JavaFX + SQLite

Полное CRUD-приложение для новичков
с обработкой ошибок

Этап 1 — Пустое окно JavaFX + Gradle

0.1. build.gradle.kts

```
plugins {
    id("java")
    id("application")
    id("org.openjfx.javafxplugin") version "0.1.0"
}

javafx {
    version = "21"
    modules("javafx.controls", "javafx.fxml")
}

application {
    mainClass.set("org.example.notes.App")
}

repositories { mavenCentral() }

dependencies {
    implementation("org.xerial:sqlite-jdbc:3.46.1.2")
}
```

Пояснения к build.gradle.kts:

- **Gradle Kotlin DSL**: Это не обычный Java-код, а Kotlin-скрипт для настройки сборки Gradle. Kotlin DSL предоставляет типобезопасную и интуитивно понятную конфигурацию по сравнению с Groovy DSL.
- **Плагины**:
  - `id("java")` — базовый плагин для Java-проектов
  - `id("application")` — позволяет создавать исполняемые приложения
  - `id("org.openjfx.javafxplugin")` — специальный плагин для интеграции JavaFX с Gradle. Версия "0.1.0" указывает на конкретную реализацию этого плагина.
- **JavaFX конфигурация**:
  - `version = "21"` — указывает версию JavaFX SDK. JavaFX с версии 11 больше не входит в состав JDK и должен подключаться отдельно.
  - `modules("javafx.controls", "javafx.fxml")` — подключает только необходимые модули JavaFX. Это оптимизирует размер итогового приложения. `javafx.controls` содержит все UI-компоненты (кнопки, таблицы и т.д.), а `javafx.fxml` необходим для загрузки интерфейсов из FXML-файлов.
- **Application конфигурация**:
  - `mainClass.set("org.example.notes.App")` — указывает класс с методом `main()`, который будет точкой входа приложения. В JavaFX это должен быть класс, наследующийся от `javafx.application.Application`.
- **Репозитории и зависимости**:
  - `mavenCentral()` — указывает Gradle искать зависимости в центральном Maven репозитории.

  - `implementation("org.xerial:sqlite-jdbc:3.46.1.2")` — подключает SQLite JDBC драйвер. Префикс `implementation` означает, что эта зависимость нужна только для компиляции и выполнения, но не будет транзитивно передаваться другим проектам.

- **Почему именно SQLite?** SQLite — это встраиваемая база данных без необходимости отдельного сервера. Она хранит данные в одном файле, что идеально подходит для десктоп-приложений и обучения. JDBC драйвер позволяет работать с SQLite через стандартный JDBC API Java.

0.2. src/main/java/org/example/notes/App.java

```
package org.example.notes;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.stage.Stage;
import org.example.notes.db.Database;

public class App extends Application {

    @Override
    public void start(Stage stage) throws Exception {
        Database.init(); // создаём файл БД и таблицу

        FXMLLoader loader = new FXMLLoader(
            App.class.getResource("/main-view.fxml"));

        Scene scene = new Scene(loader.load(), 900, 600);
        stage.setTitle("Заметки");
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch();
    }
}
```

Пояснения к App.java:

- **JavaFX Application Lifecycle**:
  - Класс `App` наследуется от `javafx.application.Application`, что делает его точкой входа для JavaFX приложения.
  - Метод `start(Stage stage)` вызывается автоматически JavaFX после инициализации. Параметр `Stage` представляет основное окно приложения.
  - Метод `main(String[] args)` обязательно должен вызывать `launch()`, который запускает JavaFX runtime и инициализирует жизненный цикл приложения.
- **FXMLLoader**:
  - `FXMLLoader` — это класс, который загружает FXML-файлы и создает соответствующие Java-объекты для UI компонентов.
  - `App.class.getResource("/main-view.fxml")` — использует ClassLoader для

поиска ресурса (FXML файла) в classpath. Префикс "/" означает поиск из корня ресурсов (src/main/resources).

  - `loader.load()` возвращает корневой узел (Node) интерфейса, который затем передается в Scene.

- **Scene и Stage**:
  - Scene — это контейнер, который управляет всем содержимым окна (всеми узлами графа сцены).
  - `new Scene(loader.load(), 900, 600)` — создает сцену с заданными размерами 900x600 пикселей.
  - `stage.setScene(scene)` — устанавливает сцену в окно.
  - `stage.show()` — делает окно видимым для пользователя.
- **Database.init()**:
  - Этот вызов инициализирует базу данных перед показом интерфейса. Это важно, потому что:
    * Гарантирует, что БД готова к работе при старте приложения
    * Избегает проблем с отсутствующими таблицами при первой загрузке
    * Выполняется в основном потоке JavaFX (UI thread), что приемлемо для инициализации, но не для длительных операций с БД
- **Пакетная структура**: Архитектура приложения разделена по пакетам:
  - `org.example.notes` — корневой пакет с основным классом приложения
  - `org.example.notes.db` — пакет для работы с базой данных
  - `org.example.notes.model` — пакет для моделей данных (сущностей)
  - `org.example.notes.controller` — пакет для JavaFX контроллеров

Этап 2 — База данных SQLite

0.3. src/main/java/org/example/notes/db/Database.java

```java
package org.example.notes.db;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class Database {
    private static final String URL = "jdbc:sqlite:notes.db";

    public static Connection connect() throws SQLException {
        return DriverManager.getConnection(URL);
    }

    public static void init() {
        String sql = """
            CREATE TABLE IF NOT EXISTS notes (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                title TEXT NOT NULL,
                content TEXT NOT NULL
            );
```

```
            """;
        try (var conn = connect();
             var stmt = conn.createStatement()) {
            stmt.execute(sql);
        } catch (SQLException e) {
            throw new RuntimeException("Не удалось создать БД", e);
        }
    }
}
```

Пояснения к Database.java:

- **JDBC (Java Database Connectivity)**:
  - JDBC — это стандартный API Java для работы с реляционными базами данных.
  - `DriverManager.getConnection(URL)` — загружает подходящий JDBC драйвер (в нашем случае SQLite) и создает соединение с базой данных.
  - URL `"jdbc:sqlite:notes.db"` использует синтаксис JDBC: `jdbc:<subprotocol>:<subn` Здесь `sqlite` — это подпротокол, а `notes.db` — имя файла базы данных в текущей директории.
- **Text Blocks (Java 15+)**:
  - `"""\n...\n"""` — это многострочные текстовые блоки, введенные в Java 15. Они значительно улучшают читаемость многострочных SQL-запросов и XML/JSON данных.
  - Преимущества перед обычными строками:
    - Сохраняют форматирование (отступы, переносы строк)
    - Не требуют экранирования кавычек внутри
    - Легко читаются и редактируются
- **Try-with-resources (Java 7+)**:
  - `try (var conn = connect(); var stmt = conn.createStatement())` — автоматически закрывает ресурсы (`Connection` и `Statement`) после выполнения блока, даже если произойдет исключение.
  - Это предотвращает утечки памяти и соединений с базой данных.
  - `var` (Java 10+) позволяет компилятору выводить тип переменной автоматически.
- **SQLite особенности**:
  - `AUTOINCREMENT` в SQLite работает иначе, чем в других СУБД. Он гарантирует, что ID никогда не будет повторно использован, даже если записи удалялись.
  - `TEXT` тип используется вместо `VARCHAR`, так как в SQLite нет разницы между этими типами (все строки хранятся как TEXT).
  - `NOT NULL` ограничение гарантирует, что поля заголовка и содержания не могут быть пустыми.
- **Статический метод init()**:

- Метод `init()` объявлен как `static`, потому что он вызывается из статического контекста (`main` метода).
- Он создает таблицу при первой инициализации приложения. Ключевое слово `IF NOT EXISTS` предотвращает ошибку, если таблица уже существует.
- При возникновении `SQLException` выбрасывается `RuntimeException`, так как невозможность создать БД на старте приложения — критическая ошибка.

Этап 3 — Модель заметки (record + JavaBean-геттеры)

0.4. src/main/java/org/example/notes/model/Note.java

```java
package org.example.notes.model;

public record Note(int id, String title, String content) {

    // Конструктор для новых заметок (id будет 0)
    public Note(String title, String content) {
        this(0, title, content);
    }

    // ←←← ВАЖНО! Добавляем обычные геттеры с префиксом get
    // Это нужно, чтобы PropertyValueFactory в FXML заработал с record
    public int getId() { return id; }
    public String getTitle() { return title; }
    public String getContent() { return content; }
}
```

Пояснения к Note.java:

- **Java Records (Java 16+)**:
  - `record` — это специальный тип класса, введенный в Java 16 для создания неизменяемых (immutable) классов данных с минимальным количеством кода.
  - Компилятор автоматически генерирует:
    - Приватные final поля (`id`, `title`, `content`)
    - Конструктор с параметрами
    - Методы `equals()`, `hashCode()`, `toString()`
    - "Канонические" геттеры (`id()`, `title()`, `content()`)
  - Records идеально подходят для DTO (Data Transfer Objects) и моделей данных.
- **Совместимость с JavaFX**:
  - JavaFX компоненты (особенно `PropertyValueFactory`) ожидают, что у моделей будут геттеры в формате JavaBean: `getPropertyName()` или свойства в формате `propertyNameProperty()`.
  - Record генерирует геттеры без префикса `get` (просто `id()`, `title()`), что несовместимо с `PropertyValueFactory`.

  - Решение: явно объявить обычные геттеры с префиксом `get` для каждого поля. Это сохраняет все преимущества record и обеспечивает совместимость с JavaFX.

- **Компактный конструктор для новых записей**:
  - Конструктор `public Note(String title, String content)` позволяет создавать новые заметки без указания ID (который будет сгенерирован базой данных).
  - Значение `0` для ID — это соглашение, указывающее, что это новая запись, которая еще не сохранена в базе.
  - Такой подход упрощает работу с новыми объектами в коде интерфейса.
- **Почему это важно**:
  - Без явных геттеров с префиксом `get` таблица JavaFX (`TableView`) будет пустой, так как `PropertyValueFactory` не сможет найти нужные свойства.
  - Это пример того, как современные Java-фичи (records) могут требовать адаптации для работы с более старыми фреймворками (JavaFX, который изначально проектировался под JavaBeans).
  - Такой подход сохраняет неизменяемость (`final` поля) и простоту records, обеспечивая при этом необходимую совместимость.
- **Альтернативные подходы**:
  - Использовать обычный класс вместо record (потеря компактности)
  - Использовать `ReadOnlyObjectWrapper` и свойства JavaFX (более сложный код)
  - Создать отдельный класс-адаптер (избыточность)
  - Текущее решение — оптимальный баланс между современностью и совместимостью.

Почему мы добавили getId(), getTitle(), getContent()?
`PropertyValueFactory` ищет именно методы с префиксом `get…` или `…Property()`. У record'ов геттеры называются просто `id()`, `title()`, поэтому без этих трёх строк таблица будет пустой. Добавив обычные геттеры — получаем и лаконичность record, и полную совместимость с FXML.

Этап 4 — Показываем все заметки в таблице (Read)

0.5. src/main/java/org/example/notes/db/NoteDao.java

```
package org.example.notes.db;

import org.example.notes.model.Note;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class NoteDao {

    public List<Note> findAll() {
        var list = new ArrayList<Note>();
        String sql = "SELECT id, title, content FROM notes ORDER BY id";
        try (Connection conn = Database.connect();
```

```
             Statement stmt = conn.createStatement();
             ResultSet rs = stmt.executeQuery(sql)) {

            while (rs.next()) {
                list.add(new Note(
                    rs.getInt("id"),
                    rs.getString("title"),
                    rs.getString("content")
                ));
            }
        } catch (SQLException e) {
            throw new RuntimeException("Ошибка чтения заметок", e);
        }
        return list;
    }

    public void insert(Note note) { /* будет дальше */ }
    public void update(Note note) { /* будет дальше */ }
    public void delete(int id) { /* будет дальше */ }
}
```

Пояснения к NoteDao.java:

- **Паттерн DAO (Data Access Object)**:
  - DAO — это паттерн проектирования, который абстрагирует и инкапсулирует все операции доступа к данным.
  - Преимущества:
    * Разделение бизнес-логики и логики доступа к данным
    * Легкость изменения источника данных (например, с SQLite на PostgreSQL)
    * Централизованная обработка ошибок работы с БД
    * Повторное использование кода доступа к данным
  - В данном случае `NoteDao` отвечает только за операции с заметками (CRUD).
- **JDBC ResultSet обработка**:
  - `ResultSet rs = stmt.executeQuery(sql)` — выполняет SQL-запрос и возвращает результат в виде объекта `ResultSet`.
  - `while (rs.next())` — перемещает курсор на следующую строку результата. Возвращает `false`, когда строк больше нет.
  - `rs.getInt("id")` и `rs.getString("title")` — получают значения столбцов по имени. Важно использовать точные имена столбцов из SQL-запроса.
  - Преобразование результата в объекты `Note` происходит сразу при чтении, что упрощает дальнейшую работу с данными.
- **Try-with-resources для ResultSet**:
  - `try` блок включает не только `Connection` и `Statement`, но и `ResultSet`.
  - Это гарантирует, что все ресурсы будут закрыты, даже если возникнет исключение при обработке результатов.
  - Закрытие `ResultSet` автоматически закрывает связанный с ним `Statement` и `Connection` (но лучше явно закрывать все).

- **Обработка исключений в DAO**:
  - `SQLException` перехватывается и оборачивается в `RuntimeException` с понятным сообщением.
  - Это упрощает обработку ошибок на уровне контроллера или сервиса.
  - Параметр `e` в конструкторе `RuntimeException` сохраняет оригинальное исключение (cause), что помогает в отладке.
  - В продакшен-коде здесь можно добавить логирование ошибки.
- **Почему List вместо ObservableList?**:
  - DAO должен возвращать просто данные, а не UI-зависимые типы (`ObservableList` из JavaFX).
  - `ObservableList` создается в контроллере, когда данные готовы для отображения.
  - Это сохраняет разделение ответственности: DAO работает с данными, контроллер — с представлением.
- **SQL-запрос особенности**:
  - `ORDER BY id` гарантирует, что заметки всегда будут отсортированы по возрастанию ID.
  - Явное указание столбцов (`SELECT id, title, content`) лучше, чем `SELECT *`, так как:
    * Код становится более устойчивым к изменениям структуры таблицы
    * Четко видно, какие данные запрашиваются
    * Может улучшить производительность (особенно если в таблице много столбцов)

0.6. src/main/resources/main-view.fxml — полный интерфейс

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox xmlns="http://javafx.com/javafx/21"
      xmlns:fx="http://javafx.com/fxml/1"
      fx:controller="org.example.notes.controller.MainController"
      spacing="20" padding="20">

    <Label text="Мои␣заметки" style="-fx-font-size:␣24px;␣-fx-font-weight:␣bold;
       "/>

    <HBox spacing="10">
        <TextField fx:id="titleField" promptText="Заголовок" prefWidth="250"/>
        <TextField fx:id="contentField" promptText="Содержание" HBox.hgrow="
           ALWAYS"/>
        <Button text="Сохранить" onAction="#saveNote"/>
    </HBox>

    <TableView fx:id="table" VBox.vgrow="ALWAYS">
        <columns>
            <TableColumn text="ID" prefWidth="60">
```

```
                <cellValueFactory><PropertyValueFactory property="id"/></
                    cellValueFactory>
            </TableColumn>
            <TableColumn text="Заголовок" prefWidth="200">
                <cellValueFactory><PropertyValueFactory property="title"/></
                    cellValueFactory>
            </TableColumn>
            <TableColumn text="Содержание" prefWidth="500">
                <cellValueFactory><PropertyValueFactory property="content"/></
                    cellValueFactory>
            </TableColumn>
        </columns>
    </TableView>

    <Button text="Удалить␣выбранную" onAction="#deleteNote"
            style="-fx-background-color:␣#dc3545;␣-fx-text-fill:␣white;"/>
</VBox>
```

Пояснения к main-view.fxml:

- **FXML — декларативный язык разметки для JavaFX**:
  - FXML позволяет описывать UI интерфейс в XML-формате, разделяя представление и логику.
  - Преимущества:
    * Визуальное проектирование интерфейса (можно использовать Scene Builder)
    * Легкость изменения дизайна без перекомпиляции Java-кода
    * Четкое разделение ответственности (дизайн vs бизнес-логика)
- **Пространства имен (namespaces)**:
  - `xmlns="http://javafx.com/javafx/21"` — указывает версию JavaFX, используемую для этого FXML.
  - `xmlns:fx="http://javafx.com/fxml/1"` — пространство имен для FXML-специфичных атрибутов (например, `fx:id`, `fx:controller`).
- **JavaFX Layout Managers**:
  - `VBox` — вертикальный контейнер, располагает дочерние элементы сверху вниз.
  - `HBox` — горизонтальный контейнер, располагает дочерние элементы слева направо.
  - `spacing="20"` — расстояние между дочерними элементами в контейнере.
  - `padding="20"` — внутренние отступы контейнера.
  - `VBox.vgrow="ALWAYS"` и `HBox.hgrow="ALWAYS"` — определяют, как компоненты растягиваются при изменении размера окна:
    * `ALWAYS` означает, что компонент займет все доступное пространство
    * Это критически важно для `TableView`, чтобы она заполняла свободное место
- **TableView и TableColumn**:

  - `TableView` — компонент для отображения табличных данных.
  - `TableColumn` определяет столбцы таблицы.
  - `cellValueFactory` определяет, как извлекать значение для каждой ячейки:
    * `PropertyValueFactory` использует Java Reflection для вызова геттеров модели
    * `property="id"` ищет метод `getId()` или `idProperty()` в объекте `Note`
    * Без явных геттеров в record это не работало бы
  - `prefWidth` задает предпочтительную ширину столбцов в пикселях.
- **Стилизация через CSS**:
  - `style="-fx-font-size: 24px; -fx-font-weight: bold;"` — встроенные CSS-стили для JavaFX.
  - `-fx-background-color: #dc3545` — цвет фона кнопки (красный Bootstrap цвет для опасных действий).
  - `-fx-text-fill: white` — белый цвет текста.
  - JavaFX использует CSS-подобный синтаксис для стилизации, но с префиксами `-fx-`.
- **Обработка событий**:
  - `onAction="#saveNote"` — связывает событие нажатия кнопки с методом `saveNote()` в контроллере.
  - `fx:id="titleField"` — присваивает уникальный идентификатор компоненту, чтобы контроллер мог получить к нему доступ через `@FXML` аннотации.
  - `fx:controller="org.example.notes.controller.MainController"` — указывает класс-контроллер, который будет обрабатывать события и логику этого FXML.
- **Почему HBox внутри VBox?**:
  - Такая вложенность контейнеров создает гибкую и адаптивную компоновку:
    * Верхний уровень (VBox) управляет вертикальным расположением
    * Средний уровень (HBox) управляет горизонтальным расположением полей ввода
    * `TableView` растягивается по вертикали (`VBox.vgrow="ALWAYS"`)
    * Поле содержания растягивается по горизонтали (`HBox.hgrow="ALWAYS"`)
  - Это стандартная практика в JavaFX для создания отзывчивых интерфейсов.

0.7. src/main/java/org/example/notes/controller/MainController.java — основа

```
package org.example.notes.controller;

import javafx.collections.FXCollections;
```

```
import javafx.fxml.FXML;
import javafx.scene.control.*;
import org.example.notes.db.NoteDao;
import org.example.notes.model.Note;

public class MainController {

    @FXML private TableView<Note> table;
    @FXML private TextField titleField;
    @FXML private TextField contentField;

    private final NoteDao dao = new NoteDao();

    @FXML
    private void initialize() {
        refresh();

        // При выборе строки — подставляем в поля
        table.getSelectionModel().selectedItemProperty()
             .addListener((obs, oldValue, newValue) -> {
                 if (newValue != null) {
                     titleField.setText(newValue.title());
                     contentField.setText(newValue.content());
                 } else {
                     titleField.clear();
                     contentField.clear();
                 }
             });
    }

    void refresh() {
        table.setItems(FXCollections.observableArrayList(dao.findAll()));
    }

    // saveNote(), deleteNote() — добавим на следующих этапах
}
```

Пояснения к MainController.java:

- **JavaFX Controller Lifecycle**:
  - Контроллер должен иметь публичный конструктор без параметров (FXMLLoader создает его через Reflection).
  - Метод, помеченный `@FXML private void initialize()`, вызывается автоматически после загрузки FXML и внедрения всех компонентов.
  - Это идеальное место для инициализации данных и установки обработчиков событий.
- **FXML-аннотации и внедрение зависимостей**:
  - `@FXML private TableView<Note> table` — аннотация указывает FXMLLoader внедрить ссылку на компонент с `fx:id="table"` из FXML.
  - Поля должны быть помечены `@FXML` и иметь совпадающие имена с `fx:id` в FXML (или использовать `fx:id` напрямую).
  - Поля обычно объявляются как `private`, но могут быть `public` или package-private (FXMLLoader использует Reflection для доступа).

- **ObservableList и FXCollections**:
  - `FXCollections.observableArrayList(dao.findAll())` преобразует обычный `List<Note>` в `ObservableList<Note>`.
  - `ObservableList` — это специальная коллекция из JavaFX, которая уведомляет UI об изменениях (добавлении, удалении, обновлении элементов).
  - `table.setItems(...)` связывает эту коллекцию с `TableView`, что позволяет таблице автоматически обновляться при изменении данных.
- **Слушатели свойств (Property Listeners)**:
  - `table.getSelectionModel().selectedItemProperty()` возвращает `ObjectProperty<N` которое отслеживает текущую выбранную строку в таблице.
  - `addListener((obs, oldValue, newValue) -> ...)` — добавляет слушатель, который вызывается при изменении выбранного элемента.
  - Параметры лямбда-выражения:
    - `obs` — наблюдаемое свойство (Observable)
    - `oldValue` — предыдущее значение
    - `newValue` — новое значение
  - Это пример реактивного программирования в JavaFX: интерфейс автоматически реагирует на изменения состояния.
- **Работа с SelectionModel**:
  - `SelectionModel` управляет выбором элементов в компонентах (таблицы, списки, деревья).
  - `selectedItemProperty()` возвращает свойство, которое содержит текущий выбранный элемент (`null`, если ничего не выбрано).
  - `clearSelection()` (используется позже) снимает выделение со всех строк.
- **Почему refresh() не помечен @FXML?**:
  - Метод `refresh()` вызывается из `initialize()` и других методов контроллера, но не связан напрямую с событиями FXML.
  - `@FXML` аннотация нужна только для методов, которые вызываются из FXML (через `onAction`) или для метода `initialize()`.
  - Делая `refresh()` package-private (без модификатора доступа), мы позволяем вызывать его из других методов контроллера, сохраняя инкапсуляцию.
- **Инверсия управления (IoC)**:
  - `private final NoteDao dao = new NoteDao()` — простой пример создания зависимостей.
  - В реальных приложениях здесь лучше использовать DI-фреймворки (Google Guice, Spring) или фабрики для создания DAO.
  - Это облегчает тестирование (можно подменить DAO на mock-объект).

Этап 5 — Добавление и редактирование (Create + Update)

В `NoteDao.java` добавляем `insert` и `update`:

```
    public void insert(Note note) {
        String sql = "INSERT INTO notes(title, content) VALUES(?, ?)";
        try (Connection c = Database.connect();
             PreparedStatement ps = c.prepareStatement(sql)) {
            ps.setString(1, note.title());
            ps.setString(2, note.content());
            ps.executeUpdate();
        } catch (SQLException e) {
            throw new RuntimeException("Не удалось добавить заметку", e);
        }
    }

    public void update(Note note) {
        String sql = "UPDATE notes SET title = ?, content = ? WHERE id = ?";
        try (Connection c = Database.connect();
             PreparedStatement ps = c.prepareStatement(sql)) {
            ps.setString(1, note.title());
            ps.setString(2, note.content());
            ps.setInt(3, note.id());
            ps.executeUpdate();
        } catch (SQLException e) {
            throw new RuntimeException("Не удалось обновить заметку", e);
        }
    }
```

Пояснения к insert() и update():

- **PreparedStatement вместо Statement**:
  - `PreparedStatement` предпочтительнее обычного `Statement` по нескольким причинам:
    - **Безопасность**: защищает от SQL-инъекций, так как параметры экранируются автоматически
    - **Производительность**: SQL-запрос компилируется один раз и может выполняться многократно с разными параметрами
    - **Читаемость**: четкое разделение SQL-шаблона и значений параметров
  - ? в SQL-запросе — это placeholders для параметров.
- **Параметризованные запросы**:
  - `ps.setString(1, note.title())` — устанавливает значение для первого placeholder (?) как строку.
  - Нумерация параметров начинается с 1 (не с 0!).
  - `ps.setInt(3, note.id())` — устанавливает значение для третьего placeholder как целое число.
  - Типы данных должны соответствовать типам столбцов в базе данных.
- **executeUpdate() vs executeQuery()**:
  - `executeUpdate()` используется для DML-команд (INSERT, UPDATE, DELETE), которые не возвращают результирующий набор.

- Возвращает количество затронутых строк (можно использовать для проверки успешности операции).
- `executeQuery()` используется для SELECT-запросов и возвращает `ResultSet`.

- **Работа с AUTOINCREMENT**:
  - В методе `insert()` мы не указываем значение для поля `id`, потому что SQLite автоматически генерирует его благодаря `AUTOINCREMENT`.
  - После вставки можно получить сгенерированный ID с помощью `ps.getGeneratedKeys` но в данном приложении это не требуется, так как мы обновляем весь список заметок через `refresh()`.
- **Почему нет возврата сгенерированного ID?**:
  - В этом CRUD-приложении после сохранения мы полностью обновляем таблицу через `dao.findAll()`, поэтому нет необходимости получать только что вставленный ID.
  - В более сложных сценариях (например, если нужно продолжить работу с новой записью) можно добавить:

```
        try (ResultSet generatedKeys = ps.getGeneratedKeys()) {
            if (generatedKeys.next()) {
                return generatedKeys.getInt(1);
            }
        }
```

В `MainController.java` — универсальный метод сохранения:

```
    @FXML
    private void saveNote() {
        String title = titleField.getText().trim();
        String content = contentField.getText().trim();

        if (title.isEmpty() || content.isEmpty()) {
            showError("Ошибка", "Заголовок и содержание не могут быть пустыми!")
    ;
            return;
        }

        Note selected = table.getSelectionModel().getSelectedItem();

        try {
            if (selected != null && selected.id() != 0) {
                dao.update(new Note(selected.id(), title, content));
            } else {
                dao.insert(new Note(title, content));
            }
            refresh();
            table.getSelectionModel().clearSelection();
            titleField.clear();
            contentField.clear();
        } catch (Exception e) {
            showError("Ошибка сохранения", e.getMessage());
        }
    }
```

Пояснения к saveNote():

- **Универсальная логика сохранения**:
  - Метод определяет, нужно ли создавать новую запись или обновлять существующую, на основе выбранного элемента в таблице.
  - `selected.id() != 0` — проверка, была ли запись уже сохранена в базе (новые записи имеют ID=0).
  - Это пример паттерна "единая точка входа" для операций создания/обновления, что упрощает UI и уменьшает дублирование кода.
- **Валидация ввода**:
  - `trim()` удаляет пробелы в начале и конце строк.
  - Проверка на пустые поля выполняется до обращения к базе данных, что улучшает пользовательский опыт.
  - `showError()` — пользовательский метод для отображения диалогов ошибок (будет реализован позже).
- **Сброс состояния интерфейса**:
  - `refresh()` — обновляет данные в таблице из базы.
  - `table.getSelectionModel().clearSelection()` — снимает выделение со строки.
  - `titleField.clear()` и `contentField.clear()` — очищают поля ввода.
  - Это создает ожидаемое поведение для пользователя после сохранения.
- **Обработка исключений**:
  - `try-catch` блок перехватывает все исключения, которые могут возникнуть при работе с базой данных.
  - `catch (Exception e)` перехватывает любые исключения, но в продакшене лучше перехватывать конкретные типы (`SQLException`).
  - `e.getMessage()` извлекает сообщение об ошибке из исключения для отображения пользователю.
  - Это предотвращает падение приложения при ошибках БД.
- **Почему selected.id() != 0?**:
  - Это соглашение, установленное в конструкторе record `Note(String, String)`, который устанавливает ID=0 для новых записей.
  - В реальных приложениях лучше использовать `Optional<Integer>` или отдельный флаг, но для учебного примера это приемлемо.
  - Альтернативный подход — проверять `selected == null` для создания новой записи, но текущая реализация позволяет редактировать выбранную запись.
- **Многопоточность в JavaFX**:
  - Весь код выполняется в UI-потоке (JavaFX Application Thread), что безопасно для обновления интерфейса.

- Однако длительные операции с базой данных могут блокировать UI. В реальных приложениях такие операции нужно выполнять в фоновых потоках с использованием `Task` или `Platform.runLater()` для обновления UI.
- Для учебного примера с SQLite это не критично, так как операции выполняются быстро.

Этап 6 — Удаление с подтверждением (Delete)

В `NoteDao.java`:

```
    public void delete(int id) {
        String sql = "DELETE FROM notes WHERE id = ?";
        try (Connection c = Database.connect();
             PreparedStatement ps = c.prepareStatement(sql)) {
            ps.setInt(1, id);
            ps.executeUpdate();
        } catch (SQLException e) {
            throw new RuntimeException("Не удалось удалить заметку", e);
        }
    }
```

Пояснения к delete():

- **Безопасное удаление**:
  - Использование `PreparedStatement` защищает от SQL-инъекций, даже при удалении по ID.
  - Условие `WHERE id = ?` гарантирует, что будет удалена только одна конкретная запись.
  - Без условия `WHERE` команда `DELETE` удалила бы все записи в таблице!
- **Проверка количества затронутых строк**:
  - `ps.executeUpdate()` возвращает количество удаленных строк.
  - В продакшен-коде можно добавить проверку:

```
        int rowsAffected = ps.executeUpdate();
        if (rowsAffected == 0) {
            throw new RuntimeException("Заметка с ID=" + id + " не
    найдена");
        }
```

  - Это помогает обнаруживать ситуации, когда запись уже была удалена другим пользователем или процессом.
- **Почему не используем CASCADE?**:
  - В SQLite можно создавать внешние ключи с каскадным удалением (`ON DELETE CASCADE`), но в нашем случае таблица `notes` не имеет связей с другими таблицами.
  - Если бы были связи, необходимо было бы предусмотреть каскадное удаление или проверку перед удалением.

В `MainController.java`:

```
    @FXML
```

```
    private void deleteNote() {
        Note selected = table.getSelectionModel().getSelectedItem();
        if (selected == null) return;

        Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
        alert.setTitle("Подтверждение");
        alert.setHeaderText(null);
        alert.setContentText("Удалить заметку «" + selected.title() + "»?");

        if (alert.showAndWait().orElse(ButtonType.CANCEL) == ButtonType.OK) {
            try {
                dao.delete(selected.id());
                refresh();
                titleField.clear();
                contentField.clear();
            } catch (Exception e) {
                showError("Ошибка удаления", e.getMessage());
            }
        }
    }
```

Пояснения к deleteNote():

- **JavaFX Alert Dialogs**:
  - `Alert` — стандартный диалоговый компонент JavaFX для отображения сообщений.
  - `Alert.AlertType.CONFIRMATION` — тип диалога с кнопками OK/Cancel.
  - `alert.showAndWait()` показывает диалог и блокирует выполнение до закрытия диалога (модальное окно).
  - `alert.showAndWait().orElse(ButtonType.CANCEL)` обрабатывает случай, когда диалог закрыт без выбора (например, нажатием крестика).
- **Безопасное пользовательское взаимодействие**:
  - Проверка `if (selected == null) return` предотвращает попытку удаления без выбранной записи.
  - Диалог подтверждения защищает от случайного удаления данных.
  - Отображение заголовка заметки в сообщении помогает пользователю подтвердить правильность действия.
  - `alert.setHeaderText(null)` убирает стандартный заголовок диалога для более чистого вида.
- **Optional API**:
  - `alert.showAndWait()` возвращает `Optional<ButtonType>`.
  - `orElse(ButtonType.CANCEL)` обеспечивает значение по умолчанию, если `Optional` пуст (диалог закрыт без выбора).
  - Это безопасный способ работы с возможностью отсутствия значения.
- **Почему не используем AlertType.WARNING?**:
  - `AlertType.CONFIRMATION` имеет стандартные кнопки OK/Cancel, которые идеально подходят для подтверждения действий.

  - `AlertType.WARNING` обычно используется для предупреждений о потенциальных проблемах, а не для подтверждения действий.
  - Стилизация красной кнопки "Удалить" в FXML уже визуально указывает на опасность действия.
- **Обработка ошибок при удалении**:
  - `try-catch` блок обрабатывает исключения, которые могут возникнуть при удалении (например, нарушение внешних ключей, проблемы с соединением).
  - После успешного удаления интерфейс сбрасывается (очистка полей и обновление таблицы).
  - Это создает последовательный пользовательский опыт.

Этап 8 — Обработка исключений (самый простой и красивый способ)

Добавляем в `MainController.java` два удобных метода:

```
    // Показывает красное окно с ошибкой
    private void showError(String header, String message) {
        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setTitle("Ошибка");
        alert.setHeaderText(header);
        alert.setContentText(message);
        alert.showAndWait();
    }

    // Показывает зелёное окно с информацией можно( использовать для успеха)
    private void showInfo(String header, String message) {
        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setTitle("Информация");
        alert.setHeaderText(header);
        alert.setContentText(message);
        alert.showAndWait();
    }
```

Пояснения к showError() и showInfo():

- **Централизованная обработка ошибок**:
  - Эти методы создают единый стиль отображения ошибок и информационных сообщений по всему приложению.
  - `Alert.AlertType.ERROR` автоматически использует красный цвет и иконку ошибки.
  - `Alert.AlertType.INFORMATION` использует синий цвет и иконку информации.
  - Это улучшает пользовательский опыт и делает приложение более профессиональным.
- **Модальные диалоги**:
  - `alert.showAndWait()` делает диалог модальным — пользователь не может взаимодействовать с основным окном, пока диалог открыт.

  - Это гарантирует, что пользователь увидит сообщение об ошибке.
  - Возвращаемое значение `Optional<ButtonType>` не используется в этих методах, так как для ошибок и информации стандартное поведение (закрытие по OK) подходит.
- **Разделение ответственности**:
  - Методы `showError()` и `showInfo()` инкапсулируют логику отображения сообщений.
  - Это позволяет легко изменить стиль всех сообщений в одном месте.
  - В будущем можно добавить логирование ошибок в файл из `showError()`.
- **Почему showAndWait(), а не show()?**:
  - `showAndWait()` блокирует выполнение до закрытия диалога, что необходимо для ошибок (пользователь должен увидеть сообщение).
  - `show()` показывает диалог асинхронно, что подходит для информационных сообщений, не требующих немедленного внимания.
  - Для ошибок используется `showAndWait()`, чтобы прервать текущий workflow и заставить пользователя увидеть ошибку.
- **Пользовательский опыт (UX)**:
  - Заголовок (`header`) и основное сообщение (`message`) разделены для лучшей читаемости.
  - Заголовок обычно краткий и описывает тип проблемы ("Ошибка валидации", "Ошибка базы данных").
  - Основное сообщение содержит детали ("Поле 'Заголовок' не может быть пустым").
  - Это соответствует лучшим практикам дизайна диалоговых окон.