Урок 4

Доступ к данным в Spring. Часть 2

Контекст постоянства. Менеджер сущностей. JPQL. Доступ к атрибутам. Каскадные операции. Управление транзакциями. Spring Data JPA. Сервис-уровень.

Контекст постоянства и EntityManager

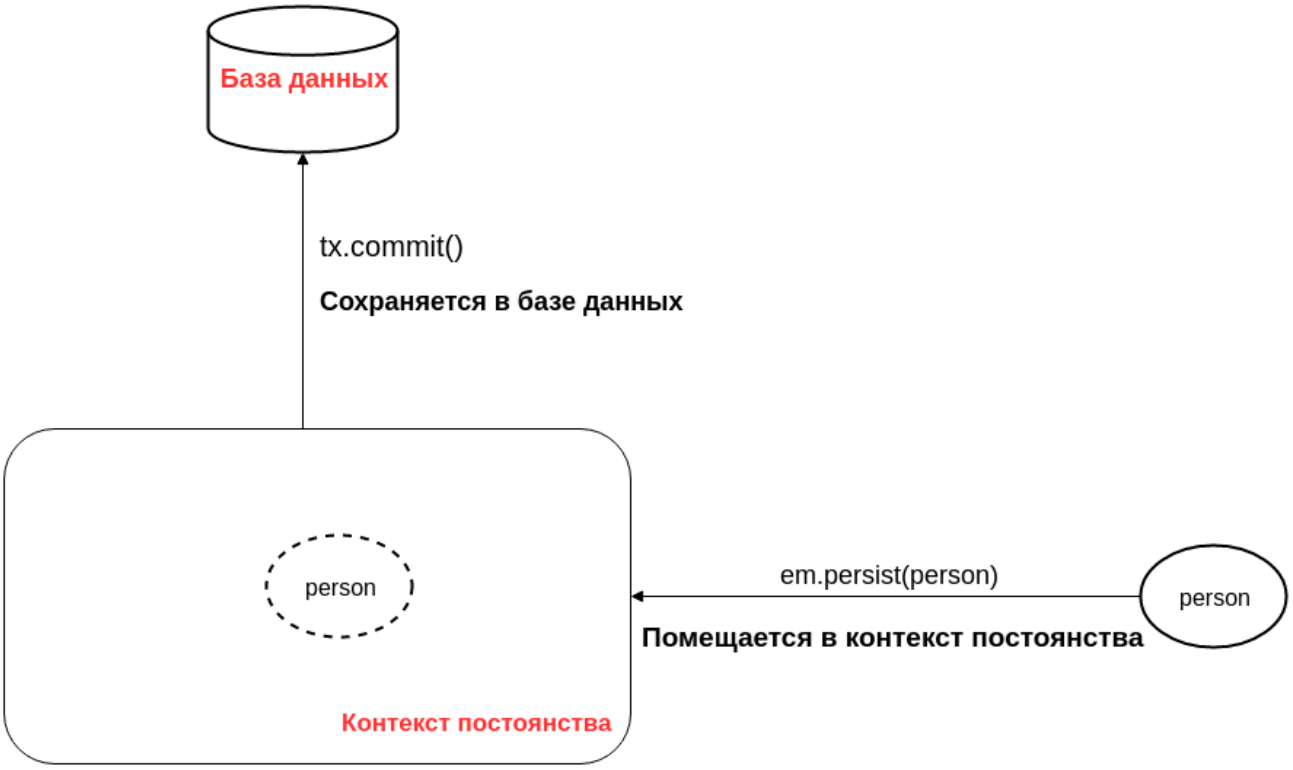
В прошлом уроке мы разобрали, что представляет собой сущность, а также изучили порядок объявления класса-сущности для объектно-реляционного отображения. Рассмотрим порядок взаимодействия наших сущностей с базой данных. Чтобы понять этот процесс, необходимо разобраться с понятием контекста постоянства.

Этот контекст представляет собой набор сущностей, которые управляются Hibernate в данный момент времени. Все управление сущностями возлагается на менеджер сущностей – класс **EntityManager**.

Он обладает полным набором CRUD-операций (**create**, **read**, **update**, **delete**). Данные операции вызываются следующими методами:

```
// Получаем фабрику менеджеров сущностей из контекста Spring, созданного ранее
      EntityManagerFactory factory =context.getBean(EntityManagerFactory.class);
// Из фабрики создаем EntityManager
      EntityManager em = factory.createEntityManager();
        Person person = new Person();
        person.setFirstname("Yuri");
        person.setLastname("Ivanov");

// Получаем транзакцию
        EntityTransaction tx = em.getTransaction();

// Открываем транзакцию
        tx.begin();

// Create (сохраняем в базе данных, тем самым сущность становится управляемой
Hibernate и заносится в контекст постоянства)
        em.persist(person);

// Подтверждаем транзакцию
        tx.commit();

        tx=em.getTransaction();
        tx.begin();

// Read (читаем сущность из базы данных по id)
        Person anotherPerson=em.find(Person.class, 1L);
        tx.commit();


        anotherPerson.setFirstname("Artem");

        tx=em.getTransaction();
        tx.begin();

// Update
        em.merge(anotherPerson);
        tx.commit();
```

Ниже графически представлен процесс сохранения в БД объекта-сущности **person**:



Данное изображение отражает всю суть работы контекста постоянства для любых операций. Главное, что следует запомнить – результат операции никак не отразится на базе данных, пока не будет произведена фиксация транзакции.

Процесс, изображенный на данной схеме, состоит из следующих этапов:

- открытие транзакции;
- сохранение объекта в контексте постоянства с помощью метода **persist**;
- сохранение объекта в базе данных после фиксации транзакции.

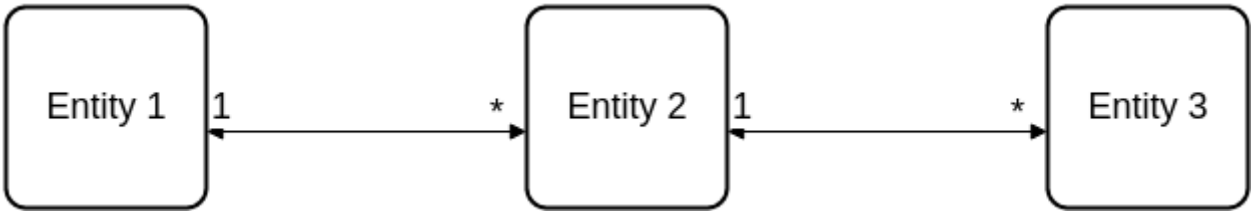
Данная последовательность характерна не только для метода **persist**, но и для остальных методов. Но в случае использования Hibernate совместно со Spring у вас не будет необходимости самостоятельно открывать и закрывать транзакции – этим будет заниматься сам Spring.

EntityManager содержит и другие методы, предназначенные для работы с сущностями:

- **detach(Object obj)** – удаляет сущность **obj** из контекста постоянства (но не из БД), и объект перестает находится под управлением Hibernate;
- **refresh(Object obj)** – синхронизирует сущность **obj** с БД. Ее поля будут иметь те значения, которые находились в столбцах строки БД на момент применения данного метода.

Доступ к атрибутам

Представим, что есть сущность, полем которой является коллекция сущностей (связь «один ко многим»), а поле дочерней сущности – тоже коллекция сущностей и так далее. В итоге совокупность связей будет выглядеть следующим образом:



Вероятно было бы то, что при получении из БД **Entity 1** за ней тянулась бы целая цепочка коллекций других сущностей, что сказалось бы на производительности негативно. Но JPA задает механизм для точного определения вида инициализации. Есть две стратегии выборки, которые задаются в атрибуте **fetch** аннотаций **@OneToOne**, **@OneToMany**, **@ManyToOne**, **@ManyToMany**:

- **LAZY** – «ленивая» выборка, при которой поля, являющиеся другими сущностями, не вытягиваются из базы данных вместе с основной сущностью, а запрашиваются отдельно (дополнительным запросом в БД), только при вызове геттера данного поля;
- **EAGER** – ранняя выборка, при которой поля, являющиеся сущностями, вытягиваются из базы вместе с основной сущностью с помощью одного запроса к БД.

Атрибут **fetch** аннотаций связи может иметь следующие значения:

- **FetchType.LAZY** – для ленивой выборки;
- **FetchType.EAGER** – для ранней выборки.

В коде задание стратегии выборки будет выглядеть так:

```
@Entity
@Table(name="country")
public class Country{

      @Id
      @GeneratedValue(strategy = GenerationType.IDENTITY)
      @Column(name="id")
      private Long id;


      @Column(name="title")
      private String title;

      @OneToMany(mappedBy="country" , fetch=FetchType.EAGER)
      private List<City> cities;

       // Геттеры и сеттеры
}
```

По умолчания каждая из связей использует стратегии следующим образом:

Аннотация	Стратегия выборки по умолчанию
@OneToOne	EAGER
@ManyToOne	EAGER
@OneToMany	LAZY
@ManyToMany	LAZY

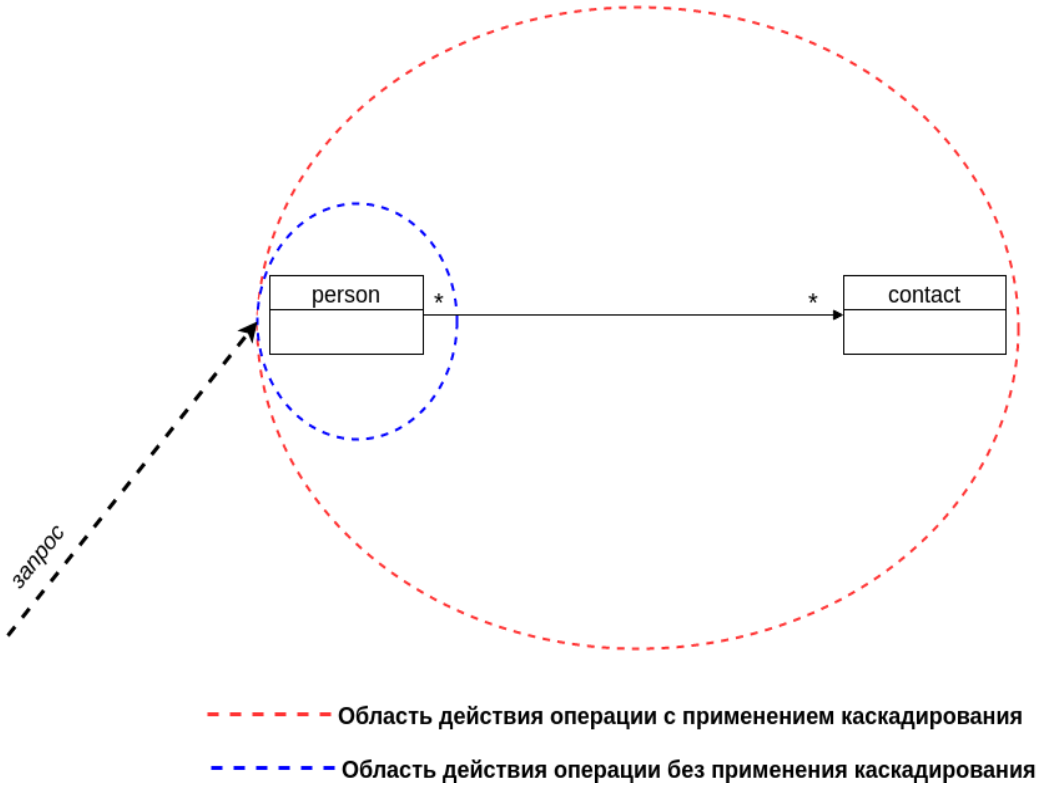
Для связи «OneToOne» можно применить только EAGER-стратегию.

Выбор стратегии остается на совести разработчика. EAGER-стратегия позволит загружать все данные с помощью небольшого количества запросов к базе данных. LAZY-стратегия не позволит заполнить всю используемую память. Вы будете контролировать, какой объект будет загружаться, но каждый раз придется осуществлять доступ к базе данных.

Задавать стратегию выборки можно и для простых атрибутов. Это можно осуществить с помощью аннотации **@Basic**.

Каскадные операции

Помимо стратегии выборки в атрибутах аннотаций связи можно указывать параметры каскадирования операций. Фактически, это означает, что при удалении сущности A из соответствующей таблицы в БД удаляется и ее дочерняя сущность B. Например, если есть сущность **Person**, которая включает в себя поле класса **Contact**, то вместе с объектом класса **Person** будут удалены строка в таблице **person** и ассоциированная с ней строка таблицы **contact**:



В коде каскадирование задается в атрибуте **cascade** аннотаций связи. Данный атрибут может принимать следующие значения перечисления **CascadeType** пакета **javax.persistence.CascadeType**:

- **CascadeType.ALL** – каскадирование будет применяться ко всем операциям;
- **CascadeType.REMOVE** – только к методу удаления;
- **CascadeType.PERSIST** – только к методу сохранения;
- **CascadeType.MERGE** – к методу обновления;
- **CascadeType.REFRESH** – к методу синхронизации с БД;
- **CascadeType.DETACH** – каскадирование применяется к методу удаления сущности из контекста постоянства (но не из БД).

В коде это выглядит так:

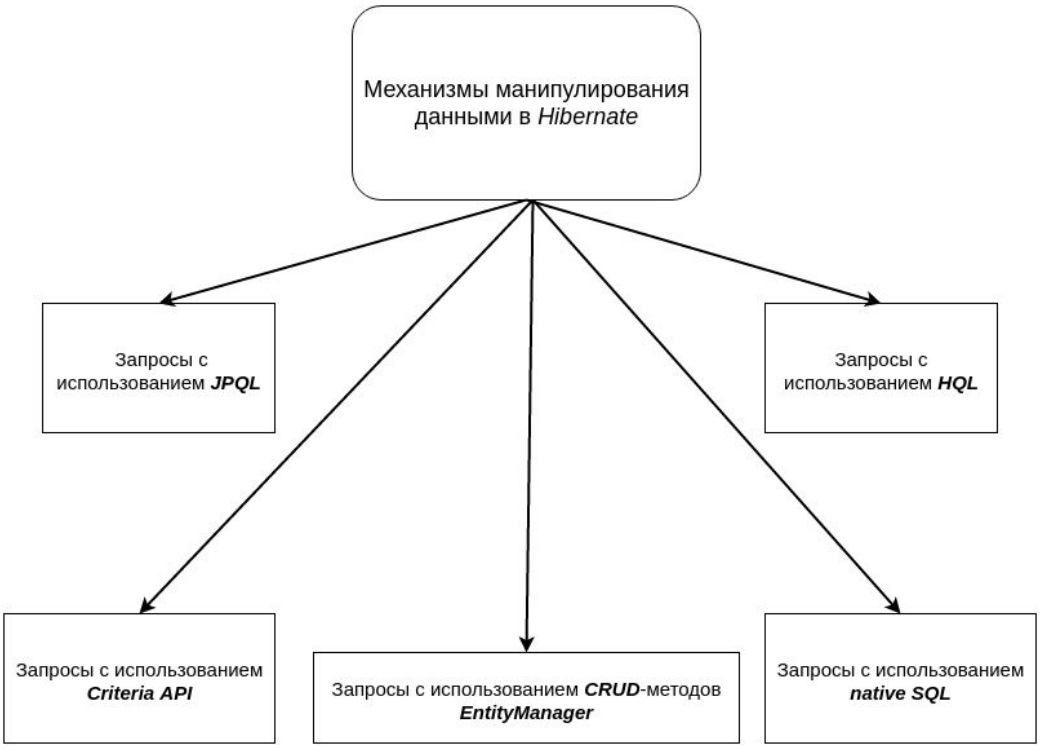
```
@ManyToOne(cascade = CascadeType.PERSIST)
@JoinColumn(name="author_id")
private Author author;
```

Кроме того, имеется возможность задавать несколько параметров:

```
@ManyToOne(cascade = { CascadeType.REMOVE, CascadeType.PERSIST})
@JoinColumn(name="author_id")
private Author author;
```

JPQL

Набор методов **CRUD**-класса **EntityManager** довольно сильно 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, данный язык запросов 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, который основан на HQL и позиционируется как более новая и стандартизованная версия этого языка. Запросы с использованием Criteria API оставим на самостоятельное рассмотрение по дополнительным материалам.

Обзор синтаксиса

Главным отличием JPQL от SQL является то, что JPQL-запросы манипулируют сущностями, т.е. 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 же используется привязка параметров, то 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 net.zt.lesson.Person(a.firstname, a.lastname) FROM Author a
```

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

Как видите, данный язык запросов практически аналогичен SQL.

Запросы

Динамические запросы

Изучим механизм, с помощью которого осуществляются запросы. Весь необходимый функционал 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 следующие методы получения результата:

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

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:

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

Заметьте, что формат названий именованных запросов рекомендует употребление имени 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. Весь функционал 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 **net.zt.funcode.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 функционала;

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:

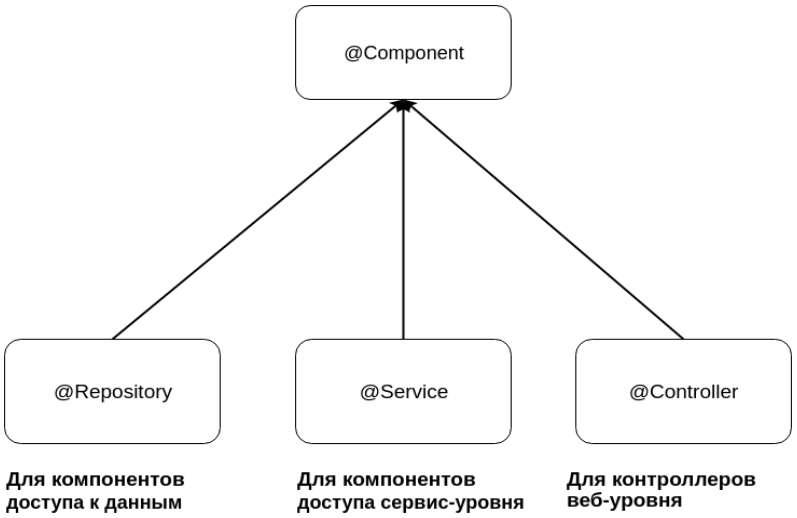
```
public interface ArticleDAO {

      // Получить все сущности Article из БД
      List<Article> findAll();

      // Сохранить сущности в БД
      void save(Article article);

      // Получить сущность из БД по id
      Article findById(long id);

      // Обновить сущность в БД
      void update(Article article);

      // Удалить сущность из БД
      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() {

		// Использует именованный запрос 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.

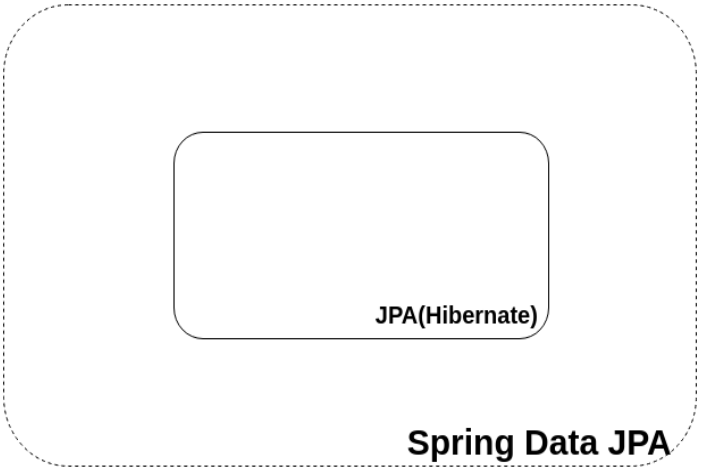


Аннотации **@Service** и **@Controller** будут рассмотрены позже.

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 в предыдущей главе. Но он состоит в основном из тривиальных задач: реализовать интерфейс, его имплементацию, внедрить менеджер сущностей и т.п. Спустя некоторое время 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> {

      // Добавляем свой метод, позволяющий искать статью по имени
      Article findByTitle(String title);

}
```

Интерфейс **ArticleRepository** расширяется интерфейсом **JpaRepository**, который типизирован классом самой сущности и классам ее поля id. Интерфейс **ArticleRepository** унаследует множество необходимых методов для совершения CRUD-операций:

- **findAll()** – получение всех сущностей;
- **findOne(Long id)** – получение сущности по id;
- **delete(Long id)** – удаление сущности с заданным id;
- **save(S entity)** – сохранение сущности (S – класс, которым типизирован интерфейс JpaRepository) и др.

В коде определен собственный метод поиска сущности по определенному критерию. Spring Data преобразовывает название метода и его сигнатуру в соответствующий запрос. Чтобы научиться писать эти методы, настоятельно рекомендую ознакомиться с шаблонами в дополнительном материале №1.

Если функционала 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 достаточно, чтобы Spring самостоятельно создал объект класса, реализующего этот интерфейс. Чтобы использовать этот класс, необходимо произвести внедрение с помощью **@Autowired** по данному интерфейсу. О конфигурировании проекта для использования Spring Data JPA поговорим в разделе «Практика».

Транзакции и уровень сервисов

Между уровнем доступа к данным и веб-уровнем, который будет рассмотрен на следующем уроке, располагается 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 определенная 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, а значит, 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;

	@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**.

Практика

В данном разделе будет модифицирован проект, разработанный в предыдущем уроке. Добавим репозитории и сервис-уровень.

Добавление зависимостей

Так как в проекте будет использоваться Spring Data JPA, необходимо добавить зависимость:

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
    <version>1.10.5.RELEASE</version>
</dependency>
```

Конфигурация

Класс конфигурации будет выглядеть так:

```
@Configuration
@EnableJpaRepositories("net.zt.funcode.repository")
@EnableTransactionManagement
@ComponentScan("net.zt.funcode.service")
public class AppConfig {


      @Bean(name="dataSource")
      public DataSource getDataSource(){
            // Создаем источник данных
            DriverManagerDataSource dataSource = new DriverManagerDataSource();
            // Задаем параметры подключения к базе данных

dataSource.setUrl("jdbc:postgresql://localhost:5432/geekbrains-lesson3");
            dataSource.setUsername("postgres");
            dataSource.setDriverClassName("org.postgresql.Driver");
            dataSource.setPassword("123");
            return dataSource;
      }

      @Bean(name="entityManagerFactory")
      public LocalContainerEntityManagerFactoryBean getEntityManager(){
           // Создаем класса фабрики, реализующей интерфейс
FactoryBean<EntityManagerFactory>
            LocalContainerEntityManagerFactoryBean factory = new
LocalContainerEntityManagerFactoryBean();
            // Задаем источник подключения
            factory.setDataSource(getDataSource());
            // Задаем адаптер для конкретной реализации JPA,
            //  указывает,  какая  именно  библиотека  будет  использоваться  в
качестве поставщика постоянства
            factory.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
                        // Указание пакета, в котором будут находиться
классы-сущности
            factory.setPackagesToScan("net.zt.funcode.domain");

            // Создание свойств для настройки Hibernate
            Properties jpaProperties = new Properties();
            //  Указание  диалекта  конкретной  базы  данных  (необходимо  для
генерации запросов Hibernate к БД)
            jpaProperties.put("hibernate.dialect",
"org.hibernate.dialect.PostgreSQLDialect");
            //  Указание  максимальной  глубины  связи  (будет  рассмотрено  в
следующем уроке)
            jpaProperties.put("hibernate.max_fetch_depth", 3);
```

```
            // Определение максимального количества строк, возвращаемых за один
запрос из БД
            jpaProperties.put("hibernate.jdbc.fetch_size", 50);
            // Определение максимального количества запросов при использовании
пакетных операций
            jpaProperties.put("hibernate.jdbc.batch_size", 10);

            // Включает логирование
            jpaProperties.put("hibernate.show_sql", true);

            factory.setJpaProperties(jpaProperties);

            return factory;

      }

      @Bean(name="transactionManager")
      public    JpaTransactionManager    transactionManager(EntityManagerFactory
entityManagerFactory){

            // Создание менеджера транзакций
            JpaTransactionManager tm= new JpaTransactionManager();
            tm.setEntityManagerFactory(entityManagerFactory);
            return tm;

      }

}
```

В данном коде появились следующие элементы:

- аннотация **@EnableJpaRepositories** – обеспечивает возможность использования Spring Data JPA. В качестве параметра указывается пакет, в котором будут находиться классы-репозитории;
- аннотация **@EnableTransactionManagement** – указывает Spring на необходимость в управлении транзакциями;
- бин **transactionManager** –менеджер транзакций, который работает «поверх» менеджера сущностей.

Создание репозиториев

Перед разработкой интерфейсов создадим пакет, в котором будут находиться все интерфейсы репозиториев (например, **net.zt.funcode.repository**).

Создадим интерфейсы репозиториев, которые будут расширять интерфейс **JpaRepository**.

Для сущностей класса **Author**:

```
@Repository
public interface AuthorRepository extends JpaRepository<Author, Long> {

}
```

Он расширяет интерфейс **JpaRepository**, которому необходимо указать два параметра для типизации (класс сущности и класс поля id сущности). По необходимости в данный интерфейс можно добавить собственные методы, используя предыдущий раздел о Spring Data JPA и дополнительный материал №1.

Для сущностей класса **Article**:

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

}
```

Для сущностей класса **Category**:

```
@Repository
public interface CategoryRepository extends JpaRepository<Category, Long > {

}
```

Создание сервис-уровня

Для уровня сервисов необходимо создать пакет, в котором будут находиться классы-сервисы, например **net.zt.funcode.service**.

Создание сервис-уровня производится в два этапа:

- создание интерфейсов;
- создание классов, реализующих данные интерфейсы.

Необходимо разработать следующие интерфейсы:

- ArticleService;
- AuthorService;
- CategoryService.

Код интерфейса **ArticleService** будет иметь следующий вид:

```
public interface ArticleService {

      public List<Article> getAll();
      public Article get(Long id);
      public void save(Article article);
}
```

Код интерфейса **AuthorService**:

```
public interface AuthorService {

      public Author get(Long id);
      public List<Author> getAll();
      public void save(Author author);
      public void remove(Author author);
}
```

Код интерфейса **CategoryService**:

```
public interface CategoryService {

      public Category get(Long id);
      public List<Category> getAll();
      public void save(Category category);
      public void remove(Category category);

}
```

Теперь разработаем классы, реализующие данные интерфейсы:

- `ArticleServiceImpl;`
- `AuthorServiceImpl;`
- `CategoryServiceImpl.`

Все вышеперечисленные классы будут являться компонентами Spring.

Код класса **ArticleServiceImpl**:

```
@Service
public class ArticleServiceImpl implements ArticleService {

	@Autowired
	private ArticleRepository articleRepo;

	@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);
	}
}
```

Класс **AuthorServiceImpl**:

```
@Service
public class AuthorServiceImpl implements AuthorService {

	@Autowired
	private AuthorRepository authorRepo;

	@Override
	@Transactional(readOnly=true)
	public Author get(Long id) {

		return authorRepo.findOne(id);
	}

	@Override
	@Transactional(readOnly=true)
	public List<Author> getAll() {

		return authorRepo.findAll();
	}

	@Override
	@Transactional
	public void save(Author author) {

		authorRepo.save(author);
	}

	@Override
	@Transactional
	public void remove(Author author) {

		authorRepo.delete(author);
	}



}
```

Класс **CategoryServiceImpl**:

```
@Service
public class CategoryServiceImpl implements CategoryService {

	@Autowired
	private CategoryRepository categoryRepo;

	@Override
	@Transactional(readOnly=true)
	public Category get(Long id) {

		return categoryRepo.findOne(id);
	}

	@Override
	@Transactional(readOnly=true)
	public List<Category> getAll() {

		return categoryRepo.findAll();
	}

	@Override
	@Transactional
	public void save(Category category) {

		categoryRepo.save(category);
	}

	@Override
	@Transactional
	public void remove(Category category) {

		categoryRepo.delete(category);

	}

}
```

Код учебного проекта

Конечный вариант кода программы, разработанной в ходе данного урока: https://github.com/Firstmol/Geekbrains-lesson4

Практическое задание

1. Изучить дополнительный материал №1.
2. Создать проект, подключить необходимые зависимости и настроить конфигурацию, как показано в данном уроке.
3. Разработать репозитории для сущностей, созданных в домашнем задании к уроку 2, которые смогли бы обеспечить:
    i. добавление объявления;

ii. удаление объявления;

iii. получение объявления по id;

iv. обновление атрибутов объявления;

v. получение всех объявлений;

vi. получение всех объявлений из данной категории;

vii. получение компании, которой принадлежит данное объявление;

viii. получение компании по id;

ix. получение всех категорий.

4. Разработать сервис-уровень для данной предметной области;
5. Выложить сделанный проект в свой репозиторий на github для проверки преподавателем.

Дополнительные материалы

1. Создание собственных методов репозитория: http://docs.spring.io/spring-data/jpa/docs/current/reference/html/#project п.5.3.2;
2. Крис Шефер, Кларенс Хо. Spring 4 для профессионалов (4-е издание) – стр. 425-450.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Крис Шефер, Кларенс Хо. Spring 4 для профессионалов (4-е издание).
2. Крейг Уоллс. Spring в действии.