**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)

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);

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

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

**Классы**

*DriverManager* – Класс, предназначенный для работы с драйверами и установления 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 того, с которым можно установить соединение (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 работы с запросами**

***Statement*** – Для работы SQL операторов без параметров, то есть статических SQL выражений, так как подвержены SQL инъекциям, поскольку он выполняет SQL-запрос без дополнительной проверки.

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. Сразу после получения набора данных его курсор находится перед первой записью и чтобы сделать её текущей необходимо вызвать метод next().  
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*** – структура для группировки запросов, подразумевающих возвращение кол-ва измененных строк или отсутствие значения (не Select) для каждого запроса (при использовании executeUpdate вернется только число обновлений для последнего запроса). 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.

Работает только с сущностью Statement.

***PreparedStatment*** – наследник statement, предназначен для SQL-запросов с параметрами и часто выполняемых операторов. Предварительно компилирует запрос, который может содержать входные параметры и выполняться несколько раз, с разным набором данных параметров, так же СУБД может определить такой запрос как ранее исполняемый и кэшировать его, за счет чего увеличится производительность. Так же защищает от SQL-инъекций.

*SQL инъекции* – внедрение вредоносного SQL запроса в основной запрос. Внедрение происходит за счет наличия параметров, которые приходят в приложение извне и используются для подстановки в основной запрос.

String line = "Select \* from new\_table where id =";  
String id = "31"; //обычный параметр, полученный от пользователя  
String injection = "31 or 1=1";//параметр с инъекцией  
try (Connection connect = DriverManager.*getConnection*("jdbc:postgresql://localhost:5432/northwind", "user", "user")) {  
 Statement statement = connect.createStatement();  
 ResultSet result = statement.executeQuery(line + id);  
 while (result.next()) {  
 System.*out*.println(result.getInt("id") + " " + result.getString("name"));  
 }  
 ResultSet injResult = statement.executeQuery(line + injection);  
 System.*out*.println("\_\_\_\_\_\_\_\_");  
 while (injResult.next()) {  
 System.*out*.println(injResult.getInt("id") + " " + injResult.getString("name"));  
 }//получаем все записи из-за того, что выполняется одно из условий SQL-выражения  
}

Варианты инъекций – Drop table, or 1=1, or ‘’ = ‘’, Union и тд.

Отличие в синтаксисе – запрос передается при создании PreparedStatment, а параметры изменяемые параметры в таком запросе заменяются на ‘?’. При вызове исполняющего метода (одного из execute) в него передаются нужные параметры для исполнения.

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);  
}

***CallableStatment*** – наследник PreparedStatment, используется для вызова хранимых процедур c помощью preparecall(). Имеет такой же принцип, как и PreparedStatment

String procedure = """  
 call sum\_values(?,?);  
 """;  
try (Connection connection = DriverManager.*getConnection*("jdbc:postgresql://localhost:5432/northwind", "user", "user")) {  
 CallableStatement callable = connection.prepareCall(procedure);  
 callable.setInt(1, 10);  
 callable.setInt(2, 35);  
 ResultSet result = callable.executeQuery();  
 while(result.next()){  
 System.*out*.println(result.getInt(1));  
 } //Можно получить результат с помощью регистрации возвращаемого параметра и его результата

}

**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

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/Array**

Blob, Clob – типы данных в некоторых БД, которым соответствуют binary large object и character large object (для хранения крупноразмерных объектов), которые имеют соответствующие классы в JDBC.

String line = "UPDATE new\_table SET image = ? where id = ?";  
try (Connection connection = DriverManager.*getConnection*("jdbc:postgresql://localhost:5432/northwind", "user", "user")) {  
 PreparedStatement preparedStatement = connection.prepareStatement(line);  
 Blob blob = connection.createBlob(); // создать блоб  
 blob.setBytes(1, Files.*readAllBytes*(Path.*of*("путь"))); //передать в него данные  
 preparedStatement.setInt(2, 31); // установка параметра id  
 preparedStatement.setBlob(1, blob); //установка параметра blob  
 int count = preparedStatement.executeUpdate(); //исполнение запроса

Вроде, при работе с BLOB и CLOB нужно убирать автомкоммит.

В PostgreSQL отсутствуют такие типы данных, для хранения данных типа BLOB используется тип данных bytea, который получается из БД соответственно, вместо CLOB используется тип данных text, который получается, как String.

**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 – шаблон проектирования, предоставляет интерфейс для работы с БД, который представляет абстракцию, обеспечивающую связь между Java-приложением и реляционной БД, то есть работа с БД реализована только в DAO классах. Представляет из себя набор классов, каждый из которых соответствует сущности в БД, которые содержат методы для выполнения CRUD-операций над данными. (См. примерную реализацию в проекте)