**Методические указания к выполнению лабораторных работ по дисциплине «Объектно-ориентированные базы данных»**

**Лабораторная работа 1**

Тема: Разработка объектно-ориентированной модели данных прикладной задачи.

Цель: Научиться разрабатывать объектно-ориентированную модель данных прикладной задачи.

Рассмотрим небольшой фрагмент задачи по ведению счета клиента банка: необходимо вести реестр клиентов, банковских счетов клиента (у каждого может быть несколько), операции по счету.

Выделим основные сущности:

Банк — содержит консолидированную информацию о клиентах, счетах, операциях.

Клиент — содержит персональные данные клиента, список принадлежащих ему счетов.

Счет — информация о номере счета, балансе счета.

Операция по счету — информация о счете, клиенте, времени транзакции, сумме по счету (отрицательная сумма означает списание средств).

Классы доступны по ссылке: https://github.com/ksenikeev/oodb/tree/master/src/main/java/ru/icmit/oodb/lab1

**Лабораторная работа 2**

Тема: JSON формат. Хранение и обработка объектов с использованием JSON формата.

JSON объект описывается в простом текстовом формате. Описание объекта расположено внутри пары фигурных скобок

{...}

Содержание описывается с помощью перечисления через запятую пар ключ=значение

{key1="value1", key2="value2"}

В качестве значения может выступать текст (заключается в двойные кавычки), число, вложенный объект, список (заключается в квадратные скобки [...]).

key="Text"

key="2019-10-15 12:23"

key=12345.67

key={key1="value1", key2="value2"}

key=["value1","value2"]

Пример работы с этим форматом доступен по ссылке:

https://github.com/ksenikeev/oodb/tree/master/src/main/java/ru/icmit/oodb/lab2

**Лабораторная работа 3**

Тема: XML формат. XSD схема. Хранение и обработка объектов с использованием XML формата.

XML (англ. eXtensible Markup Language) — расширяемый язык разметки, предназначенный для хранения и передачи данных.

С помощью XML описываются данные в виде структурированного документа. Документ состоит из элементов, каждый из которых начинается с открывающего тега и завершается закрывающим тегом

<tag\_name>**value**</tag\_name>

Между тегами указывается значение элемента. Каждый тег должен иметь закрывающую его пару.

Эемент может содержать атрибуты, которые перечисляются внутри открывающего тега:

<tag\_name atribute1="a\_value1" atribute2="a\_value1">**value**</tag\_name>

Значением элемента может выступать вложенный элемент.

<tag1>

<tag2>**value**</tag2>

</tag2>

Таким образом структура XML документа иерархическая, во главе иерархии находится корневой элемент.

При использовании вложенных структур закрывающие теги должны строго соответствовать порядку вложенности (т. е. Вложенный элемент должен закрыться раньше, чем закроется родительский элемент).

Пример XML документа, описывающего группу института:

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

<institut>

<name>**ИВМиИТ**</name>

<groupp>

<name>09-751</name>

<year>2017</year>

<students>

<student>Bulat</student>

<student>Karina</student>

<student>Marat</student>

</students>

</groupp>

</institut>

Можно и так:

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

<institut name="ИВМиИТ">

<groupp name="09-751" year="2017">

<students>

<student name="Bulat"/>

<student name="Karina"/>

<student name="Marat"/>

</students>

</groupp>

</institut>

Документы при этом не эквивалентны.

**Инструменты работы с XML**

Мы будем рассматривать JAXB (Java Architecture for XML Binding), технологию, позволяющую преобразовывать объект Java в XML документ, и обратно XML документ в объект Java.

В первом случае преобразование именуют «marshalling».

Обратное преобразование - «unmarshalling».

Преобразование объекта будем рассматривать на следующей структуре классов (в дополнение к описанию полей указаны аннотации, управляющие процессом преобразования):

**Bank.java**

**package** ru.icmit.oodb.lab3.model;

**import** javax.xml.bind.annotation.XmlElement;

**import** javax.xml.bind.annotation.XmlElementWrapper;

**import** javax.xml.bind.annotation.XmlRootElement;

**import** java.util.List;

@XmlRootElement(name = **"bank"**)

**public class** Bank {

**private** String **name**;

**private** List<Client> **clients**;

**private** List<Transaction> **transactions**;

**private** List<BankAccount> **bankAccounts**;

**public** String getName() {

**return name**;

}

**public void** setName(String name) {

**this**.**name** = name;

}

@XmlElementWrapper(name = **"clients"**)

@XmlElement(name = **"client"**)

**public** List<Client> getClients() {

**return clients**;

}

**public void** setClients(List<Client> clients) {

**this**.**clients** = clients;

}

@XmlElementWrapper(name = **"transactions"**)

@XmlElement(name = **"transaction"**)

**public** List<Transaction> getTransactions() {

**return transactions**;

}

**public void** setTransactions(List<Transaction> transactions) {

**this**.**transactions** = transactions;

}

@XmlElementWrapper(name = **"bankAccounts"**)

@XmlElement(name = **"bankAccount"**)

**public** List<BankAccount> getBankAccounts() {

**return bankAccounts**;

}

**public void** setBankAccounts(List<BankAccount> bankAccounts) {

**this**.**bankAccounts** = bankAccounts;

}

@Override

**public** String toString() {

**return "Bank: "** +

**"name='"** + **name** + **'\n'** +

**", clients="** + **clients** + **'\n'** +

**", transactions="** + **transactions** + **'\n'** +

**", bankAccounts="** + **bankAccounts**;

}

}

**Client.java**

**package** ru.icmit.oodb.lab3.model;

**import** javax.xml.bind.annotation.XmlElement;

**import** javax.xml.bind.annotation.XmlElementWrapper;

**import** java.util.List;

**public class** Client **extends** Person {

**public** Client(String firstName, String lastName, String phoneNumber, String email) {

**super**(firstName, lastName, phoneNumber, email);

}

**public** Client() {

}

**private** List<BankAccount> **accounts**;

@XmlElementWrapper(name = **"accounts"**)

@XmlElement(name = **"account"**)

**public** List<BankAccount> getAccounts() {

**return accounts**;

}

**public void** setAccounts(List<BankAccount> accounts) {

**this**.**accounts** = accounts;

}

@Override

**public** String toString() {

**return "Client: "** + getLastName() +

**", accounts="** + **accounts** +

**'\n'**;

}

}

**Person.java**

**package** ru.icmit.oodb.lab3.model;

**public class** Person {

**private** String **firstName**;

**private** String **lastName**;

**private** String **phoneNumber**;

**private** String **email**;

...

}

**Transaction.java**

**package** ru.icmit.oodb.lab3.model;

**import** java.util.Date;

**public class** Transaction {

**private** BankAccount **bancAccount**;

**private** Date **transactionDate**;

**private** Client **client**;

**private double amount**;

...

}

**BankAccount.java**

**package** ru.icmit.oodb.lab3.model;

**import** javax.xml.bind.annotation.XmlElement;

**public class** BankAccount {

**private long accountNumber**;

**private double balance**;

**public** BankAccount(**long** accountNumber) {

**this**.**accountNumber** = accountNumber;

**this**.**balance** = 0;

}

**public** BankAccount() {

}

@XmlElement(name = **"accountNumber"**)

**long** getAccountNumber() {

**return accountNumber**;

}

@XmlElement(name = **"balance"**)

**double** getBalance() {

**return balance**;

}

**public void** setAccountNumber(**long** accountNumber) {

**this**.**accountNumber** = accountNumber;

}

**public void** setBalance(**double** balance) {

**this**.**balance** = balance;

}

**protected void** increaseBalance(**double** cash) {

**this**.**balance** += cash;

}

**protected boolean** reduceBalance(**double** cash) {

**if** (**balance** < cash) {

**return false**;

} **else** {

**this**.**balance** -= cash;

**return true**;

}

}

@Override

**public** String toString() {

**return "BankAccount:"** +

**"accountNumber="** + **accountNumber** +

**", balance="** + **balance** +

**'\n'**;

}

}

Перечисленные выше классы описывают предметную область «Банк». Основной класс Bank содержит список клиентов (List<Client> **clients**), список расчетных счетов (List<BankAccount> **bankAccounts)**, список транзакций (List<Transaction> **transactions )**.

**Аннотации JAXB**

Наша задача — сохранить объект-Банк в XML файл bank.xml

Для указания корневого элемента XML документа мы используем аннотацию

@XmlRootElement(name = **"bank"**)

над определением класса Bank.

Для управляемого формирования элементов XML документа используют аннотации

@XmlElement(name = **"…"**), проставляемые над методом get...() соответствующего полю класса. Если аннотация не используется, то в качестве имени элемента будет использовано имя поля класса.

Списки удобно преобразовывать, используя элемент-обертку над набором одноименных элементов списка. В нашем случае мы хотим преобразовать в XML элемент поле List<Client> **clients**.

<clients>

<client>...</client>

<client>...</client>

...

<client>...</client>

</clients>

Такая структура формируется с помощью аннотаций:

@XmlElementWrapper(name = **"clients"**)

@XmlElement(name = **"client"**)

**public** List<Client> getClients() {

**return clients**;

}

Далее рассмотрим методы работы JAXB с подготовленными к преобразованию классами.

Основные методы преобразования описаны в классе Lab3XMLService

**package** ru.icmit.oodb.lab3;

**import** ru.icmit.oodb.lab3.model.Bank;

**import** javax.xml.bind.JAXBContext;

**import** javax.xml.bind.JAXBException;

**import** javax.xml.bind.Marshaller;

**import** javax.xml.bind.Unmarshaller;

**import** java.io.File;

**public class** Lab3XMLService {

*/\*\**

*\* метод saveBankData сохраняет объект типа Bank в XML документ*

*\*/*

**public static void** saveBankData(Bank bank) {

**try** {

JAXBContext context = JAXBContext.*newInstance*(Bank.**class**);

Marshaller marshaller = context.createMarshaller();

*// устанавливаем флаг для читабельного вывода XML в JAXB*

marshaller.setProperty(Marshaller.***JAXB\_FORMATTED\_OUTPUT***, Boolean.***TRUE***);

*// маршаллинг объекта в файл*

marshaller.marshal(bank, **new** File(**"bank.xml"**));

} **catch** (JAXBException e) {

e.printStackTrace();

}

}

*/\*\**

*\* Метод loadBankFromXML преобразует XML документ в объект типа Bank*

*\*/*

**public static** Bank loadBankFromXML() {

**try** {

*// создаем объект JAXBContext - точку входа для JAXB*

JAXBContext jaxbContext = JAXBContext.*newInstance*(Bank.**class**);

Unmarshaller un = jaxbContext.createUnmarshaller();

**return** (Bank) un.unmarshal(**new** File(**"bank.xml"**));

} **catch** (JAXBException e) {

e.printStackTrace();

}

**return null**;

}

}

**Следующий пример демонстрирует сохранения объекта в XML документ:**

**package** ru.icmit.oodb.lab3;

**import** ru.icmit.oodb.lab3.model.Bank;

**import** ru.icmit.oodb.lab3.model.BankAccount;

**import** ru.icmit.oodb.lab3.model.Client;

**import** java.util.ArrayList;

**import** java.util.List;

*/\*\**

*\* Пример, демонстрирующий сохранение (marshalling) объекта в XML документ*

*\*/*

**public class** Lab3MainSave {

**public static void** main(String[] args) {

*// Создаем объект банк*

Bank bank = **new** Bank();

*// Создаем объекты - клиенты банка*

Client client1 = **new** Client(**"Иван"**,**"Иванов"**,**""**,**""**);

Client client2 = **new** Client(**"Петр"**,**"Петров"**,**""**,**""**);

*// Добавляем объекту client1 расчетный счет*

BankAccount account = **new** BankAccount(1299876);

List<BankAccount> accounts = **new** ArrayList<>();

accounts.add(account);

client1.setAccounts(accounts);

List<Client> clients = **new** ArrayList<>();

clients.add(client1);

clients.add(client2);

*// Прикрепляем клиентов к банку*

bank.setClients(clients);

*// сохраняем объект в в XML документ*

Lab3XMLService.*saveBankData*(bank);

}

}

**Загрузка данных в объект из XML документа:**

**package** ru.icmit.oodb.lab3;

**import** ru.icmit.oodb.lab3.model.Bank;

**import** ru.icmit.oodb.lab3.model.BankAccount;

**import** ru.icmit.oodb.lab3.model.Client;

**import** java.util.ArrayList;

**import** java.util.List;

*/\*\**

*\* Пример, демонстрирующий преобразование XML документа в объект*

*\*/*

**public class** Lab3MainLoad {

**public static void** main(String[] args) {

Bank bank = Lab3XMLService.*loadBankFromXML*();

System.***out***.println(bank);

}

}

Примеры доступны по ссылке:

https://github.com/ksenikeev/oodb/tree/master/src/main/java/ru/icmit/oodb/lab3

# Java Architecture for XML Binding (JAXB),

<https://www.oracle.com/technical-resources/articles/javase/java-architecture-XML-binding.html>

**Лабораторная работа 4**

**Лабораторная работа 5**

Тема: пользовательские типы данных PostgreSQL

CREATE TYPE person AS (

firstname character varying (255),

lastname character varying (255),

middlename character varying (255),

gender char (1),

birthdate date

);

CREATE TYPE library\_reader AS (

reader person,

abonement\_number character varying (255),

recording\_date date

);

CREATE SEQUENCE library\_person\_seq;

CREATE TABLE library\_person (

id bigint default nextval('library\_person\_seq'),

client person,

CONSTRAINT library\_person\_pkey PRIMARY KEY (id)

);

CREATE SEQUENCE library\_client\_seq;

CREATE TABLE library\_client (

id bigint default nextval('library\_client\_seq'),

client library\_reader,

CONSTRAINT library\_client\_pkey PRIMARY KEY (id)

);

INSERT INTO library\_person (client) VALUES

(ROW('ИВАН','ИВАНОВ','ИВАНОВИЧ','М','2001-10-01'));

INSERT INTO library\_person (client) VALUES

(ROW('ПЕТР','ПЕТРОВ','ПЕТРОВИЧ','М','2000-02-11'));

INSERT INTO library\_person ( client.lastname, client.gender, client.firstname) VALUES

( 'ПОТАПОВА','Ж','ЖАННА');

UPDATE library\_person set client.birthdate = '1999-05-28' where

(client).lastname like 'ПЕТРОВ';

SELECT (client).firstname from library\_person;

<https://postgrespro.ru/docs/postgrespro/10/rowtypes>

**Лабораторная работа 6**

Тема: Анализ объектной модели данных.

**Лабораторная работа 7**

Тема: Принципы построения ORM.

Запрос к базе данных (PostgreSQL) извлекающий список таблиц:

SELECT

table\_name

FROM

information\_schema.tables

WHERE

table\_type = 'BASE TABLE'

AND

table\_schema NOT IN ('pg\_catalog', 'information\_schema');

Запрос к базе данных (PostgreSQL) извлекающий список полей указанной таблицы

SELECT a.attname

FROM pg\_catalog.pg\_attribute a

WHERE a.attrelid = (

SELECT c.oid FROM pg\_catalog.pg\_class c

LEFT JOIN pg\_catalog.pg\_namespace n ON n.oid = c.relnamespace

WHERE pg\_catalog.pg\_table\_is\_visible(c.oid) AND c.relname = '**TABLE\_NAME**'

)

AND a.attnum > 0 AND NOT a.attisdropped

**Лабораторная работа 8**

Тема: Управляющие конструкции ORM

**Лабораторная работа 9**

**Тема**: Принципы проектирования объектно-ориентированной БД с использованием спецификации JPA.

**Задание**: По имеющейся предметной области создать объектную модель данных с использованием требований спецификации JPA. Создать программу, генерирующую структуру реляционной БД в СУБД PostgreSQL.

Эта тема - начало описания решения проблемы долговременного хранения объектов (object persistence) в «промышленных» масштабах на базе технологии JPA.

Java Persistence API — спецификация (JSR 338: Java Persistence API, version 2.2), описывающая концепцию объектно-реляционного отображения (Object/Relational Mapping, ORM) на прикладном уровне. Спецификация описывает хранение информации, структурированной в объектной модели, в реляционной базе данных.

Такой способ хранения нельзя назвать естественным, но в силу большой распространенности и эффективности современных реляционных СУБД, концепция ORM преобладает над другими способами хранения объектно-ориентированных данных.

Имеет место и обратная ситуация — изначально представленные в реляционной модели сущности преобразуют в объекты для работы с ними на уровне реализации логики «бизнес»- процессов.

Мы будем исходить из определения данных в объектной модели. Реляционная модель будет строиться автоматически на основе указаний (удобнее давать их в виде аннотаций). Манипулирование данными будем осуществлять на уровне объектной модели. При необходимости оптимизации, будем рассматривать манипуляции данными на уровне реляционной модели.

Практическая работа будет осуществляться с помощью фреймворка Hibernate, реализующего спецификацию JPA.

**Проектирование объектной модели данных для работы с ORM**

Проектирование объектной модели данных для работы с ORM происходит с большой оглядкой на реляционную структуру. Для описания нюансов объектно-реляционного отображения мы будем использовать аннотации пакета javax.persistence.\*:

@Entity

@Table

@Id

@Column

@ManyToOne

@OneToOne

@OneToMany

и ряд других.

Опишем кратко назначение этих аннотаций.

@Entity — описание класса как сущности

@Id — описание члена класса как первичного ключа

@Column — детализирует свойства поля таблицы, представляющего член класса

@ManyToOne — описывает связь типа «многие-к-одному» посредством члена класса, типизированного связанным классом.

@OneToOne — описывает связь типа «один-к-одному» посредством члена класса, типизированного связанным классом.

@OneToMany — описывает связь типа «один-ко-многим» посредством списка значений, типизированного связанным классом.

**Пример объектно-ориентированной базы данных.**

Рассмотрим три связанных сущности Клиент — Счет — Транзакция.

class Account {

Client client;

String accountNumber;

}

class Client {

String name;

String passport;

String address;

}

class AccountTransaction {

Date transactionDate;

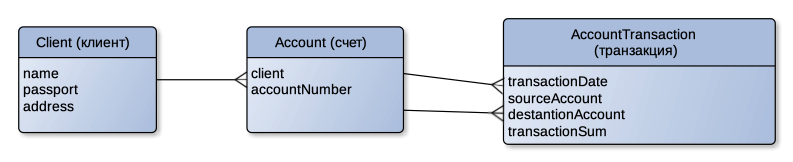
Account sourceAccount;

Account destantionAccount;

Double transationSum;

}

Диаграмма демонстрирует связи, впрочем достаточно очевидные из описания объектов:



Внесем дополнительные указания, которые позволят нам сохранять данные в реляционной БД (в приведенных ниже примерах опущены для краткости стандартные методы get... set...):

@Entity

public class Account {

@Id

private Long id;

@ManyToOne

private Client client;

@Column(length=20)

private String accountNumber;

...

}

@Entity

public class Client {

@Id

private Long id;

private String name;

private String passport;

private String address;

...

}

@Entity

public class AccountTransaction {

@Id

private Long id;

@Temporal(TemporalType.*TIMESTAMP*)

private Date transactionDate;

@ManyToOne

private Account sourceAccount;

@ManyToOne

private Account destantionAccount;

private Double transationSum;

...

}

Аннотация @Entity указывает JPA системе необходимость работы с классом как сущностью, представленной таблицей (по умолчанию имя таблицы совпадает с именем класса).

Несмотря на наличие потенциальных ключей в составе членов класса мы приняли решение использовать фиктивный целочисленный первичный ключ @Id private Long id; - такая унификация удобнее (но это совсем не обязательно, JPA позволяет использовать ключи других типов, в том числе и составные).

В классе AccountTransaction мы использовали аннотацию @Temporal над членом transactionDate, чтобы указать тип данных в реляционной БД для хранения значений этого поля.

Аннотация **@ManyToOne** формирует принцип работы на уровне БД с объектной ссылкой (описывающей связь многие к одному). Такая ссылка (представленная членом класса) трансформируется во внешний ключ. Например в классе Account имеется член client (@ManyToOne private Client client;). Тогда в соответствующей таблице account должно быть целочисленное поле client\_id.

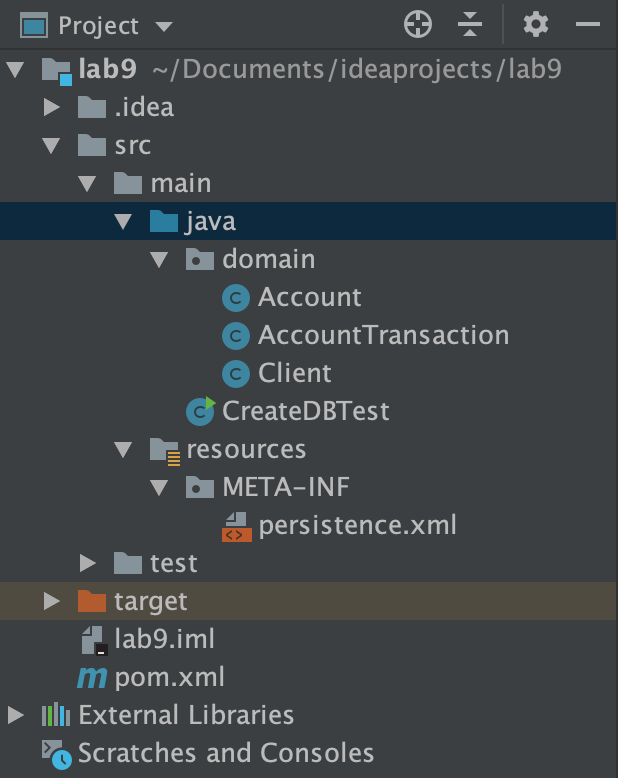
Связи типа «многие-ко-многим» и «один-ко-многим» в объектной модели могут быть представлены коллекциями:

@OneToMany

private Collection<Account> accounts;

**Рассмотрим программу, которая создаст базу данных по имеющемуся описанию объектной модели.**

Структура maven проекта представлена на рисунке ниже:



pom.xml:

<?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>ru.icmit.oodb.lab9</groupId>

<artifactId>lab9</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

<version>42.2.6.jre7</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-entitymanager</artifactId>

<version>5.4.3.Final</version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

Задание параметров JPA (таких как имя базы данных, пользователь, JDBC драйвер …) можно осуществить несколькими способами. Один из них — создать в папке META-INF конфигурационный файл persistence.xml

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

<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence

http://xmlns.jcp.org/xml/ns/persistence/persistence\_2\_2.xsd" version="2.2">

<persistence-unit name="lab9">

<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

<properties>

<property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver"/>

<property name="javax.persistence.jdbc.url" value="jdbc:postgresql://127.0.0.1:5432/lab9"/>

<property name="javax.persistence.jdbc.user" value="postgres"/>

<property name="javax.persistence.jdbc.password" value="post"/>

<property name="hibernate.show\_sql" value="true"/>

<property name="hibernate.hbm2ddl.auto" value="update"/> <!-- none, create -->

</properties>

</persistence-unit>

</persistence>

В этом файле важно правильно задать:

имя модуля ( параметр persistence-unit)

имя базы данных (в параметре javax.persistence.jdbc.url)

имя пользователя и пароль

режим автоматической генерации структуры БД (hibernate.hbm2ddl.auto); мы поставили update, что означает автоматическое изменение структуры при изменении объектной модели.

Сама программа:

import javax.persistence.EntityManagerFactory;

import javax.persistence.Persistence;

public class CreateDBTest {

public static void main(String[] args) {

EntityManagerFactory emf =

Persistence.*createEntityManagerFactory*("lab9");

emf.close();

}

}

Метод Persistence.*createEntityManagerFactory*("lab9") имеет значение аргумента — имя persistence модуля.

Перед выполнением программы необходимо создать БД (JDBC драйвер PostgreSQL не делает это автоматически)

CREATE DATABASE lab9;

Вывод в консоль результатов работы программы:

Hibernate: create table Account (id int8 not null, accountNumber varchar(20), client\_id int8, primary key (id))

Hibernate: create table AccountTransaction (id int8 not null, transactionDate timestamp, transationSum float8, destantionAccount\_id int8, sourceAccount\_id int8, primary key (id))

Hibernate: create table Client (id int8 not null, address varchar(255), name varchar(255), passport varchar(255), primary key (id))

Hibernate: alter table if exists Account add constraint FKpei4spdwejr1mbu98yngbojd8 foreign key (client\_id) references Client

Hibernate: alter table if exists AccountTransaction add constraint FKs23agqnpkek8o7ox75g9b6k69 foreign key (destantionAccount\_id) references Account

Hibernate: alter table if exists AccountTransaction add constraint FKe5dbw5uaryjr1d58h93710pyh foreign key (sourceAccount\_id) references Account

Здесь мы видим скрипты создания структуры БД.

**Лабораторная работа 10**

**Автоматическая генерация значений первичного ключа в JPA**

Напомню, в многопользовательской среде при использовании фиктивных первичных ключей (т. е. не естественных в отличии от таких как номер паспорта, ИНН, СНИЛС …) возникает проблема уникальности при генерации значений. Такие проблемы обычно разрешаются на стороне СУБД.

Наша цель — изучить связь между определениями объектов и механизмами СУБД генерации значений первичного ключа. Рассматриваем связку с СУБД PostgreSQL. В примерах структура БД будет создаваться автоматически по описанию объектной модели.

Член класса-сущности, выступающий в роли первичного ключа отмечается аннотацией @Id.

1. В «ручном режиме» генерации (назначаем id программно) никаких дополнительных средств определения не требуется.

2. При делегировании генерации значений первичного ключа в СУБД используем аннотацию @GenratedValue. Поведение определяется атрибутом strategy (со значением из enum GenerationType { AUTO, SEQUENCE, IDENTITY, TABLE }) AUTO — значение по умолчанию.

**AUTO:**

@Id

@GeneratedValue

private Long id;

При этом в БД описание поля id

id bigint NOT NULL PRIMARY KEY

Для генерации значений будет использоваться последовательность с именем

hibernate\_sequence

Причем эта последовательность будет использована *для всех таблиц и первичных ключей*, определенных таким образом.

**SEQUENCE:**

Дополнительно необходимо определить генератор @SequenceGenerator

@Id

@SequenceGenerator(name="tblgen", sequenceName="tbl\_seq", allocationSize=1)

@GeneratedValue(strategy=GenerationTypeю.SEQUENCE, generator="tblgen")

private Long id;

БД описание поля id

id bigint NOT NULL PRIMARY KEY

Для генерации значений будет использоваться последовательность с именем tbl\_seq

**IDENTITY**:

@Id

@GeneratedValue(strategy=GenerationTypeю.IDENTITY)

private Long id;

Для такого определения будет сформирована последовательность по правилу *имятаблицы\_id\_seq.* Эта последовательность будет использована для задания уникального значения первичного ключа. В БД определение таблицы примет вид

id bigint NOT NULL PRIMARY KEY DEFAULT nextval('*имятаблицы\_id\_seq'*);

3. Пример: в таблице person первичный ключ id генерируется последовательностью person\_seq

Тогда объект Person дополним следующим образом

**@Id**

**@GeneratedValue**(**strategy**=GenerationType.SEQUENCE, **generator**="personIdGenerator")

**@SequenceGenerator**(**name**="personIdGenerator", **sequenceName**="person\_seq", **allocationSize**=1)

**private** Long id;

**Лабораторная работа 13**

**Тема**: стратегии отображения в реляционной СУБД наследуемых классов.

Понятие наследования входит в базовые принципы объектно-ориентированного программирования. Естественным будет активно использовать этот принцип. Но при отображении сложного объекта, включенного в иерархию наследования, возникает ряд вопросов с его представлением в реляционной базе данных.

* При наследовании информация оказывается распределенной по нескольким объектам.
* Как строить таблицы в таком случае?
* Как восстанавливать потом иерархию классов из таблиц при запросах?

Спецификация JPA предлагает несколько стратегий для разрешения этих вопросов. Специальным образом размечая классы (мы будем делать это с помощью аннотаций), мы даем указание на определенное поведение системы ORM.

Рассмотрим разные стратегии на примере двух классов Person и Client:

class Person {

String name;

}

class Client extends Person {

String passport;

String address;

}

@MappedSuperClass

Проставляем аннотацию @MappedSuperClass над определение класса предка

@MappedSuperclass

public class **Person** {

@Id

private Long **id**;

String **name**;

…

}

Класс Person может быть абстрактным.

@Entity

public class **Client** extends Person{

private String **passport**;

private String **address**;

…

}

Таким образом определенная иерархия наследования будет отображаться на таблицу

CREATE TABLE **client**

(

**id** bigint NOT NULL,

**name** character varying(255),

**address** character varying(255),

**passport** character varying(255),

CONSTRAINT client\_pkey

PRIMARY KEY (id)

);

Мы видим, что все атрибуты объединены в одну таблицу client.

@Inheritance(strategy = *JOINED*)

Проставляем аннотацию @Inheritance(strategy = *JOINED*) над определение класса предка

@Entity

@Inheritance(strategy = *JOINED*)

public class **Person** {

@Id

private Long **id**;

String **name**;

…

}

Класс Person может быть абстрактным.

@Entity

public class **Client** extends Person{

private String **passport**;

private String **address**;

…

}

Классы отображаются на таблицы

CREATE TABLE person

(

**id** bigint NOT NULL,

**name** character varying(255),

CONSTRAINT person\_pkey

PRIMARY KEY (id)

);

CREATE TABLE client

(

**id** bigint NOT NULL,

**address** character varying(255),

**passport** character varying(255),

CONSTRAINT client\_pkey

PRIMARY KEY (id),

CONSTRAINT person\_fk

FOREIGN KEY (id)

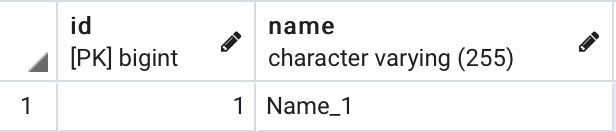
REFERENCES person (id) ...

);

Сохраняя класс Client, мы получаем одновременно записи в две таблицы с одинаковым значением первичного ключа:

**client**

**person**



Замечание: *такая стратегия формирует связь типа «один к одному».*

@Inheritance(strategy = *SINGLE\_TABLE*)

Эта стратегия предполагает отображение на одну таблицу с именем корневого класса предка

@Entity

@Inheritance(strategy =*SINGLE\_TABLE*)

public class **Person** {

@Id

private Long **id**;

String **name**;

…

}

@Entity

public class **Client** extends Person{

private String **passport**;

private String **address**;

…

}

Отображаемая таблица строится по имени корневого класса предка (Person в нашем случае)

CREATE TABLE **person**

(

**dtype** character varying(31)NOT NULL,

**id** bigint NOT NULL,

**name** character varying(255),

**address** character varying(255),

**passport** character varying(255),

CONSTRAINT person\_pkey PRIMARY KEY (id)

);

Появляется дополнительное поле **dtype** character varying(31) — хранит информацию об имени класса потомке (ПОМНИТЕ ПРО ОГРАНИЧЕНИЕ В 31 СИМВОЛ, ИМЕНУЯ КЛАССЫ!)

@Inheritance(strategy = *TABLE\_PER\_CLASS*)

Отдельная независимая таблица на класс, при этом класс-потомок отображается на таблицу, включающую поля с нижележащих иерархий.

В этом случае для каждого класса необходимо наличие поля с аннотацией @Id

@Entity

@Inheritance(strategy=*TABLE\_PER\_CLASS*)

public class **Person** {

@Id

private Long **id**;

String **name**;

…

}

@Entity

public class **Client** extends Person{

@Id

private Long **id**;

private String **passport**;

private String **address**;

…

}

CREATE TABLE person

(

**id** bigint NOT NULL,

**name** character varying(255),

CONSTRAINT person\_pkey PRIMARY KEY (id)

);

CREATE TABLE public.client

(

**id** bigint NOT NULL,

**name** character varying(255),

**address** character varying(255),

**passport** character varying(255),

CONSTRAINT client\_pkey PRIMARY KEY (id)

);

Выбор стратегии определяется логикой работы приложения и инструментарием, который обеспечивает эту логику.

**Лабораторная работа 14**

**Тема**: Связи сущностей (Один к одному, Один ко многим, Многие к одному, Многие ко многим)

Эту тему имеет смысл рассматривать с разных сторон проектирования: *прямого* (объектная модель → реляционная модель) и *обратного* (реляционная модель → объектная модель). Даже при прямом проектировании, соображения оптимизации хранения и обработки данных приводят к необходимости рассмотрения частного фрагмента модели с точки зрения обратного проектирования.

Рассмотрим поведение «по умолчанию».

При наличии в объекте атрибутов, являющимися в свою очередь сущностными объектами, мы имеем модель связанных сущностей, например:

|  |  |
| --- | --- |
| class Bank {  Long id;  String name;  } | class Client {  Long id;  String lastName;  **Bank** **bank**;  } |

или «зеркальный» пример:

|  |  |
| --- | --- |
| class Bank {  Long id;  String name;  List<**Client**> **clients**;  } | class Client {  Long id;  String lastName;  } |

Традиционно связи между сущностями описываются в терминах «Один к одному», «Один ко многим», «Многие к одному», «Многие ко многим».

1. Если идти от объекта сущности к атрибуту сущности (client → bank), то в первом случае мы имеем множество клиентов одного выделенного банка. Это пример связи «Многие к одному».

В этом случае для организации корректного отображения объекта на таблицы используем аннотацию

**@ManyToOne**

Как происходит отображение объектного атрибута на поле таблицы? Спецификация описывает правило по умолчанию: имяАтрибута\_имяПервичногоКлючаСвязаннойТаблицы,

обычно получается что-то типа **bank\_id**.

|  |  |  |
| --- | --- | --- |
| @Entity public class Bank {  @Id  @Column(name = "pk\_bank")  private Long id; | **→** | CREATE TABLE public.bank  (  **pk\_bank** bigint NOT NULL,  ... |
| @Entity public class Client {    @ManyToOne  private Bank bank; | **→** | CREATE TABLE public.client  (  id bigint NOT NULL,  **bank\_pk\_bank** bigint,  ... |

Если хотим задать имя поля в таблице принудительно, то используем дополнительную аннотацию

**@JoinColumn(name="new\_name")**

|  |  |  |
| --- | --- | --- |
| @JoinColumn(name="bank\_fk") @ManyToOne private Bank bank; | **→** | CREATE TABLE public.client  (  id bigint,  **bank\_fk** bigint ... |

2. Связь **«Один к одному»** часто возникает при нормализации реляционной базы данных. В объектных моделях она чаще приходит из наследования (этот случай мы разбирали), но встречается и как атрибутивная (на уровне однонаправленных или обоюдонаправленных ссылок посредством членов классов), например

|  |  |  |
| --- | --- | --- |
| @Entity public class Client {   @OneToOne  private PersonInfo personInfo; ... | **→** | @Entity public class PersonInfo {   private String phone;  private String email;  ... |

|  |  |  |
| --- | --- | --- |
| CREATE TABLE public.client  (  ...  personinfo\_id bigint, ... | **→** | CREATE TABLE public.personinfo  (  id bigint NOT NULL,  email character varying(255) ,  phone character varying(255) , ... |

Как видно из примера тип связи «Один к одному» задается аннотацией

**@OneToOne**

Как и в типе связи «Многие к одному» для именования поля в таблице, отличным от значения по умолчанию способом используем аннотацию

**@JoinColumn(name="new\_name")**

3. Связь **«Один ко многим»**. Возникает (или моделируется) при наличии в классе атрибута типа «множество» (Collection, Set, List, Map, …). Дальнейшее поведение определяется аннотацией

**@OneToMany**

|  |  |  |
| --- | --- | --- |
| @Entity public class **BankAccount** {  @Id  private Long id; … | **→** | CREATE TABLE bankaccount  (  id bigint NOT NULL,  ... |
| @Entity public class **Client** {  **@OneToMany** private  Collection<BankAccount> **accounts**; | **→** | CREATE TABLE client  (  id bigint NOT NULL,    ... |
|  |  | CREATE TABLE **client**\_**bankaccount**  (  client\_id bigint NOT NULL,  **accounts**\_id bigint NOT NULL,  CONSTRAINT uk\_1 UNIQUE accounts\_id),  CONSTRAINT fk1 FOREIGN KEY (client\_id) …  CONSTRAINT fk2 FOREIGN KEY (accounts\_id) … |

При необходимости специального именования связывающей таблицы, над коллекцией можно использовать аннотацию

**@JoinTable(name="table\_name")**

4. Связь **«Многие ко многим»**. Возникает (или моделируется) при наличии в классе атрибута типа «множество» (Collection, Set, List, Map, …). Дальнейшее поведение определяется аннотацией

**@ManyToMany**

Отличия между аннотациями @OneToMany и @ManyToMany проявляются на уровне ограничений реляционной БД (в первом случае добавляется ограничение уникальности, во втором — его нет).

|  |  |  |
| --- | --- | --- |
| @Entity public class **Address**  {  @Id  private Long id; … | **→** | CREATE TABLE address  (  id bigint NOT NULL,  ... |
| @Entity public class **Client** {  @ManyToMany private List<Address> **addresses**; | **→** | CREATE TABLE client  (  id bigint NOT NULL,    ... |
|  |  | CREATE TABLE **client\_address**  (  client\_id bigint NOT NULL,  **addresses\_id** bigint NOT NULL,  CONSTRAINT fk1 FOREIGN KEY (client\_id) …  CONSTRAINT fk2 FOREIGN KEY (addresses\_id) … |

При необходимости специального именования связывающей таблицы, над коллекцией можно использовать аннотацию

**@JoinTable(name="table\_name")**

Дополнительные опции

@JoinTable(name="client\_bankaccount",  
 [joinColumns=@JoinColumn](mailto:joinColumns%3D@JoinColumn)(name="ba\_id"),

inverseJoinColumns=@JoinColumn(name="accounts\_id"  
 *//,nullable=false,*  
 *//unique=true*  
)

Пример реализации связи Один-ко-многим посредством карты

@Entity

public class Item {

@Id int id;

...

@ElementCollection

@MapKeyColumn(name="IMAGE\_NAME")

@Column(name="IMAGE\_FILENAME")

@CollectionTable(name="IMAGE\_MAPPING")

Map<String, String> images; // map from image name to filename

...

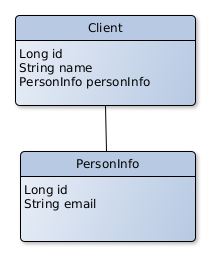
}

**Лабораторная работа 15**

**Тема**: Манипулирование данными в связанных сущностях. Каскадные операции.

При манипулировании с данными в связанных сущностях можно использовать два подхода

1. Сохранять (изменять, удалять) связанные объекты последовательно, начиная снизу



|  |  |  |
| --- | --- | --- |
| @Entity public class Client {   @OneToOne  private PersonInfo personInfo; ... | **→** | @Entity public class PersonInfo {   private String phone;  private String email;  ... |

Client client = new Client();  
client.setName("Клиент 1");  
  
PersonInfo personInfo = new PersonInfo();  
personInfo.setPhone("11111");  
personInfo.setEmail("email");  
  
client.setPersonInfo(personInfo);

entityManager.persist(personInfo);

entityManager.persist(client);

2. используя каскадные операции

|  |  |  |
| --- | --- | --- |
| @Entity public class Client {   @OneToOne(cascade = CascadeType.*ALL*)  private PersonInfo personInfo; ... | **→** | @Entity public class PersonInfo {   private String phone;  private String email;  ... |

Client client = new Client();  
client.setName("Клиент 1");  
  
PersonInfo personInfo = new PersonInfo();  
personInfo.setPhone("11111");  
personInfo.setEmail("email");  
  
client.setPersonInfo(personInfo);

// Здесь достаточно сохранить клиента, связанные объекты будут

// сохранены автоматически рекурсивно

entityManager.persist(client);

Аннотации @OneToOne, @ManyToOne, @OneToMany, @ManyToMany имеют атрибут cascade, который может принимать одно или несколько значений

CascadeType.*PERSIST*,

CascadeType.*MERGE*,  
CascadeType.*REFRESH*,

CascadeType.*REMOVE*,

CascadeType.*DETACH*,

CascadeType.*ALL*

Например

@ManyToOne(cascade={CascadeType.*PERSIST*, CascadeType.*MERGE*}).

Использование этих значений определяет каскадное поведение на различных операциях:

CascadeType.*PERSIST —* выполняется каскадное сохранение связанных объектов.

CascadeType.*MERGE* *—* выполняется каскадное обновление.  
CascadeType.*REFRESH —* выполняется каскадное присоединение сущности к менеджеру сущностей.

CascadeType.*REMOVE* *—* выполняется каскадное удаление.

CascadeType.*DETACH* *—* выполняется каскадное отсоединение объектов от менеджера сущностей.

CascadeType.*ALL—* все перечисленные выше операции выполняются каскадно.

**Лабораторная работа 16**

**Тема**: Управление менеджером сущностей.

**Лабораторная работа 17**

**Тема**: Управление транзакциями.

Напомним основные принципы вкладываемые в понятие *транзакции*, сформулированные в абревиатуре ACID: atomicity (атомарность), consistency (непротиворечивость), isolation (изолированность), durability (долговечность).

Мы будем различать транзакции уровня СУБД и транзакции уровня информационной системы (системные транзакции).

На уровне JDBC каждый SQL запрос по умолчанию выполняется в выделенной транзакции. Для помещения нескольких SQL инструкций в одну транзакцию необходимо отключить механизм «по умолчанию». Это породит начало общей транзакции:

connection.setAutoCommit(false);

…

завершаем транцакцию

connection.commit();

или откатываем изменения

connection.rollback();

Программное управление транзакциями

entityManager.getTransaction().begin();

…

entityManager.getTransaction().commit();

entityManager.getTransaction().rollback();

Декларативное управление транзакциями в SpringFramework

1) В конфигурации разрешаем использование менеджера транзакций

@EnableTransactionManagement

2) Настраиваем менеджер транзакций в классе-конфигурации

@Bean  
public PlatformTransactionManager transactionManager() {  
 JpaTransactionManager transactionManager = new JpaTransactionManager();  
 transactionManager.setEntityManagerFactory(entityManagerFactory());  
  
 return transactionManager;  
}

Методы, в которых задействованы операции с БД, изменяющие данные, аннотируем как

@Transactional

(аннотацию используем либо из пакета javax.transaction.Transactional

, либо org.springframework.transaction.annotation.Transactional)

Поведение транзакции определяется атрибутом value типа Transactional.TxType (в первом случае), и propagation типа Propagation (во втором).

Возможные значения:

REQUIRED (по умолчанию) — метод будет работать в транзакции (существующей, или вновь запущенной);

REQUIRES\_NEW - метод будет работать в транзакции, если транзакия уже задейсвована, то она завершается, и запускается новая;

MANDATORY - метод требует наличия активной транзакции, иначе будет выброшено исключение;

SUPPORTS - метод будет выполняться независимо от наличия действующей транзакции (используется в операциях выборки);

NOT\_SUPPORTED - указание на то, что метод не требует наличия транзакции (используется в операциях с оперативной памятью);

Использование аннотации из пакета org.springframework.transaction.annotation.Transactional

позволяет задать уровень изоляции:

@Transactional(isolation = Isolation.*DEFAULT*)

Возможные варианты:

Isolation.*READ\_UNCOMMITTED*

Isolation.*READ\_COMMITTED*

Isolation.*REPEATABLE\_READ*

isolation = Isolation.*SERIALIZABLE*

**Лабораторная работа 18**

**Тема**: Операции с использованием языка манипулирования объектами

Query Language

QL\_statement::=select\_statement|update\_statement|delete\_statement

Ключевые слова:

ABS, ALL, AND, ANY, AS, ASC, AVG, BETWEEN, BOTH, BY, CASE, CLASS, COALESCE, CONCAT, COUNT, CURRENT\_DATE, CURRENT\_TIME, CURRENT\_TIMESTAMP,

DELETE, DESC, DISTINCT, ELSE, EMPTY, END, ENTRY, ESCAPE, EXISTS, FALSE, FETCH,

FROM, FUNCTION, GROUP, HAVING, IN, INDEX, INNER, IS, JOIN, KEY, LEADING, LEFT,

LENGTH, LIKE, LOCATE, LOWER, MAX, MEMBER, MIN, MOD, NEW, NOT, NULL, NULLIF,

OBJECT, OF, ON, OR, ORDER, OUTER, SELECT, SET, SIZE, SOME, SQRT, SUB-

STRING, SUM, THEN, TRAILING, TREAT, TRIM, TRUE, TYPE, UPDATE, UPPER,

VALUE, WHEN, WHERE

Обновление части данных объекта(ов) по условию:

**update\_statement :: = update\_clause [where\_clause]**

**UPDATE Client c SET c.name = :name WHERE c = :c**

Удаление объекта(ов) по условию:

**delete\_statement :: = delete\_clause [where\_clause]**

**DELETE FROM Client c WHERE c = :c**

**DELETE FROM Client c WHERE c IN :list**

**DELETE FROM Client c WHERE** **c.bank IS EMPTY**

Выборка:

**select\_statement :: = select\_clause from\_clause [where\_clause] [groupby\_clause]**

**[having\_clause] [orderby\_clause]**

SELECT

FROM Client AS c JOIN c.accounts AS a

WHERE a.account = "11111";

from\_clause ::=

FROM identification\_variable\_declaration

{, {identification\_variable\_declaration | collection\_member\_declaration}}\*

identification\_variable\_declaration ::= range\_variable\_declaration { join | fetch\_join }\*

range\_variable\_declaration ::= entity\_name [ AS ] identification\_variable

join ::= join\_spec join\_association\_path\_expression [AS] identification\_variable

[join\_condition]

fetch\_join ::= join\_spec FETCH join\_association\_path\_expression

join\_association\_path\_expression ::=

join\_collection\_valued\_path\_expression |

join\_single\_valued\_path\_expression |

TREAT( join\_collection\_valued\_path\_expression AS subtype ) |

TREAT( join\_single\_valued\_path\_expression AS subtype )

join\_collection\_valued\_path\_expression::=

identification\_variable.{single\_valued\_embeddable\_object\_field.}\*collection\_valued\_field

join\_single\_valued\_path\_expression::=

identification\_variable.{single\_valued\_embeddable\_object\_field.}\*single\_valued\_object\_field

join\_spec ::= [ LEFT [ OUTER ] | INNER ] JOIN

join\_condition ::= ON conditional\_expression

collection\_member\_declaration ::=

IN ( collection\_valued\_path\_expression ) [AS] identification\_variable

Полезные ссылки:

<https://github.com/ksenikeev/oodb> — примеры программ к лабораторным работам

<https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html>

<https://codernet.ru/books/java/java_persistence_api_i_hibernate/> (только для ознакомления с оглавлением)

<https://download.oracle.com/otndocs/jcp/persistence-2_2-mrel-eval-spec/index.html>