**JDBC**

JDBC – стандарт для взаимодействия Java-приложения с реляционными БД, представляет собой набор классов и интерфейсов для работы с различными реляционными БД. Реализация интерфейсов JDBC API осуществляется за счет JDBC-драйвера.

JDBC-драйвер – jar-файл, который обеспечивает взаимодействие между Java-приложением и конкретной БД. Предоставляет реализацию JDBC API для конкретной реляционной БД (адаптер Java-приложения к БД – преобразует запрос на Java в формат, который БД сможет обработать и выполнить).

Стандартизированный API обеспечивает независимость Java-приложения от БД, с которой она взаимодействует, то есть можно легко переключаться между разными БД, изменив драйвер на соответствующий, при этом не меняя реализацию приложения.

**Общий план подключения и работы:**

1. Подключение драйвера к проекту: добавление его в classpath(file-> proj structure-> modules) или подключение зависимости.
2. Загрузка класса этого драйвера и его регистрация – с помощью Class.forName() или DriverManager.registerDriver() – с JDBC 4 это происходит автоматически при создании подключения.

Class.*forName*("org.postgresql.Driver");  
  
Driver driver = new org.postgresql.Driver();  
DriverManager.*registerDriver*(driver);

2) Создание подключения (получение объекта Connection) с помощью DriverManager.getConnection().

3) Дальнейшая работа с классами, представляющими работу с SQL запросами (Statement, PreparedStatment, CallableStatment).

**Основные классы:**

*DriverManager* – Класс, предназначенный для работы с драйверами и установкой соединения между Java приложением и БД. Основное назначение – регистрация драйверов и установление соединения.

registerDriver() – явно можно не вызывать, т.к. метод вызывается в static блоке класса Driver при его загрузке.

getConnection() – опрашивает каждый загруженный драйвер для определения драйвера для установки соединения (действует простым перебором и устанавливает соединение с первым соответствием).

*Connection* – устанавливается с помощью DriverManager.getConnection(), посредством передачи в него данных для подключения. Включает в себя методы для работы с транзакциями (commit, rollback, setautocommit, savepoint), работа со специальными типами (blob, clob, array, xml), исполнение SQL запросов (statement, preparestatment, callablestatment).

**Классы для исполнения запросов:**

***Statement*** – Для работы со статичными SQL-запросами, не требующими параметров (для DDL операций или простых select).

***PreparedStatment*** – наследник statement, предназначен для запросов, которые могут содержать конфигурируемые параметры, т.е. может исполняться несколько раз с разным набором этих параметров.

***CallableStatment*** – наследник PreparedStatment, конфигурируются входные и выходные параметры, используется для вызова хранимых процедур c помощью preparecall(). Имеет общий принцип работы с PreparedStatment.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Характеристика** | Statment | PreparedStatement | |  | | --- | | CallableStatement | |
| **Использование** | Простые статичные SQL-запросы | Запросы с параметрами | Вызов хранимых процедур |
| **Параметризация** | Нет | Да | Да – параметризация входных и выходных параметров |
| **SQL Injection** | Уязвим | Защищен | Защищен |
| **Производительность** | Ниже – повторная компиляция при каждом исполнении | Выше – одна компиляция | Аналогично PreparedStatement |
| **Цель** | Выполнение простых запросов | Выполнение динамических запросов с параметрами | Вызов логики в хранимых процедурах |
| **Поддержка входных/выходных параметров** | Нет | Только входные параметры | Поддержка IN, OUT и INOUT параметров |

Отличие в синтаксисе – запрос передается при создании PreparedStatment, а параметры в таком запросе заменяются на ‘?’, значение которых должно быть установлено до исполнения запроса.

String query = """  
 insert into new\_table(name)  
 values (?),(?);  
 """; //Вопросы указывают на изменяемые параметры  
try (Connection connection = DriverManager.*getConnection*("jdbc:postgresql://localhost:5432/northwind", "user", "user")) {  
 PreparedStatement prep = connection.prepareStatement(query);  
 prep.setString(1, "new name");  
 prep.setString(2, "another name");  
 //Каждый знак вопроса задается определенным числовым типом с помощью метода set и передается его номер  
 int updateNumber = prep.executeUpdate(); //Исполняется как Statement  
 System.*out*.println(updateNumber);  
}

Execute() – Исполняет SQL-запрос, возвращает Boolean, поэтому предпочтительное использование – DDL операции, что не подразумевает возвращаемые данные (возвращает true, если результат запроса resultset и false в остальных случаях). При этом все таки можно получить кол-во измененных записей или ResultSet посредством вызова getUpdateCount(), getResultSet() и др. на соответствующем Statement.

executeUpdate() – отличается возвращаемым значением, возвращает число затронутых операцией записей (-1 если таких нет). Используется, когда не требуется получить множество результатов запроса.

executeQuerry() – возвращает resultset, который содержит множество результатов запроса, используется, в случае, если запрос подразумевает возвращение некоторого множества. Используется в запросах типа select или запросах, использующих returning.

SetQueryTimeout() – устанавливает максимальное время ожидания запроса к БД.

SetfetchSize() – определяет максимальный размер результирующего набора, который будет получать приложение. Т.е. результат общей выборки не изменится, но приложение будет получать ее пакетами указанного размера, что уменьшает нагрузку на клиентскую часть за счет уменьшения кол-ва записей, которые нужно хранить, при этом увеличивает кол-во запросов к БД.

SetMaxRows()– устанавливает максимальное значение результирующего набора.

***ResultSet*** – содержит набор данных для извлечения. Работает как курсор или итератор – для получения следующего значения используется next() (изначально курсор не указывает ни на один результат). Содержание полей текущей записи доступно через вызовы методов getInt(), getFloat(), getString(), getDate() и т.д.

Для работы с результатами, которые в перспективе могут иметь значение null нужно использовать getObject() при работе с ResultSet, метод требует передать тип объекта, к которому нужно кастить результат (Class.class)

***Batch*** – механизм для группировки SQL-запросов и отправки их в БД как один пакет. Используется для запросов, которые не имеют возвращаемого результата (не используется для select). Используется только со statement, т.к. prepareStatement создается под каждый запрос. Необходим для оптимизации производительности путем уменьшение накладных расходов на связь с базой данных за счет уменьшения числа обращений между приложением и базой данных. Взаимодействие осуществляется посредством методов addBatch() – добавляет запрос в пакет, executeBatch() – исполняет пакет запросов, clearBatch()- очищает пакет.

**Magic constant** – константы для настроек параметров ResultSet при создании Statement.

*Константы направления движения по результату:*

**TYPE\_FORWARD\_ONLY** – только вперед (по умолчанию)

**TYPE\_SCROLL\_INSENSITIVE** – вперед и назад, не чувствителен к изменению значений результата в БД

**TYPE\_SCROLL\_SENSITIVE** – вперед и назад, чувствителен к изменению значений результата в БД (изменения не поддерживаются в PostgreSQL)

*Редактирование данных с помощью ResultSet:*

**CONCUR\_READ\_ONLY** – только для чтения (по умолчанию)

**CONCUR\_UPDATABLE** – можно изменять данные в БД через ResultSet

*Константы для возвращения генерируемых ключей* (вызываются у Statment):

**RETURN\_GENERATED\_KEYS –** возвращает генерируемые ключи

**NO\_GENERATED\_KEYS –** не возвращает ключи (по умолчанию)

[Код с использованием констант](https://habr.com/ru/sandbox/112346/)

Помимо константы, можно использовать RETURNING в SQL-запросе, для возврата ключей. При использовании RETURNING ключи получаются как возвращаемое значение (использовать с ExecuteQuery)

ResultSet resultSet = statement.executeQuery(sql);

При использовании константы сначала нужно исполнить запрос, а затем получить ключи:

ResultSet resultSet = statement.execute(sql);  
ResultSet resultSet = statement.getGeneratedKeys()

**Blob/Clob**

Blob (binary large object), Clob (character large object) – типы данных в некоторых БД для хранения крупноразмерных бинарных или текстовых файлов, которые имеют соответствующие классы в JDBC.

В PostgreSQL отсутствуют такие типы данных, для хранения данных типа BLOB используется тип данных bytea, вместо CLOB используется тип данных text и вместо стандартного способа создания таких объектов (с помощью connection.creteBlob/Clob и конфигурирования этого объекта) используется установка соответствующих параметров prepareStatement (setBytes, setString).

**Получение нескольких множеств**

String sql = """  
 select \* from new\_table where id > 35;  
 select \* from new\_table where id > 36;  
 """;  
try (Connection connection = DriverManager.*getConnection*("jdbc:postgresql://localhost:5432/northwind", "user", "user")) {  
 PreparedStatement preparedStatement = connection.prepareStatement(sql);  
 boolean hasRes = preparedStatement.execute();  
 ResultSet res = preparedStatement.getResultSet();  
 //код, для получения нескольких множеств результатов  
 while (hasRes) {  
 if (res.isClosed()) break;  
 while (res.next()) {  
 System.*out*.println(res.getInt(1) + " " + res.getString(2));  
 }  
 hasRes = preparedStatement.getMoreResults(); //проверяем, есть ли еще множества результатов  
 }  
}

**DAO (Data access object)** – паттерн проектирования, который используется для абстрагирования и инкапсуляции доступа к данным реляционной БД.

Структурно выражен с помощью Entity – является проекцией каждой таблицы БД, где кортеж таблицы выражен объектом этого класса – поля класса соответствуют атрибутам таблицы (таблицы с отношением многие ко многим не мапятся). DAO слой – инкапсулирует логику взаимодействия с базой данных (Синглтон, не содержит состояния, хранит запросы в виде статических полей)

[Практика](https://www.youtube.com/watch?v=S6tQJk6CH38&list=PLt91xr-Pp57QEnGJPuI4R_VqEXikn2a76&index=12&ab_channel=JavaGuru)