**Контрольные вопросы(лаб 7)**

**1. Что такое транзакция?**

*Транзакция* — минимальная логически осмысленная операция, которая имеет смысл и может быть совершена только полностью.

*Транзакция* (информатика) — группа логически объединённых последовательных операций по работе с данными, обрабатываемая или отменяемая целиком.

**2. В чем её отличие JPA от Hibernate?**

JPA - спецификация, Hibernate - реализация JPA + свои особенности.

Hibernate одна из самых популярных открытых реализаций последней версии спецификации (JPA 2.1). Даже скорее самая популярная, почти стандарт де-факто. То есть JPA только описывает правила и API, а Hibernate реализует эти описания, впрочем у Hibernate (как и у многих других реализаций JPA) есть дополнительные возможности, не описанные в JPA (и не переносимые на другие реализации JPA).

**3. Что такое Entity?**

Entity это легковесный хранимый объект бизнес логики (persistent domain object). Основная программная сущность это entity класс, который так же может использовать дополнительные классы, который могут использоваться как вспомогательные классы или для сохранения состояния еntity.

**4. Конфигурационный файл Hibernate?**

Hibernate требует заранее знать – где найти информацию о сопоставлении, которая определяет, как ваши классы Java связаны с таблицами базы данных. Hibernate также требует набора параметров конфигурации, связанных с базой данных и других связанных параметров. Вся такая информация обычно предоставляется в виде стандартного файла свойств Java с именем hibernate.properties или в виде файла XML с именем hibernate.cfg.xml.

Ниже приведен список важных свойств, которые можно будет настроить для баз данных в автономной ситуации:

*hibernate.dialect* - Это свойство заставляет Hibernate генерировать соответствующий SQL для выбранной базы данных

*hibernate.connection.driver\_class* - Класс драйвера JDBC.

*hibernate.connection.url* - URL JDBC для экземпляра базы данных.

*hibernate.connection.username* - Имя пользователя базы данных.

*hibernate.connection.password* - Пароль базы данных.

*hibernate.connection.pool\_size* - Ограничивает количество соединений, ожидающих в пуле соединений базы данных Hibernate.

*hibernate.connection.autocommit* - Позволяет использовать режим автоматической фиксации для соединения JDBC.

**5. Аннотация @Autowired?**

Аннотация @Autowired отмечает конструктор, поле или метод как требующий автозаполнения инъекцией зависимости Spring. Данная аннотация впервые появилась в Spring 2.5.

Используя эту аннотацию, не нужно заботиться о том, как лучше всего передать классу или bean'у экземпляр другого bean'a. Фреймворк Spring сам найдет нужный bean и подставит его значение в свойство, которое отмечено аннотацией @Autowired. (Бин — создаваемый Spring-ом объект класса, который можно внедрить в качестве значения поля в другой объект.)

Свойства класса с аннотацией @Autowired заполняются соответствующими значениями сразу после создания bean'а и перед тем, как любой из методов класса будет вызван.

@Autowired

private FieldService propertyService;

Традиционное использование аннотации.

@Autowired

public void setSetterService(SetterService setterService) {

this.setterService = setterService;

}

Только один конструктор может выполнять эту аннотацию. Этот конструктор может быть любого типа (private, protected), а не только pubic.

@Autowired

public BeanContainer(ConstructorService constructorService) {

this.constructorService = constructorService;

}

Аннотация @Autowire может быть использована в методе с любым именем и с любым количеством принимаемых параметров. В этом случае Spring попытается присвоить каждому(!) аргументу значение соответствующих bean'а. Метод не обязан быть public. Например:

@Autowired

public void multipleArguments(SetterAService setterAService,

SetterBService setterBService) {

this.setterAService = setterAService;

this.setterBService = setterBService;

}

С помощью аннотации @Qualifier можно отметить конкретного кандидата для автозаполнения если кандидатов несколько.

**6. Что такое JPQL (Java Persistence query language)?**

*Java Persistence Query Language (JPQL)* — платформенно-независимый объектно-ориентированный язык запросов, являющийся частью спецификации Java Persistence API (JPA).

JPQL используется для написания запросов к сущностям, хранящимся в реляционной базе данных. JPQL во многом похож на SQL, но в отличие от последнего, оперирует запросами, составленными по отношению к сущностям JPA, в отличие от прямых запросов к таблицам базы данных.

В дополнение к получению объектов (SELECT-запросы), JPQL поддерживает запросы, основанные на операторах UPDATE и DELETE.

JPQL может использоваться в NamedQuery (через аннотации или XML) или в динамических запросах, используя EntityManager createQuery() API.

JPQL (Java Persistence query language) это язык запросов, практически такой же как SQL, однако вместо имен и колонок таблиц базы данных, он использует имена классов Entity и их атрибуты. В качестве параметров запросов так же используются типы данных атрибутов Entity, а не полей баз данных. В отличии от SQL в JPQL есть автоматический полиморфизм (см. следующий вопрос). Также в JPQL используется функции которых нет в SQL: такие как KEY (ключ Map’ы), VALUE (значение Map’ы), TREAT (для приведение суперкласса к его объекту-наследнику, downcasting), ENTRY и т.п.

**7. Интерфейс JpaRepository?**

*JpaRepository* – это интерфейс фреймворка Spring Data предоставляющий набор стандартных методов JPA для работы с БД.

**8. Аннотация @Id, @GeneratedValue?**

Аннотация @GeneratedValue для JPA. В JPA значение примечания @GeneratedValue главным образом заключается в создании уникально идентифицированного первичного ключа для объекта (JPA требует, чтобы каждый объект должен иметь один и только один первичный ключ). @GeneratedValue обеспечивает генерацию первичного ключа. стратегия. Аннотация @GeneratedValue имеет два атрибута, а именно, стратегию и генератор, где значением атрибута генератора является строка со значением по умолчанию "", которое объявляет имя генератора первичного ключа (соответствует генераторам первичного ключа с тем же именем @SequenceGenerator и @TableGenerator ).

JPA предоставляет разработчикам четыре стратегии генерации первичных ключей, которые определены в классе перечисления GenerationType, включая GenerationType.TABLE, GenerationType.SEQUENCE, GenerationType.IDENTITY и GenerationType.AUTO.

С помощью аннотации @Id мы указываем первичный ключ (Primary Key) данного класса.

**9. Как в проекте Maven подключить JPA?**

Создаем maven проект и добавляем следующую зависимость:

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>JPAHiberanateExample</groupId>

<artifactId>JPAHiberanateExample</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.25</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-entitymanager</artifactId>

<version>4.2.1.Final</version>

</dependency>

<dependency>

<groupId>org.hibernate.javax.persistence</groupId>

<artifactId>hibernate-jpa-2.0-api</artifactId>

<version>1.0.1.Final</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

</dependency>

</dependencies>

</project>

Одной из крутых возможностей JPA – это создание таблиц в базе данных на основе существующий классов сущностей.

Entity – это сущность какого-то объекта, который можно описать некоторыми атрибутами.

В качестве предметной области мы возьмем магазин по продаже автомобилей.

Первая сущность – это Автомобиль:

package com.devcolibri.entity;

import java.util.Date;

public class Car {

private String name; //Название авто

private Date releaseDate; //дата выпуска

private float cost; //стоимость

public Car(String name, Date releaseDate, float cost) {

this.name = name;

this.releaseDate = releaseDate;

this.cost = cost;

}

public Car() {

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Date getReleaseDate() {

return releaseDate;

}

public void setReleaseDate(Date releaseDate) {

this.releaseDate = releaseDate;

}

public float getCost() {

return cost;

}

public void setCost(float cost) {

this.cost = cost;

}

@Override

public String toString() {

return "Car{" +

"name='" + name + '\'' +

", releaseDate=" + releaseDate +

", cost=" + cost +

'}';

}

}

В сущности Car мы переопределили метод toString(), чтобы выводя объект в консоль мы видели его в человеческом виде.

Вторая сущность – это продавец:

package com.devcolibri.entity;

public class Seller {

private String fullName; // Полное имя

private int age; // Возвраст

private float salary; // Зар. плата

public Seller(String fullName, int age, float salary) {

this.fullName = fullName;

this.age = age;

this.salary = salary;

}

public Seller() {

}

public String getFullName() {

return fullName;

}

public void setFullName(String fullName) {

this.fullName = fullName;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public float getSalary() {

return salary;

}

public void setSalary(float salary) {

this.salary = salary;

}

@Override

public String toString() {

return "Seller{" +

"fullName='" + fullName + '\'' +

", age=" + age +

", salary=" + salary +

'}';

}

}

Создаем БД.

CREATE SCHEMA `carshop` DEFAULT CHARACTER SET utf8 COLLATE utf8\_general\_ci;

Теперь нужно проаннотировать наши сущности я на примере Car класса.

package com.devcolibri.entity;

import javax.persistence.\*;

import java.util.Date;

@Entity

@Table(name = "cars")

@NamedQuery(name = "Car.getAll", query = "SELECT c from Car c")

public class Car {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private long id;

@Column(name = "name", length = 32)

private String name; //Название авто

@Column(name = "date")

@Temporal(TemporalType.TIMESTAMP)

private Date releaseDate; //дата выпуска

@Column(name = "cost")

private float cost; //стоимость

public Car(String name, Date releaseDate, float cost) {

this.name = name;

this.releaseDate = releaseDate;

this.cost = cost;

}

public Car() {

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Date getReleaseDate() {

return releaseDate;

}

public void setReleaseDate(Date releaseDate) {

this.releaseDate = releaseDate;

}

public float getCost() {

return cost;

}

public void setCost(float cost) {

this.cost = cost;

}

public long getId() {

return id;

}

@Override

public String toString() {

return "Car{" +

"id=" + id +

", name='" + name + '\'' +

", releaseDate=" + releaseDate +

", cost=" + cost +

'}';

}

}

Теперь создадим конфигурационный файл для JPA называется он ‘src\main\resources\META-INF\persistence.xml‘:

<?xml version="1.0" encoding="UTF-8"?>

<persistence xmlns="http://java.sun.com/xml/ns/persistence"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/persistence

http://java.sun.com/xml/ns/persistence/persistence\_1\_0.xsd"

version="1.0">

<persistence-unit name="COLIBRI" transaction-type="RESOURCE\_LOCAL">

<provider>org.hibernate.ejb.HibernatePersistence</provider>

<properties>

<property name="hibernate.connection.driver\_class" value="com.mysql.jdbc.Driver"/>

<property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/carshop"/>

<property name="hibernate.connection.username" value="root"/>

<property name="hibernate.connection.password" value="root"/>

<property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>

<property name="hibernate.hbm2ddl.auto" value="update"/>

</properties>

</persistence-unit>

</persistence>

Создаем класс-сервис, который будет обеспечивать работу с БД **CarService.java**:

package com.devcolibri.crud;

import com.devcolibri.entity.Car;

import javax.persistence.EntityManager;

import javax.persistence.Persistence;

import javax.persistence.TypedQuery;

import java.util.List;

public class CarService {

public EntityManager em = Persistence.createEntityManagerFactory("COLIBRI").createEntityManager();

public Car add(Car car){

em.getTransaction().begin();

Car carFromDB = em.merge(car);

em.getTransaction().commit();

return carFromDB;

}

public void delete(long id){

em.getTransaction().begin();

em.remove(get(id));

em.getTransaction.commit();

}

public Car get(long id){

return em.find(Car.class, id);

}

public void update(Car car){

em.getTransaction().begin();

em.merge(car);

em.getTransaction().commit();

}

public List<Car> getAll(){

TypedQuery<Car> namedQuery = em.createNamedQuery("Car.getAll", Car.class);

return namedQuery.getResultList();

}

}

Для класса **CarService** мы напишем следующий тестовый класс который проверит работоспособность методов:

package com.devcolibri.testing;

import com.devcolibri.crud.CarService;

import com.devcolibri.entity.Car;

import org.junit.After;

import org.junit.AfterClass;

import org.junit.Before;

import org.junit.Test;

import java.util.ArrayList;

import java.util.Date;

import java.util.List;

public class CarServiceTest {

CarService service = new CarService();

[@Test](https://devcolibri.com/%d0%bf%d0%be%d0%bb%d1%8c%d0%b7%d0%be%d0%b2%d0%b0%d1%82%d0%b5%d0%bb%d0%b8/test/)

public void testSaveRecord() throws Exception {

//Создаем автомобиль для записи в БД

Car car1 = new Car();

car1.setName("BMW");

car1.setCost(20000);

car1.setReleaseDate(new Date());

//Записали в БД

Car car = service.add(car1);

//Вывели записанную в БД запись

System.out.println(car);

}

[@Test](https://devcolibri.com/%d0%bf%d0%be%d0%bb%d1%8c%d0%b7%d0%be%d0%b2%d0%b0%d1%82%d0%b5%d0%bb%d0%b8/test/)

public void testDeleteRecord() throws Exception {

//Создаем автомобиль для записи в БД

Car car1 = new Car();

car1.setName("Ferrari");

car1.setCost(100000);

car1.setReleaseDate(new Date());

//Записуем его в БД

Car car = service.add(car1);

//Удвлем его с БД

service.delete(car.getId());

}

[@Test](https://devcolibri.com/%d0%bf%d0%be%d0%bb%d1%8c%d0%b7%d0%be%d0%b2%d0%b0%d1%82%d0%b5%d0%bb%d0%b8/test/)

public void testSelect() throws Exception {

//Создаем автомобиль для записи в БД

Car car1 = new Car();

car1.setName("Citroen‎");

car1.setCost(30000);

car1.setReleaseDate(new Date());

//Записываем в БД

Car car = service.add(car1);

//Получние с БД Citroen‎

Car carFromDB = service.get(car.getId());

System.out.println(carFromDB);

}

[@Test](https://devcolibri.com/%d0%bf%d0%be%d0%bb%d1%8c%d0%b7%d0%be%d0%b2%d0%b0%d1%82%d0%b5%d0%bb%d0%b8/test/)

public void testUpdate() throws Exception {

//Создаем автомобиль для записи в БД

Car car1 = new Car();

car1.setName("Lambordshini");

car1.setCost(5000000);

car1.setReleaseDate(new Date());

//Записываем в БД

car1 = service.add(car1);

car1.setCost(0);

//Обновляем

service.update(car1);

//Получаем обновленую запись

Car car2 = service.get(car1.getId());

System.out.println(car2);

}

public void testGetAll(){

//Создаем автомобиль для записи в БД

Car car1 = new Car();

car1.setName("Lexus");

car1.setCost(300000);

car1.setReleaseDate(new Date());

//Создаем автомобиль для записи в БД

Car car2 = new Car();

car2.setName("Fiat");

car2.setCost(20000);

car2.setReleaseDate(new Date());

//Создаем автомобиль для записи в БД

Car car3 = new Car();

car3.setName("Porsche");

car3.setCost(458000);

car3.setReleaseDate(new Date());

//Сохраняем все авто

service.add(car1);

service.add(car2);

service.add(car3);

//Получаем все авто с БД

List<Car> cars = service.getAll();

//Выводим полученый список авто

for(Car c : cars){

System.out.println(c);

}

}

}

**10. Методы JpaRepository?**

<S extends T> S save(S entity); – сохраняет переданную сущность.

Optional<T> findById(ID primaryKey); – возвращает сущность, идентифицированную данным ID.

Iterable<T> findAll(); – возвращает все сущности.

long count(); – возвращает количество сущностей.

void delete(T entity); – удаляет переданную сущность.

boolean existsById(ID primaryKey); – указывает, существует ли сущность с данным ID.

flush() – производит запись сохраненных данных непосредственно в БД.

findAll() – получить Список всех доступных сущностей в базе данных

findAll(…) – получить Список всех доступных сущностей и отсортировать их с помощью предоставленного условия

save(…) – s ave an Iterable сущностей. Здесь мы можем передать несколько объектов, чтобы сохранить их в пакете

flush() – f добавить все отложенные задачи в базу данных

saveAndFlush(…) – сохраните объект и немедленно сбросьте изменения

deleteInBatch(…) – удаление итеративного объекта. Здесь мы можем передать несколько объектов, чтобы удалить их в пакете