GeekBrains

Основы веб-разработки на Spring Framework

Доступ к данным в Spring. Часть 2

DAO. Spring Data JPA. Сервис-уровень.

Оглавление

Доступ к базе данных из Spring приложения

На прошлом занятии мы рассмотрели принципы работы с Hibernate, теперь пришло время применить это на практике при разработке веб-приложения.

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:

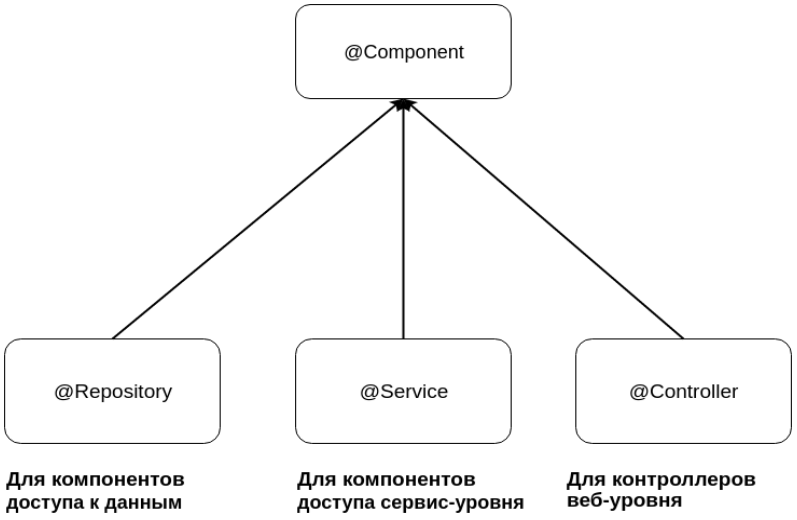
```
public interface ArticleDAO {
  List<Article> findAll();
  void save(Article article);
  Article findById(Long id);
  void update(Article article);
  void delete(Article article);
}
```

В приведенном интерфейсе нет специальных аннотаций. Реализация данного интерфейса:

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

  @Override
  public List<Article> findAll() {
                      return      entityManager.createQuery("from      Article",
Article.class).getResultList();
  }

  @Override
  public void save(Article article) {
```

```
    entityManager.persist(article);
  }

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

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

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

Приведенный выше класс также является компонентом Spring, но он помечен аннотацией **@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

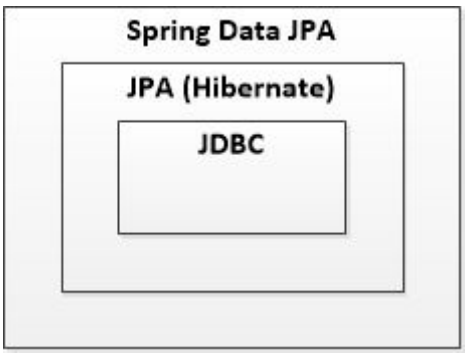
Долгое время для организации доступа к данным использовался подход, описанный выше. Но он состоит в основном из тривиальных задач: реализовать интерфейс, его имплементацию, внедрить менеджер сущностей и подобных. При этом от сущности к сущности код базовых операций (сохранения, чтения, удаления и обновления данных) не особо отличается, если в коде выше, в

ArticleDAOImpl заменить слово/класс Article на любой другой класс (например, Product), то все будет работать. Давайте проверим:

```
public interface ProductDAO {
  List<Product> findAll();
  void save(Product product);
  Product findById(Long id);
  void update(Product product);
  void delete(Product product);
}

@Repository
public class ProductDAOImpl implements ProductDAO {
  @PersistenceContext
  private EntityManager entityManager;

  @Override
  public List<Product> findAll() {
                        return      entityManager.createQuery("from      Product",
Product.class).getResultList();
  }

  @Override
  public void save(Product product) {
    entityManager.persist(product);
  }

  @Override
  public Product findById(Long id) {
    return entityManager.find(Product.class, id);
  }

  @Override
  public void update(Product product) {
    entityManager.merge(product);
  }

  @Override
  public void delete(Product product) {
    entityManager.remove(product);
  }
}
```

Как видите каких-то изменений в коде кроме замены слова Article/article -> Product/product никаких больше изменений не произошло. Если мы возьмем 100 таких классов, то получим столько же очень похожих кусков кода. Чтобы избавить от написания одного и того же кода, фреймворк Spring предоставляет модуль Spring Data JPA.

Spring Data JPA, фактически, является очередной абстракцией над Hibernate (Hibernate же является реализацией спецификации JPA). Ну а на самом нижнем уровне по-прежнему JDBC.



Что нам дает и зачем нужен еще один уровень абстракции? Spring Data генерирует стандартный код для работы с базами данных, для этого достаточно указать сущность для которой нужно сгенерировать этот код и тип ID этой сущности. Например, если мы создадим интерфейс и унаследуем его от CrudRepository:

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

То нам будет “из коробки” доступен набор CRUD-операций по работе с сущностью Article: save(...), saveAll(...), findAll(), findById(...), count(), delete(...), deleteById(...), deleteAll(), deleteAll(...) и т.д. Из названий методов должно быть понятно что они позволяют делать. То есть мы можем заинжектить подобный репозиторий в сервис и спокойно с ним работать, весь код, реализующий эти методы подготовит сам Spring.

Spring Data JPA предоставляет следующий набор репозиториев. Каждый следующий расширяет возможности предыдущего.

1. **Базовый интерфейс**

```
public interface Repository<T, ID> {
}
```

2. **CrudRepository** - предоставляет набор базовых CRUD операций

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {
  <S extends T> S save(S var1);
  <S extends T> Iterable<S> saveAll(Iterable<S> var1);
  Optional<T> findById(ID var1);
  boolean existsById(ID var1);
  Iterable<T> findAll();
  Iterable<T> findAllById(Iterable<ID> var1);
  long count();
  void deleteById(ID var1);
  void delete(T var1);
  void deleteAll(Iterable<? extends T> var1);
  void deleteAll();
}
```

3. **PagingAndSortingRepository** - добавляет возможность использования пагинации и сортировки

```
public interface PagingAndSortingRepository<T, ID extends Serializable> extends
CrudRepository<T, ID> {
  Iterable<T> findAll(Sort sort);
  Page<T> findAll(Pageable pageable);
}
```

4. **JpaRepository**

```
public interface JpaRepository<T, ID> extends PagingAndSortingRepository<T, ID>,
QueryByExampleExecutor<T> {
  List<T> findAll();
  List<T> findAll(Sort var1);
  List<T> findAllById(Iterable<ID> var1);
  <S extends T> List<S> saveAll(Iterable<S> var1);
  void flush();
  <S extends T> S saveAndFlush(S var1);
  void deleteInBatch(Iterable<T> var1);
  void deleteAllInBatch();
  T getOne(ID var1);
  <S extends T> List<S> findAll(Example<S> var1);
  <S extends T> List<S> findAll(Example<S> var1, Sort var2);
}
```

В таблице ниже приведено назначение стандартных методов:

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

Помимо методов, предоставляемых репозиториями Spring Data, мы можем создавать свои собственные. Функционал методов будет определяться их названием. Если создать метод **Author findByTitle(String title)**, то мы сможем искать статьи по названию; **Author findByTitleAndAuthor** - по

названию и автору; **List<Author> findAllByAuthor(String author)** - все статьи по указанному автору. То есть Spring Data разбирает имя метода на части и по нему формирует запрос.

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

В коде определен собственный метод поиска сущности по заданному критерию. Spring Data преобразовывает название метода и его сигнатуру в соответствующий запрос. В таблице ниже приведены основные конструкции, которые могут быть использованы в названии методов репозитория.

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)

Разберем несколько примеров из таблицы выше:

- **findByFirstname(String firstname)** – поиск пользователя с указанным именем;
- **findByLastnameAndFirstname(String lastname, String firstname)** – поиск пользователя с указанными фамилией и именем;
- **findByAgeIsNull()** – поиск пользователей, у которых поле age не заполнено;

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. 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 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 articleRepository;
```

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

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

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

  @Override
  @Transactional
  public void save(Article article) {
    articleRepository.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 рассмотрим пример добавления репозитория и сервис-уровня.

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>2.0.9.RELEASE</version>
</dependency>
```

Конфигурация

Класс конфигурации будет выглядеть так:

```
@Configuration
@EnableJpaRepositories("com.geekbrains.app.repositories")
@EnableTransactionManagement
@ComponentScan("com.geekbrains")
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:mysql://localhost:3306/geekbrains-lesson3");
    dataSource.setUsername("geek");
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setPassword("geek");
    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("com.geekbrains");

    // Создание свойств для настройки Hibernate
    Properties jpaProperties = new Properties();

    // Указание диалекта конкретной базы данных
  jpaProperties.put("hibernate.dialect", "org.hibernate.dialect.MySQLDialect");

    // Указание максимальной глубины связи
    jpaProperties.put("hibernate.max_fetch_depth", 3);

    // Максимальное количество строк, возвращаемых за один запрос из БД
    jpaProperties.put("hibernate.jdbc.fetch_size", 50);

    // Максимальное количество 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, com.geekbrains.app.repositories). И добавим интерфейсы репозиториев, которые будут расширять интерфейс **JpaRepository**.

Для сущностей класса **Author**:

```
@Repository
public interface AuthorsRepository 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 ArticlesRepository extends JpaRepository<Article, Long> {
}
```

Создание сервис-уровня

Для уровня сервисов необходимо создать пакет, в котором будут находиться классы-сервисы, — com.geekbrains.app.services.

Сервис-уровень создается в два этапа:

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

Необходимо разработать следующие интерфейсы:

- **ArticleService**;
- **AuthorService**;

Код интерфейса **ArticleService**:

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

Код интерфейса **AuthorService**:

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

Теперь разработаем классы, реализующие данные интерфейсы:

- **ArticleServiceImpl**;
- **AuthorServiceImpl**;

Они будут являться компонентами Spring.

Код класса **ArticleServiceImpl**:

```
@Service
public class ArticleServiceImpl implements ArticleService {
  private ArticlesRepository articlesRepository;

  @Autowired
  public void setArticlesRepository(ArticlesRepository articlesRepository) {
    this.articlesRepository = articlesRepository;
  }

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

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

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

Класс **AuthorServiceImpl**:

```
@Service
public class AuthorServiceImpl implements AuthorService {
  private AuthorsRepository authorsRepository;

  @Autowired
  public void setAuthorsRepository(AuthorsRepository authorsRepository) {
    this.authorsRepository = authorsRepository;
  }

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

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

  @Override
  @Transactional
  public void save(Author author) {
    authorsRepository.save(author);
  }

  @Override
  @Transactional
  public void remove(Author author) {
    authorsRepository.delete(author);
  }
}
```

Практическое задание

1. Создать сущность «товар» (id, название, стоимость) и соответствующую таблицу в БД. Заполнить таблицу тестовыми данными (20 записей).
2. Сделать страницу, в которую будут выведены эти записи.
3. С помощью GET-запроса указывать фильтрацию по:
    a. только минимальной,
    b. только максимальной,
    c. или минимальной и максимальной цене.
4. * Добавить постраничное отображение (по 5 записей на странице).

Дополнительные материалы

1. Создание собственных методов репозитория (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.