego zapytania stosujemy prefiks

- count…By

Przykład :

```
long countByFirstName(String firstName);
```

Temat zapytania

Istnieje możliwość dodania dodatkowego elementu znajdującego się po prefiksie, którego Spring Data nie interpretuje, ale dzięki temu możemy tworzyć bardziej adekwatne nazwy.

Poniższe będą tożsame

```
long countByFirstName(String firstName);
long countPeopleByFirstName(String firstName);
long countSomePeopleByFirstName(String firstName);
long countSomeByFirstName(String firstName);
```

Temat zapytania

Dodatkowe ciągi znaków w przykładach to:

- People
- SomePeople
- Some

Nazwa umieszczona po prefiksie nie ma znaczenia, określenie jakiej encji dotyczy zapytanie odbywa się przez definicję repozytorium:

```
public interface PersonRepository extends JpaRepository<Person, Long> { }
```

Operatory

Tworząc bardziej skomplikowane przykłady możemy korzystać z operatorów logicznych

- And
- Or

Lub wyrażeń

- Between
- LessThan
- GreaterThan
- Like

Ilość pobranych elementów

Tworząc sygnatury metod określamy czy zwracane wartości mają być pojedynczym obiektem np:

```
Person findPersonByAgeGreaterThan(int age);
```

W przypadku gdy nasze zapytanie zwraca więcej niż jeden wynik otrzymamy wyjątek

`javax.persistence. NonUniqueResultException` - aby się przed nim zabezpieczyć możemy określić limit zwracanych wierszy.

W celu otrzymania listy obiektów definicja powinna wyglądać następująco:

```
List<Person> findByAgeGreaterThan(int age);
```

Przykłady

Wyszukiwanie pierwszej osoby dla zadanego imienia lub nazwiska:

```
Person findFirstByFirstNameOrLastName(String firstName, String lastName);
```

Wyszukiwanie pierwszej osoby dla zadanego imienia oraz nazwiska:

```
Person findFirstByFirstNameAndLastName(String firstName, String lastName);
```

Lista osób w wieku powyżej zadanego:

```
List<Person> findByAgeGreaterThan(int age);
```

Przykład wykorzystania

Przykład wykorzystania metody w kontrolerze:

```
@Controller
public class RepoTestController {
    @Autowired
    PersonRepository personRepository;
    @RequestMapping("/PeopleByAge")
    @ResponseBody
    public String getPeopleByAge() {
        List<Person> list = personRepository.findByAgeGreaterThan(12);
        for (Person person : list) {
            System.out.println(person.getFirstName() + " " + person.getAge());
        }
        return "result";
    }
}
```

Wspierane słowa kluczowe

Prezentowane przykłady to tylko wycinek możliwości, pełna lista:

https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation

Wykonaj zadania z działu

Tworzenie zapytań

Zapytania szczegółowe

Własne zapytania

Mimo ogromnych możliwości tworzenia zapytań za pomocą nazw metod, nie zawsze okazują się wystarczające lub nazwa metody jest bardzo długa.

W takich przypadkach możemy skorzystać z adnotacji **@Query**, w którym umieścimy odpowiednie zapytanie.

```
@Query("select u from User u where u.emailAddress = ?1")
User findByEmailAddress(String email);
```

Parametr metody **`findByEmailAddress`** o nazwie **`email`** zostanie wykorzystany jako parametr zapytania określony za pomocą oznaczenia **?1**.

Korzystamy tutaj z języka zapytań **JPQL** - omawianego już podczas zajęć.

Pamiętajmy że **JPQL** domyślnie operuje na nazwach atrybutów klasy, a nie na nazwach kolumn z bazy danych.

Własne zapytania - przykład

```
public interface UserRepository extends JpaRepository<User, Long> {
    @Query("select u from User u where u.emailAddress = ?1")
    User findByEmailAddress(String emailAddress);
    @Query("select u from User u where u.firstname like %?1")
    List<User> findByFirstnameEndsWith(String firstname);
    @Query(value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1",
            nativeQuery = true)
    User findByEmailAddress(String emailAddress);
}
```

Parametr **nativeQuery** - informuje, że tworzymy zapytanie w natywnym SQL.

Nazwane parametry

Tworząc zapytania zamiast polegać na parametrach ustawionych w odpowiedniej kolejności możemy wykorzystać możliwość wstawiania nazwanych parametrów

Tworzymy je za pomocą znaku dwukropka oraz nazwy przekazywanego parametru **:paramName**

Parametry w metodzie musimy dodatkowo opisać za pomocą adnotacji **@Param("paramName")**.

```
public interface UserRepository extends JpaRepository<User, Long> {
  @Query("select u from User u where u.firstname = :firstname or
         u.lastname = :lastname")
  User findByLastnameOrFirstname(@Param("lastname") String lastname,
                        @Param("firstname") String firstname);
}
```

Własne funkcjonalności

Aby skorzystać z możliwości własnych implementacji korzystających z **EntityManagera**, musimy skorzystać z możliwości wielodziedziczenia interfejsów.

Więcej na temat dziedziczenia:
https://docs.oracle.com/javase/tutorial/java/IandI/multipleinheritance.html

Podczas generowania implementacji interfejsów dla repozytoriów, **Spring Data** wyszukuje klasy o takich samych nazwach jak nazwy interfejsów, ale zakończone sufiksem **Impl**.

Własne funkcjonalności

Taka funkcjonalność może być przydatna, gdy w projekcie mamy już zapytania korzystające z EntityManagera.

Dzięki temu możemy wzbogacić istniejącą aplikację o możliwość korzystania ze **Spring Data** bez konieczności zmiany już istniejącego kodu.

Tworzenie interfejsu

Aby skorzystać z mechanizmu włączania własnej implementacji do generowanej przez Spring Data, należy utworzyć własny interfejs.

```
package pl.coderslab.dao;
import pl.coderslab.model.Person;
public interface PersonRepoCustom {
    public Person myCustomFindById(Long id);
    public void changeLastName(String originalLastName, String newLastName);
}
```

Implementacja interfejsu

Następnie jego implementację, której nazwa musi się pokrywać z nazwą repozytorium z dodatkowym suffiksem **Impl** np.: PersonRepositoryImpl.

```
@Repository
public class PersonRepositoryImpl
implements PersonRepoCustom { //.... }
```

Na kolejnych slajdach znajduje się przykładowa implementacja interfejsu.

Implementacja interfejsu

```
package pl.coderslab.dao;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import javax.transaction.Transactional;
import org.springframework.stereotype.Repository;
import pl.coderslab.model.Person;
@Repository
@Transactional
public class PersonRepositoryImpl implements PersonRepoCustom {
    @PersistenceContext
    private EntityManager entityManager;
}
```

Implementacja interfejsu

```
package pl.coderslab.dao;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import javax.transaction.Transactional;
import org.springframework.stereotype.Repository;
import pl.coderslab.model.Person;
@Repository
@Transactional
public class PersonRepositoryImpl implements PersonRepoCustom {
    @PersistenceContext
    private EntityManager entityManager;
}
```

Wstrzykujemy obiekt managera encji.

Implementacja interfejsu

Uzupełniamy implementację metod:

```
@Override
public Person myCustomFindById(Long id) {
    return entityManager.find(Person.class, id);

}
@Override
public void changeLastName(String originalLastName, String newLastName) {
    Query q = entityManager
            .createQuery("Update Person p set p.lastName = :newLastName" +
                    " where p.lastName= :originalLastName");
    q.setParameter("newLastName", newLastName)
        .setParameter("originalLastName", originalLastName)
        .executeUpdate();
}
```

Modyfikacja repozytorium

Ostatnim krokiem jest modyfikacja repozytorium tab by rozszerzało nasz dodatkowy interfejs.

```
public interface PersonRepository extends JpaRepository<Person, Long>,
                                          PersonRepoCustom { //...}
```

Wykonaj zadania z działu

Zapytania szczegółowe