**Приложения Reading Diary**

**Автор:** Елизавета Руденко

<https://github.com/rudenyel/sql-java.git>

**Приобретенные навыки:**

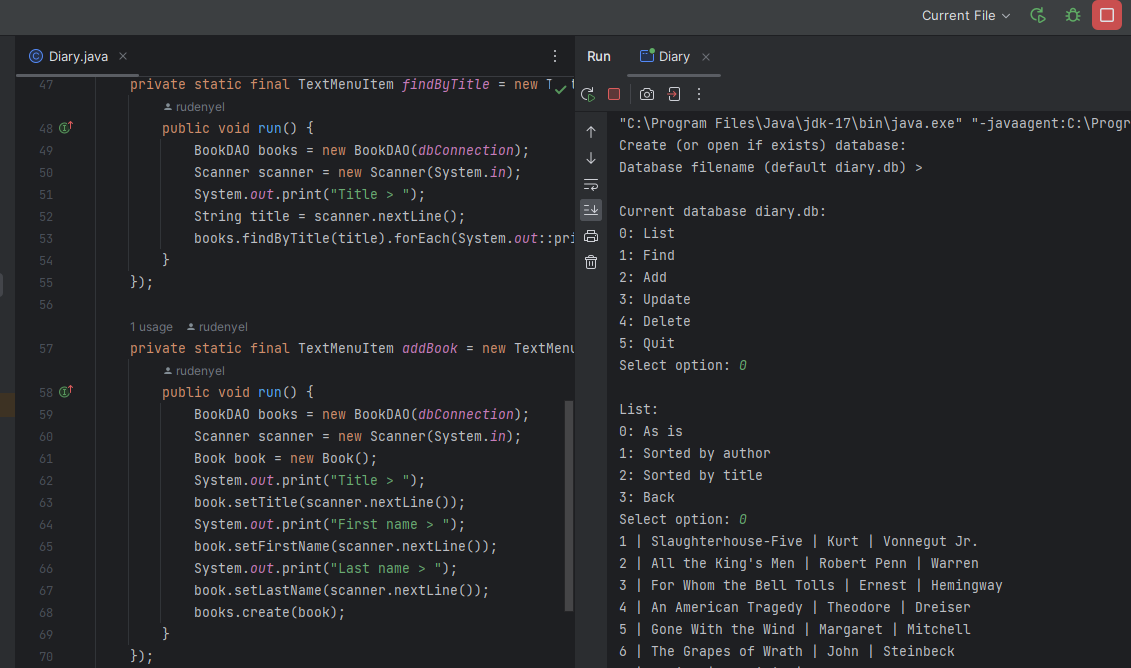
Just finished the course “Learning JDBC” by Frank Moley! Check it out:

<https://www.linkedin.com/learning/certificates/f92632247717b134999c50be2d62e22386a8205cb87c2c29b59cbdb2b48ea45b>

**Задача:**

Разработаем приложение позволяющее работать с дневником, в котором хранятся названия книг и авторы. Приложение должно уметь вставлять и удалять записи, сортировать записи при выводе на экран, искать книги по названию и автору. Приложение использует двухуровненое текстовое меню.

<https://github.com/rudenyel/sql-java/blob/main/src/main/java/cz/murkaliza/diary/Diary.java>



**Инструменты:**

Для хранения данных будем использовать SQLite – компактную, встраиваемую систему управления реляционными базами данных (СУБД). Она отличается от многих других СУБД, таких как MySQL или PostgreSQL, тем, что она встраивается непосредственно в приложение, хранится в одном файле, не требуя отдельного сервера баз данных.

OpenJDK 17.0.8

IntelliJ IDEA 2023.2.1 Community Edition

Apache Maven 3.9.5

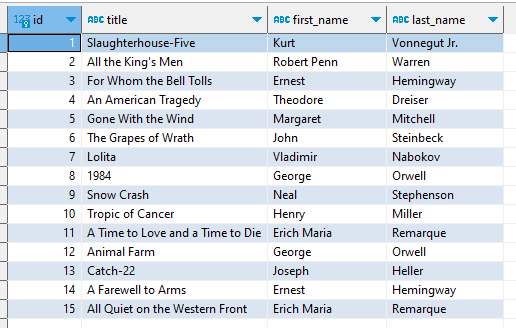
SQLite JDBC 3.44.1.0

DBeaver Community 23

**Реляционная модель и проектирование базы данных**

Основой современной технологии баз данных является реляционная модель данных – абстрактная теория данных, основная на некоторых положениях математики (в основном теории множеств и предикативной логики). Принципы реляционной модели были изначально заложены доктором Коддом (Codd), сотрудником IBM. В конце 1968 года Кодд, математик по образованию, впервые осознал, что математические дисциплины можно использовать, чтобы привнести в область управления базами данных строгие принципы и точность.

Реляционная база данных хранит информацию в таблицах (в реляционной теории– отношения) со строками (кортежами) и столбцами (атрибутами).



Отношение делится на две части: заголовок и тело. Заголовок – это множество атрибутов, а тело – это множество кортежей. Количество атрибутов называется степенью, а количество кортежей – кардинальным числом. Отношения обладают четырьмя очень важными свойствами:

**1. В них нет одинаковых кортежей.** Это свойство следует из того факта, что тело отношения – это математическое множество (кортежей), а множество в математике по определению не содержит одинаковых элементов. Важным следствием этого факта, является то, что всегда существует первичный ключ – уникальное значение в пределах отношения (в нашем случае это поле id). Первичные ключи обеспечивают основной механизм адресации на уровне кортежей в реляционной системе. Единственный гарантируемый системой способ точно указать на какой-нибудь кортеж – это указать значение некоторого первичного ключа.

**2. Кортежи не упорядочены (сверху вниз).** Это свойство также следует из того, что тело отношения – это математическое множество (кортежей), а простые множества в математике не упорядочены.

**3. Атрибуты не упорядочены (слева направо).** Это свойство следует из того факта, что заголовок отношения также определен как множество (атрибутов).

Первые три свойства отношений служат хорошей иллюстрацией того факта, что отношение и таблица – это не одно и то же. Хотя дальше в тексте мы будем использовать название таблица.

**4. Все значения атрибутов атомарные.** Отношение, удовлетворяющее этому условию, называется нормализованным, или представленным в первой нормальной форме. В нашем случае мы нормализовали таблицу, разделив не только название книги и автора, но и имя и фамилию. Это ускорит поиск по названию и по фамилии автора, позволит, например, сортировать дневник как по названию, так и по автору.

Существуют вторая нормальная форма, третья и т.д. В нашем случае мы могли бы привести наше отношение ко второй нормальной форме, создав таблицу authors (id, first\_name, last\_name), оставив в таблице books поле id\_author и связав эти таблицы отношением один ко многим authors.id -> books.id\_author. Это уменьшило бы размеры базы данных, ведь множество повторяющихся строк с именами и фамилиями авторов были бы заменены числовым идентификатором. Это также уменьшает процент ошибок при наборе и исправлении текста – имя и фамилия автора находится в одном месте, в таблице Authors.

Так же мы могли бы решить проблему хранения нескольких авторов для одной книги, введя промежуточную таблицу с соответствием id\_books - id\_authors. Но мы остановимся в проектировании базы данных на одной таблице.

**Язык запросов SQL**

Для работы с базами данными используется язык структурированных запросов (SQL). Инструкции SQL можно использовать для хранения, обновления, удаления, поиска и извлечения информации из базы данных.

Язык SQL используется также для управления и оптимизации базы данных. Например, после создания базы данных мы можем создать в ней таблицу books с помощью SQL-запроса:

CREATE TABLE IF NOT EXISTS books (

id INTEGER PRIMARY KEY AUTOINCREMENT,

title VARCHAR(127) CHECK(title != ''),

first\_name VARCHAR(127) CHECK(first\_name != ''),

last\_name VARCHAR(127) CHECK(last\_name != ''))

Он создает в ней таблицу books с уникальным первичным ключом id, который автоматически увеличивается на единицу при добавлении новых записей. Также в этой таблице создаются текстовые поля title, first\_name, last\_name размером до 127 символов, которые не могут быть пустыми строками.

Мы можем добавить данные в таблицу с помощью команды INSERT:

INSERT INTO books (title, first\_name, last\_name) VALUES ('1984', 'George', 'Orwell')

Значение первичного ключа при операциях вставки указывать не нужно – его значение контролирует база данных.

Удаление записи:

DELETE FROM books WHERE id = 1

Для выполнения запросов выборки данных используется команда SELECT. Например, для отображения данных, отсортированных по фамилии (по возрастанию):

SELECT \* FROM books ORDER BY last\_name

и в обратном порядке:

SELECT \* FROM books ORDER BY last\_name DESC

Вместо \* можно указать конкретные имена столбцов, которые мы хотим видеть в выборке:

SELECT last\_name, first\_name FROM books ORDER BY last\_name

В этом запросе есть повторяющиеся имена и фамилии авторов, их можно сгруппировать и не выводить повторяющиеся значения:

SELECT last\_name, first\_name

FROM books

GROUP BY last\_name

ORDER BY last\_name

Можно выбрать авторов с конкретным именем:

SELECT last\_name, first\_name

FROM books

WHERE first\_name = 'Kurt'

ORDER BY last\_name

или с именем, начинающимся на конкретную букву:

SELECT last\_name, first\_name

FROM books

WHERE first\_name LIKE 'E%'

ORDER BY last\_name

GROUP BY last\_name

Также в запросы, как и в функции, можно передавать параметры, которые позволяют использовать один и тот же запрос для различных входных данных:

SELECT title FROM books WHERE last\_name = :last\_name

При запуске этого запроса нужно ввести или передать через API параметр last\_name

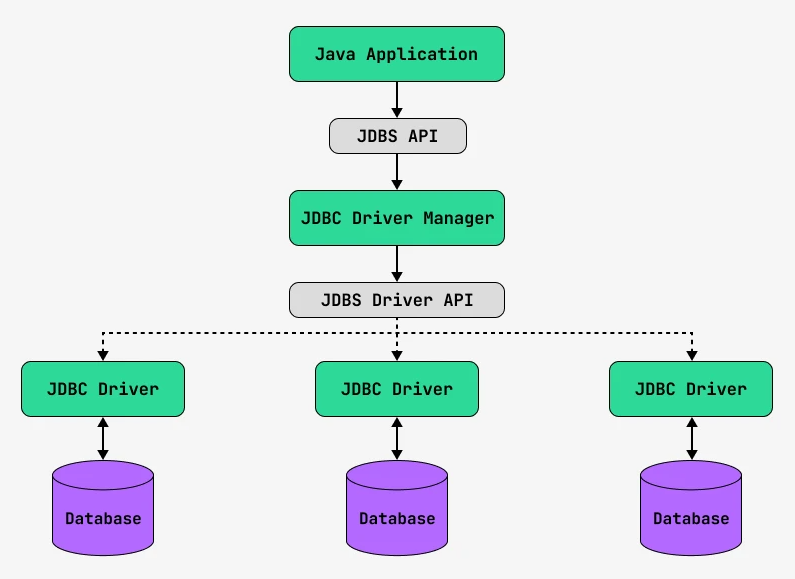
**JDBC**

Java Database Connectivity – это стандартный API для независимого соединения Java-приложения с различными базами данных. JDBC решает следующие задачи: создание соединения с базой данной; создание и выполнение SQL-запросов для получения записей, просмотр и модификация этих записей. Если говорить в целом, то JDBC – это библиотека, которая обеспечивает целый набор классов и интерфейсов для доступа к различным источниками данных. Для доступа к каждой конкретной БД необходим специальный JDBC-драйвер, который является адаптером Java-приложения к БД.

JDBC в общем виде состоит из двух слоев:

JDBC API – обеспечивает соединение Java-приложение– JDBC-менеджер;

JDBC Driver API – обеспечивает соединение JDBC- менеджер – драйвер БД.



Обычный программный цикл работы с JDBC API выглядит следующим образом:

<https://github.com/rudenyel/sql-java/blob/main/src/main/java/cz/murkaliza/jdbc/ExampleJDBC.java>

С помощью объекта **Driver Manager** мы устанавливаем соединение с базой данных (интерфейс **Connection**). Все взаимодействия с БД происходят исключительно через Connection. Например, для выполнения SQL-запросов мы используем объекты, созданные с использованием этого интерфейса. В этот объект так же передаются параметры SQL-запроса. Экземпляры класса **ResultSet** содержат данные, которые были получены в результате выполнения SQL-запроса. Он работает как итератор и “пробегает” по полученным данным. **SQLException** обрабатывает все ошибки, которые могут возникнуть при работе с БД.

**DAO, DTO patterns**

Одним из наиболее распространенных паттернов при работе с базами данных является паттерн DAO (Data Access Object). DAO обеспечивает слой абстракции между кодом JDBC API и основной программой, в частности, бизнес-логикой. По сути, DAO является реализацией слоя отображения реляционных данных в объекты и наоборот.

В классическом варианте DAO содержит только стандартные CRUD-методы (create, read, update, delete). Клиент вызывает эти методы получая или передавая в качестве аргумента так называемый DTO (Data Transfer Object). DTO предоставляет независимый набор данных и является полностью инкапсулированным объектом (записи таблицы), который также может содержать подобъекты (связанные записи другой таблицы). Основное предназначение DTO – передача данных между классами и модулями.

Итак, DTO входом и выходом одного DAO должен быть один DTO и все его дочерние объекты. Следуя этой концепции значительно упрощается написание кода, вся работа с базой данных скрыта внутри объектов DAO:

**Основная программа**

diary.cpp: <https://github.com/rudenyel/sql-cpp/blob/main/diary.cpp>

В начале работы программы вводится имя файла с базой данных. Если такой базы не существует в текущей папке, то она создается и в ней появляется пустая таблица books. Если просто нажать ввод, то откроется база данных diary.db с тестовым набором данных. Далее отображается текстовое меню с набором действий, которые можно производить с дневником чтения.