Warstwa dostępu do danych

Z wykorzystaniem Spring oraz Spring Data

Czym jest Spring Data?

Spring Data stanowi moduł oparty na frameworku Spring, którego celem jest stworzenie przejrzystego i spójnego modelu dostępu do danych.

Spring Data pozwala między innymi na:

- ułatwienie dostępu do relacyjnych i nierelacyjnych baz danych
- wydzielenie poziomu abstrakcji dla mapowania relacyjno - obiektowego (ORM)
- tworzenie dynamicznych zapytań do składowanych zasobów
- integrację i ujednolicenie dostępu do danych z różnych typów źródeł
- wyeliminowanie powtarzalnego kodu w czasie tworzenia kolejnych repozytoriów

Najważniejsze moduły Spring Data

W pakiecie Spring Data kluczową rolę odgrywają moduły:

- Spring Data Commons - współdzielona infrastruktura dla innych modułów Spring Data
- Spring Data JPA - wsparcie dla JPA
- Spring Data MongoDB - wsparcie dla dokumentów i repozytoriów MongoDB.
- Spring Data Redis - konfiguracja i wsparcie dla bazy Redis
- Spring Data REST - budowa aplikacji opartych na stylu architektury REST

Architektura Spring Data

Spring Data poszerza koncepcję `Repository<T, ID>` czyli uogólnioną warstwę dostępu do magazynów danych i encji.

`Repository<T, ID>` jest interfejsem, gdzie `T` jest typem przechowywanym, a `ID` jest typem klucza głównego. Ten interfejs jest rozszerza inne interfejsy, takie jak:

- `CrudRepository<T, ID>`
- `PagingAndSortingRepository<T, ID>`

Architektura Spring Data

Implementujemy encje, a następnie tworzymy interfejs lub zbiory interfejsów, aby opisać sposób manipulowania tą encją między systemem, a bazą danych.

- Interfejs `Repository<T, ID>` sam w sobie jest tylko znacznikiem, nie ma metod.

Tworzenie repozytorium obejmuje trzy pliki:

- konfiguracja Springa
- entity type (obiekt reprezentujący dane, zawierający primary key)
- interfejs implementujący `Repository<T, ID>` (w praktyce implementuje się `CrudRepository<T, ID>` lub `PagingAndSortingRepository<T, ID>`).

Istnieje wiele odmian interfejsów, które umożliwiają różne poziomy dostępu do metod.

Metody udostępniane przez interfejs CrudRepository

- `<S extends T> S save(S entity)`
- `<S extends T> Iterable<S> saveAll(Iterable<S> entities)`
- `Optional<T> findById(ID id)`
- `Iterable<T> findAll()`
- `long count()`
- `void delete(T entity)`
- `void deleteAll(Iterable<? extends T> entities)`
- `boolean existsById(ID id)`

Autoimplementacja metod

Spring Data wykorzystuje dynamiczne proxy do tworzenia implementacji na podstawie wprowadzonej nazwy metody.

Wspierane są takie słowa - klucze jak:

- find - wyszukiwanie encji spełniających konkretne kryteria
- get - wyszukiwanie encji spełniających konkretne kryteria
- count - zliczanie encji spełniających konkretne kryteria
- modyfikatory:
  - Top, Bottom
  - First
  - Distinct
  - OrderBy
  - Between
  - Containing
- wyrażenia logiczne - AND, OR

```
3 usages  new *
@Repository
public interface AuthorRepositoryMysql extends CrudRepository<Author,Integer> {
    2 usages  new *
    Author findByLibraryKey(String libraryKey);
    1 usage  new *
    Author getAuthorByBio(String bio);
    1 usage  new *
    long countAuthorByNameAndBooks(String name, Set<Book> books);
    1 usage  new *
    Author findDistinctFirstByBioContaining(String text);
}
```

```
3 usages  new *
@Repository
public interface AuthorRepositoryMysql extends CrudRepository<Author,Integer> {
    findDistinctFirstBy
    findBy
    findDistinctTopBy
    findAuthorBy
    findAuthorsBy
    findAllBy
    findDistinctBy
    findFirstBy
    findTopBy
    Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards  Next Tip
    Author find
}
```

Definiowanie własnych zapytań do bazy danych

Spring Data zapewnia wiele sposób ma definiowanie zapytań do bazy danych. Jeden z nich stanowi wykorzystanie adnotacji `@Query`.

Aby pokazać, że metoda ma zwracać rezultat danego zapytania wystarczy:

- oznaczyć ją adnotacją `@Query` gdzie jako parametr value podać pożądane zapytanie
- ustawić parametr `nativeQuery = true`
- wykorzystać adnotację `@Param` do określenia wymaganych parametrów zapytania
- pamiętać o możliwości wykorzystania klauzul takich jak - `WHERE, ORDER BY`

```
@Query(value = "SELECT * FROM book e ORDER BY e.library_key LIMIT :limit" , nativeQuery = true)
Iterable<Book> findBooksByLimit(@Param("limit") int limit);
```

JPA - Java Persistence API

Java Persistence API to ORM przeznaczony do języka programowania Java.

ORM (Object-Relational Mapping) - służy do odzwierciedlania architektury obiektowej systemu na relacyjną bazę danych.

Przykładowe implementacje JPA

- Hibernate
- OpenJpa
- EclipseLink
- DataNucleus

Jak rozpocząć pracę z JPA?

Prace z JPA rozpoczynamy od modyfikacji pliku budowania zależności projektu – czyli build.gradle lub pom.xml

Konfiguracja gradle.build

Do poprawnego działania JPA wymagane będzie dodanie kilku zależności:

Spring Boot JPA – obsługa JPA

```
implementation 'org.springframework.boot:spring-boot-starter-data-jpa:3.0.0'
```

PostgreSQL – sterownik do obsługi bazy danych

```
implementation 'org.postgresql:postgresql:42.5.1'
```

Lombok – dodatkowa biblioteka, dzięki której unikniemy boilerplate code

```
compileOnly 'org.projectlombok:lombok:1.18.20'
annotationProcessor 'org.projectlombok:lombok:1.18.20'
```

Budowanie projektu - model

Klasa, która posiada pełne odzwierciedlenie po stronie bazy. To znaczy wszystkie jej pola znajdą się w bazie danych w tabeli o nazwie i kolumnach analogicznych jak obiekt tej klasy.

```java
@Data
@Entity(name = "articles")
@NoArgsConstructor
@RequiredArgsConstructor
public class Article implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long idArticle;

    @Column(nullable = false)
```

Nazwa	Zależność	Opis
@Entity(name = "articles")	JPA	Adnotacja z JPA wskazująca, że dana klasa jest modelem i posiada jej implementacje w bazie danych.
@Data	Lombok	Generuje gettery, settery, konstruktor tip
@NoArgsConstructor	Lombok	Generuje konstruktor bez parametrów
@Id	JPA	Oznacza kolumnę będąca Id modelu.
@GeneratedValue	JPA	Ustawienie strategii, np. auto inkrementacja.
@Column(nullable = false)	JPA	Adnotacja umożliwiająca dodanie określonej charakterystyki dla danej kolumny
@ManyToOne	JPA	Określenie kolumny z relacją Wiele do jeden
@OneToMany	JPA	Określenie kolumny z relacją Jeden do wiele.
@JoinColumn(name="id_article_type")	JPA	Określenie nazwy kolumny do złączeń

Budowanie projektu - repozytorium

Interfejs służący do późniejszej komunikacji z bazą danych.

```java
@Repository
public interface ArticleRepository extends JpaRepository<Article, Long> {

    @Query(value = """
            SELECT * FROM articles JOIN article_type USING(id_article_type)
            JOIN users USING(id_user) JOIN users_data USING(id_user)
            WHERE permission=:level AND verificated=true ORDER BY date, time""",
            nativeQuery = true)
    List<Article> findByUserPermissionLevel(@Param("level") int level);

}
```

Najważniejszą adnotacją jest **@Repository**, oznaczająca interfejs jako repozytorium.

Domyślnie JPA umożliwia budowanie automatycznych zapytań za pomocą nazwy funkcji, natomiast jest możliwość zdefiniowania własnego zapytania za pomocą @Query.

Budowanie projektu - serwis

Klasa, która odpowiada za logikę biznesową aplikacji. W jej implementacji odwołujemy się do beanów repozytoriów.

```java
@Service
public class ArticleService {

    private final ArticleRepository articleRepository;
    private final UserRepository userRepository;
    private final TagRepository tagRepository;

    @Autowired
    public ArticleService(ArticleRepository articleRepository,
                          UserRepository userRepository,
                          TagRepository tagRepository) {
        this.articleRepository = articleRepository;
        this.userRepository = userRepository;
        this.tagRepository = tagRepository;
    }
```

Adnotacja @Service oznacza klasę jako serwis. Dodajemy pola o typach utworzonych repozytoriów, następnie przekazujemy je wszystkie w konstruktorze.

Nad konstruktorem dodajemy adnotację @Autowired, z pomocą której wstrzykujemy zależności repozytoriów.

Połączenie z bazą danych

Podpięcie bazy danych następuje w pliku application.properties

spring.datasource.url=jdbc:{sterownik_bazy}://${address}:{port}/{baza} – nakierowanie na bazę danych

spring.datasource.username = user – login do operatora bazy danych

spring.datasource.password = password – hasło do operatora bazy danych

spring.jpa.hibernate.ddl-auto = update – tryb w jakim ma się uruchamiać JPA, update – aktualizacja bazy

Testy

Do testów dostarczono adnotację @DataJpaTest.

Dostarcza ona środowisko z bazą danych, dzięki której można przetestować w przyjazny sposób działanie zapytań.

```
@DataJpaTest
class UserEntityRepositoryTest {

    @Autowired private DataSource dataSource;
    @Autowired private JdbcTemplate jdbcTemplate;
    @Autowired private EntityManager entityManager;
    @Autowired private UserRepository userRepository;

    @Test
    void injectedComponentsAreNotNull(){
        assertThat(dataSource).isNotNull();
        assertThat(jdbcTemplate).isNotNull();
        assertThat(entityManager).isNotNull();
        assertThat(userRepository).isNotNull();
    }
}
```

MongoDB

- Nierelacyjna baza przechowująca dane w postaci dokumentów z pomocą JSON-a.
- Należy do grupy języków NoSQL.
- Bardzo wysoka skalowalność oraz elastyczność.
- Obsługa JPA

Połączenie z MongoDB

W celu połączenia naszej aplikacji z bazą danych MongoDB musimy dodać nowe zależności do konfiguracji projektu:

Spring Boot Starter Data MongoDB:

- compile("org.springframework.boot:spring-boot-starter-data-mongodb")

Flapdoodle embed mongo:

- testCompile 'de.flapdoodle.embed:de.flapdoodle.embed.mongo:2.2.0'

Przykład klasy modelu w MongoDB

Spring Data do komunikacji z MongoDB jest wykorzystywane podobnie jak dzieje się to w przypadku relacyjnej bazy danych, drobne różnice pojawiają się w klasie modelowej.

```
@Document
@Data
@NoArgsConstructor
@RequiredArgsConstructor
public class Artist implements BaseArtist<String> {
 @Id
 String id;
 @NonNull
 String name;
}
```

Przykład klasy Artist

- Zamiast z adnotacji @Entity korzystamy z @Document.
- Klucz id jest stosowany jako typ String.

Podsumowanie

Do tej pory można zauważyć, jak Spring Data ułatwia dostęp do przechowywania danych. Pozwalając na dostęp zarówno do relacyjnej bazy danych, jak i do bazy MongoDB za pomocą dwóch zestawów klas, które są bardziej podobne niż odmienne - nawet jeśli bazowe modele danych są niezwykle różne. Ten sam rodzaj korzyści można zaobserwować z dostępu do prawie każdej bazy danych, która ma wsparcie w Spring Data.

Warto zauważyć, że interfejsy, można w zasadzie dość łatwo podłączyć do front-end'u. Zachowana została transakcyjność, a w efekcie prostota i elastyczność; dzięki JPA można otrzymać wsparcie dla większości (jeśli nie wszystkich) relacyjnych baz danych, a zmiana kodu dla MongoDB jest trywialna. Można migrować do innych baz danych, takich jak Cassandra czy Neo4J, przy równie niewielkim wysiłku.