УДК 004.415

РЕАЛИЗАЦИЯ ЗАГРУЗКИ НАСТРОЕК КОНФИГУРАЦИИ ИЗ БАЗЫ ДАННЫХ В JAVA-SPRING ПРИЛОЖЕНИИ

Сафина Г.Ф.

к. ф.-м. н, доцент,

ФГБОУ ВО «Уфимский университет науки и технологий», Нефтекамский филиал, Нефтекамск, Россия

Коняев Ю.С.

студент,

ФГБОУ ВО «Уфимский университет науки и технологий», Нефтекамский филиал, Нефтекамск, Россия

Аннотация

Статья посвящена реализации хранения настроек приложения в базе данных при разработке на языке программирования Java Spring. Продемонстрированы безопасные функции чтения и записи значений, инициализации настроек по умолчанию, а также их восстановления. Приведены практические примеры предложенного пошаговой реализации хранения настроек приложения.

Ключевые слова: Java, Spring, базы данных, настройки приложения, конфигурация, кэш, микросервисная архитектура.

IMPLEMENTATION OF LOADING CONFIGURATION SETTINGS FROM A DATABASE IN A JAVA-SPRING APPLICATION

Safina G.F.

PhD, Associate Professor,

Neftekamsk branch of the Ufa University of Science and Technology,

Neftekamsk, Russia

Konyaev Yu.S.

student,
Neftekamsk branch of the Ufa University of Science and Technology,
Neftekamsk, Russia

Abstract

The article is devoted to the implementation of storing application settings in a database when developing in the Java Spring programming language. Demonstrated the safe functions of reading and writing values, initializing default settings, as well as restoring them. Practical examples of the proposed step-by-step implementation of storing application settings are given.

Keywords: Java, Spring, databases, application settings, configuration, cache, microservice architecture.

В процессе работы любого приложения может возникнуть необходимость как-либо поменять важные настройки, которые влияют на несколько модулей программы или на систему в целом [1-5]. При хранении настроек традиционным способом – то есть в виде строк в текстовых файлах, их невозможно обновить или добавить, пока приложение работает [6, 7]. Для того, чтобы внести какие-либо изменения при таком способе конфигурации, необходимо перезапускать всё приложение – что недопустимо, если приложение уже находится на сервере и используется клиентами. Чтобы избежать прерывания работы приложения, настройки можно хранить в базе данных.

С учетом вышесказанного в представленной работе продемонстрируем конкретную реализацию хранения настроек приложения в базе данных на языке Java. Для хранения конфигураций в виде объектов нами было решено использовать специальный формат JSON – JavaScript Object Notation. Это текстовый формат обмена данными, основанный на JavaScript [8]. Этот формат позволяет записать любой объект языка Java в виде формализованного текста,

который затем можно сериализовать и поместить на хранение в базу данных и при необходимости извлечь из базы и десериализовать обратно в объект.

Первым шагом для решения поставленной задачи было создание таблицы properties_db для хранения настроек. Данная таблица имеет всего четыре поля для записи: уникального идентификатора каждой настройки, её типа, значения и описания. Для создания таблицы использовался файл миграции базы данных.

Далее создадим класс ConfigUnit – интерфейс, позволяющий создавать настройки в виде объектов, экземпляр этого класса можно назвать единицей настройки. Его методы соответствуют полям уже созданной нами таблицы, название описание хранятся в виде строк, идентификатор – в виде специального класса, а сама настройка, её значение – в виде объекта:

```
public interface ConfigUnit {
    default ConfigUnitKey getKey()
    String name();
    Object getDefaultValue();
    String getDescription();
```

Класс ConfigUnitKey, возвращаемый методом getKey, на основании логики нашего приложения из поля имени позволяет извлечь значение идентификатора настройки. Реализацией единицы настройки будут enum-классы, например Config, который в качестве имени передаёт постоянную:

```
    enum Config implements ConfigUnit {
        SSA_EMAIL_CREDS(getDefMailCreds(), "Email creds for
verification");
        final Object defaultValue;
        final String description;
    }
```

При создании экземпляра данного класса вызывается метод, возвращающий значение настройки в виде JSON:

```
    private static String getDefMailCreds() {
            return """{
                        "name": "example@mail.ru",
                        "password": "string password",
                        "imap": "imap.mail.ru"
                    }""";
        }
```

Центральным в нашем решении является класс ConfigsRdb, который отвечает за чтение и запись настроек из созданной нами таблицы. В ходе работы были реализованы следующие методы(функции):

Метод инициализации конфигурации, который при запуске приложения записывает в базу данных все значения из заранее подготовленного списка с обязательными настройками. Аннотация @PostConstruct из библиотеки Lombok сообщает, что данный метод запустится сразу при старте приложения.

```
@Inject Optional<List<NeedsConfig>> configsOptional;
    @PostConstruct
    void init() {
        configsOptional.ifPresent(configs -> {
            configs.forEach(config -> {
                for (ConfigUnit unit : config.getConfigUnits()) {
                    String propertyId = unit.getKey().id();
                    defaultConfigs.put(propertyId, unit);
                    if (!propertyExists(propertyId)) {
                        insertDefaultProperty(unit); }
                }
            });
```

```
        });
    }
```

Метод инициализации кеша, который будет хранить часто запрашиваемые значения настроек в оперативной памяти для более быстрого доступа к ним при последующих обращениях. В нашем проекте для работы с хэшем используется специализированная библиотека Caffeine:

```
    @PostConstruct
        void initCache() {
            long ttl = configProps.getTtlSec();
            this.cache = Caffeine.newBuilder()
                    .expireAfterWrite(Duration.ofSeconds(ttl))
                    .build();
        }
```

Непосредственно для записи значения настройки методом конфигурации в цикле вызывается приватный метод insertDefaultProperty, который формирует запрос в базу данных на запись строки с полями, взятыми из переданного экземпляра единицы настройки:

```
        private void insertDefaultProperty(ConfigUnit unit) {
            String sql = """
                    INSERT INTO _conf.properties_db (property_id,
property_category, value, description)
                    VALUES (?, ?, ?, ?)
                    """;
            jdbc.update(sql,
                    unit.getKey().id(),
                    unit.getKey().cat(),
                    unit.getDefaultValue().toString(),
```

```
                unit.getDescription()
            );
        }
```

Методы для чтения и записи в таблицу значений настроек примитивных типов – строк, булевых значений, числовых значений разной размерности, а также объектов – таких как экземпляры ConfigUnit. Для записи любого значения в таблицу его надо привести к строчному виду. Вот пример реализации двух методов записи, все они обращаются к приватному методу updateValue, который формирует простой запрос обновления таблицы на основании переданного идентификатора и значения:

```
    @Override
    public void setFloat(ConfigUnit unit, float value) {
        updateValue(unit.getKey().id(), Float.toString(value));}

    @Override
    public <T> void setObject(ConfigUnit unit, T value) {
        String jsonString = json.from(value);
        updateValue(unit.getKey().id(), jsonString);}

    private void updateValue(String propertyId, String value) {
    String sql = "UPDATE _conf.properties_db SET value = ? WHERE
property_id = ?";
            jdbc.update(sql, value, propertyId);
            cache.put(propertyId, value);
        }
```

Именно метод updateValue помещает в хэш переданное значение настройки, которое можно будет извлечь при последующих обращениях.

В случае чтения записей из таблицы необходимо проверить их наличие, чтобы избежать ошибок в работе программы. Если из таблицы будет извлечено пустое значение – программа сообщит об ошибке не прекращая работу:

```
        @Override
        public long getAsLong(ConfigUnit unit) throws
NumberFormatException {
            String value = readValue(unit.getKey().id());
            if (value == null) {
                throw new NumberFormatException("Config value for key "
+ unit.getKey().id() + " is null");
            }
            return Long.parseLong(value);
        }
```

Для получения записей из таблицы все get-методы обращаются к приватной функции, которая прежде всего обращается к кэшу, и если метод уже принимал точно такие же значения – то результат будет взят из памяти, без обращения к таблице.

```
    private String readValue(String propertyId) {
            return cache.get(propertyId, id -> {
                String sql = "SELECT value FROM _conf.properties_db
WHERE property_id = ?";
                List<String> result = jdbc.query(sql,
                        (resultSet, rowNum) ->
resultSet.getString("value"),
                        id);
                return result.isEmpty() ? restoreValue(propertyId) :
result.getFirst();
            });
```

```
    }
```

Здесь в ходе продумывания логики программы мы обратили внимание на такой сценарий – если во время работы программы в таблице по какой-либо причине не окажется искомой строки, то метод readValue, а следовательно, и все get-методы вернут пустое значение, даже если настройка на самом деле существует в виде объекта и была записана при запуске приложения.

Для решения этой проблемы был создан словарь, использующий в качестве ключей строки-идентификаторы и хранящий единицы настроек:

```
    private final Map<String, ConfigUnit> defaultConfigs = new HashMap<>();
```

Значения настроек по умолчанию в этот список помещаются при исполнении метода инициализации. Для возвращения значений, в случае их необнаружения, обратно в базу данных был написан метод восстановления restoreValue, который вызывается при отсутствии искомого значения в базе. Он проверят наличие потерянной настройки в упомянутом выше словаре и при её наличии – заново записывает значение в таблицу и его же возвращает:

```
    private String restoreValue(String propertyId) {
            if (defaultConfigs.containsKey(propertyId)) {
                ConfigUnit restoredUnit = defaultConfigs.get(propertyId);
                insertDefaultProperty(restoredUnit);
                return restoredUnit.getDefaultValue().toString();
            }
            return null;
        }
```

Таким образом, в работе продуман и разработан механизм хранения настроек приложения не непосредственно в файлах конфигурации, а в виде объектов, записываемых в базу данных.

Реализованы методы чтения и записи значений, застрахованные от возможных ошибок формата данных или отсутствия искомого значения в базе данных. Решение построено на принципах микросервисной архитектуры и в сути своей универсально при решении похожих задач в других приложениях, использующих Java-Spring.

Библиографический список:

1. Oracle documentation website with information on Java, Java EE, and JavaSE [Электронный ресурс] // https://docs.oracle.com/. – URL: https://docs.oracle.com/ (Дата обращения 29.01.2026).

2. Spring framework [Электронный ресурс] // https://spring.io/. – URL: https://spring.io/ (Дата обращения 02.02.2026).

3. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Паттерны объектно-ориентированного проектирования / Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. – Спб: Питер, 2020. – 448 с.

4. Васюткина И. А. Разработка серверной части web-приложений на Java: учебное пособие / И. А. Васюткина. – Новосибирск: НГТУ, 2021. – 83 с.

5. Java 2019 – The state of Developer Ecosystem in 2019 infographic / JetBrains. – URL: https://www.jetbrains.com/lp/devecosystem-2019/java/ (дата обращения: 02.02.2026).

6. Spring Guides / Broadcom. – URL: https://spring.io/guides (дата обращения: 29.01.2026).

7. Java EE Documentation / Oracle. – URL: http://www.oracle.com/ technetwork/java/javaee/documentation/index.html (дата обращения: 02.02.2026).

8. Батулин Н.А., Сафина Г.Ф. Реализация базовой защиты Java-приложения с помощью класса Security фреймворка Spring /

В сборнике: Наука XXI века: технологии, управление, безопасность. Материалы III Национальной научной конференции. – Курган, 2024. – С. 3-6.