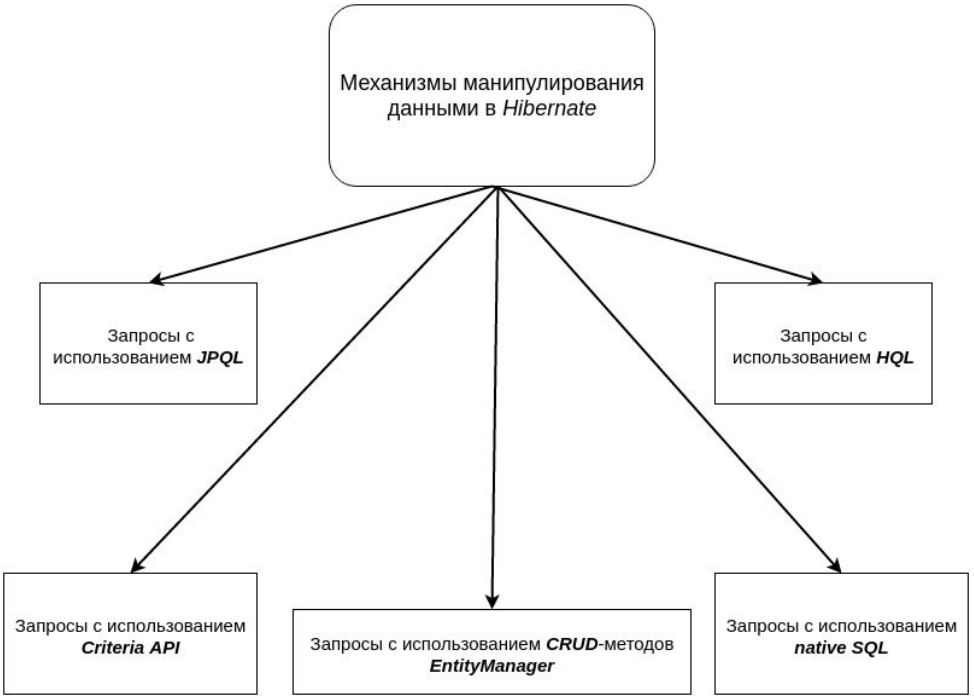
Java для молодых специалистов Банка ВТБ

Урок 15

Spring Data

JPQL

Набор методов CRUD-класса **EntityManager** ограничивает возможности по манипулированию сущностями. Например, метод **find()** позволяет искать сущность только по идентификатору. Для создания более гибких запросов нужно использовать другие механизмы:



- **Запросы с использованием JPQL (Java Persistence Query Language)** — объектно-ориентированный язык запросов, который описан в спецификации JPA. В отличие от SQL, он оперирует сущностями на уровне кода, а в дальнейшем поставщик постоянства транслирует эти запросы в SQL-запросы к БД;
- **Запросы с использованием HQL (Hibernate Query Language)** — аналогичен JPQL, но используется только в Hibernate;
- **Запросы с использованием CRUD-методов** — запросы к базе данных с помощью методов, рассмотренных в предыдущей главе;
- **Запросы с использованием Criteria API** — запросы, которые последовательно формируются с помощью объектов и методов.

Стоит отметить, что все языки запросов очень похожи на SQL. Рассмотрим синтаксис языка запросов JPQL.

Обзор синтаксиса

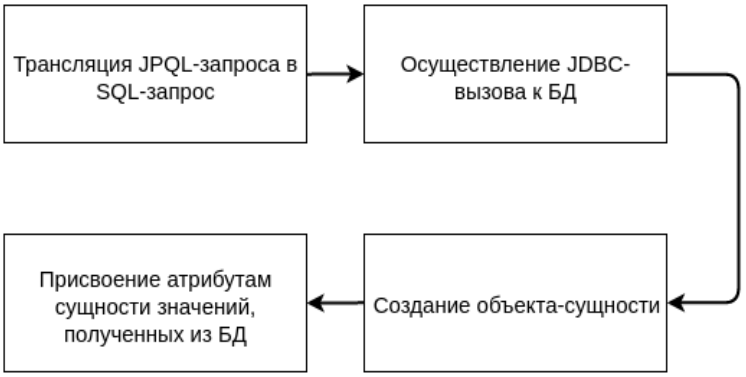
Главное отличие JPQL от SQL состоит в том, что JPQL-запросы манипулируют сущностями, то есть объектами классов. Самый простой JPQL-запрос, который делает выборку всех объектов-сущностей класса **Article** из базы данных, выглядит так:

```
SELECT a FROM Article a
```

Особенности этого запроса:

- оператор **FROM** указывает на класс (в данном случае — класс **Article**), выборку объектов которого необходимо сделать из соответствующей ему таблицы в базе данных;
- в блоке оператора **FROM** указывается псевдоним класса (в данном случае — **a**).

При использовании JPQL-запросов происходит следующее:



Если необходимо применить определенный критерий поиска, то запрос будет выглядеть так:

```
SELECT a FROM Article a WHERE a.id=2
```

В данном случае псевдоним **a** используется для доступа к атрибутам класса.

Возвращать можно не только объекты, но и атрибуты класса. Запрос, возвращающий атрибуты объекта, может выглядеть следующим образом:

```
SELECT a.firstname, a.lastname FROM Author a
```

Если используется привязка параметров, запрос может быть таким:

```
SELECT a.firstname, a.lastname FROM Author a WHERE a.id=?1
```

В случае именованных параметров:

```
SELECT a.firstname, a.lastname FROM Author a WHERE a.id=:id
```

Можно указать поставщику постоянства, что необходимо создать объекты из возвращаемых из БД значений. Например:

```
SELECT NEW com.geekbrains.Person(a.firstname, a.lastname) FROM Author a
```

Класс **Person** не обязан являться сущностью, но должен содержать конструктор с указанной в запросе сигнатурой.

Запросы

Динамические запросы

Изучим механизм, с помощью которого осуществляются запросы. Вся необходимая функциональность содержится в методах класса **EntityManager**.

Рассмотрим основные методы:

- **createQuery(String jpqlString)** — метод, принимающий строку JPQL-запроса и возвращающий объект класса **Query**;
- **createNamedQuery(String name)** — метод для именованных запросов, принимающий их названия и возвращающий объекты класса **Query**;
- **createNativeQuery(String sqlString)** — метод для запросов с использованием SQL, возвращает объект класса **Query** и других.

Эти методы имеют свои перегруженные аналоги, принимающие дополнительный параметр типа **Class** или **Class<T>**, которые помогают избежать лишних преобразований типов.

Но стоит отметить, что все вышеперечисленные методы не обеспечивают выполнение запроса как такового — для этого необходимо использовать:

- **getSingleResult()** — для получения одиночного объекта в качестве конечного результата запроса;
- **getResultList()** — для получения коллекции объектов как конечного результата запроса;
- и другие.

Например, получение всех авторов может выглядеть следующим образом:

```
// Осуществление запроса, возвращающего коллекцию
List<Author> authors = em.createQuery("SELECT a FROM Author a",
Account.class).getResultList();

// Осуществление запроса, возвращающего одиночный результат
Author author = em.createQuery("SELECT a FROM Author a WHERE a.id = 1",
Account.class).getResultList();
```

Именованные запросы

Именованные запросы более производительны, чем динамические. Это связано с тем, что преобразование JPQL-запроса в SQL происходит сразу после запуска приложения. Чтобы выполнить именованный запрос, в классе, к которому он будет осуществляться, необходимо объявить аннотацию **@NamedQuery**, содержащую следующие атрибуты:

- **name** — название именованного запроса;

- **query** — JPQL-строка запроса.

Можно объявлять несколько именованных запросов с помощью множественной аннотации **@NamedQueries**.

Объявление именованных запросов:

```
@Entity
@Table(name="author")
@NamedQueries({
  @NamedQuery(name="Author.findAll", query="SELECT a FROM Author a"),
    @NamedQuery(name="Author.findById",  query="SELECT  a  FROM  Author  a  WHERE
a.id=:id ")
})
public class Author{
  // Fields, getter and setters
}
```

В данном листинге показан пример объявления двух именованных запросов:

- метода **Author.findAll** для получения всех авторов;
- запроса **Author.findById** (аналог метода **find** класса **EntityManager**), использующий именованные параметры.

Заметьте, что формат названий именованных запросов рекомендует употреблять имя класса в качестве префикса, а само название отражает операцию и критерий.

В коде использование именованных запросов будет выглядеть так:

```
List<Author>        authors        =        em.createNamedQuery("Author.findAll",
Author.class).getResultList();
Author          author          =          em.createNamedQuery("Author.findById",
Author.class).setParameter("id", 1).getSingleResult();
```

DAO

DAO (Data Access Object) — это уровень доступа к данным. Вся функциональность DAO основывается на классе **EntityManager**. Чтобы создать DAO-уровень для сущности, необходимо выполнить следующие действия:

- создать отдельный пакет для классов уровня доступа к данным, например **com.geekbrains.dao**;
- создать интерфейс доступа к сущности, например **ArticleDAO**;
- в интерфейсе объявить методы, исходя из набора требующихся операций над сущностью;
- создать класс, имплементирующий данный интерфейс, и реализовать в нем методы интерфейса, используя **EntityManager** для обеспечения функциональности.

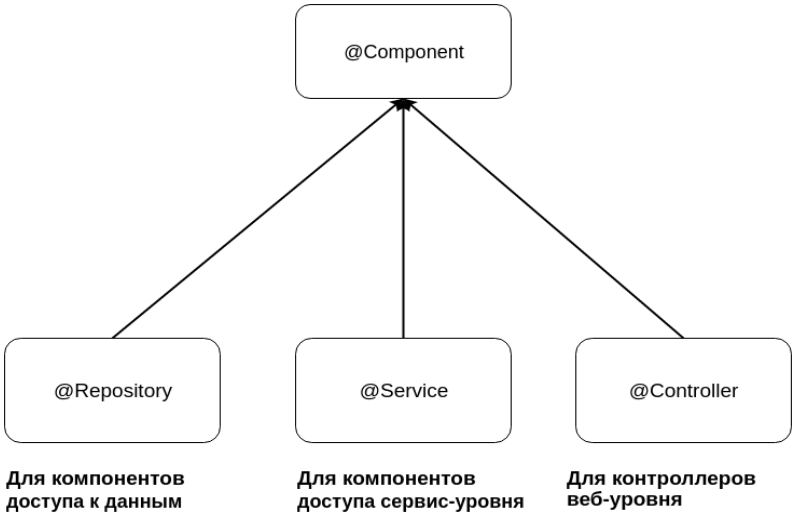
Предположим, что для сущности **Article** необходимо реализовать следующие операции: поиск всех сущностей, сохранение сущности **Article**, получение сущности по id, обновление сущности, удаление сущности по id. Исходя из этого, интерфейс доступа будет выглядеть так:

```
public interface ArticleDAO {
  List<Article> findAll();
  void save(Article article);
  Article findById(long id);
  void update(Article article);
  void delete(Article article);
}
```

В приведенном интерфейсе нет специальных аннотаций. Реализация данного интерфейса:

```
@Repository
public class ArticleDAOImpl implements ArticleDAO {
  @PersistenceContext
  private EntityManager em;

  @Override
  public List<Article> findAll() {
    return em.createNamedQuery("Article.findAll", Article.class).getResultList();
  }

  @Override
  public void save(Article article) {
    em.persist(article);
  }

  @Override
  public Article findById(long id) {
    return em.find(Article.class,id);
  }

  @Override
  public void update(Article article) {
    em.merge(article);
  }

  @Override
  public void delete(Article article) {
    em.remove(article);
  }
}
```

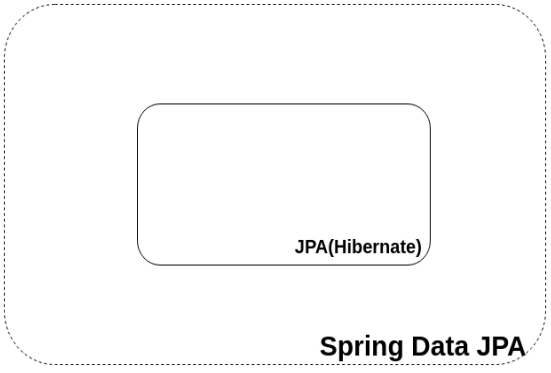
Приведенный выше класс также является компонентом Spring, но он аннотирован как **@Repository**. Это уточняющая аннотация по отношению к **@Component**. Указывает на то, что данный компонент необходим для доступа к данным. Фактически, **@Repository** является одним из видов аннотации **@Component**. Данный класс также будет управляться контейнером Spring и будет пригодным для внедрения в другие классы и компоненты.



В данном коде происходит внедрение **EntityManager** с помощью аннотации **@PersistenceContext**, а не **@Autowired**. Это связано с тем, что в проекте могут использоваться сразу несколько источников данных, а аннотация **@PersistenceContext** обладает широким набором атрибутов, которые необходимы для точного указания настроек контекста постоянства.

Spring Data JPA

Долгое время для организации доступа к данным использовался подход, описанный выше. Но он состоит в основном из тривиальных задач: реализовать интерфейс, его имплементацию, внедрить менеджер сущностей и подобных. Потом появилась альтернатива — использование Spring Data JPA. Фактически, это очередная абстракция **EntityManager**, но она избавляет от рутинной работы. Кроме того, появился легкий способ написания запросов к БД — без использования JPQL. Spring Data JPA является абстракцией над Hibernate, который реализует спецификацию JPA.



Используя Spring Data JPA, мы оперируем репозиториями, а не DAO. Объявление репозитория, аналогичного DAO-классу в предыдущей главе, будет выглядеть так:

```
@Repository
public interface ArticleRepository extends JpaRepository<Article, Long> {
  Article findByTitle(String title);
}
```

Интерфейс **ArticleRepository** расширяется интерфейсом **JpaRepository**, который типизирован классом самой сущности и классом ее поля id.

Интерфейс **JpaRepository<T, ID>** предоставляет множество методов для CRUD-операций:

Метод	Назначение
`long count()`	Возвращает количество доступных сущностей
`void delete(T entity)`	Удаляет указанную сущность
`void deleteAll()`	Удаляет все сущности
`void deleteAll(Iterable<? extends T> entities)`	Удаляет указанный набор сущностей
`void deleteById(ID id)`	Удаляет сущность с указанным id
`boolean existsById(ID id)`	Проверяет существование сущности с указанным id
`Iterable<T> findAll()`	Возвращает все объекты данного типа
`Iterable<T> findAllById(Iterable<ID> ids)`	Получает все объекты по набору id
`Optional<T> findById(ID id)`	Возвращает сущность по id
`<S extends T> S save(S entity)`	Сохраняет указанную сущность
`<S extends T> Iterable<S> saveAll(Iterable<S> entities)`	Сохраняет указанный набор сущностей

В коде определен собственный метод поиска сущности по заданному критерию. Spring Data преобразовывает название метода и его сигнатуру в соответствующий запрос.

Ключевое слово	Пример имени метода	JPQL код
`And`	`findByLastnameAndFirstname`	`… where x.lastname = ?1 and x.firstname = ?2`
`Or`	`findByLastnameOrFirstname`	`… where x.lastname = ?1 or x.firstname = ?2`
`Is, Equals`	`findByFirstname, findByFirstnameIs, findByFirstnameEquals`	`… where x.firstname = ?1`
`Between`	`findByStartDateBetween`	`… where x.startDate between ?1 and ?2`
`LessThan`	`findByAgeLessThan`	`… where x.age < ?1`
`LessThanEqual`	`findByAgeLessThanEqual`	`… where x.age <= ?1`
`GreaterThan`	`findByAgeGreaterThan`	`… where x.age > ?1`
`GreaterThanEqual`	`findByAgeGreaterThanEqual`	`… where x.age >= ?1`
`After`	`findByStartDateAfter`	`… where x.startDate > ?1`
`Before`	`findByStartDateBefore`	`… where x.startDate < ?1`
`IsNull`	`findByAgeIsNull`	`… where x.age is null`
`IsNotNull,NotNull`	`findByAge(Is)NotNull`	`… where x.age not null`
`Like`	`findByFirstnameLike`	`… where x.firstname like ?1`

NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1(parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1(parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1(parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> ages)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firstame) = UPPER(?1)

Если функциональности методов **JpaRepository** недостаточно, а описать запрос через название метода проблематично, то можно воспользоваться JPQL:

```
@Query("select a from Article a where a.author = :author")
List<Article> findArticleByAuthor(@Param("author") Author author);
```

Этих объявлений вполне хватит, чтобы Spring самостоятельно создал объект класса, реализующего этот интерфейс. Чтобы использовать этот класс, необходимо произвести внедрение с помощью **@Autowired** по данному интерфейсу.

Для подключения к Spring Boot проекту модуля Spring Data JPA используется следующая зависимость:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Транзакции и уровень сервисов

В сервис-уровень помещена необходимая бизнес-логика, которая оперирует данными, получаемыми из уровня доступа к данным и веб-уровня. В методах сервис-уровня происходит работа с транзакциями.

Управление транзакциями бывает двух видов:

- управление приложением — явное открытие и фиксация транзакций разработчиком;

- управление контейнером — управление транзакциями делегируется контейнеру, в котором выполняется приложение.

По определению, открытие и фиксация транзакции происходит на сервис-уровне. Ведь на нем выполняется бизнес-логика, которая может оперировать несколькими сущностями, а значит, несколькими классами уровня доступа к данным. Но каким образом организовать транзакции на сервис-уровне, если наш **EntityManager** инкапсулирован в классах DAO-уровня и поэтому вызывать в сервис-уровне метод **em.getTransaction()** невозможно? В данном случае необходимо делегировать управление транзакциями контейнеру.

Чтобы указать контейнеру, что в методе необходимо открыть транзакцию и зафиксировать ее по окончании выполнения метода, нужно использовать аннотацию **@Transactional**:

```
@Service
public class ArticleServiceImpl implements ArticleService {
  @Autowired
  private ArticleRepository articleRepo;

  public List<Article> getAll() {
    return articleRepo.findAll();
  }

  @Override
  @Transactional(readOnly=true)
  public List<Article> getAll() {
    return articleRepo.findAll();
  }

  @Override
  @Transactional(readOnly=true)
  public Article get(Long id) {
    return articleRepo.findOne(id);
  }

  @Override
  @Transactional
  public void save(Article article) {
    articleRepo.save(article);
  }
}
```

Здесь ко всем методам сервиса применяется аннотация **@Transactional**. Она указывает контейнеру, что необходимо открыть транзакции перед началом выполнения кода метода и закрыть их после того, как весь код метода выполнен. Если транзакция подразумевает только чтение из БД, то можно воспользоваться атрибутом **readOnly** и указать значение **true**.

Практическое задание

1. Перенесите всю работу с товарами на репозитории;
2. Добавьте пагинацию, с выводом на страницу по 10 товаров; (с фронтендом можете не заморачиваться, просто сделайте ссылки на страницы 1, 2, 3, 4, 5)

3. * Перенесите фильтры на работу через Specification

Дополнительные материалы

1. Кей С. Хорстманн, Гари Корнелл Java. Библиотека профессионала. Том 1. Основы // Пер. с англ. — М.: Вильямс, 2014. — 864 с.
2. Брюс Эккель. Философия Java // 4-е изд.: Пер. с англ. — СПб.: Питер, 2016. — 1 168 с.
3. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.
4. Г. Шилдт. Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 720 с.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

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