GeekBrains

Основы веб-разработки на Spring Framework

Java Persistence API / Hibernate. Часть 1

Java Persistence API. Hibernate. Понятие сущности. Объектно-реляционное отображение. Контекст постоянства. Менеджер сущностей. Доступ к атрибутам

Оглавление

Определение Java Persistence API

Взаимодействие с базой данных является неотъемлемой частью при разработке enterprise-приложения. Как правило, для хранения фактографических данных используются реляционные базы данных. В этих БД информация хранится в таблицах, которые могут быть связаны между собой определенными отношениями. Так как обычно при разработке приложений используется объектно-ориентированный подход, возникает задача связать реляционную структуру БД с объектной моделью приложения. Для этого была разработана технология объектно-реляционного отображения (**Object-Relational Mapping**). Эта технология, дает возможность работать с таблицами БД в терминах классов и, наоборот, преобразовывать значения полей классов в данные, пригодные для хранения в БД.

Одной из таких технологий является спецификация Java Persistence API (JPA), которая предоставляет способ управлять реляционными данными в разрабатываемом приложении.

JPA является спецификацией (стандартом), в которой определен набор интерфейсов и аннотаций. Все эти интерфейсы и аннотации находятся внутри пакета «java.persistence».

Существует несколько распространенных реализаций JPA: «Hibernate», «EclipseLink», «Toplink». Существуют также расширения JPA, такие как «Spring Data JPA». Мы будем рассматривать именно Hibernate.

<u>В архитектуре JPA используются следующие классы и интерфейсы:</u>

EntityManager – это интерфейс, который обеспечивает основные действия, такие, как: выборка, добавление и удаление записи, управление транзакциями т.д. В Hibernate используется наследник EntityManager - **Session**.

EntityManagerFactory – это фабричный класс, который создает и управляет несколькими экземплярами «EntityManager». В Hibernate используется наследник EntityManagerFactory - **SessionFactory**.

Persistence – класс, который содержит статические методы для создания «EntityManagerFactory». Данный класс можно рассматривать как главную входную точку в функциональность JPA, но непосредственно он используется редко, т.к. существует множество более удобных методов работы с JPA.

EntityTransaction – интерфейс, который используется для управления транзакциями. Для каждого «EntityManager» создается отдельный экземпляр «EntityTransaction» через метод «getTransaction».

Entity – это объекты-сущности, которые соответствуют таблицам базы данных.

Query - это интерфейс, который используется для получения записей из базы данных.

Давайте приступим к рассмотрению принципов работы с Hibernate.

Подключение Hibernate к проекту

Для работы с PostgreSQL и подключения Hibernate к Maven проекту, необходимо добавить зависимости:

```
<dependency>
    <groupId>org.hibernate</groupId>
```

```
    <artifactId>hibernate-core</artifactId>
    <version>5.4.3.Final</version>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.2.5</version>
</dependency>
```

Первая зависимость - ядро Hibernate, вторая - JDBC драйвер для работы непосредственно с PostgreSQL. Далее, в ресурсы проекта добавляем файл настроек hibernate.cfg.xml.

```
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property
name="connection.driver_class">org.postgresql.Driver</property>
        <property
name="connection.url">jdbc:postgresql://localhost:5432/postgres</property>
        <property name="connection.username">postgres</property>
        <property name="connection.password">admin</property>
        <property name="connection.pool_size">1</property>
        <property
name="dialect">org.hibernate.dialect.PostgreSQL94Dialect</property>
        <property name="show_sql">true</property>
        <property name="hbm2ddl.auto">none</property>
        <property name="current_session_context_class">thread</property>

        <mapping class="com.geekbrains.hibernate.Person"/>
    </session-factory>
</hibernate-configuration>
```

Давайте посмотрим что означают все эти свойства в конфигурационном файле.

- **connection.driver_class** - имя класса JDBC-драйвера для БД к которой мы должны подключиться. Данный драйвер должен быть подключен к проекту как maven-зависимость или каким-то другим способом добавлен в classpath проекта;
- **connection.url** - url для подключения к серверу базы данных. В данном случае сервер базы данных развернут на локальной машине, на стандартном порту 5432, и имя базы данных - postgres;
- **connection.username**, **connection.password** - логин/пароль пользователя;
- **connection.pool_size** - размер пула соединений;
- **dialect** - задает класс-описатель диалекта SQL для конкретной БД. Диалект нужен чтобы Hibernate корректно формировал запросы для используемой СУБД. Нужно указать класс, который соответствует типу и версии БД, которая у вас установлена;
- **show_sql** - включает отображение в логах сервера SQL запросов, которые HIbernate выполняет в процессе работы. Очень полезна в учебных целях и для отладки, но может очень сильно снизить производительность приложения. Поэтому в рабочих версиях приложения ее обычно отключают;

- **hibernate.hbm2ddl.auto** - очень важная настройка, которая задает режим Hibernate для работы со структурой таблиц в БД. Разберем некоторые возможные значения этой настройки
    - **none** - никаких действий при запуске приложения не выполняется. Если структура таблиц не соответствует структуре классов-сущностей приложения то в процессе работы могут возникнуть ошибки
    - **create** - база данных будет создана при запуске приложения, причем все ранее находившиеся в ней таблицы будут удалены и все данные из них потеряны.
    - **update** - структура БД будет обновляться в соответствии со структурой классов-сущностей в приложении. Этот режим наиболее удобен на начальном этапе разработки и в учебных проектах.
    - **validate** - при запуске приложения осуществляется проверка структуры таблиц БД на соответствие структуре классов сущностей в приложении. Если структуры не соответствуют, то приложение завершится с ошибкой. Данный режим чаще всего используется в реальных приложениях.
- **current_session_context_class** - указание области видимости сессии, в данном случае для каждого потока будет своя сессия;
- **mapping class** - перечисление хранимых классов.

Объектно-реляционное отображение и понятие сущности

Обычные объекты классов Java сохраняют свое состояние в оперативной памяти компьютера, но после завершения работы программы вся информация о них теряется. Объекты-сущности — это Java объекты, которые сохраняют свое состояние в базе данных, что обеспечивает их долгосрочное хранение. Чтобы сохранить состояние объекта, в БД должна располагаться таблица, соответствующая классу этого объекта. Сущность — это объект, которым управляет класс **EntityManager**, относящийся к Hibernate.

Чтобы объекты класса являлись сущностями, должны выполняться следующие условия:

- Наличие аннотации **@Entity**;
- Наличие поля, помеченного аннотацией **@Id**, в котором будет храниться уникальный идентификатор сущности;
- Наличие конструктора без аргументов (конструктора по-умолчанию) с модификатором доступа protected или public. Допускается объявление перегруженных конструкторов;
- Отсутствие модификатора **final** в объявлении класса;
- Отсутствие final в полях, ссылающихся на другие сущности, и в их геттерах;
- Класс сущность должен быть верхнеуровневым классом. Перечисления и интерфейсы не могут быть сущностями;
- Сущностями могут являться как обычные классы, так и абстрактные;
- Доступ к полям сущности должен осуществляться через геттеры/сеттеры, сами поля не должны быть public;

Ниже можно видеть пример объявления класса сущности:

```
@Entity
```

```java
@Table(name = "persons")
public class Person {
  @Id
  @GeneratedValue
  @Column(name = "id")
  private Long id;

  @Column(name = "first_name")
  private String firstname;

  @Column(name = "last_name")
  private String lastname;

  // Геттеры и сеттеры
  // ...

  public Person() {
  }
}
```

Для хранения объектов приведенного выше класса должна быть создана таблица вида:

```sql
CREATE TABLE persons (id bigserial primary key, first_name varchar(255),
last_name varchar(255));
```

Когда класс снабжен аннотацией **@Entity**, все его поля по умолчанию будут отображаться в столбцы ассоциируемой таблицы. Если нет необходимости в отображении какого-либо поля, необходимо пометить его аннотацией **@Transient**.

Большинство аннотаций из данного листинга интуитивно понятны, но стоит подробнее рассмотреть **@GeneratedValue**.

Важно отметить, что объект становится объектом-сущностью только после сохранения в базе данных. Кроме того, любой объект-сущность должен иметь свой уникальный идентификатор. Но разработчику совсем не обязательно заботиться об этом. Как правило, это значение можно (и желательно) получить из самой базы данных. Большинство БД предоставляют собственные механизмы генерации значений id. Значение может инжектироваться в поле id объекта-сущности после его сохранения. Для этого необходимо использовать аннотацию **@GeneratedValue**. В зависимости от механизма генерации значений id в базе данных, атрибуту **strategy** аннотации **@GeneratedValue** присваиваются различные значения из перечисления **GenerationType**.

Атрибут **strategy** может иметь следующие значения:

- **GenerationType.SEQUENCE** — говорит о том, что значение id будет генерироваться с помощью sequence-генератора, созданного разработчиком в базе данных. При использовании данной стратегии необходимо дополнительно указывать имя генератора в атрибуте **name** аннотации **@GeneratedValue**;
- **GenerationType.IDENTITY** — указывает поставщику постоянства, что значение id необходимо получать непосредственно из столбца «id» таблицы, в которую мэппится данный объект-сущность;

- **GenerationType.AUTO** — предоставлялет Hibernate возможность самостоятельно выбрать стратегию для получения id, исходя из используемой СУБД;
- **GenerationType.TABLE** — говорит о том, что для получения значения id необходимо использовать определенную таблицу в БД, содержащую набор чисел.

Оптимальный подход — использование **GenerationType.IDENTITY**. Аннотация говорит Hibernate о том, что после сохранения объекта в базе данных необходимо получить значение из столбца, на который отображается атрибут id, и присвоить его объекту-сущности. А каким образом в этом столбце появится значение после вставки строки с информацией об объекте, остается на совести разработчика. Данный подход удобен при использовании СУБД PostgreSQL, в которой такому столбцу можно задать тип **serial** — и СУБД будет автоматически генерировать значения для данного столбца после вставки строки.

Контекст постоянства

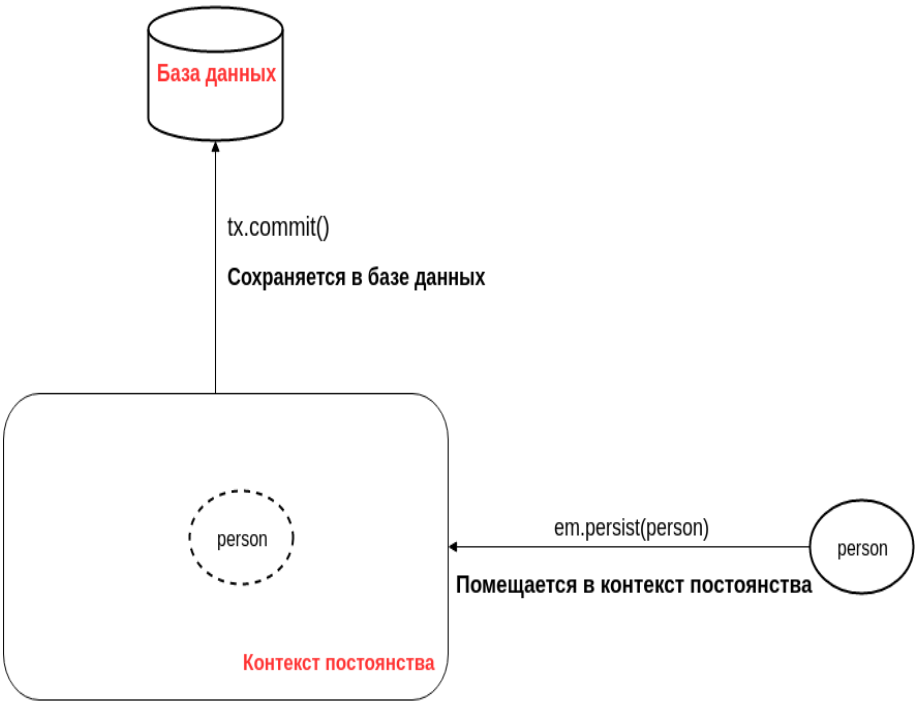
Рассмотрим порядок взаимодействия наших сущностей с базой данных. Чтобы понять этот процесс, необходимо разобраться с понятием контекста постоянства. Этот контекст представляет собой набор сущностей, которые управляются Hibernate в данный момент. Все управление сущностями возлагается на менеджер сущностей — **EntityManager**, или **Session** в случае Hibernate.

Он обладает полным набором CRUD-операций. Данные операции вызываются следующими методами:

```
public static void main(String[] args) {
    // Получаем фабрику менеджеров сущностей
    EntityManagerFactory factory = new Configuration()
            .configure("hibernate.cfg.xml")
            .buildSessionFactory();
    // Из фабрики создаем EntityManager
    EntityManager em = factory.createEntityManager();

    Person person = new Person("Ivan", "Ivanov");

    // Открываем транзакцию
    em.getTransaction().begin();
    // Create (сохраняем в базе данных, и благодаря этому сущность
    // становится управляемой Hibernate и заносится в контекст постоянства)
    em.persist(person);
    // Подтверждаем транзакцию
    em.getTransaction().commit();

    em.getTransaction().begin();
    // Read (читаем сущность из базы данных по id)
    Person anotherPerson = em.find(Person.class, 1L);
    em.getTransaction().commit();
    anotherPerson.setFirstname("Artem");

    em.getTransaction().begin();
    // Update
    em.merge(anotherPerson);
    em.getTransaction().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.

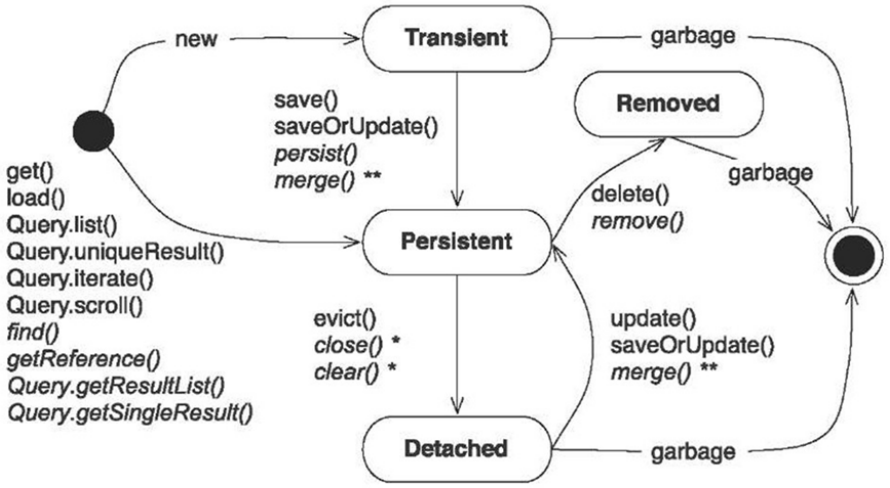
Данная последовательность характерна не только для метода **persist**, но и для остальных. В случае использования Hibernate совместно со Spring не будет необходимости самостоятельно открывать и закрывать транзакции — этим будет заниматься сам Spring.

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 находиться 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.

Жизненный цикл JPA-сущностей

Сущности в JPA могут находиться в нескольких возможных состояниях переход между которыми определяется жизненным циклом.

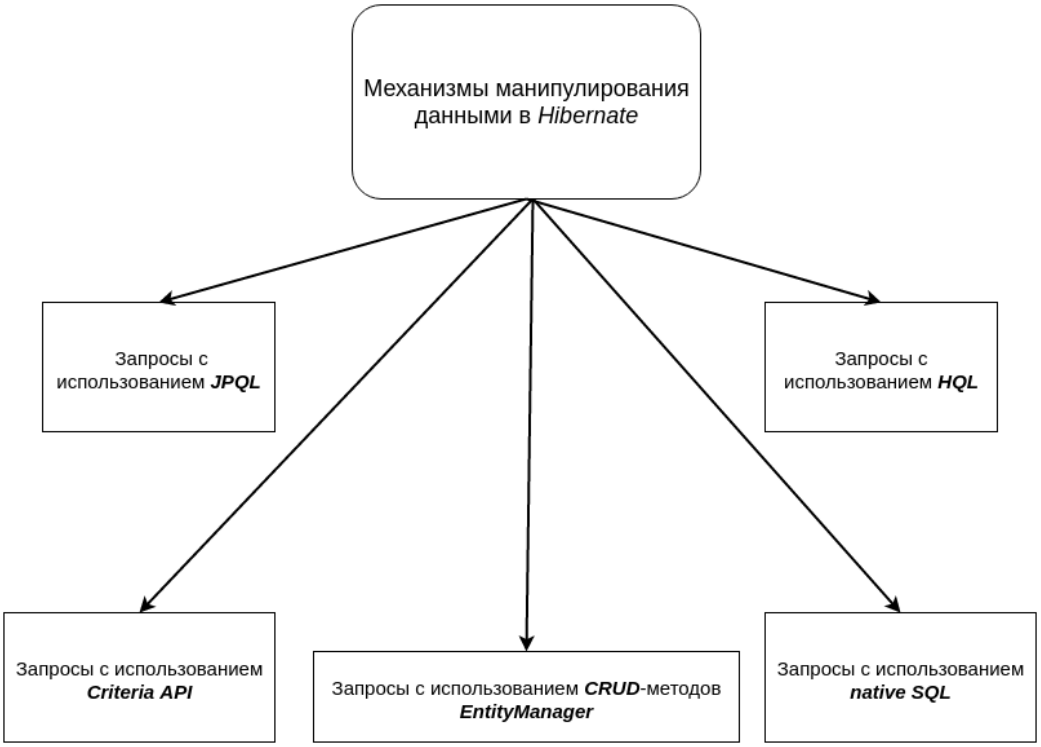


После создания объекта сущности оператором new сущность находится в статусе «Transient», на этом этапе это обычный класс никак не связанный с JPA. Для того чтобы сущность попала под управление «EntityManager», нужно вызывать метод «persist()». После этого класс сущности переходит в состояние «Persistent». Следует отметить, что при вызове метода persist() JPA предполагает, что сущности с таким идентификатором не существует. Под идентификатором понимается поле с аннотацией @Id из класса сущности.

Вывод сущности из управления происходит в тот момент, когда ей присваивается статус «detached», это происходит в момент выполнения метода «commit()» объекта «EntityTransaction», или метода «close()» объекта «EntityManager». При использовании аннотаций для управления транзакциями эти действия происходят автоматически.

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, который основан на HQL и позиционируется как более новая и стандартизованная версия этого языка.

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:

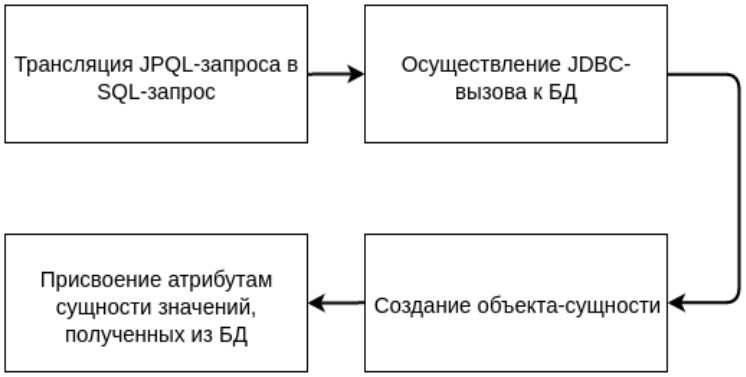
```
SELECT a FROM Article a
```

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",
Author.class).getResultList();

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

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:

- **Author.findAll** для получения всех авторов;
- **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();
```

Практическое задание

1. Создайте сущность Product (Long id, String title, int price) и таблицу в базе данных для хранения объектов этой сущности;
2. Создайте класс ProductDao и реализуйте в нем логику выполнения CRUD-операций над сущностью Product (Product findById(Long id), List<Product> findAll(), void deleteById(Long id), Product saveOrUpdate(Product product));
3. * Вшить ProductDao в веб-проект, и показывать товары, лежащие в базе данных. Помните что в таком случае SessionFactory или обертку над ней надо будет делать в виде Spring бина.

Дополнительные материалы

1. https://hibernate.org/orm/documentation/5.4/
2. **GEEKCHANGE: "Java. Изучаем Hibernate ORM для работы с базами данных"** https://www.youtube.com/watch?v=emg94BI2Jao (Введение в Hibernate)

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://hibernate.org/orm/documentation/5.4/