

# SQL




Артём  
Романов



## Артём Романов

Инженер по обеспечению качества

 [romanov-artyom](#)

 [Романов Артём](#)

 [@Artem\\_Telegram](#)



# План занятия

1. [Введение](#)
2. [Задача](#)
3. [SOL](#)
4. [JDBC](#)
5. [DBUtils](#)
6. [DBUnit](#)
7. [Итоги](#)



# Введение



## Постоянное хранилище данных

Приложениям и сервисам требуются постоянные хранилища данных, обеспечивающие сохранность и консистентность информации при перезапусках и различных сбоях.

Можно, конечно, хранить всё в файликах, но как вы думаете, чем плох такой подход?



# Функциональность

Зачастую требуется гораздо большая функциональность, чем просто хранение данных:

- структура и консистентность информации — защита от того, что приложение внесёт неправильные данные или в неправильном формате;
- конкурентный доступ — возможность не только «одновременно» читать, но и «одновременно»\* модифицировать;
- транзакционность — выполнение неаотомарных операций в виде одного атомарного блока;
- и другие.

---

Примечание:\* речь не всегда именно об истинной одновременности, некая её эмуляция, незаметная пользователю тоже сойдёт.



# Базы данных

Необходимость данной функциональности привела к тому, что появился целый класс продуктов, решающих эту задачу.

Само хранилище данных называют базой данных (БД), а систему, управляющую им — СУБД.

Достаточно часто оба эти термина используют как взаимозаменяемые, поэтому мы акцентироваться на этом тоже не будем.



# История

История развития у БД достаточно богатая, она привела к тому, что наиболее распространёнными на сегодняшний день являются SQL базы данных.

Поэтому в рамках этой лекции мы будем рассматривать только их.

При этом другие не-SQL БД также вполне себе используются и достаточно популярны.





# Задача



# Задача

Напомню задачу, которую мы решаем — протестировать:

- операцию перевода денег с карты одного клиента на карту другого клиента через Интернет Банк;
- операцию перевода денег с одной карты на другую одного и того же клиента через Интернет Банк.



## Доступ к СУБД

На этот раз разработчики решили, что не будут предоставлять нам никакого сервисного режима, хардкодить данные и т.д.

Сказали: «раз уж вы умеете разворачивать базу данных в Docker'е, вот вам описание нужных таблиц, там всё и смотрите».

Вопрос к аудитории: какие риски прямого доступа к БД из тестов вы помните?



## Доступ к СУБД

Соответственно, наша задачи на сегодняшнюю лекцию — рассмотреть:

- какие инструменты у нас есть для разворачивания БД;
- как выполнять в БД запросы;
- как взаимодействовать в БД из Java (от простого к сложному).

Имея этот набор знаний, вы без труда сможете решить задачу в формате, описанном на предыдущем слайде.



SQL



# SQL

Прежде, чем мы перейдём непосредственно к работе, нам необходимо вспомнить некоторые основы языка SQL.

SQL – общее название языка, используемого в SQL базах данных.

Зачастую он подразделяется на:

- DDL – Data Definition Language;
- DML – Data Manipulation Language;
- DRL (или DQL) – Data Retrieval (Query) Language.

# Диалекты

На язык SQL разрабатываются стандарты, но ключевое, что нам нужно знать — хотя большинство БД и поддерживает стандарты, но почти каждая обладает собственным диалектом.

Диалект — это некоторые особенности синтаксиса конкретной БД, возможно, не поддерживаемые в других.

Например, в зависимости от БД ключевое слово может писаться в различных вариациях:

- `AUTOINCREMENT`;
- `AUTO_INCREMENT`;
- отсутствовать вообще (не поддерживаться).

Но общие выражения, в целом, будут похожи.

# MySQL

Для экспериментов мы будем использовать MySQL в Docker контейнере.

Если у вас отсутствует возможность использовать Docker, то воспользуйтесь Docker Playground.

`docker-compose.yml`:

```
version: '3.7'
services:
  mysql:
    image: mysql:8.0.18
    ports:
      - '3306:3306'
    volumes:
      - ./data:/var/lib/mysql
    environment:
      - MYSQL_RANDOM_ROOT_PASSWORD=yes
      - MYSQL_DATABASE=app
      - MYSQL_USER=app
      - MYSQL_PASSWORD=pass
```



# Подключаемся

БД бывают достаточно разные — от файловых, которые подключаются в виде обычно библиотеки, до клиент-серверных.

Для серверных чаще всего в состав контейнера входит и инструмент командной строки (CLI), который позволяет подключаться к серверу.

В случае MySQL этот инструмент называется `mysql`:

```
docker-compose exec mysql mysql -u app -d app -p
```

Первое `mysql` — это имя сервиса из файла Docker Compose, второе — имя исполняемого файла.

Флаг `-u app` — указание пользователя, `app` — база данных, `-p` — подключение с паролем



## mysql

Клиент каждой СУБД обладает собственным набором команд, синтаксисом и особенностями.

Узнать о работе того или иного клиента можно на официальных страницах справки.

# GUI

Если вы плохо владеете консольным клиентом, то на выход приходят менеджеры с графическим интерфейсом.

Например:

- [DBeaver](#);
- клиент, встроенный в IDEA Ultimate;
- и другие.

Кроме того, вы можете использовать образ [Adminer](#) (см.документацию на странице образа MySQL).

# SQL

В качестве учебной задачи мы рассмотрим моделирование минимальной системы Интернет Банка, в которой есть пользователи (с логинами и паролями), подтверждение по SMS, и те же операции перевода денег.

Напоминаем, что в рамках SQL мы будем рассматривать структуру БД как систему взаимосвязанных таблиц.

Таблицы состоят из столбцов и строк:

- Столбцы имеют конкретный тип и ограничения и определяют структуру информации;
- Строки представляют из себя хранимые данные.

# DDL

Важно: сначала создаётся сама таблица и её структура и только потом в неё можно вносить данные.

При этом структуру таблицы вы можете менять и после внесения данных в неё.

Структура информации определяется подмножеством DDL (Data Definition Language):

```
CREATE TABLE table_name (  
    column_name column_type column_options,  
    column_name column_type column_options,  
    ...  
    CONSTRAINT constraint_name constraint_expression  
);
```

Полный синтаксис можно посмотреть [на странице документации](#).

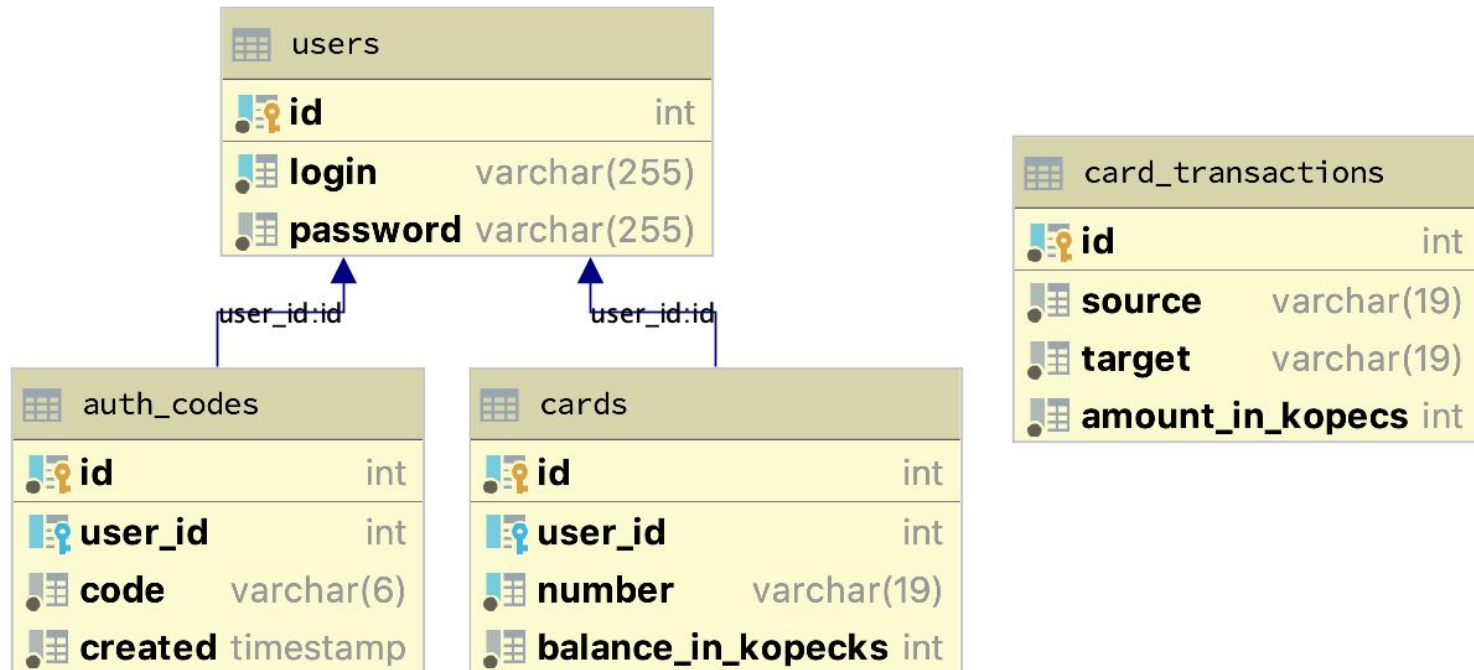
```
CREATE TABLE users
(
    id          INT PRIMARY KEY AUTO_INCREMENT,
    login       VARCHAR(255) UNIQUE NOT NULL,
    password    VARCHAR(255)        NOT NULL
);

CREATE TABLE cards
(
    id          INT PRIMARY KEY AUTO_INCREMENT,
    user_id     INT NOT NULL,
    number      VARCHAR(19) UNIQUE NOT NULL,
    balance_in_kopecks INT          NOT NULL DEFAULT 0,
    FOREIGN KEY (user_id) REFERENCES users (id)
);

CREATE TABLE auth_codes
(
    id          INT PRIMARY KEY AUTO_INCREMENT,
    user_id     INT NOT NULL,
    code       VARCHAR(6) NOT NULL,
    created     TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users (id)
);

CREATE TABLE card_transactions
(
    id          INT PRIMARY KEY AUTO_INCREMENT,
    source     VARCHAR(19) NOT NULL,
    target     VARCHAR(19) NOT NULL,
    amount_in_kopecks INT          NOT NULL,
    created     TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

# Cxema



# Ключевые моменты

- `NOT NULL` — в колонку нельзя записать `NULL`;
- `CHECK ( expression )` — перед вставкой/обновлением проверяется истинность выражения;
- `DEFAULT expression` — значение колонки по умолчанию (если иное не указано);
- `UNIQUE` — ограничение уникальности (не может быть других строк с таким же значением);
- `PRIMARY KEY` — первичный ключ (уникальная идентификация строки среди всех строк);
- `REFERENCES table_name (columns)` — внешний ключ (ссылка на ту же таблицу, либо на другую).





# NULL

NULL — специальный маркер в рамках СУБД, означающий отсутствие значения.

Ключевое: поле, помеченное как NULL — удовлетворяет требованиям уникальности, т.к. NULL ничему не равен, включая самому себе.

Для работы с NULL используются специальные выражения: IS NULL, IS NOT NULL.



## DROP & ALTER TABLE

- `DROP TABLE table_name` — удаление таблицы со всеми данными. **Не используйте на Production!**
- `ALTER TABLE table_name ...` — изменение структуры таблицы. **Не используйте на Production!**

# Создание таблицы

**Q:** В какой момент создаётся таблица? Должны ли мы создавать её руками сами?

**A:** Всё зависит от проекта, варианты:

- схему нужно самому создавать вручную или запуская скрипт инициализации;
- ПО само при запуске создаст необходимую структуру данных;
- и другие варианты.

Т.е. вам нужно обязательно уточнять этот момент, т.к. без базы вы ничего не протестируете.



# DML

К ключевым запросам манипуляции данными относятся:

- `INSERT` — вставка данных;
- `UPDATE` — обновление данных;
- `DELETE` — удаление данных.

# INSERT

Самые распространённые форма:

```
INSERT INTO users (id, login, password) VALUES (1, "vasya", "password");
INSERT INTO users (id, login, password) VALUES (2, "petya", "password");
INSERT INTO cards (id, user_id, number, balance_in_kopecks) VALUES
(1, 1, "5559000000000001", 1000000),
(2, 1, "5559000000000002", 1000000);

INSERT INTO card_transactions (source, target, amount_in_kopecks) VALUES
("5559000000000001", "5559000000000002", 10000);
```

Указывается таблица, в которую будет производиться вставка данных и столбцы (для остальных столбцов устанавливается либо автогенерируемое значение, либо default'ное, либо NULL).

**Обратите внимание:** пароли почти никогда не хранятся в открытом виде (с номерами карт возможны варианты).

# UPDATE

Самые распространённая форма:

```
UPDATE cards SET balance_in_kopecks = balance_in_kopecks - 10000  
WHERE number = "5559000000000001";
```

```
UPDATE cards SET balance_in_kopecks = balance_in_kopecks + 10000  
WHERE number = "5559000000000002";
```

Естественно такие операции (эти две и предыдущий `INSERT`) должны заворачиваться в транзакцию.

Часть с `WHERE` не является обязательной, но если вы её опустите — обновятся все строки в таблице.



# DELETE

Самая распространённая форма:

```
DELETE FROM auth_codes WHERE created < NOW() - INTERVAL 5 MINUTES;
```

# WHERE

В части `WHERE` можно писать различные логические выражения, определяющие, подходит строка для модификации/удаления или нет.

Вообще говоря, `WHERE` не обязателен в запросах `UPDATE` и `DELETE` и если его не указать, то будут обновлены/удалены все записи. **Будьте с этим осторожны!**

В `WHERE` поддерживаются:

- операторы сравнения: `<`, `>`, `<=`, `>=`, `<>` (синоним `!=`), `=`;
- проверка на вхождение в перечисляемый список значений или интервал: `IN (A, B, C)`, `BETWEEN A AND B`;
- проверка на `NULL`: `IS NULL`, `IS NOT NULL`;
- логические операторы `AND`, `OR`, `NOT`;
- и другие возможности в зависимости от используемой БД.



# DQL (DRL)

Несмотря на то, что синтаксис оператора SELECT наиболее богатый в рамках всего SQL, мы рассмотрим лишь ключевые возможности.

```
-- выборка всех столбцов и всех строк из таблицы users (осторожно на больших таблицах)
SELECT * FROM users;
-- выборка только определённых столбцов
SELECT id, login FROM users;
-- выборка по условию
SELECT balance_in_kopecks FROM cards WHERE number = "5559000000000002";
-- вычисляемые столбцы
SELECT balance_in_kopecks / 100 AS balance_in_rub FROM cards
WHERE number = "5559000000000002";
```

# Агрегирующие запросы

Агрегирующие запросы позволяют вам получать результат выполнения агрегирующей функции над группой строк.

Например: найти количество, среднее, min, max, сумму:

```
-- группируем по всей таблице
SELECT max(cards.balance_in_kopecks) FROM cards;
-- сначала фильтруем по user_id, потом группируем
SELECT sum(balance_in_kopecks) FROM cards WHERE user_id = 1;

-- группируем по user_id (количество карт каждого пользователя)*
-- важно: будут посчитаны карты в привязке к пользователю (т.е. petya не отобразится)
SELECT count(*) FROM cards GROUP BY user_id;
```

---

Примечание\*: в наборе столбцов можно использовать только те столбцы, по которым идёт группировка, + результаты агрегирующих функций.



## Дополнительно

Мы настоятельно рекомендуем вам ознакомиться с вопросами получения данных одновременно из нескольких таблиц:

- подзапросами;
- объединениями (JOIN'ами).

Кроме того, желательно иметь представление о транзакциях и VIEW.



# JDBC



# Автоматизация

Это всё здорово: что мы можем подключиться к консоли и там выполнять запросы.

Но нам же нужно автоматизировать\*:

- мы хотим возможность выставлять нужное состояние в БД;
- мы хотим возможность проверять состояние в БД после действий в интерфейсе.

---

Примечание:\* естественно, вы должны помнить о всех рисках подобного подхода.

# JDBC

В стандартной библиотеке Java уже определён стандарт взаимодействия с SQL базами данных.

Типы этого стандарта располагаются в пакете `java.sql`.

Каждый производитель БД предоставляет собственный драйвер, позволяющий подключаться к БД и соответствующий стандарту.

```
dependencies {  
    testImplementation 'org.junit.jupiter:junit-jupiter:5.5.2'  
    testImplementation 'mysql:mysql-connector-java:8.0.18'  
    testImplementation 'com.github.javafaker:javafaker:1.0.1'  
}
```

# Ключевые типы

- `DriverManager` – простая реализация подключения к базе через драйвер;
- `Connection` – абстракция подключения;
- `Statement/PreparedStatement` – абстракция выполняемого запроса;
- `ResultSet` – абстракция набора получаемых результатов;
- `SQLException` – исключение.

# Общий алгоритм

1. Через `DriverManager` получаем `Connection`;
2. Через `Connection` создаём statement'ы (`Statement` для запросов без параметров, `PreparedStatement` для запросов с параметрами)\*
3. Выполняем запросы:
  - `execute` – general purpose;
  - `executeUpdate` – для UPDATE/INSERT/DELETE (возвращает количество затронутых строк);
  - `executeQuery` – для SELECT (возвращает `ResultSet`);
4. Закрываем всё: `Connection`, `Statement` / `PreparedStatement`, `ResultSet`.

---

Примечание:\* вообще говоря, это больше связано с безопасностью, но лучше сразу привыкнуть делать именно так.



```
// Пример вставки данных
@BeforeEach
void setUp() throws SQLException {
    val faker = new Faker();
    val dataSQL = "INSERT INTO users(login, password) VALUES (?, ?)";

    try (
        val conn = DriverManager.getConnection(
            "jdbc:mysql://192.168.99.100:3306/app", "app", "pass"
        );
        val dataStmt = conn.prepareStatement(dataSQL);
    ) {
        dataStmt.setString(1, faker.name().username());
        dataStmt.setString(2, "password");
        dataStmt.executeUpdate();
        dataStmt.setString(1, faker.name().username());
        dataStmt.setString(2, "password");
        dataStmt.executeUpdate();
    }
}
```

# Вопросы

**Q:** Почему мы не вычищаем все таблицы, а генерируем данные?

**A:** Это хороший вопрос. Как только мы установили ограничения внешнего ключа на некоторые таблицы, то мы не можем просто так удалять данные. Например, мы не можем удалить пользователей, пока у них есть карты.

Есть несколько стратегий решения этой проблемы:

- очищать таблицы в правильном порядке
- заворачивать всю очистку в транзакцию
- каждый раз воссоздавать структуру таблиц с нуля (DROP + CREATE)
- генерировать уникальные данные

У каждого есть минусы и плюсы.

# Вопросы

**Q:** Что за `try (...)`?

**A:** Это `try-with-resources`, конструкция языка, позволяющая не писать нам вызов метода `close`. Компилятор сам сгенерирует правильный вызов закрытия ресурсов.

**Q:** А вопросики?

**A:** Это placeholder'ы — точки в `PreparedStatement`, в которые можно подставлять параметры с помощью `set*`.

```
// Пример чтения данных
@Test
void stubTest() throws SQLException {
    val countSQL = "SELECT COUNT(*) FROM users;";
    val cardsSQL = "SELECT id, number, balance_in_kopecks FROM cards WHERE user_id = ?;";

    try (
        val conn = DriverManager.getConnection(
            "jdbc:mysql://192.168.99.100:3306/app", "app", "pass"
        );
        val countStmt = conn.createStatement();
        val cardsStmt = conn.prepareStatement(cardsSQL);
    ) {
        try (val rs = countStmt.executeQuery(countSQL)) {
            if (rs.next()) {
                // выборка значения по индексу столбца (нумерация с 1) – лучше выбирать по имени
                val count = rs.getInt(1);
                // TODO: использовать
                System.out.println(count);
            }
        }

        cardsStmt.setInt(1, 1);
        try (val rs = cardsStmt.executeQuery()) {
            while (rs.next()) {
                val id = rs.getInt("id");
                val number = rs.getString("number");
                val balanceInKopecks = rs.getInt("balance_in_kopecks");
                // TODO: сложить всё в список
            }
        }
    }
}
```



## Эмоции

**Q:** Это ужасно!!! Неужели каждый раз придётся это делать?

**A:** Конечно же нет, чаще всего собственные обёртки для работы с JDBC и используют их (вспомните пример с `DataHelper`). Но один раз их всё-таки придётся написать или воспользоваться уже готовыми решениями.



# Apache Commons DBUtils

# DbUtils

Apache Commons — это набор библиотек, предоставляющих различные удобства, которых не хватало (или не хватает) в стандартной библиотеке Java.

Apache Commons DbUtils — набор классов, делающих работу с JDBC проще.

```
dependencies {  
    ...  
    testImplementation 'commons-dbutils:commons-dbutils:1.7'  
}
```

## Ключевые типы

- `QueryRunner` — «исполнитель запросов»;
- `ResultSetHandler` — функциональный интерфейс, преобразующий `ResultSet` в объект нужного типа;
- `ScalarHandler` — реализация интерфейса `ResultSetHandler`, отображающая первую «колонку» его в объект нужного типа;
- `BeanHandler` — реализация интерфейса `ResultSetHandler`, отображающая первую «строку» его в объект нужного типа;
- `BeanListHandler` — реализация интерфейса `ResultSetHandler`, отображающая все «строки» в объекты нужного типа.

Настоятельно рекомендуем ознакомиться с [JavaDoc'ами](#) на эти классы.





```
// Пример вставки данных
@BeforeEach
void setUp() throws SQLException {
    val faker = new Faker();
    val runner = new QueryRunner();
    val dataSQL = "INSERT INTO users(login, password) VALUES (?, ?)";

    try (
        val conn = DriverManager.getConnection(
            "jdbc:mysql://192.168.99.100:3306/app", "app", "pass"
        );

    ) {
        // обычная вставка
        runner.update(conn, dataSQL, faker.name().username(), "pass");
        runner.update(conn, dataSQL, faker.name().username(), "pass");
    }
}
```

```
// Пример чтения данных
@Test
void stubTest() throws SQLException {
    val countSQL = "SELECT COUNT(*) FROM users;";
    val usersSQL = "SELECT * FROM users;";
    val runner = new QueryRunner();

    try (
        val conn = DriverManager.getConnection(
            "jdbc:mysql://192.168.99.100:3306/app", "app", "pass"
        );
    ) {
        val count = runner.query(conn, countSQL, new ScalarHandler<>());
        System.out.println(count);
        val first = runner.query(conn, usersSQL, new BeanHandler<>(User.class));
        System.out.println(first);
        val all = runner.query(conn, usersSQL, new BeanListHandler<>(User.class));
        System.out.println(all);
    }
}
```

# Реализация

Важно: вы всегда можете посмотреть внутрь классов реализации и увидеть, как там всё устроено:

```
// ScalarHandler
public T handle(ResultSet rs) throws SQLException {

    if (rs.next()) {
        if (this.columnName == null) {
            return (T) rs.getObject(this.columnIndex);
        }
        return (T) rs.getObject(this.columnName);
    }
    return null;
}
```



## DbUtils

Вы должны понимать, что мы только для простоты восприятия написали всё сразу в коде тестов.

Вам же нужно все подобные «вспомогательные» методы для работы с данными не «размазывать» по тестам, а выносить в отдельный класс, скрывая конкретную реализацию.



# DbUnit & Database Rider



## DbUnit & Database Rider

Естественно, это лишь первый шаг на пути работы с БД из тестов.

Существуют гораздо более продвинутые инструменты, позволяющие вам практически целиком контролировать состояние БД.

Мы рекомендуем вам по возможности обратить внимание на проекты [DbUnit](#) и [Database Rider](#).



# Итоги



## Итоги

Сегодня мы рассмотрели возможности по работе с БД из Java, в том числе библиотеку DbUtils.

Всегда с осторожностью подходите к интеграции на этом уровне, т.к. схема БД может меняться, что может повлечь за собой нарушение логики тестов.

Кроме того, вставка некорректных данных (не все логические связи может контролировать БД) может привести к изменению в работе SUT и, как следствие, отсутствию доверия к результатам тестов.



# Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задавайте **в чате** мессенджера Slack.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

**Важно:** начиная с сегодняшнего дня вам нужно сдавать все домашние задания только с использованием PageObject.

Домашние задания, которые реализованы без использования PageObject будут сразу отправляться на доработку.

**Задавайте вопросы и  
пишите отзыв о лекции!**

**Артем Романов**



[romanov-artyom](https://romanov-artyom.netology.ru)



[Романов Артём](#)



[@Artem\\_Telegram](#)