

JAVA

ADVANCED



ЧТО БЫЛО В ПРОШЛЫЙ РАЗ?

- Некоторые идеи чистого кода
 - YAGNI
 - DRY
 - KISS
 - SOLID
- Слоеная архитектура приложений
- Начало знакомство со Spring

ЧТО БУДЕТ СЕГОДНЯ?

- Низкоуровневая работа с БД. JDBC
- База данных H2
- DataSource
- Продолжаем про Spring
- Еще немного Git

JAVA DATABASE CONNECTIVITY (JDBC)

- платформенно-независимый стандарт взаимодействия Java-приложений с различными СУБД
- Низкоуровневое, сложное и многословное средство работы с базами данных в Java
- Самое быстрое
- Более удобные инструменты “накручены” поверх JDBC

JDBC ДРАЙВЕР

- Прежде чем идти в БД, нам нужно подключить драйвер, соответствующий той БД, которую используем
- Обычно, эти драйверы предоставляются как библиотеки и мы можем просто стянуть их с репозитория Maven
 - [Драйвер для PostgreSQL](#)
 - [Драйвер для MariaDB/MySQL](#)
 - [Драйвер для Oracle](#)

БАЗА ДАННЫХ H2

- Является базой данных типа “in-memory”
- Однако, может работать и в режиме файла
- Широко применяется в юнит-тестах
- Подключим ее драйвер

Драйвер H2

```
<dependency>  
  <groupId>com.h2database</groupId>  
  <artifactId>h2</artifactId>  
  <version>2.3.232</version>  
</dependency>
```

- не забываем удалить score, так как нам она нужна не только в тестах

СТАТИЧЕСКИЕ ЧЛЕНЫ КЛАССА

- Ключевым словом `static` в Java обозначаются элементы самого класса (а не объекта)
 - Поля
 - Методы
 - Вложенные классы
 - Статические блоки инициализации
 - Конструкторы
- Эти члены класса могут быть использованы без создания экземпляра класса
- Часто используются в утилитных классах (к примеру: `System`, `Math`)

СТАТИЧЕСКИЕ ЧЛЕНЫ КЛАССА

КОНСТАНТЫ

- В Java нет отдельного слова для определения константы, но ее можно определить, задав ключевые слова `static final`
- При этом, принято, чтобы имя такого поля писалось большими буквами, разделенными нижним подчеркиванием

```
private static final double THRESHOLD_VALUE = 75.0;
```


ИНИЦИАЛИЗАЦИЯ

- Статические поля инициализируются только при создании первого объекта или при обращении к его статическим данным
- Конструктор неявно является статическим методом, так что при его вызове предыдущий пункт тоже выполняется

БЛОК ИНИЦИАЛИЗАЦИИ

(СТАТИЧЕСКИЙ)

- Язык Java позволяет сгруппировать несколько действий по инициализации объектов `static` в статическом блоке инициализации
- Этот блок выполняется только при создании первого объекта или при обращении к его статическим данным.

```
public class InitExample {  
    static int i;  
    static {  
        i = 47;  
    }  
}
```

БЛОК ИНИЦИАЛИЗАЦИИ

(НЕСТАТИЧЕСКИЙ)

- Помимо конструкторов, можно определить блоки нестатической инициализации объекта класса
- Секция инициализации экземпляра выполняется раньше любых конструкторов

```
public class InitExample {  
    private final int i;  
    private int j = 0;  
  
    {  
        i = 10;  
        j++;  
    }  
}
```

КЛАСС DAO

- Расшифровывается, как Data Access Object
- В нем прячется низкоуровневая логика по работе с БД и преобразованием полученных строк из таблиц в объекты Java.
И наоборот
- В нашем примере (ветка **lesson_4.1**) это класс TaskJdbcDao в пакете `ru.akhitev.dao`
- Добавим в него сразу загрузку h2 драйвера

```
@Repository
public class TaskJdbcDao implements TaskRepository {
    ...
    static {
        try {
            Class.forName("org.h2.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
    ...
}
```

ru.akhitev.dao.TaskJdbcDao

JDBC

СОЕДИНЕНИЕ К БД

- Используя `DriverManager` мы создаем `Connection` к базе данных
 - В нашем случае это база данных `task`
 - у которой задаем логин и пароль `admin/admin`
- При этом помним, что соединение - дефицитный ресурс, требующий немалых затрат на установление.
- Как правило, база данных создает поток исполнения и порождает дочерний процесс для каждого соединения.
- Но количество параллельных соединений, как правило, ограничено, поскольку большое число установленных соединений замедляет работу базы данных
- Как итог - не забываем закрывать соединения!

```
@Repository
public class TaskJdbcDao implements TaskRepository {
    ...
    private Connection getConnection() throws SQLException {
        return DriverManager.getConnection("jdbc:h2:mem:task;DB_CLOSE_DELAY=-1",
            "admin", "admin");
    }

    private void closeConnection(Connection connection) {
        if (connection == null) return;
        try {
            connection.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

ru.akhitev.dao.TaskJdbcDao

JDBC

STATEMENT

- Класс, позволяющий выполнять запросы к БД
- Существует простой Statement, но его использовать не стоит
- Правильнее использовать PreparedStatement, так как он нейтрализует sql-инъекции, в то время, как Statement к ним уязвим
 - Еще PreparedStatement позволяет в удобном виде задавать параметры запроса
- Есть еще CallableStatement, но он используется для вызова хранимых процедур
- Не забываем закрывать Statement!

```
@Repository
public class TaskJdbcDao implements TaskRepository {
    private static final String DDL_QUERY = ""
        "Create table tasks (ID int primary key, code varchar(10), title varchar(50))
        """;
    public void initDataBase() {
        Connection connection = null;
        try {
            connection = getConnection();
            PreparedStatement statement = connection.prepareStatement(DDL_QUERY);
            statement.executeUpdate();
            statement.close();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            closeConnection(connection);
        }
    }
    ...
}
```

JDBC

А ЕСЛИ ЗАПРОС С ПАРАМЕТРАМИ?

- В самом запросе для PreparedStatement пишем знаки вопроса ?
- А позже указываем, какие значения должны быть, указывая их порядок

```
@Repository
public class TaskJdbcDao implements TaskRepository {
    ...
    private static final String INSERT_TASK_QUERY = ""
        INSERT INTO tasks(code, title) VALUES (?, ?)
        "";
    ...
    @Override
    public void add(Task task) {
        Connection connection = null;
        try {
            connection = getConnection();
            PreparedStatement statement = connection.prepareStatement(INSERT_TASK_QUERY);
            statement.setString(1, task.getCode());
            statement.setString(2, task.getTitle());
            statement.execute();
            statement.close();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            closeConnection(connection);
        }
    }
    ...
}
```

ru.akhitev.dao.TaskJdbcDao

JDBC

А ЕСЛИ НУЖНО ЗАПРОСИТЬ ДАННЫЕ?

- Для этого выполняется запрос методом `executeQuery()` у `statement`-а
- Он возвращает `ResultSet`
- Этот полученный `ResultSet` нужно перебирать и вытаскивать из него данные по номеру столбца в запросе или по имени
- И да, его тоже надо закрывать :)

```
@Repository
public class TaskJdbcDao implements TaskRepository {
    ...
    @Override
    public List<Task> findAll() {
        Connection connection = null;
        List<Task> tasks = new ArrayList<>();
        try {
            connection = getConnection();
            PreparedStatement statement = connection.prepareStatement(FIND_ALL_QUERY);
            ResultSet resultSet = statement.executeQuery();
            while (resultSet.next()) {
                Task task = new Task(resultSet.getString("code"),
                    resultSet.getString("title"));
                tasks.add(task);
            }
            resultSet.close();
            statement.close();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            closeConnection(connection);
        }
        return tasks;
    }
}
```


DATASOURCE

- Если посмотреть на класс `TaskJdbcDao`, то заметим, что нарушается SOLID
 - Мы напрямую лезем в `DriverManager`,
 - храним в DAO-классе данные для подключения
- Есть класс, который помогает определить все данные для работы с БД и потом внедрять его как зависимость - `DataSource`
- Есть отличная реализация `BasicDataSource` от Apache
- Чтобы ее подключить, нужно добавить зависимость [apache commons-dbcp2](#)
- Переключимся на ветку **lesson_4.2**, где она уже есть

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-dbcp2</artifactId>
  <version>2.13.0</version>
</dependency>
```

pom.xml

BASICDATASOURCE

ОПРЕДЕЛЯЕМ В КОНФИГУРАЦИИ SPRING

```
@Configuration
@ComponentScan(basePackageClasses = Application.class)
public class SpringConfig {
    private static final String DB_DRIVER_CLASS_NAME = "org.h2.Driver";
    private static final String DB_URL = "org.h2.Driver";
    private static final String DB_USER_NAME = "sa";
    private static final String DB_PASSWORD = "sa";

    @Bean
    @Lazy
    public DataSource dataSource() {
        try {
            BasicDataSource dataSource
                = new BasicDataSource();
            dataSource.setDriverClassName(DB_DRIVER_CLASS_NAME);
            dataSource.setUrl(DB_URL);
            dataSource.setUsername(DB_USER_NAME);
            dataSource.setPassword(DB_PASSWORD);
            return dataSource;
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

- Мы видим новую для нас аннотацию @Bean
 - Это альтернативный способ создания bean-ов в Spring
 - В этом случае, мы можем сами сконструировать объект и передать его в контекст через return
 - Эта аннотация обрабатывается только классе, помеченном аннотацией @Configuration
 - По умолчанию, имя bean-а в контексте Spring будет совпадать с именем метода, помеченного @Bean
- Аннотация @Lazy показывает Spring-у, что этот bean нужно инициализировать “лениво”, то есть тогда, когда в нем появится необходимость

ЧТО ТЕПЕРЬ С НАШИМ DAO?

```
@Repository
public class TaskJdbcDao implements TaskRepository {
    ...
    private final DataSource dataSource;

    public TaskJdbcDao(DataSource dataSource) {
        this.dataSource = dataSource;
        initDataBase();
    }
    ...
}
```

ru.akhitev.dao.TaskJdbcDao

- Это уже значительно сокращает код
- У нас пропадает блок статической инициализации и лишние методы
- Но нельзя ли применить тут try-with-resources и автоматом закрывать Connection, Statement, ResultSet?

TRY-WITH-RESOURCES

- С Java 7 появилась конструкция `try-with-resources`
- В нее мы передаем объект, реализующий `AutoCloseable`
- Java сама закрывает соединение в конце работы блока и сама обрабатывает исключения, которые возникают в этот момент

TRY-WITH-RESOURCES И JDBC

```
@Repository
public class TaskJdbcDao implements TaskRepository {
    ...
    @Override
    public List<Task> findAll() {
        List<Task> tasks = new ArrayList<>();
        try (Connection connection = dataSource.getConnection();
            PreparedStatement statement = connection.prepareStatement(FIND_ALL_QUERY);
            ResultSet resultSet = statement.executeQuery();) {
            while (resultSet.next()) {
                Task task = new Task(resultSet.getString("code"),
                    resultSet.getString("title"));
                tasks.add(task);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return tasks;
    }

    @Override
    public Optional<Task> findByCode(String code) {
        List<Task> tasks = new ArrayList<>();
        try (Connection connection = dataSource.getConnection();
            PreparedStatement statement = connection.prepareStatement(FIND_BY_CODE_QUERY);) {
            statement.setString(1, code);
        }
    }
}
```

ru.akhitev.dao.TaskJdbcDao

GIT

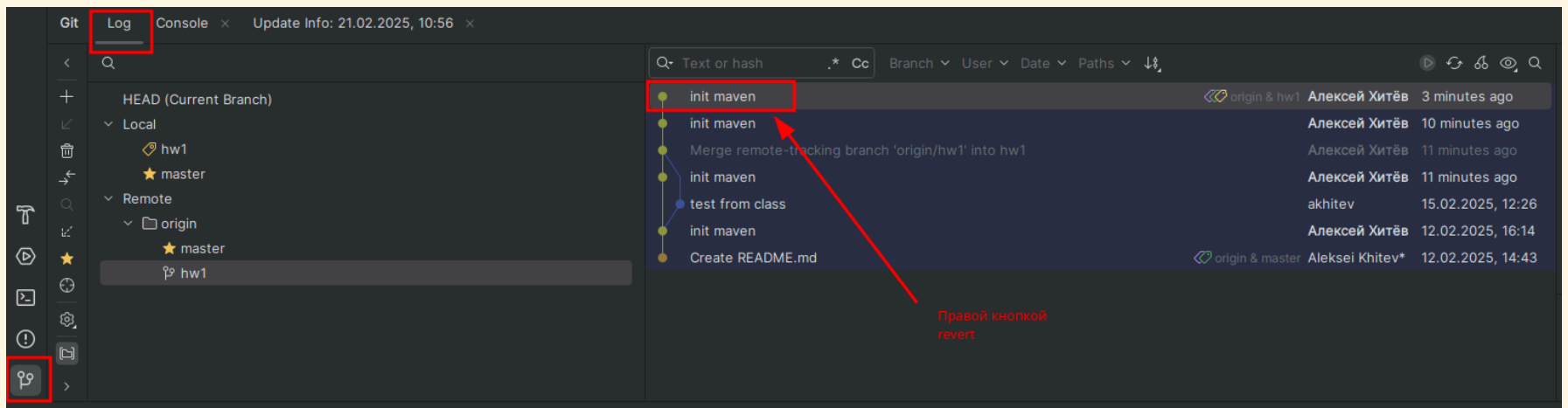
ФАЙЛ .GITIGNORE

- Чтобы при коммите не видеть кучу мусора (`target`, `.idea`) и прочее, обычно создается файл `.gitignore`, в котором указываются паттерны пути и файлы, которые нужно игнорировать
- Кроме того, помогает не коммитить лишнее
- Создадим такой в своем корне проекта с таким содержимым
- точка вначале имени файла обязательна

```
.idea/**  
target/**
```

ОТКАТ КОММИТА

- Чтобы откатить commit, можно воспользоваться командой `revert`



- После этого не забываем выполнить `push`
- Эта команда сделает контр-коммит, который откатит изменения, внесенные в выбранном коммите

GIT

УБИРАЕМ ФАЙЛЫ ИЗ РЕПОЗИТОРИЯ

- Команда `git rm --cache <файл>` позволяет удалить из репозитория ненужный файл (локально он останется жив)
- Команду можно натравить сразу на каталог, использовав ключ - r

```
$ git rm -r --cache ./target
rm 'target/classes/ru/akhitev/App.class'
$ git commit -m "removed useless"
[hw1 bf7bafa] removed useless
1 file changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 target/classes/ru/akhitev/App.class
$ git push
```


ЗАДАНИЕ 4.1

- добавить реализацию репозитория на базе JDBC
- Настроить DataSource в контексте Spring

ЗАДАНИЕ 4.1

Полный текст задания с критериями приемки лежит в репозитории

https://github.com/aleksei-khitev/java_adv_22b01_02_e/blob/main/tasks/task_4_1.pdf

В СЛЕДУЮЩЕЙ СЕРИИ

- Spring JDBC
- Swing
- Mockito
- Многомодульные проекты Maven

ОБЩИЕ РЕСУРСЫ

- [Таблица с прогрессом](#)
Пароль: student25
- [Группа в Telegram](#)
- Репозиторий с материалами
https://github.com/aleksei-khitev/java_adv_22b01_02_e

ПОЛЕЗНЫЕ МАТЕРИАЛЫ

- [Документация Java: про JDBC](#)
- [Документация Apache: про commons-dbcp2](#)
- [Статья на Baeldung: про Spring Bean](#)

