





JDBC

Java Database Connectivity (JDBC) — определяет API для доступа к базе данных из Java.

Для установления соединения нужен JDBC драйвер для конкретной БД.

Рассмотрим простой пример подключения к БД и выполнения простого запроса.

Допустим в СУБД PostgreSQL есть БД с таблицей `course`, в которой следующие данные:

 id	 title	 duration	 price
1	Python	150	20000
2	PHP	174	25000
3	Ruby	135	17000
4	Java	145	30000
5	Scala	120	18000
6	Clojure	90	13000
7	Haskell	156	9000

Для начала создадим maven проект и добавим в зависимость jdbc драйвер для PostgreSQL:

```
<?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.ifmo</groupId>
  <artifactId>jdbc-example</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <postgres-jdbc-driver.version>9.1-901-1.jdbc4</postgres-jdbc-driver.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>${postgres-jdbc-driver.version}</version>
    </dependency>
  </dependencies>
</project>
```

Теперь подключим драйвер, установим соединение и выполним простой запрос:

```
public class JdbcExample {
    public static void main(String[] args) throws ClassNotFoundException, SQLException {
        // Загружаем нужный нам JDBC драйвер
        Class.forName("org.postgresql.Driver");

        // Выполняем подключение, используя загруженный драйвер: ifmo - название БД в
        PostgreSQL СУБД
        try (Connection con =
            DriverManager.getConnection("jdbc:postgresql://localhost:5432/ifmo", "ifmo", "q1w2e3")) {
            // Создаем statement
            try (Statement stmt = con.createStatement()) {
                // Выполняем запрос к БД
                try (ResultSet rs = stmt.executeQuery("SELECT * FROM course")) {
                    // Перемещаем курсор по результатам
                    while (rs.next()) {
                        // Извлекаем конкретные значения из ResultSet
                        int id = rs.getInt("id");
                        String title = rs.getString("title");
                        int duration = rs.getInt("duration");
                        double price = rs.getDouble("price");

                        System.out.printf("id: %s, title: %s, duration: %s, price: %s\n",
                            id, title, duration, price);
                    }
                }
            }
        }
    }
}
```

Если все указано верно, то программа выведет в консоль:

```
id: 1, title: Python, duration: 150, price: 20000.0  
id: 2, title: PHP, duration: 174, price: 25000.0  
id: 3, title: Ruby, duration: 135, price: 17000.0  
id: 4, title: Java, duration: 145, price: 30000.0  
id: 5, title: Scala, duration: 120, price: 18000.0  
id: 6, title: Clojure, duration: 90, price: 13000.0  
id: 7, title: Haskell, duration: 156, price: 9000.0
```

Устанавливая соединение мы указали следующие данные:

jdbc — тип соединения,
postgresql — имя загруженного jdbc драйвера,
//localhost — хост СУБД,
5432 — порт, на котором СУБД слушает входящие подключения,
ifmo — имя конкретной БД:

```
jdbc:postgresql://localhost:5432/ifmo
```

имя пользователя и пароль.

Класс `Connection` предоставляет множество методов для работы с БД, такие как получения метаданных (информация о таблицах, хранимых процедурах и т. п.), управления транзакциями, информации о текущей схеме, подключении и так далее.

Но так же дает возможность создавать `Statement` — фактически запрос к БД.

`Statement` так же богата методами, но нас интересуют в первую очередь `execute()`, `executeQuery()`, `executeUpdate()`.

Результатом выполнения запроса является `ResultSet`, он использует полученный от СУБД курсор и перемещается по полученным строкам, позволяя извлекать конкретные данные.

`Connection`, `Statement`, `ResultSet` должны быть закрыты после окончания работы с ними.

Таким образом, мы можем выполнить любой запрос, например с фильтром:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM course WHERE title='Java'")
```

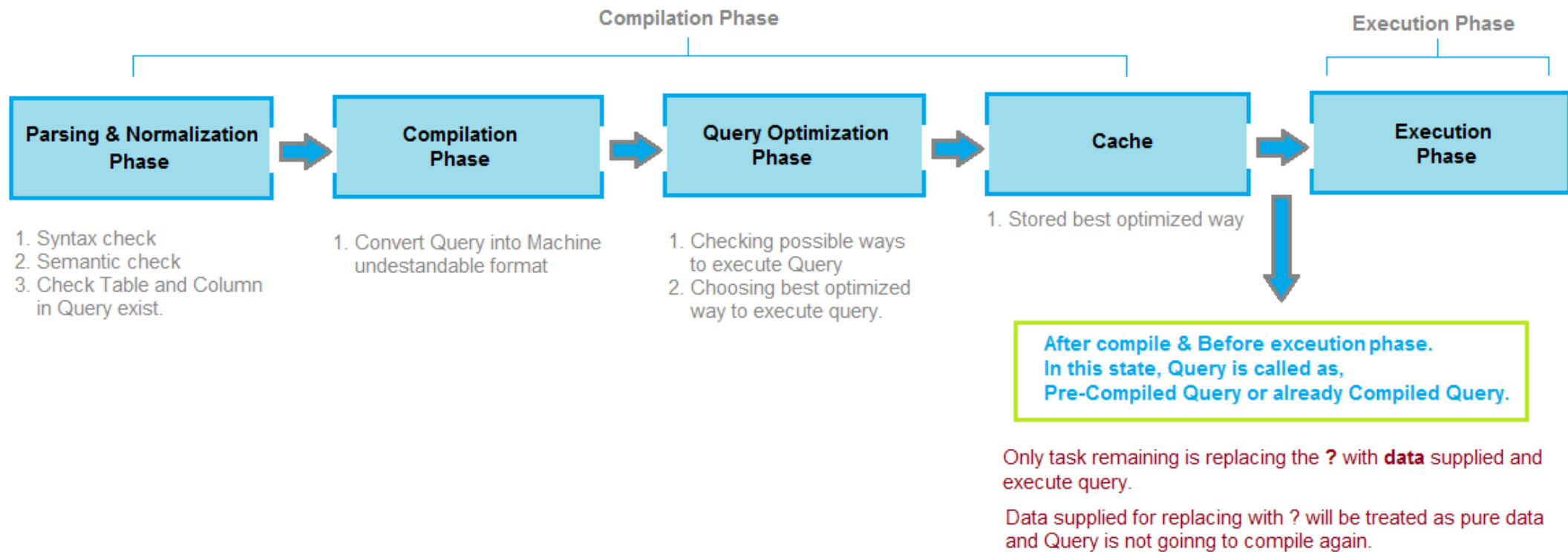
Вставка, изменение данных или DML, DDL выполняется при помощи `executeUpdate()`, который возвращает количество измененных строк:

```
stmt.executeUpdate("INSERT INTO course(title, duration, price) VALUES  
( 'JavaScript', 102, 34000)");
```

Но помимо Statement существует так же PreparedStatement. Его особенность заключается в первую очередь в том, что он обрабатывается, оптимизируется и кэшируется СУБД один раз при первом исполнении, после чего каждый последующий запрос выполняется гораздо быстрее.

Одним из приятных побочных эффектов такого поведения является защита от SQL инъекций, т. к. аргументы отправляются в СУБД в сыром виде и легко экранируются.

Query Execution Phases



Beauty of Prepare Statement

Найдем Java курсы с помощью PreparedStatement:

```
// Подготавливаем запрос, который будет заэкширован, а аргументы заменяем ?
PreparedStatement prepared = con.prepareStatement("SELECT * FROM course WHERE
title=?");
// Устанавливаем на места ? конкретные аргументы
prepared.setString(1, "Java");
// Выполняем запрос к БД
try (ResultSet rs = prepared.executeQuery()) {
    // Перемещаем курсор по результатам
    while (rs.next()) {
        // Извлекаем конкретные значения из ResultSet
        int id = rs.getInt("id");
        String title = rs.getString("title");
        int duration = rs.getInt("duration");
        double price = rs.getDouble("price");

        System.out.printf("id: %s, title: %s, duration: %s, price: %s\n", id, title,
duration, price);
    }
}
```

Пример транзакций:

```
// Выключаем автокоммит транзакций  
con.setAutoCommit(false);
```

```
// Создаем statement
```

```
try (Statement stmt = con.createStatement()) {  
    // Выполняем запрос к БД
```

```
    int rows = stmt.executeUpdate("INSERT INTO course(title, duration, price)  
VALUES ('NodeJS', 102, 34000)");  
    System.out.println(rows);
```

```
    // Если все хорошо завершаем транзакцию  
    con.commit();
```

```
}  
catch (SQLException e) {  
    // В случае ошибки откатываем изменения  
    con.rollback();  
}
```

Но у работы с Connection есть ряд недостатков, например, он не обязан быть потокобезопасным, что вынуждает программиста или открывать/закрывать соединение на каждый запрос или сохранять его отдельно для каждого потока.

Для решения такой задачи обычно используются пулы соединений. Они поддерживают заданное число подключений к БД и прекрасно работают в многопоточной среде.

Рассмотрим работу пула потоков на примере c3p0.

Для этого подключим его к нашему maven проекту:

```
<dependency>  
  <groupId>c3p0</groupId>  
  <artifactId>c3p0</artifactId>  
  <version>0.9.1.2</version>  
</dependency>
```

Добавим DataSource класс, который будет хранить ссылку на пул:

```

public class DataSource {
    private static volatile DataSource dataSource;
    private ComboPooledDataSource cpds;

    private DataSource() throws PropertyVetoException {
        cpds = new ComboPooledDataSource();
        cpds.setDriverClass("org.postgresql.Driver");
        cpds.setJdbcUrl("jdbc:postgresql://localhost:5432/ifmo");
        cpds.setUser("ifmo");
        cpds.setPassword("q1w2e3");

        cpds.setMinPoolSize(1);
        cpds.setMaxPoolSize(5);
        cpds.setMaxStatements(180);
    }

    public static DataSource instance() {
        if (dataSource == null) {
            synchronized (DataSource.class) {
                if (dataSource == null) {
                    try {
                        dataSource = new DataSource();
                    }
                    catch (PropertyVetoException e) {
                        throw new IllegalStateException(e);
                    }
                }
            }
        }

        return dataSource;
    }

    public Connection connection() throws SQLException {
        return cpds.getConnection();
    }

    public static Connection getConnection() throws SQLException {
        return instance().connection();
    }
}

```

А дальше просто получаем Connection из пула и работаем с ним как обычно:

```
public static void main(String[] args) throws SQLException {
    try (Connection con = DataSource.getConnection()) {
        try (PreparedStatement prepared = con.prepareStatement("SELECT * FROM course
WHERE title=?")) {

            prepared.setString(1, "Java");

            try (ResultSet rs = prepared.executeQuery()) {
                while (rs.next()) {
                    int id = rs.getInt("id");
                    String title = rs.getString("title");
                    int duration = rs.getInt("duration");
                    double price = rs.getDouble("price");

                    System.out.printf("id: %s, title: %s, duration: %s, price:
%s\n", id, title, duration, price);
                }
            }
        }
    }
}
```

с3p0 позволяет кэшировать PreparedStatement для получения максимальной производительности.