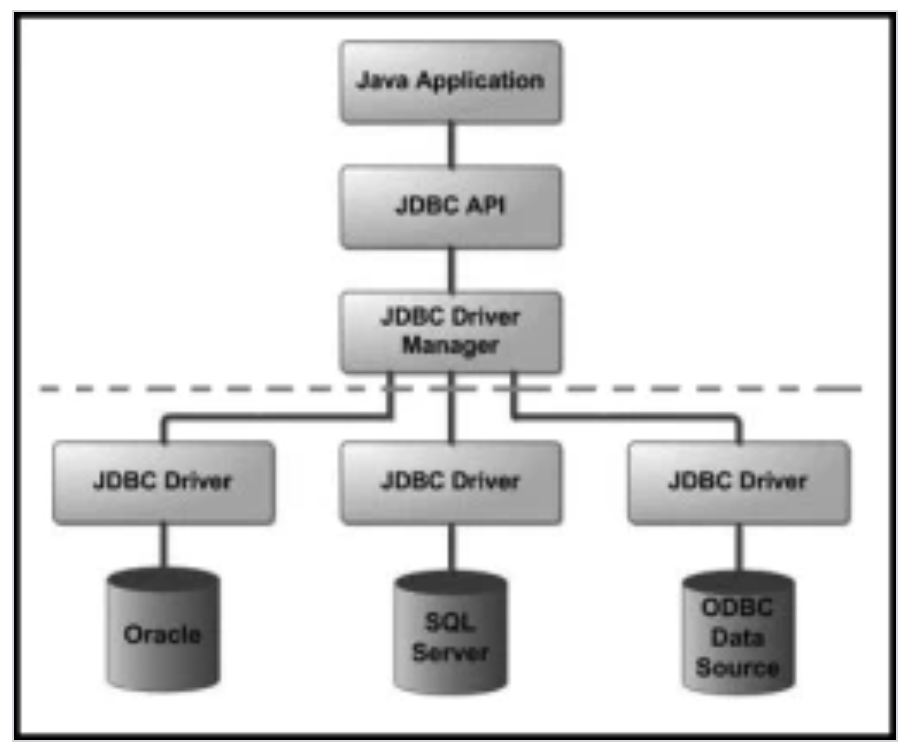
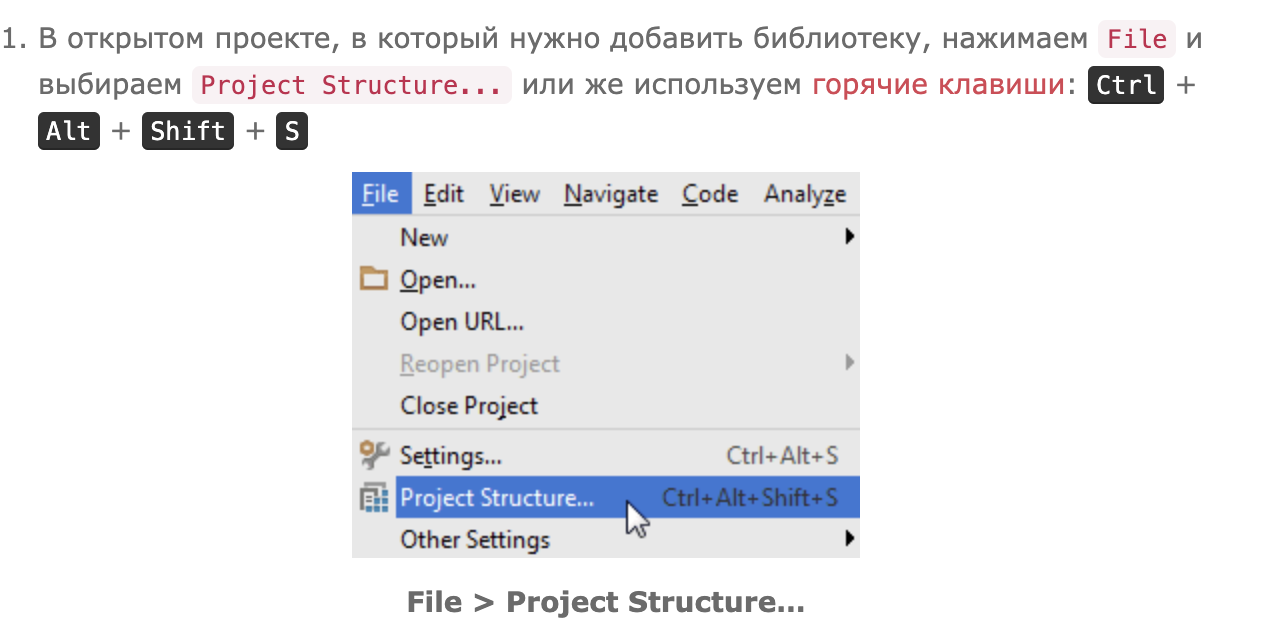
**JDBC**

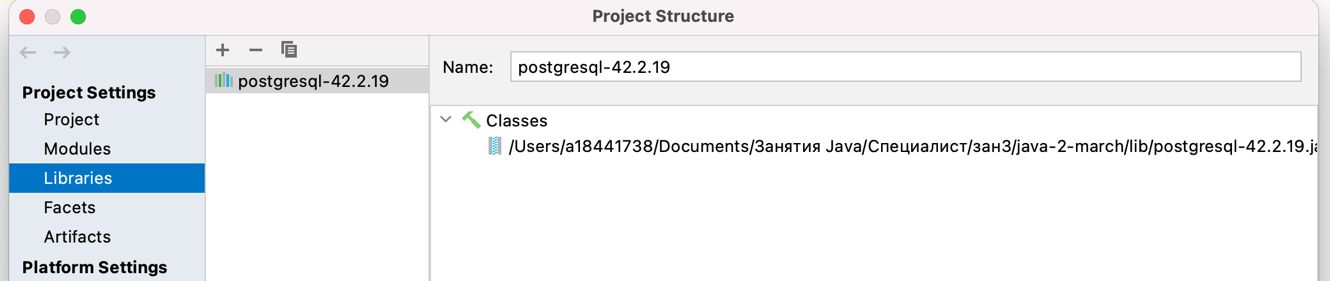
**JDBC API (Java DataBase Connectivity — соединение с базами данных на Java)** нужен для подключения к Базе данных. Он есть уже из коробки и на нем построены все фреймворки для работы с бд.

****

Однако, для каждой СУДБ нужен еще свой JBDC драйвер, который необходимо скачать либо с сайта разработчика СУДБ, либо с ресурса какого-нибудь тематического, либо как dependencies подгрузить и тогда он сам будет скачан. Если скачивать как драйвер, он будет jar – соответственно, его можно скопировать в папку lib (ее ). По факту это пакет с интерфейсами + спецификации, необходимыми для корректного сообщения с конкретной бд.



Там, соответственно, плюсик и ищем расположение модуля, добавляем



Если добавляли не из lib, то попадет в ExternalLibraries проекта.

В любом случае, проверить, подгрузилась ли библиотека, можно начав впечатывать класс из нее. Если выдаст, то все ок – можно работать.

Основные команды SQL (в идеале, будут в файле добавлены по SQL):

1. Создание таблиц

drop table if exists book;  
drop table if exists author;  
  
create table author(  
 author\_id serial primary key,  
 first\_name varchar(25),  
 last\_name varchar(25)  
);  
  
create table book(  
 book\_id serial primary key,  
 title text not null,  
 author\_id int REFERENCES author  
);  
  
insert into author (first\_name, last\_name) values ('Бьёрн', 'Страуструп');  
insert into author (first\_name, last\_name) values ('Брюс', 'Эккель');  
insert into author (first\_name, last\_name) values ('Блох', 'Джошуа');  
insert into author (first\_name, last\_name) values ('Дональд', 'Кнут');  
insert into author (first\_name, last\_name) values ('Роберт', 'Мартин');  
insert into author (first\_name, last\_name) values ('Джон', 'Скит');  
  
insert into book (title, author\_id) values ('Язык программирования C++', 1);  
insert into book (title, author\_id) values ('Java Concurrency на практике', 3);  
insert into book (title, author\_id) values ('Java. Эффективное программирование', 3);  
insert into book (title, author\_id) values ('Философия Java', 2);  
insert into book (title, author\_id) values ('Дизайн и эволюция языка C++', 1);  
insert into book (title, author\_id) values ('Искусство программирования', 4);  
insert into book (title, author\_id) values ('Философия C++. Введение в стандартный C++', 2);  
insert into book (title, author\_id) values ('C# для профессионалов', 6);  
insert into book (title, author\_id) values ('Чистый код. Создание, анализ и рефакторинг', 5);  
insert into book (title, author\_id) values ('Гибкая разработка программ на Java и C++. Принципы, паттерны и методики', 5);

2) Основные select ‘ы

- выводит колонки из таблицы по условию

select  
 author\_id, first\_name, last\_name  
from  
 author  
where  
 first\_name like 'Д%о%'  
order by author\_id;

- выводит всю таблицу  
  
select \* from author;

- выводит колонки обеих таблиц по условию  
  
select  
 \*  
from  
 author a, book b  
where  
 a.author\_id = b.author\_id;

- выводит колонки таблиц, которые связаны условием

select  
 \*  
from  
 author a  
 inner join book b  
 on a.author\_id = b.author\_id;

- выводит колонки таблиц, которые связаны условием – тут модный using

select  
 \*  
from  
 author a  
 inner join book b  
 using (author\_id);  
  
- тут из второй таблицы только одна колонка выведена будет (left join)  
select  
 a.\*, b.title  
from  
 author a  
 left join book b  
 using (author\_id);  
  
- distinct + ilike

select distinct  
 a.first\_name,  
 a.last\_name  
from  
 author a  
 inner join book b  
 using (author\_id)  
where  
 b.title ilike '%java%';

-выведет всего автора + еще создаст колонку с единицей  
select 1 from author;

3) Функции и процедуры

-создание функций (возвращают результат)

create or replace function inc(integer) returns integer  
language plpgsql  
as $$  
begin  
 return $1 + 1;  
end;  
$$  
  
create or replace function my\_sum(integer, integer) returns integer  
language plpgsql  
as $$  
begin  
 return $1 + $2;  
end;  
$$

-вызов функции  
  
select my\_sum(1,3);

-создание процедуры (не возвращает результат)  
create or replace procedure insert\_author (varchar, varchar)  
language plpgsql  
as $$  
begin  
 insert into author (first\_name, last\_name) values ($1, $2);  
end;  
$$

-вызов процедуры  
call insert\_author('323r2', '23r23r');  
  
  
create or replace procedure insert\_author\_and\_book (varchar, varchar, varchar)  
language plpgsql  
as $$  
declare  
 a\_id integer;  
begin  
 insert into author (first\_name, last\_name)  
 values ($1, $2)  
 returning author\_id into a\_id;  
 insert into book (title, author\_id) values ($3, a\_id);  
end;  
$$  
  
call insert\_author\_and\_book('323r2', '23r23r', '!!!');

Связи между таблицами

* Один к одному
* Один ко многим
* Многие ко многим

*Для обеспечения связи многие ко многим нужна Промежуточная таблица и Foreign Key (чтобы автоматизировать изменение эл-тов промежуточной таблицы при update или delete):*

drop table if exists orders CASCADE;  
 drop table if exists products CASCADE;  
 drop table if exists orders\_products ;  
  
 create table orders(  
 order\_id serial primary key,  
 created\_at timestamp (0)  
 );  
  
 create table products(  
 product\_id serial primary key,  
 product\_name varchar(25),  
 price int  
 );  
  
 create table orders\_products(  
 order\_id int REFERENCES orders (order\_id) ON DELETE CASCADE ON UPDATE CASCADE,  
 product\_id int REFERENCES products (product\_id) ON DELETE CASCADE ON UPDATE CASCADE  
 );

Работа с бд из IDE

Для настройки бд – нужно, чтобы она была запущена на сервере. Это можно сделать, запустив ее, используя свой комп как сервер или подключиться к удаленному серверу, на котором запущена бд (как мой амазон). Также, чтобы не проверять из консоли бд жедательно установить среду типа MySQL workbench или PG4Admin или SQL Workbench (это амазоновская всеядная).

Соответственно, также нужно создать подключение к БД из IDEA

Через jdbc это делается так:

Connection c = DriverManager.*getConnection*("jdbc:postgresql://rdsdatabase-x.cplmjbu0dc7d.us-east-2.rds.amazonaws.com:5432/mydbasex","postgres","postgres"); -

Делается через Try блок (или Try with resourses)

Можно создать файл .properties (к примеру, db.properties) и кинуть в корень проекта



- тогда логин и пароль будут взяты с него:

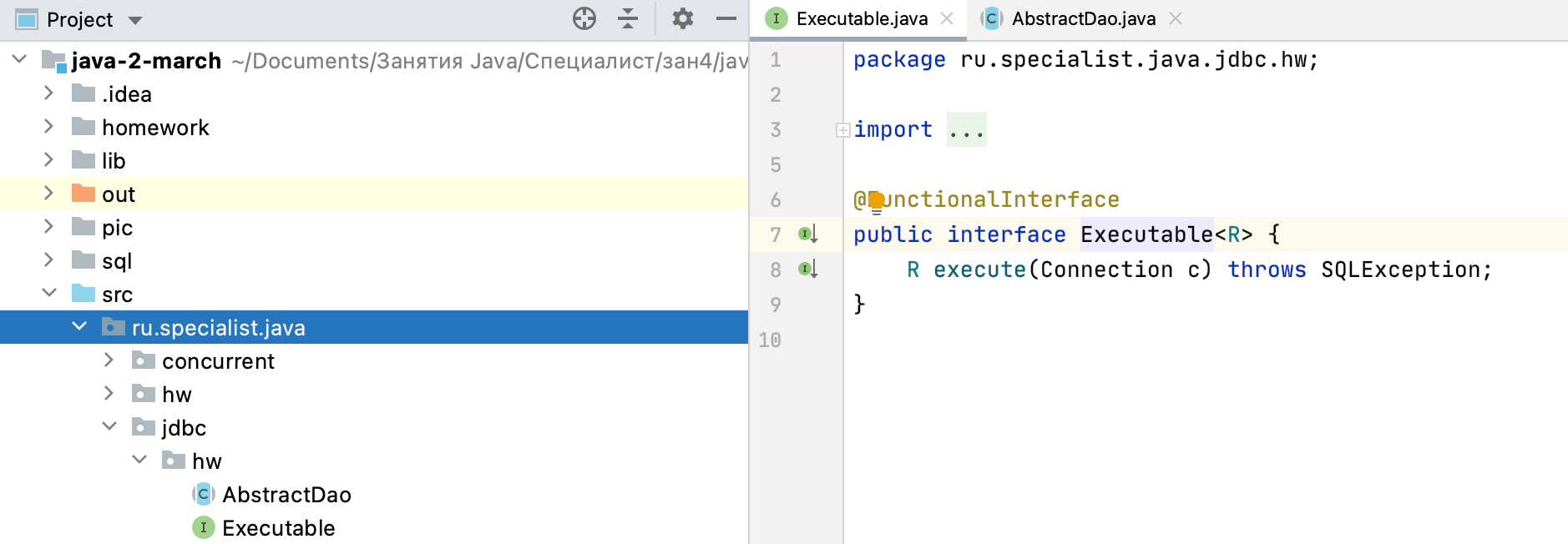
Properties prop = new Properties();  
prop.load(Files.*newInputStream*(Paths.*get*("db.properties")));

Connection c = DriverManager.*getConnection*(*URL*, prop));

+NB

Также можно создать отдельный интерфейс + абстрактный класс для создания подключения:

abstract public class AbstractDao {  
  
 private String url;  
 private String login;  
 private String password;  
  
 {  
 try (InputStream input = new FileInputStream("db-hw.properties")) {  
 Properties prop = new Properties();  
 prop.load(input);  
 url = prop.getProperty("db.url");  
 login = prop.getProperty("db.login");  
 password = prop.getProperty("db.password");  
 } catch (FileNotFoundException e) {  
 e.printStackTrace();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 protected <R> R execute(Executable<R> executable) {  
 try (Connection c = DriverManager.*getConnection*(url, login, password)) {  
 return executable.execute(c);  
 } catch (SQLException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
}



\_\_\_\_\_\_\_\_\_\_\_

Запросы из IDEA через JDBC:

1. Делаются через Statement statement = c.createStatement();

Результат выводится в специальный Set (ResultSet), из которого циклом можно вытаскивать данные:

public class JdbcExample {  
  
 private static final String *URL* = "jdbc:postgresql://localhost:5432/postgres";  
  
 public static void main(String[] args) throws ClassNotFoundException, IOException {  
*// Class.forName(Driver.class.getCanonicalName());* Properties prop = new Properties();  
 prop.load(Files.*newInputStream*(Paths.*get*("db.properties")));  
  
 try (Connection c = DriverManager.*getConnection*(*URL*, prop)){  
 Statement statement = c.createStatement();  
 ResultSet set = statement.executeQuery("select\n" +  
 "\ta.author\_id,\n" +  
 "\ta.first\_name, \n" +  
 "\ta.last\_name\n" +  
 "from \n" +  
 "\tauthor a");  
  
 while (set.next()){  
 System.*out*.printf("%d: %s %s%n",  
*// set.getInt("author\_id"),* set.getInt(1),  
 set.getString(2),  
 set.getString(3));  
 }  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 }  
}

1. Чтобы избежать SQL -инъекций (когда можно навредить данным в БД извне при подстановке хитрого значения в запрос), можно использовать форму, в которой запрос поставляется в statement не конкатенацией, а через спец символ - ?

+ так бытрее будет отрабатывать база, тк у нее есть спец память для занесения операций, которые уже были:

Если просто выполнить запрос, то statement.executeQuery();

public class JdbcStatement {  
  
 private static final String *URL* = "jdbc:postgresql://localhost:5432/postgres";  
  
 public static void main(String[] args) throws ClassNotFoundException, IOException {  
  
 Properties prop = new Properties();  
 prop.load(Files.*newInputStream*(Paths.*get*("db.properties")));  
  
 try (Scanner scanner = new Scanner(System.*in*);  
 Connection c = DriverManager.*getConnection*(*URL*, prop)){  
 String word = scanner.nextLine();  
  
 PreparedStatement statement = c.prepareStatement("select title from book where title ilike ?");  
 statement.setString(1, "%" + word + "%");  
  
 ResultSet set = statement.executeQuery();  
  
 List<String> books = new ArrayList<>();  
 while (set.next()){  
 books.add(set.getString(1));  
 }  
  
 System.*out*.println(books);  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 }  
}

1. А можно запрос сделать константой – типа Query +

Можно вытащить сгенерированные ключи (primary key autoincrement) - через

параметр Statement.*RETURN\_GENERATED\_KEYS* + statement.executeUpdate(); + ResultSet set = statement.getGeneratedKeys();

то есть, один сет, который зависит от Query тянет значения, а другой – сгенерированные ключи

Причем, если между ними будет другой запрос – statement, то его значения будут тянуться через его переменную:

private static final String *URL* = "jdbc:postgresql://localhost:5432/postgres";  
private static final String *QUERY* = "insert into author (first\_name, last\_name) values (?, ?)";  
  
 public static void main(String[] args) throws ClassNotFoundException, IOException {  
  
 Properties prop = new Properties();  
 prop.load(Files.*newInputStream*(Paths.*get*("db.properties")));  
  
 try (Scanner scanner = new Scanner(System.*in*);  
 Connection c = DriverManager.*getConnection*(*URL*, prop)){  
 System.*out*.println("Enter first name:");  
 String firstName = scanner.nextLine();  
  
 System.*out*.println("Enter last name:");  
 String lastName = scanner.nextLine();  
  
 PreparedStatement statement = c.prepareStatement(*QUERY*, Statement.*RETURN\_GENERATED\_KEYS*);  
 statement.setString(1, firstName);  
 statement.setString(2, lastName);  
  
 statement.executeUpdate();  
  
 Statement select = c.createStatement();  
 select.executeQuery("select \* from author");  
  
 ResultSet set = statement.getGeneratedKeys();  
 while (set.next()){  
 System.*out*.println("generated id = " + set.getString(1));  
 }  
  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 }  
}

1. Вызов процедуры. – через c.prepareCall

public class ProcedureExample {  
  
 private static final String *URL* = "jdbc:postgresql://localhost:5432/postgres";  
  
 public static void main(String[] args) throws ClassNotFoundException, IOException {  
  
 Properties prop = new Properties();  
 prop.load(Files.*newInputStream*(Paths.*get*("db.properties")));  
  
 try (Scanner scanner = new Scanner(System.*in*);  
 Connection c = DriverManager.*getConnection*(*URL*, prop)){  
  
 System.*out*.println("Enter first name:");  
 String firstName = scanner.nextLine();  
  
 System.*out*.println("Enter last name:");  
 String lastName = scanner.nextLine();  
  
 System.*out*.println("Enter book title:");  
 String title = scanner.nextLine();  
  
 CallableStatement statement = c.prepareCall("call insert\_author\_and\_book(?, ?, ?)");  
 statement.setString(1, firstName);  
 statement.setString(2, lastName);  
 statement.setString(3, title);  
  
 statement.execute();  
  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 }  
}

1. Вызов функции – тоже через c.prepareCall

Но ? ее ретурн тайпа не через setX а через statement.registerOutParameter(1, Types.*INTEGER*);

public class FunctionExample {  
  
 private static final String *URL* = "jdbc:postgresql://localhost:5432/postgres";  
  
 public static void main(String[] args) throws ClassNotFoundException, IOException {  
  
 Properties prop = new Properties();  
 prop.load(Files.*newInputStream*(Paths.*get*("db.properties")));  
  
 try (Scanner scanner = new Scanner(System.*in*);  
 Connection c = DriverManager.*getConnection*(*URL*, prop)){  
  
 int num1 = Integer.*parseInt*(scanner.nextLine());  
 int num2 = Integer.*parseInt*(scanner.nextLine());  
  
 CallableStatement statement = c.prepareCall("{? = call my\_sum(?, ?)}");  
 statement.registerOutParameter(1, Types.*INTEGER*);  
 statement.setInt(2, num1);  
 statement.setInt(3, num2);  
  
 statement.execute();  
  
 System.*out*.println(statement.getInt(1));  
  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 }  
}

1. SQL Транзакции – это операции, которые должны быть атомарными – то есть цепочка SQL- запросов, которые должны быть выполнены последовательно, и если какой-то из них вызывает ошибку, то остальные должны откатываться

Достигается  это коммитом:

public class JdbcTransaction {  
  
 private static final String *URL* = "jdbc:postgresql://localhost:5432/postgres";  
 private static final String *QUERY* = "insert into author (first\_name, last\_name) values (?, ?)";  
  
 public static void main(String[] args) throws ClassNotFoundException, IOException {  
  
 Properties prop = new Properties();  
 prop.load(Files.*newInputStream*(Paths.*get*("db.properties")));  
  
 try (Scanner scanner = new Scanner(System.*in*);  
 Connection c = DriverManager.*getConnection*(*URL*, prop)) {  
  
 c.setAutoCommit(false);  
  
 Statement statement = c.createStatement();  
 statement.execute("insert into author (first\_name, last\_name) values ('111', '111')");  
 statement.execute("insert into author (first\_name, last\_name) values ('222', '222')");  
 statement.execute("insert into author (first\_name, last\_name) values ('333', '333', '333')");  
 statement.execute("insert into author (first\_name, last\_name) values ('444', '444')");  
  
 c.commit();  
  
 } catch (SQLException e) {  
 e.printStackTrace();  
 } }}

* c.setAutoCommit(true); - это обычный построчный коммит
* Причем, открыв c.setAutoCommit(false); его не нужно открывать в след раз, после того как закоммитили - просто после следующая цепочка запросов, после которой c.commit();  
   и тд
* Если между c.setAutoCommit(false); и c.setAutoCommit(false); не вписать запросов, то эксепшн Не выдаст

public class JdbcTransaction {  
  
 private static final String *URL* = "jdbc:postgresql://rdsdatabase-x.cplmjbu0dc7d.us-east-2.rds.amazonaws.com:5432/mydbasex";  
 private static final String *QUERY* = "insert into author (first\_name, last\_name) values (?, ?)";  
  
 public static void main(String[] args) throws ClassNotFoundException, IOException {  
  
 Properties prop = new Properties();  
 prop.load(Files.*newInputStream*(Paths.*get*("db.properties")));  
  
 try (Scanner scanner = new Scanner(System.*in*);  
 Connection c = DriverManager.*getConnection*(*URL*, prop)) {  
  
 c.setAutoCommit(false);  
 c.commit();  
  
 Statement statement = c.createStatement();  
 statement.execute("insert into author (first\_name, last\_name) values ('111', '111')");  
 statement.execute("insert into author (first\_name, last\_name) values ('222', '222')");  
 statement.execute("insert into author (first\_name, last\_name) values ('333', '333', '333')");  
 statement.execute("insert into author (first\_name, last\_name) values ('444', '444')");  
  
 c.commit();  
  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 }  
}

Вывод:

org.postgresql.util.PSQLException: ERROR: INSERT has more expressions than target columns

Позиция: 66

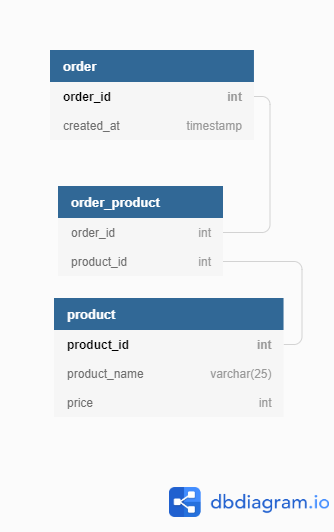
at org.postgresql.core.v3.QueryExecutorImpl.receiveErrorResponse(QueryExecutorImpl.java:2553)

Соответственно, изменения не будут внесены никакие, тк 4 запроса в случае такого коммита – это единая операция

Работа с бд из JDBC – пример

Есть 2 таблицы:

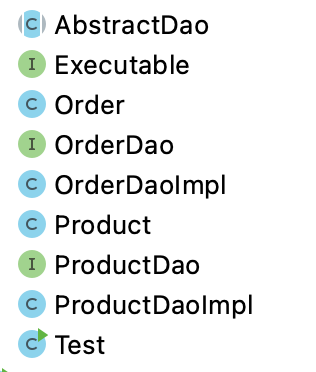
Orders  и Products с отношением многие ко многим



Нужно обеспечить  СRUD операции + формирование List для заказов, в которых есть определенный продукт и тп

Свой кодик я сюда плюсовать не буду – там подключение я каждый раз добавляю (в каждом методе) и еще есть усложнения всякие + toString перезаписан и конструкторы сделал– добавлю просто код от препода, хоть мой ближе к сердцу и понятнее, а свой мб на гитхаб выложу

Соответственно, в результате должно получиться следующее:



**Order и Product** - классы с полями колонок таблиц Order и Product. Писать класс под промежуточную таблицу не требуется – ее создают в б/д изначально, и если прописывают Foreign key’и, то в целом при апдейтах или удалениях из Order и Product,  ее менять не надо. Но если метод по изменению состава заказа, то нужно будет сначала delete сделать по всем строкам где этот номер заказа, а затем по-новой занести нужное.

import java.time.LocalDateTime;  
  
public class Order {  
  
 private int orderId;  
 private LocalDateTime createdAt;  
  
 public int getOrderId() {  
 return orderId;  
 }  
  
 public void setOrderId(int orderId) {  
 this.orderId = orderId;  
 }  
  
 public LocalDateTime getCreatedAt() {  
 return createdAt;  
 }  
  
 public void setCreatedAt(LocalDateTime createdAt) {  
 this.createdAt = createdAt;  
 }  
}

\_\_

public class Product {  
  
 private int productId;  
 private String productName;  
 private int price;  
  
 public int getProductId() {  
 return productId;  
 }  
  
 public void setProductId(int productId) {  
 this.productId = productId;  
 }  
  
 public String getProductName() {  
 return productName;  
 }  
  
 public void setProductName(String productName) {  
 this.productName = productName;  
 }  
  
 public int getPrice() {  
 return price;  
 }  
  
 public void setPrice(int price) {  
 this.price = price;  
 }  
}

AbstractDAO и Executable – это интерфейс и класс (абстрактый – просто для того, чтобы экземпляр не создавать) для создания подключения

import java.sql.Connection;  
import java.sql.SQLException;  
  
@FunctionalInterface  
public interface Executable<R> {  
 R execute(Connection c) throws SQLException;  
}

import java.sql.SQLException;  
import java.util.Properties;  
  
abstract public class AbstractDao {  
  
 private String url;  
 private String login;  
 private String password;  
  
 {  
 try (InputStream input = new FileInputStream("db-hw.properties")) {  
 Properties prop = new Properties();  
 prop.load(input);  
 url = prop.getProperty("db.url");  
 login = prop.getProperty("db.login");  
 password = prop.getProperty("db.password");  
 } catch (FileNotFoundException e) {  
 e.printStackTrace();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 protected <R> R execute(Executable<R> executable) {  
 try (Connection c = DriverManager.*getConnection*(url, login, password)) {  
 return executable.execute(c);  
 } catch (SQLException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
}

Test – соответственно для тестов (вместо main). Тесты сделаны не через Assert. и jUnit, методв assert написан вручную внизу

import java.time.LocalDateTime;  
import java.time.temporal.ChronoUnit;  
import java.util.List;  
import java.util.Objects;  
  
public class Test {  
  
 public static void main(String[] args) {  
  
 ProductDao productDao = new ProductDaoImpl();  
 OrderDao orderDao = new OrderDaoImpl();  
  
 *clearTables*(productDao, orderDao);  
  
 int product1Id = productDao.createProduct("Sunglasses", 100);  
 int product2Id = productDao.createProduct("Phone", 350);  
 int product3Id = productDao.createProduct("Book", 10);  
  
 int order1Id = orderDao.createOrder(List.*of*(product1Id, product2Id));  
 int order2Id = orderDao.createOrder(List.*of*(product3Id));  
  
 *assertEquals*(orderDao.getOrderById(order1Id).getOrderId(), order1Id);  
 *assertEquals*(productDao.listProductsByOrderId(order1Id).size(), 2);  
 *assertEquals*(productDao.listProductsByOrderId(order2Id).size(), 1);  
  
 orderDao.updateOrder(order1Id, List.*of*(product1Id, product2Id, product3Id));  
  
 *assertEquals*(productDao.listProductsByOrderId(order1Id).size(), 3);  
 *assertEquals*(orderDao.listOrders().size(), 2);  
  
 productDao.updateProduct(product3Id, "e-Book", 100);  
  
 *assertEquals*(orderDao.listOrdersByProductName("e-Book").size(), 2);  
 *assertEquals*(orderDao.listOrdersCreatedBetween(  
 LocalDateTime.*now*().minus(1, ChronoUnit.*DAYS*),  
 LocalDateTime.*now*().plus(1, ChronoUnit.*DAYS*)  
 ).size(), 2);  
  
 orderDao.deleteOrder(order1Id);  
 *assertEquals*(orderDao.listOrdersByProductName("e-Book").size(), 1);  
  
 *assertEquals*(productDao.listProducts().size(), 3);  
 productDao.deleteProduct(product1Id);  
 *assertEquals*(productDao.listProducts().size(), 2);  
  
 }  
  
 private static void clearTables(ProductDao productDao, OrderDao orderDao) {  
 orderDao.listOrders().forEach(o -> orderDao.deleteOrder(o.getOrderId()));  
 productDao.listProducts().forEach(p -> productDao.deleteProduct(p.getProductId()));  
 }  
  
 private static void assertEquals(Object real, Object expected){  
 if (!Objects.*equals*(real, expected))  
 throw new AssertionError(String.*format*("'%s' doesn't match expected '%s'", real, expected));  
 }  
}

OrderDAO  и ProductDAO – это интерфейсы с методами, необходимыми для прохождения Test, реализованными в классах OrderDAOImpl и ProductDAOImpl:

import java.time.LocalDateTime;  
import java.util.List;  
  
public interface OrderDao {  
  
 *//Создать заказ (возвращается ID созданного заказа)* int createOrder(List<Integer> productIds);  
  
 *//Изменить существующий заказ* void updateOrder(int orderId, List<Integer> productIds);  
  
 *//Удалить заказ* void deleteOrder(int orderId);  
  
 *//Получить все заказы* List<Order> listOrders();  
  
 *//Получить заказы, созданные между двумя датами* List<Order> listOrdersCreatedBetween(LocalDateTime from, LocalDateTime to);  
  
 *//Получить заказы, содержащие товар с указанным именем* List<Order> listOrdersByProductName(String productName);  
  
 *//Получить заказ по ID* Order getOrderById(int orderId);  
  
  
}

import java.sql.\*;  
import java.time.LocalDateTime;  
import java.util.ArrayList;  
import java.util.List;  
  
public class OrderDaoImpl extends AbstractDao implements OrderDao {  
  
 @Override  
 public int createOrder(List<Integer> productIds) {  
 return execute(c -> {  
 PreparedStatement statement = c.prepareStatement(  
 "insert into \"order\" (created\_at) values (?)",Statement.*RETURN\_GENERATED\_KEYS*);  
 statement.setTimestamp(1, Timestamp.*valueOf*(LocalDateTime.*now*()));  
 statement.executeUpdate();  
 ResultSet set = statement.getGeneratedKeys();  
  
 set.next();  
 int orderId = set.getInt(1);  
 statement = c.prepareStatement("insert into \"order\_product\" values (?, ?)");  
 for (Integer productId : productIds){  
 statement.setInt(1, orderId);  
 statement.setInt(2, productId);  
 statement.executeUpdate();  
 }  
 return orderId;  
 });  
 }  
  
 @Override  
 public void updateOrder(int orderId, List<Integer> productIds) {  
 execute(c -> {  
 PreparedStatement statement = c.prepareStatement(  
 "delete from \"order\_product\" where order\_id = ?");  
 statement.setInt(1, orderId);  
 statement.executeUpdate();  
  
 statement = c.prepareStatement("insert into \"order\_product\" values (?, ?)");  
 for (Integer productId : productIds){  
 statement.setInt(1, orderId);  
 statement.setInt(2, productId);  
 statement.executeUpdate();  
 }  
 return null;  
 });  
 }  
  
 @Override  
 public void deleteOrder(int orderId) {  
 execute(c -> {  
 PreparedStatement statement = c.prepareStatement(  
 "delete from \"order\_product\" where order\_id = ?");  
 statement.setInt(1, orderId);  
 statement.executeUpdate();  
  
 statement = c.prepareStatement(  
 "delete from \"order\" where order\_id = ?");  
 statement.setInt(1, orderId);  
 statement.executeUpdate();  
 return null;  
 });  
 }

@Override  
 public List<Order> listOrders() {  
 return execute(c -> {  
 Statement statement = c.createStatement();  
 ResultSet set = statement.executeQuery("select order\_id, created\_at from \"order\"");  
 return toList(set);  
 });  
 }  
  
 @Override  
 public List<Order> listOrdersCreatedBetween(LocalDateTime from, LocalDateTime to) {  
 return execute(c -> {  
 PreparedStatement statement = c.prepareStatement(  
 "select order\_id, created\_at " +  
 "from \"order\" " +  
 "where created\_at between ? and ?");  
 statement.setTimestamp(1, Timestamp.*valueOf*(from));  
 statement.setTimestamp(2, Timestamp.*valueOf*(to));  
 ResultSet set = statement.executeQuery();  
 return toList(set);  
 });  
 }  
  
 @Override  
 public List<Order> listOrdersByProductName(String productName) {  
 return execute(c -> {  
 PreparedStatement statement = c.prepareStatement(  
 "select o.order\_id, o.created\_at " +  
 "from \"order\" o join order\_product op " +  
 "on o.order\_id = op.order\_id join product p " +  
 "on op.product\_id = p.product\_id " +  
 "where p.product\_name = ?");  
 statement.setString(1, productName);  
 ResultSet set = statement.executeQuery();  
 return toList(set);  
 });  
 }  
  
 @Override  
 public Order getOrderById(int orderId) {  
 return execute(c -> {  
 PreparedStatement statement = c.prepareStatement(  
 "select order\_id, created\_at " +  
 "from \"order\" where order\_id = ?");  
 statement.setInt(1, orderId);  
 ResultSet set = statement.executeQuery();  
 return toOrder(set);  
 });  
 }  
  
  
 private List<Order> toList(ResultSet set) throws SQLException {  
 List<Order> orders = new ArrayList<>();  
 while (set.next()){  
 Order order = new Order();  
 order.setOrderId(set.getInt(1));  
 order.setCreatedAt(set.getTimestamp(2).toLocalDateTime());  
 orders.add(order);  
 }  
 return orders;  
 }  
  
 private Order toOrder(ResultSet set) throws SQLException{  
 if (set.next()) {  
 Order order = new Order();  
 order.setOrderId(set.getInt(1));  
 order.setCreatedAt(set.getTimestamp(2).toLocalDateTime());  
 return order;  
 } else {  
 return null;}}}

\_\_\_\_\_\_

import java.util.List;  
  
public interface ProductDao {  
  
 *//Создать товар (возвращается ID созданного товара)* int createProduct(String productName, int price);  
  
 *//Изменить существующий товар* void updateProduct(int productId, String productName, int price);  
  
 *//Удалить товар* void deleteProduct(int productId);  
  
 *//Получить все товары* List<Product> listProducts();  
  
 *//Получить товары, относящиеся к заказу с указанным ID* List<Product> listProductsByOrderId(int orderId);  
  
}

import java.sql.PreparedStatement;  
import java.sql.ResultSet;  
import java.sql.SQLException;  
import java.sql.Statement;  
import java.util.ArrayList;  
import java.util.List;  
  
public class ProductDaoImpl extends AbstractDao implements ProductDao {  
 @Override  
 public int createProduct(String productName, int price) {  
 return execute(c -> {  
 PreparedStatement statement = c.prepareStatement(  
 "insert into product (product\_name, price) values (?, ?)", Statement.*RETURN\_GENERATED\_KEYS*);  
 statement.setString(1, productName);  
 statement.setInt(2, price);  
 statement.executeUpdate();  
 ResultSet set = statement.getGeneratedKeys();  
  
 set.next();  
 return set.getInt(1);  
 });  
 }  
  
 @Override  
 public void updateProduct(int productId, String productName, int price) {  
 execute(c -> {  
 PreparedStatement statement = c.prepareStatement(  
 "update product " +  
 "set product\_name = ?, " +  
 "price = ? " +  
 "where product\_id = ?");  
 statement.setString(1, productName);  
 statement.setInt(2, price);  
 statement.setInt(3, productId);  
 statement.executeUpdate();  
 return null;  
 });  
 }  
  
 @Override  
 public void deleteProduct(int productId) {  
 execute(c -> {  
 PreparedStatement statement = c.prepareStatement(  
 "delete from product where product\_id = ?");  
 statement.setInt(1, productId);  
 statement.executeUpdate();  
 return null;  
 });  
 }  
  
 @Override  
 public List<Product> listProducts() {  
 return execute(c -> {  
 Statement statement = c.createStatement();  
 ResultSet set = statement.executeQuery("select p.product\_id, p.product\_name, p.price " +  
 "from product p");  
 return toList(set);  
 });  
 }  
  
 @Override  
 public List<Product> listProductsByOrderId(int orderId) {  
 return execute(c -> {  
 PreparedStatement statement = c.prepareStatement(  
 "select p.product\_id, p.product\_name, p.price " +  
 "from product p join order\_product op on p.product\_id = op.product\_id " +  
 "where op.order\_id = ?");  
 statement.setInt(1, orderId);  
 ResultSet set = statement.executeQuery();  
 return toList(set);  
 });  
 }  
  
 private Product toProduct(ResultSet set) throws SQLException {  
 if (set.next()) {  
 Product product = new Product();  
 product.setProductId(set.getInt(1));  
 product.setProductName(set.getString(2));  
 product.setPrice(set.getInt(3));  
 return product;  
 } else {  
 return null;  
 }  
 }  
  
 private List<Product> toList(ResultSet set) throws SQLException {  
 List<Product> products = new ArrayList<>();  
 while (set.next()){  
 Product product = new Product();  
 product.setProductId(set.getInt(1));  
 product.setProductName(set.getString(2));  
 product.setPrice(set.getInt(3));  
 products.add(product);  
 }  
 return products;  
 }  
}

Вот так вот. У меня все менее изящно, более громоздко, но тоже работает☺