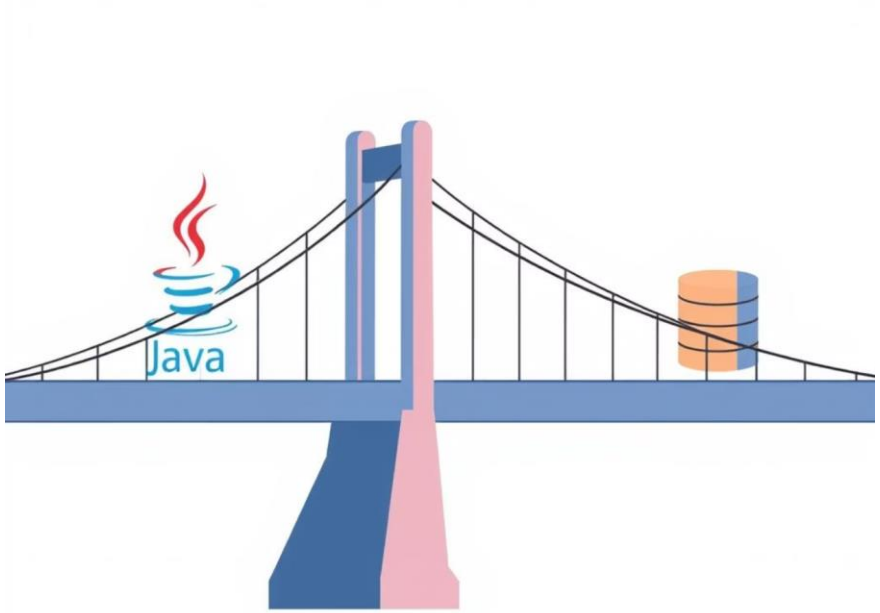
JDBC

Что такое JDBC и зачем он нужен

JDBC (Java Database Connectivity) — это **мост между Java и любой SQL базой данных**.

Вы пишете код один раз, используя JDBC API, а сам JDBC через специальный драйвер разговаривает с нужной вам БД. Можете переключаться между PostgreSQL, MySQL, Oracle, SQL Server — а ваш код остается одинаковым.



Архитектура JDBC: основные компоненты

1

Слой 1 — ваше приложение
Это Java код, который вы пишете. OrderService, UserService, любая бизнес-логика.

2

Слой 2 — JDBC API.
Это интерфейсы, описанные в пакете java.sql. Три самых главных:
Connection — соединение к БД
Statement — способ выполнить SQL команду
ResultSet — результаты запроса

3

Слой 3 — JDBC Driver.
Это конкретная реализация для каждой БД. Для PostgreSQL — это файл postgresql-42.7.3.jar, для MySQL — mysql-connector-java, для Oracle — ojdbc.jar. Драйвер реализует эти интерфейсы конкретно для своей БД.

4

Слой 4 — сама база данных. PostgreSQL, MySQL, Oracle, какая угодно.

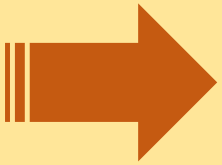
БАЗОВЫЙ ЦИКЛ

Этап 1 — создать соединение.

```
Connection conn = DriverManager.getConnection(
        "jdbc:postgresql://localhost:5432/shop_db",
        "postgres",
        "password”
 ););
```

Используем DriverManager и передаем:

URL БД (где она находится, какой порт)
Имя пользователя
Пароль
Получаем объект Connection

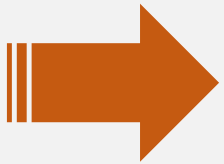


Этап 2 — выполнить SQL.

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT *
FROM orders");
```

Создаем Statement (это способ отправить SQL в БД), и выполняем запрос.

Для SELECT используем executeQuery().
Для INSERT, UPDATE, DELETE используем executeUpdate().



Этап 3 — обработать результаты.

```
while (rs.next()) {
    String name = rs.getString("name");
    int price = rs.getInt("price");
    System.out.println(name + " " + price);
}
```

Проходим по результатам. rs.next() — переходит на следующую строку. rs.getString() — получает значение колонки как String, rs.getInt() — как int.

Важно: в конце закрыть соединение.

```
conn.close();
stmt.close();
rs.close();
```

SQL INJECTION

Представьте, вы разработчик, пишете функцию входа. Пользователь вводит login. И вы конкатенируете его прямо в SQL запрос:

```
String login = userInput;
String sql = "SELECT * FROM users WHERE login
= '" + login + "'";
stmt.executeQuery(sql);
```

Это вход выглядит как:

```
SELECT * FROM users WHERE login = 'ivan'
```

Предположим, пользователь вводит вместо 'ivan' вот это: 'admin' -- (кавычка, слово admin, кавычка, два минуса).

```
SELECT * FROM users WHERE login = 'admin' --'
```

В SQL два минуса — это комментарий, т. е. всё что дальше — игнорируется. То есть запрос становится:

```
SELECT * FROM users WHERE login = 'admin'
```

Запрос вернет данные, для пользователя с логином admin, БЕЗ аутентификации.

Решение — PreparedStatement:

Вместо того чтобы конкатенировать строки, мы используем плейсхолдер — знак вопроса '?'.

Потом мы передаем параметр отдельно через setString(). JDBC сам экранирует все спецсимволы.

Правило: НИКОГДА не конкатенируйте параметры в SQL. ВСЕГДА используйте PreparedStatement с плейсхолдерами."

```
String sql = "SELECT * FROM users WHERE login
= ?";
PreparedStatement pstmt =
conn.prepareStatement(sql);
pstmt.setString(1, userInput);
pstmt.executeQuery();
```

CONNECTION POOLING

Следующая проблема — производительность.
Создание соединения к базе данных — это достаточно медленно.
Потому что нужно установить TCP соединение, авторизоваться в БД, инициализировать сессию.
Решение — Connection Pool.

Без pooling:

```
Запрос 1: создать соединение (~100ms) → execute
(~1ms) → закрыть
Запрос 2: создать соединение (~100ms) → execute
(~1ms) → закрыть
Запрос 1000: создать соединение (~100ms) →
execute (~1ms) → закрыть

Итого: 100+ секунд!
```

С Connection Pool (HikariCP):

```
Инициализация: создать 10 готовых соединений
(~200ms один раз)

Запрос 1: взять из пула (~0.1ms) → execute
(~1ms) → вернуть в пул
Запрос 2: взять из пула (~0.1ms) → execute
(~1ms) → вернуть в пул
Запрос 1000: взять из пула (~0.1ms) → execute
(~1ms) → вернуть в пул

Итого: 1-2 секунды!
```

DAO PATTERN

Допустим, у вас есть OrderService, где вы производите различные сложные операции с заказами, внутри которых выполняете SQL запросы:

При добавлении все новых и новым методов для работы с заказами возникают определенные проблемы:

- SQL код повторяется везде
- Если нужно поменять БД, меняете везде
- Нельзя тестировать бизнес-логику без БД
- Грязная архитектура

```
class OrderService {
    public void createOrder(Order order) {
        Connection conn = ...;
        PreparedStatement pstmt =
conn.prepareStatement(
            "INSERT INTO orders ..."
        );
        // 20 строк SQL кода
        ...
    }

    public Order getOrder(Long id) {
        Connection conn = ...;
        PreparedStatement pstmt =
conn.prepareStatement(
            "SELECT * FROM orders WHERE id = ?"
        );
        // Еще 15 строк SQL кода
        ...
    }
}
```

Правильный способ — DAO Pattern:

Вы создаете отдельный класс OrderDAO, который содержит ВСЕ SQL операции с заказами:

Преимущества:

- SQL код в одном месте
- Бизнес-логика отделена от БД
- Легко менять БД (переписываете только DAO)
- Легко тестировать (можешь подделать DAO в тестах)
- Чистая архитектура

```
class OrderService {
    private OrderDAO orderDAO;

    public void createOrder(Order order) {
        orderDAO.save(order);  // Просто!
    }

    public Order getOrder(Long id) {
        return orderDAO.findById(id);  // Просто!
    }
}

class OrderDAO {
    public void save(Order order) {
        // SQL код только здесь
        conn.prepareStatement("INSERT INTO orders
...");
    }

    public Order findById(Long id) {
        // SQL код только здесь
    }
}
```

ТРАНЗАКЦИИ

Представьте реальный сценарий: вы создаете заказ в магазине. Нужно сделать две операции:

Вставить запись в таблицу orders
Уменьшить количество товара на 5 единиц

Без транзакций:

```
1. INSERT INTO orders ... -- успешно
2. UPDATE products SET stock = stock - 5 ... -- ошибка БД
```

В результате, заказ создан, но товар не уменьшился, образовалась несогласованность данных. Это проблема.

С транзакциями:

```
try {
    conn.setAutoCommit(false);  // Отключаем
автокоммит

    // Операция 1
    pstmt1.execute("INSERT INTO orders ...");

    // Операция 2
    pstmt2.execute("UPDATE products SET stock =
stock - 5 ...");

    // Всё успешно? Сохраняем ВСЕ операции
    conn.commit();

} catch (SQLException e) {
    // Ошибка? Отменяем ВСЕ операции
    conn.rollback();
    throw e;
}
```

Транзакция гарантирует ACID свойства:

A — Atomicity (Атомарность): Либо обе операции успешны, либо ни одна. Не может быть полуформленного заказа.

C — Consistency (Согласованность): Данные остаются в корректном состоянии. Если товара было 100, и мы уменьшили на 5, то стало 95. Не может остаться 100.

I — Isolation (Изоляция): Если два клиента одновременно покупают последний товар в магазине, один из них получит ошибку.

D — Durability (Надежность): Когда вы выполнили commit(), данные на диске. Если сервер упал, заказ остался в базе. Не потеряется.

BEST PRACTICES

Рекомендация 1 — Всегда PreparedStatement.

Мы уже обсудили SQL Injection. Никогда не конкатенируйте параметры в SQL. Всегда используйте '?' и setXxx() методы.

Рекомендация 2 — Try-with-resources.

```
try (Connection conn =
dataSource.getConnection();
     PreparedStatement pstmt =
conn.prepareStatement(sql)) {
    // код
}  // Connection и PreparedStatement
закроются автоматически!
```

Это гарантирует что ресурсы закроются, даже если произойдет исключение.

Рекомендация 3 — Connection Pool.

Не используйте DriverManager.getConnection() каждый раз. Используйте pool:

```
DataSource dataSource =
DatabaseConfig.getDataSource();
Connection conn =
dataSource.getConnection();  // из пула!
```

Рекомендация 4 — DAO Pattern.

SQL код не должен быть в сервисах. Должен быть в отдельных DAO классах.

Рекомендация 5 — Логирование.

Вы должны видеть какие SQL запросы выполняются. Это помогает найти проблемы:

```
logger.info("Executing query: " + sql);
```

Рекомендация 6 — Правильная обработка исключений.

```
// НЕПРАВИЛЬНО
} catch (SQLException e) {
    // молчим
}

// ПРАВИЛЬНО
} catch (SQLException e) {
    logger.error("Database error while
creating order", e);
    throw new RuntimeException("Cannot create
order", e);
}
```

Рекомендация 7 — Batch операции.

Если вы вставляете 1000 заказов, не делайте 1000 отдельных executeUpdate(). Используйте addBatch():

```
for (Order order : orders) {
    pstmt.setLong(1, order.getId());
    pstmt.addBatch();  // В памяти
}
pstmt.executeBatch();  // Один раз в БД!
```

Рекомендация 8 — Индексы.

```
CREATE INDEX idx_status ON orders(status);
```

Без индекса запрос 'SELECT * FROM orders WHERE status = PENDING' сканирует ВСЕ строки в таблице. С индексом находит нужные строки сразу.

ПРИМЕР

5 таблиц:

- **customers** — клиенты магазина.

Колонки: id, name, email, phone.

- **products** — товары.

Колонки: id, name, price, stock_quantity.
Реальные товары: ASUS VivoBook ноутбук за 50 тысяч рублей, LG монитор за 15 тысяч, Corsair клавиатура, Logitech мышь и т.д.

- **orders** — заказы.

Колонки: id, customer_id, status, total_amount, delivery_address, created_at.

- **order_items** — содержимое заказа.

Какие товары в каком заказе, в каких количествах.

- **order_status_history** — аудит лог.

История всех изменений статуса заказа.

4 клиента: Иван, Мария, Николай, Анна.

6 товаров: ASUS ноутбук, LG монитор, Corsair клавиатура, Logitech мышь, подставка для монитора, USB хаб.

5 заказов. Каждый заказ с реальными данными: кто заказал, какие товары, сколько, какой статус (PENDING, CONFIRMED, SHIPPED, DELIVERED).

Структура проекта:

DatabaseConfig.java — инициализирует HikariCP пул. 10 соединений, готовых к работе.

Order.java — модель заказа. Содержит поля: id, customerId, status, totalAmount, deliveryAddress и т.д.

OrderDAO.java — все SQL операции с заказами.

Application.java — entry point. Запускает всё: DatabaseConfig, миграции, демонстрирует CRUD операции.

DatabaseConfig. HikariCP инициализация:

```
public class DatabaseConfig {
    private static HikariDataSource dataSource;

    public static HikariDataSource getDataSource() {
        if (dataSource == null) {
            HikariConfig config = new HikariConfig();
            config.setJdbcUrl("jdbc:postgresql://localh
ost:5432/shop_db");

            config.setUsername("postgres");
            config.setPassword("admin");
            config.setMaximumPoolSize(10);
            config.setConnectionTimeout(20000);

            dataSource = new HikariDataSource(config);
        }
        return dataSource;
    }
}
```

OrderDAO.save().
Создание заказа:

```java
public long save(Order order) {
    String sql = "INSERT INTO orders (customer_id, status, total_amount, " +
                 "delivery_address, created_at) " +
                 "VALUES (?, ?, ?, ?, NOW()) RETURNING id";

    try (Connection conn = dataSource.getConnection();
         PreparedStatement pstmt = conn.prepareStatement(sql)) {

        pstmt.setLong(1, order.getCustomerId());
        pstmt.setString(2, order.getStatus().toString());
        pstmt.setBigDecimal(3, order.getTotalAmount());
        pstmt.setString(4, order.getDeliveryAddress());

        try (ResultSet rs = pstmt.executeQuery()) {
            if (rs.next()) {
                return rs.getLong("id");
            }
        }
    } catch (SQLException e) {
        throw new RuntimeException("Cannot save order", e);
    }
    return -1;
}
```

OrderDAO.findByCustomerId().
Получение заказов:

```
public List<Order> findByCustomerId(long customerId) {
    String sql = "SELECT * FROM orders WHERE customer_id = ? " +
                 "ORDER BY created_at DESC";
    List<Order> orders = new ArrayList<>();

    try (Connection conn = dataSource.getConnection();
         PreparedStatement pstmt = conn.prepareStatement(sql)) {

        pstmt.setLong(1, customerId);

        try (ResultSet rs = pstmt.executeQuery()) {
            while (rs.next()) {
                Order order = mapResultSetToOrder(rs);
                orders.add(order);
            }
        }
    } catch (SQLException e) {
        throw new RuntimeException("Cannot fetch orders", e);
    }

    return orders;
}
```

Application.java. Полный цикл:

```
public static void main(String[] args) {
    try {
        // 1. Инициализируем DataSource
        DataSource ds = DatabaseConfig.getDataSource();
        System.out.println("HikariCP pool инициализирован");

        // 2. Запускаем Flyway миграции
        Flyway flyway = Flyway.configure()
            .dataSource(ds)
            .locations("classpath:db/migration")
            .load();
        flyway.migrate();
        System.out.println("Миграции выполнены");

        // 3. Демонстрация CRUD
        OrderDAO orderDAO = new OrderDAO(ds);

        // CREATE — создаем заказ
        Order order = new Order(
            1L,  // customer_id
            new BigDecimal("1599.99"),  // total_amount
            "ул. Пушкина, 10"  // delivery_address
        );
        long orderId = orderDAO.save(order);
        System.out.println("Заказ создан: ID=" + orderId);
```

```
// READ — получаем по ID
        Order found = orderDAO.findById(orderId).get();
        System.out.println("Найден заказ: " + found);

        // UPDATE — меняем статус
        orderDAO.updateStatus(orderId, OrderStatus.SHIPPED);
        System.out.println("Статус изменен на SHIPPED");

        // LIST — получаем все заказы клиента
        List<Order> customerOrders = orderDAO.findByCustomerId(1L);
        System.out.println("Заказы клиента: " +
customerOrders.size());

        // STATS — считаем статистику
        OrderDAO.OrderStats stats = orderDAO.getStats();
        System.out.println("Всего заказов: " + stats.totalOrders);
        System.out.println("Выручка: " + stats.totalRevenue);

        // DELETE — удаляем заказ
        orderDAO.delete(orderId);
        System.out.println("Заказ удален");

    } finally {
        DatabaseConfig.closeDataSource();
    }
}
```

Спасибо за внимание!