

L3. Базы данных для бота: SQLite

Практический минимум SQL и интеграция в команды бота

Наш технологический стек

Основные инструменты

- Python 3.11
- pyTelegramBotAPI (TeleBot)
- Long polling
- SQLite через sqlite3
- python-dotenv
- Стандартный logging

Принцип простоты

Локально, без Docker/webhooks/ORM. Код и БД работают на вашей машине.

Обязательно: .env не коммитим, bot.db можно добавить в .gitignore, чтобы база не push'илась на GitHub.

□ **Важно:** Простая связка → быстрее к рабочему боту. Меньше DevOps — больше кода!



Зачем базы данных в боте?

Проблема памяти

RAM стирается при перезапуске бота. Все данные теряются навсегда

SQLite – идеальное решение

Встроена в Python, нулевая настройка, один файл БД

Практические применения

Заметки пользователей, задачи, квизы, счётчики



Вопрос в зал: что теряем, если хранить всё в list() или dict()?

Ментальная модель для новичков



Таблица ≈ лист Excel

Структурированные данные в строках и столбцах. Каждая таблица решает свою задачу.



Строка ≈ запись

Одна строка = одна заметка, один пользователь, одна задача. Горизонтальная единица данных.



Столбец ≈ поле

Вертикальная характеристика: ID, текст заметки, дата создания. У всех записей одинаковые поля.

Первый объект — таблица `notes` для заметок пользователя. Схема: `id`, `user_id`, `text`, `created_at`.

<code>id</code>	<code>user_id</code>	<code>text</code>	<code>created_at</code>
1	123456	Купить молоко	2024-01-15
2	789012	Сделать ДЗ	2024-01-16

SQL-минимум: CREATE – создаём таблицу

```
CREATE TABLE IF NOT EXISTS notes (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    text TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
CREATE INDEX IF NOT EXISTS idx_notes_user ON notes(user_id);
```

Ключевые элементы

- PRIMARY KEY AUTOINCREMENT — авто-ID
- NOT NULL — обязательные поля
- Индекс по user_id ускоряет поиск

 **Опасно:** Опечатка в имени таблицы/столбца → "no such column"

SQL-минимум: INSERT – добавляем данные

```
INSERT INTO notes(user_id, text) VALUES (?, ?);
```

Безопасность превыше всего

Всегда используйте плейсхолдеры ? — никаких f-строк в SQL! Это защищает от SQL-инъекций.

Правильно: VALUES (?, ?)

Неправильно: VALUES ({user_id}, '{text}')

Мини-упражнение

Добавьте 2 заметки для user_id=123:

1. "Купить молоко"
2. "Сделать домашнее задание"

Best practice: Валидируйте длину текста на стороне Python



SQL-минимум: SELECT – читаем данные

```
SELECT id, text, created_at FROM notes  
WHERE user_id = ?  
ORDER BY id DESC  
LIMIT ?;
```

1

Базовый список

Показываем заметки конкретного пользователя, сортируем по ID (новые сверху).

2

Счётчик записей

```
SELECT COUNT(*) AS total FROM notes WHERE user_id = ?;
```



Опасно: Забыли WHERE user_id = ? → покажете чужие данные

SQL-минимум: SELECT с поиском

```
SELECT id, text FROM notes  
WHERE user_id = ? AND text LIKE '%' || ? || '%'  
ORDER BY id DESC  
LIMIT 10;
```

Поиск подстроки

LIKE — простой поиск подстроки в тексте. Чувствителен к регистру по умолчанию, но для учебного курса это подходит.

Конструкция '%' || ? || '%' безопасно обрамляет поисковый запрос символами %.

Ограничения

- Лимитируйте выдачу (10-50 записей)
- Проверяйте пустые результаты
- Обрабатывайте короткие запросы

SQL-минимум: UPDATE — обновляем записи

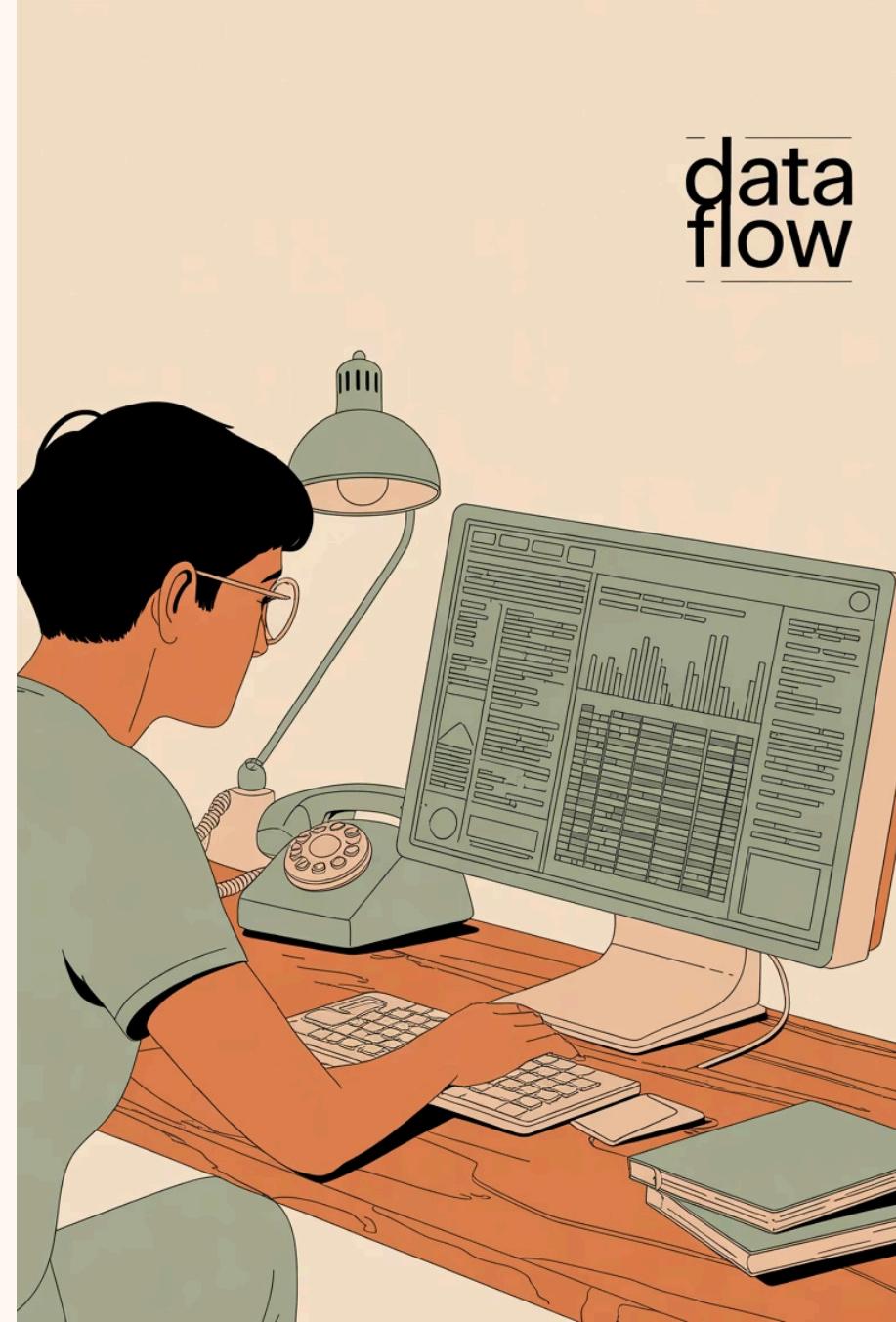
```
UPDATE notes  
SET text = ?  
WHERE user_id = ? AND id = ?;
```

Двойная защита

Фильтруем и по `user_id`, и по `id`, чтобы не трогать чужие записи. Это критически важно для безопасности.

Проверка результата

В коде проверяем `rowcount` курсора — сколько строк было обновлено. `0` = запись не найдена.





SQL-минимум: DELETE – удаляем записи

```
DELETE FROM notes  
WHERE user_id = ? AND id = ?;
```

Безопасное удаление

Та же логика: фильтруем по `user_id` и `id` одновременно. Никогда не удаляем без WHERE!

Проверка входных данных

- ID должен быть числом
- Пользователь должен быть авторизован
- Проверяем `rowcount > 0`



Опасно: Пропуск WHERE удалит ВСЁ. Защищайтесь проверками ввода!

Обёртка над sqlite3: db.py (часть 1)

```
# db.py
import sqlite3, logging
from config import DB_PATH

log = logging.getLogger(__name__)

def _connect():
    conn = sqlite3.connect(DB_PATH)
    conn.row_factory = sqlite3.Row
    conn.execute("PRAGMA foreign_keys = ON")
    return conn

def init_db():
    schema = """
CREATE TABLE IF NOT EXISTS notes (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    text TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE INDEX IF NOT EXISTS idx_notes_user ON notes(user_id);
"""

    with _connect() as conn:
        conn.executescript(schema)
    log.info("DB initialized: %s", DB_PATH)
```

Принцип "одно действие — одно подключение" снижает риск "database is locked".

Обёртка над sqlite3: db.py (часть 2)

```
def add_note(user_id: int, text: str) -> int:
    with _connect() as conn:
        cur = conn.execute(
            "INSERT INTO notes(user_id, text) VALUES (?, ?)",
            (user_id, text)
        )
        return cur.lastrowid

def list_notes(user_id: int, limit: int = 10):
    limit = max(1, min(int(limit), 50))
    with _connect() as conn:
        cur = conn.execute(
            "SELECT id, text, created_at FROM notes "
            "WHERE user_id = ? ORDER BY id DESC LIMIT ?",
            (user_id, limit)
        )
        return cur.fetchall()

def search_notes(user_id: int, needle: str, limit: int = 10):
    with _connect() as conn:
        cur = conn.execute(
            "SELECT id, text FROM notes "
            "WHERE user_id = ? AND text LIKE '%' || ? || '%' "
            "ORDER BY id DESC LIMIT ?",
            (user_id, needle, limit)
        )
        return cur.fetchall()

def update_note(user_id: int, note_id: int, text: str) -> bool:
    with _connect() as conn:
        cur = conn.execute(
            "UPDATE notes SET text=? WHERE user_id=? AND id=?",
            (text, user_id, note_id)
        )
        return cur.rowcount > 0
```



config.py: настройки и логирование

```
# config.py
import os, logging
from dotenv import load_dotenv

load_dotenv()

TOKEN = os.getenv("TOKEN")
DB_PATH = os.getenv("DB_PATH", "bot.db")

if not TOKEN:
    raise RuntimeError("Нет TOKEN в .env")

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s [%(levelname)s] %(name)s: %(message)s"
)
```

Переменные окружения

TOKEN — обязательный токен бота

DB_PATH — путь к файлу БД (по умолчанию bot.db)

❑ .env не коммитим! В репозитории храним только .env.example

Интеграция с TeleBot: основы

```
# main.py (фрагменты)
import telebot, logging
from config import TOKEN
import db

bot = telebot.TeleBot(TOKEN)
db.init_db()

@bot.message_handler(commands=['start'])
def start(message):
    bot.reply_to(message,
        "Привет! Команды: /note_add, /note_list [N], "
        "/note_find <строка>, /note_edit <текст>, "
        "/note_del <id>, /note_count"
    )
)
```

Инициализация

Сразу после создания бота вызываем `db.init_db()` для создания таблиц.

Разбор аргументов

Используем `split()` и проверки `.isdigit()` для обработки параметров команд.

Команды добавления и просмотра заметок

```
@bot.message_handler(commands=['note_add'])

def note_add(message):
    parts = message.text.split(maxsplit=1)
    if len(parts) < 2 or not parts[1].strip():
        bot.reply_to(message, "Формат: /note_add текст заметки")
        return
    note_id = db.add_note(message.from_user.id, parts[1].strip())
    bot.reply_to(message, f"Добавил заметку #{note_id}")

@bot.message_handler(commands=['note_list'])

def note_list(message):
    parts = message.text.split(maxsplit=1)
    limit = int(parts[1]) if len(parts) == 2 and parts[1].isdigit() else 10
    rows = db.list_notes(message.from_user.id, limit=limit)
    if not rows:
        bot.reply_to(message, "Заметок пока нет.")
        return
    bot.reply_to(message, "Твои заметки:\n" +
"\n".join(f'{r["id"]}: {r["text"]}' for r in rows))
```

Команды поиска и редактирования

```
@bot.message_handler(commands=['note_find'])
def note_find(message):
    parts = message.text.split(maxsplit=1)
    if len(parts) < 2:
        bot.reply_to(message, "Формат: /note_find подстрока")
        return
    rows = db.search_notes(message.from_user.id, parts[1])
    result = "\n".join(f"{r['id']}: {r['text']}" for r in rows)
    bot.reply_to(message, result if rows else "Ничего не нашёл.")

@bot.message_handler(commands=['note_edit'])
def note_edit(message):
    parts = message.text.split(maxsplit=2)
    if len(parts) < 3 or not parts[1].isdigit():
        bot.reply_to(message, "Формат: /note_edit <новый текст>")
        return
    ok = db.update_note(message.from_user.id, int(parts[1]), parts[2].strip())
    bot.reply_to(message, "Готово." if ok else "Не нашёл такую заметку.")
```

Команды удаления и подсчёта

```
@bot.message_handler(commands=['note_del'])  
def note_del(message):  
    parts = message.text.split(maxsplit=1)  
    if len(parts) < 2 or not parts[1].isdigit():  
        bot.reply_to(message, "Формат: /note_del ")  
        return  
    ok = db.delete_note(message.from_user.id, int(parts[1]))  
    bot.reply_to(message, "Удалено." if ok else "Не нашёл такую заметку.")  
  
@bot.message_handler(commands=['note_count'])  
def note_count(message):  
    count = db.count_notes(message.from_user.id)  
    bot.reply_to(message, f"У тебя {count} заметок.")  
  
if __name__ == "__main__":  
    bot.infinity_polling(skip_pending=True)
```

- Всегда проверяйте формат команд и возвращайте понятные сообщения об ошибках

ДЕМО: проверяем работу "end-to-end"

01

/note_add купить молоко

Бот должен ответить с ID созданной заметки

02

/note_list 5

Видим нашу заметку в списке

03

/note_find молоко

Находим по подстроке

04

/note_edit 1 купить молоко и хлеб

Проверяем обновление записи

05

/note_del 1, затем /note_count

Удаляем и проверяем счётчик

Проверьте: DB_PATH существует, логи выводятся в консоль, интернет-соединение активно.

Типовые ошибки и их решения

database is locked

Делайте короткоживущие соединения (как в `with _connect():`), избегайте долгих операций в хэндлерах.

no such table

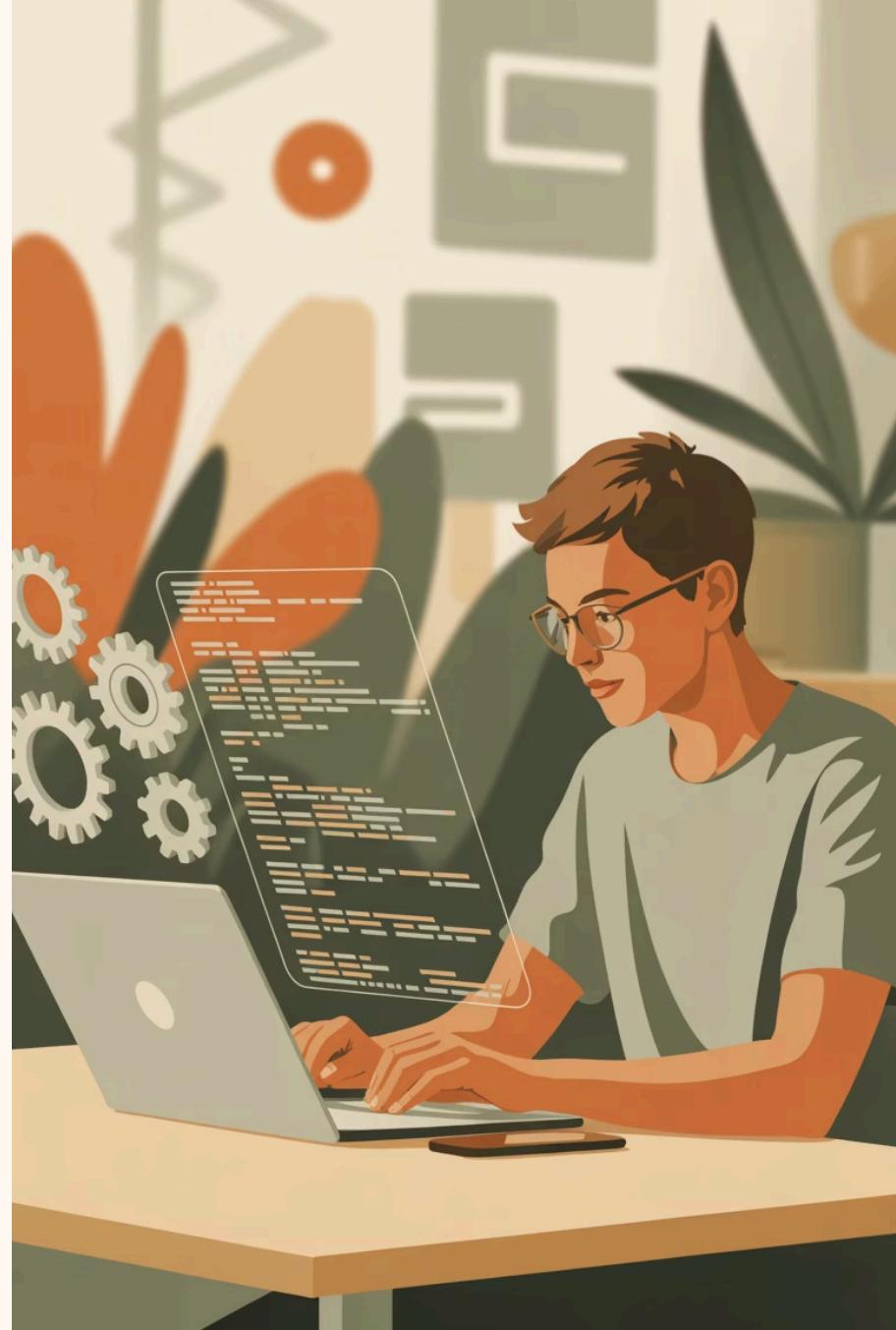
Не вызвали `init_db()` или запускаете из другой папки. Проверьте путь к БД.

Пустые результаты

Это нормально! Отвечайте дружелюбно: "не нашёл", "заметок пока нет".

Безопасность .env

Токены и `bot.db` не коммитим. Используйте `.gitignore`!





Второй пример: «Трекер задач»

Что будем создавать

- Хранить задачи пользователя (title, is_done) в SQLite
- Команды управления задачами
- Фильтрация по статусу выполнения

Модель: одна таблица tasks с индексами по user_id и is_done

Новые команды

- /task_add <текст>
- /task_list [all | done]
- /task_done <id>
- /task_clear_done

SQL-схема для tasks

```
CREATE TABLE IF NOT EXISTS tasks (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    title TEXT NOT NULL,
    is_done INTEGER NOT NULL DEFAULT 0, -- 0=false, 1=true
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX IF NOT EXISTS idx_tasks_user ON tasks(user_id);
CREATE INDEX IF NOT EXISTS idx_tasks_done ON tasks(user_id, is_done);
```

Важные решения

- Булево как INTEGER (0/1)
- DEFAULT 0 для новых задач
- Составной индекс для фильтрации

 Опасно: не храните булево как TEXT — используйте 0/1 (INTEGER)

Расширяем init_db (db.py)

```
def init_db():
    schema = """
CREATE TABLE IF NOT EXISTS notes (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    text TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE INDEX IF NOT EXISTS idx_notes_user ON notes(user_id);

CREATE TABLE IF NOT EXISTS tasks (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    title TEXT NOT NULL,
    is_done INTEGER NOT NULL DEFAULT 0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE INDEX IF NOT EXISTS idx_tasks_user ON tasks(user_id);
CREATE INDEX IF NOT EXISTS idx_tasks_done ON tasks(user_id, is_done);
"""

    with _connect() as conn:
        conn.executescript(schema)
```

- ❑ В учебном проекте держим всю схему в одном месте — в init_db()

CRUD для задач (db.py)

```
def add_task(user_id: int, title: str) -> int:
    with _connect() as conn:
        cur = conn.execute(
            "INSERT INTO tasks(user_id, title) VALUES (?, ?)",
            (user_id, title)
        )
        return cur.lastrowid

def list_tasks(user_id: int, filter_mode: str = "active", limit: int = 50):
    filter_mode = (filter_mode or "active").lower()

    if filter_mode == "active":
        where = "user_id = ? AND is_done = 0"
    elif filter_mode == "done":
        where = "user_id = ? AND is_done = 1"
    else:
        where = "user_id = ?"

    query = f"SELECT id, title, is_done FROM tasks WHERE {where} ORDER BY id DESC LIMIT ?"
    with _connect() as conn:
        cur = conn.execute(query, (user_id, limit))
        return cur.fetchall()

def mark_task_done(user_id: int, task_id: int) -> bool:
    with _connect() as conn:
        cur = conn.execute(
            "UPDATE tasks SET is_done = 1 WHERE user_id = ? AND id = ?",
            (user_id, task_id)
        )
        return cur.rowcount > 0
```

- Best practice: всегда проверяйте rowcount у UPDATE/DELETE операций

Хэндлеры TeleBot для задач

```
@bot.message_handler(commands=['task_add'])
def task_add(message):
    parts = message.text.split(maxsplit=1)
    if len(parts) < 2 or not parts[1].strip():
        bot.reply_to(message, "Формат: /task_add текст задачи")
        return
    task_id = db.add_task(message.from_user.id, parts[1].strip())
    bot.reply_to(message, f"Добавил задачу #{task_id}")

@bot.message_handler(commands=['task_list'])
def task_list(message):
    mode = "active"
    parts = message.text.split(maxsplit=1)
    if len(parts) == 2:
        arg = parts[1].strip().lower()
        if arg in ("all", "done", "все", "готово", "сделано"):
            mode = "all" if arg in ("all", "все") else "done"
    rows = db.list_tasks(message.from_user.id, filter_mode=mode, limit=20)
    if not rows:
        bot.reply_to(message, "Задач не найдено.")
        return
    def mark(r): return "Done" if r["is_done"] else "not Done"
    lines = [f"{mark(r)} {r['id']}: {r['title']}" for r in rows]
    bot.reply_to(message, "Твои задачи:\n" + "\n".join(lines))

@bot.message_handler(commands=['task_done'])
def task_done(message):
    parts = message.text.split(maxsplit=1)
    if len(parts) < 2 or not parts[1].isdigit():
        bot.reply_to(message, "Формат: /task_done <id>")
        return
    ok = db.mark_task_done(message.from_user.id, int(parts[1]))
    bot.reply_to(message, "Готово." if ok else "Не нашёл такую задачу.")

@bot.message_handler(commands=['task_clear_done'])
def task_clear_done(message):
    n = db.clear_done(message.from_user.id)
    bot.reply_to(message, f"Удалено выполненных задач: {n}")
```

ДЕМО: сценарий для задач



- ☐ Важно: проверьте, что DB_PATH указывает на записываемый файл рядом с проектом

Валидация и UX-мелочи



Ограничения длины

Ограничите длину title (например, ≤ 200 символов) на стороне Python до вставки в БД



Дружелюбные ответы

«Не нашёл задачу», «Список пуст» вместо технических ошибок



Визуальные индикаторы

Смайлики / упрощают чтение списков задач

- ❑ Опасно: не отвечайте просто «Ок» без контекста — дублируйте ID/название операции

Квиз: вопросы 1-4

1. Что делает `row_factory = sqlite3.Row`?

- Даёт доступ к столбцам по имени
- Включает автокоммит
- Создаёт индекс

2. Почему нельзя форматировать SQL через f-строки?

- Медленно работает
- Опасно (SQL-инъекции)
- Не работает в sqlite3

3. Как удалить ОДНУ заметку с `id=5` текущего пользователя?

- `DELETE FROM notes WHERE id=5;`
- `DELETE FROM notes WHERE user_id=? AND id=?;`
- `DELETE FROM notes WHERE text LIKE '%5%';`

4. Что делает `lastrowid` у курсора?

- Возвращает последний AUTO ID
- Количество строк
- SQL-код

Квиз: вопросы 1-4

1. Что делает `row_factory = sqlite3.Row?`

- А: Даёт доступ к столбцам по имени
- В: Включает автокоммит
- С: Создаёт индекс

2. Почему нельзя форматировать SQL через f-строки?

- А: Медленно работает
- **В: Опасно (SQL-инъекции)**
- С: Не работает в sqlite3

3. Как удалить ОДНУ заметку с `id=5` текущего пользователя?

- А: `DELETE FROM notes WHERE id=5;`
- **В: `DELETE FROM notes WHERE user_id=? AND id=?;`**
- С: `DELETE FROM notes WHERE text LIKE '%5%';`

4. Что делает `lastrowid` у курсора?

- А: Возвращает последний AUTO ID
- В: Количество строк
- С: SQL-код

КВИЗ: ВОПРОСЫ 5-7

5. Что вернёт rowCount после UPDATE?

- Количество изменённых строк
- Всегда 1
- ID обновлённой строки

6. Где хранить TOKEN?

- В .env файле
- В main.py
- В README

7. Что делает skip_pending=True?

- Пропускает накопившиеся апдейты при старте
- Отключает polling
- Перезапускает бота

КВИЗ: ВОПРОСЫ 5-7

5. Что вернёт rowCount после UPDATE?

- А: Количество изменённых строк
- В: Всегда 1
- С: ID обновлённой строки

6. Где хранить TOKEN?

- А: В .env файле
- В: В main.py
- С: В README

7. Что делает skip_pending=True?

- А: Пропускает накопившиеся апдейты при старте
- В: Отключает polling
- С: Перезапускает бота

Квиз: финальные вопросы 8-10

8. Чего НЕ хватает в запросе для поиска?

```
SELECT id, text FROM notes WHERE text LIKE '%' || ? || '%'
```

- Ограничения по user_id
- LIMIT
- ORDER BY

9. Чем опасен DELETE FROM tasks; без WHERE?

- Медленно работает
- Удалит все строки
- Ничем не опасен

10. Что включаем в init_db()?

- CREATE TABLE/INDEX, один раз при старте
- Только SELECT
- Только INSERT

Отличная работа! Теперь вы знаете основы интеграции SQLite с Telegram-ботами

Квиз: финальные вопросы 8-10

8. Чего НЕ хватает в запросе для поиска?

```
SELECT id, text FROM notes WHERE text LIKE '%' || ? || '%'
```

- A: Ограничения по user_id
- B: LIMIT
- C: ORDER BY

9. Чем опасен DELETE FROM tasks; без WHERE?

- A: Медленно работает
- B: Удалит все строки
- C: Ничем не опасен

10. Что включаем в init_db()?

- A: CREATE TABLE/INDEX, один раз при старте
- B: Только SELECT
- C: Только INSERT

Отличная работа! Теперь вы знаете основы интеграции SQLite с Telegram-ботами

Пагинация длинных списков

Необходимость пагинации

1 Сообщение Telegram имеет ограничение по длине, поэтому длинные списки выдачи нужно разбивать на страницы для пользователя.

Реализация

2 Для "нарезки" данных на страницы в SQL используются операторы `LIMIT` и `OFFSET`. В Python реализуется небольшая функция для вычисления смещения и отправки запроса.

Пример команды

3 Пользователь может запросить конкретную страницу, например: `/note_list <page> <size>`.

SQL: Извлечение данных с пагинацией

```
SELECT id, text, created_at
FROM notes
WHERE user_id = ?
ORDER BY id DESC
LIMIT ? OFFSET ?;
```

Python: Функция list_notes_page

```
def list_notes_page(user_id: int, page: int = 1, size: int = 10):
    page = max(1, page); size = max(1, min(size, 50))
    offset = (page - 1) * size
    with _connect() as conn:
        cur = conn.execute(
            "SELECT id, text, created_at FROM notes "
            "WHERE user_id = ? ORDER BY id DESC LIMIT ? OFFSET ?",
            (user_id, size, offset)
        )
        return cur.fetchall()

def _send_long(bot, chat_id, text, chunk=4000):
    for i in range(0, len(text), chunk):
        bot.send_message(chat_id, text[i:i+chunk])
```

Важно: Формат команды `/note_list 2 15` означает: страница 2, по 15 записей на странице.

Поиск без учёта регистра



- Пользователям привычно: «**МОЛОКО**» ≈ «**Молоко**». Ваш бот должен быть таким же гибким.
- В SQLite это просто: используйте оператор `COLLATE NOCASE` или функцию `LOWER()`.

SQL: Извлечение данных с учётом регистра

```
SELECT id, text  
FROM notes  
WHERE user_id = ?  
AND text LIKE '%' || ? || '%' COLLATE NOCASE  
ORDER BY id DESC  
LIMIT 10;
```

Подсказка: для кириллицы чаще хватает `LOWER(text) LIKE LOWER('%' || ? || '%')`. Этот подход обеспечивает надёжную работу с различными алфавитами.

Ограничения в схеме: CHECK/UNIQUE

Валидация на уровне БД

Часть правил валидации данных можно «закрепить»
прямо в схеме таблицы, обеспечивая целостность данных
независимо от логики приложения.

Предотвращение ошибок

Используйте ограничения, чтобы автоматически пресекать
пустые/слишком длинные строки и дубликаты, повышая
надёжность вашей базы данных.

Пример SQL-схемы с ограничениями:

```
CREATE TABLE IF NOT EXISTS notes (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    text TEXT NOT NULL CHECK(length(text) BETWEEN 1 AND 500),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(user_id, text) ON CONFLICT IGNORE
);
```

- ❑ **Опасно:** не забывайте обрабатывать ситуацию «ничего не вставилось» (например, из-за UNIQUE-конфликта) на стороне Python.

Обработка ошибок sqlite3: дружелюбный UX

Перехват исключений

Ловите специфические исключения (`IntegrityError`, `DatabaseError`) и отвечайте пользователю понятными, человечными сообщениями, вместо технических ошибок.

Проверка изменений

Для операций `UPDATE` и `DELETE` всегда проверяйте свойство `cursor.rowcount`, чтобы убедиться, что изменения были успешно применены.

Пример безопасной функции для добавления заметки с обработкой возможных конфликтов:

```
import sqlite3, logging
log = logging.getLogger(__name__)

def add_note_safe(user_id: int, text: str) -> int | None:
    try:
        return add_note(user_id, text) # Предполагается, что add_note() вызывает выполнение SQL
    except sqlite3.IntegrityError:
        log.warning("Constraint failed (user=%s)", user_id)
        # Например, вернуть None или специальный код ошибки для UI
        return None
    except sqlite3.DatabaseError as e:
        log.exception("DB error: %s", e)
        # Обработка других ошибок базы данных
        return None
```

Резервная копия и экспорт данных

- Создавайте **резервные копии** базы данных, используя встроенный метод SQLite `backup()`.
- Реализуйте **экспорт данных** (например, заметок) в TXT-файл и отправляйте его пользователю как документ Telegram.

Код: Создание резервной копии базы данных

```
import sqlite3
from config import DB_PATH

def backup_to(path: str = "backup.db"):
    with sqlite3.connect(DB_PATH) as src, sqlite3.connect(path) as dst:
        src.backup(dst)
```

Код: Хэндлер для экспорта заметок

```
@bot.message_handler(commands=['note_export'])
def note_export(message):
    # Извлекаем до 1000 заметок пользователя
    rows = db.list_notes(message.from_user.id, limit=1000)
    if not rows:
        bot.reply_to(message, "Экспортировать нечего.")
        return

    # Создаем временный файл для экспорта
    fname = f"notes_{message.from_user.id}.txt"
    with open(fname, "w", encoding="utf-8") as f:
        for r in rows:
            f.write(f"{r['id']}\t{r['text']}\n") # Записываем ID и текст заметки

    # Отправляем файл пользователю
    with open(fname, "rb") as f:
        bot.send_document(message.chat.id, f)
```

- Важно:** Файлы `backup.db` и другие сгенерированные экспорты не должны попадать в систему контроля версий (например, Git). Добавьте их в файл `.gitignore!`

Мини-миграции: PRAGMA user_version

Для управления изменениями в схеме базы данных SQLite можно использовать механизм мини-миграций, основанный на специальной прагме.

→ Версионирование схемы

Используйте `PRAGMA user_version` как счётчик для отслеживания текущей версии схемы непосредственно в самой базе данных.

→ Пошаговое обновление

Реализуйте функцию миграции, которая последовательно применяет изменения структуры базы данных, если текущая версия ниже требуемой.

Пример кода: Функция для мини-мigrations

```
def migrate():
    with _connect() as conn:
        ver = int(conn.execute("PRAGMA user_version").fetchone()[0])
        if ver < 1:
            conn.executescript("""
                CREATE TABLE IF NOT EXISTS notes (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    user_id INTEGER NOT NULL,
                    text TEXT NOT NULL,
                    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                );
                CREATE INDEX IF NOT EXISTS idx_notes_user ON notes(user_id);
            """)
            conn.execute("PRAGMA user_version = 1")
            ver = 1
        if ver < 2:
            conn.execute("ALTER TABLE notes ADD COLUMN archived INTEGER NOT NULL DEFAULT 0")
            conn.execute("PRAGMA user_version = 2")
```

□ **Важно:** Вызывайте функцию `migrate()` при старте приложения рядом с инициализацией базы данных (`init_db()`).

Меньше «database is locked»: WAL и тайм-аут

SQLite, будучи файловой базой данных, по умолчанию может испытывать проблемы с конкурентным доступом, что приводит к ошибкам «database is locked». Для повышения производительности и надёжности в условиях одновременного чтения и записи, особенно в ботах, применяются следующие подходы:

- Короткоживущие соединения и WAL- журналирование

Использование **WAL (Write-Ahead Logging)** позволяет операциям чтения и записи происходить одновременно без взаимных блокировок. Это значительно улучшает параллелизм, так как записи идут в отдельный файл журнала, пока основной файл базы данных остаётся доступным для чтения. Для учебного бота это особенно удобно, поскольку он прост в настройке и эффективен для большинства сценариев.

- Установка `busy_timeout`

Если база данных временно заблокирована другой операцией, `busy_timeout` предписывает SQLite подождать указанное время (в миллисекундах), прежде чем выдать ошибку. Это помогает избежать мгновенных сбоев при кратковременных блокировках, делая приложение более устойчивым к пиковым нагрузкам.

Пример: Настройка соединения с базой данных

```
def _connect():
    conn = sqlite3.connect(DB_PATH, timeout=5.0) # Таймаут для открытия соединения
    conn.row_factory = sqlite3.Row
    conn.execute("PRAGMA foreign_keys = ON")
    conn.execute("PRAGMA journal_mode = WAL") # Включаем Write-Ahead Logging
    conn.execute("PRAGMA busy_timeout = 5000") # 5 секунд ожидания при занятой БД
    return conn
```

- **Важно:** Несмотря на преимущества WAL, всё равно держимся подхода «одно действие — одно подключение» для максимальной простоты и предсказуемости поведения бота.

Связи и JOIN: категории для задач

Для более эффективной организации задач часто требуется привязка к категориям. Это позволяет группировать задачи и упрощать навигацию. Рассмотрим, как реализовать такую связь и извлекать данные с её использованием.

→ Модель 1→N: категория → задачи

Одна категория может содержать множество задач, но каждая задача принадлежит только одной категории. Это классическая связь "один ко многим".

→ LEFT JOIN для списков

Используем `LEFT JOIN` для получения списка задач вместе с информацией о их категориях, даже если категория не указана.

SQL-схема для категорий

```
CREATE TABLE IF NOT EXISTS categories (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    title TEXT NOT NULL
);
-- При миграции схемы tasks:
-- ALTER TABLE tasks ADD COLUMN category_id INTEGER REFERENCES categories(id);
```

Выборка задач с категориями (LEFT JOIN)

```
SELECT t.id, t.title, t.is_done, c.title AS cat
FROM tasks t
LEFT JOIN categories c ON c.id = t.category_id
WHERE t.user_id = ?
ORDER BY t.id DESC
LIMIT 20;
```

Этот запрос выбирает до 20 последних задач пользователя, включая название категории. Если у задачи нет категории, поле `cat` будет `NULL`.

Отчёты: GROUP BY и «стата за 7 дней»



Для понимания динамики использования бота и получения ценных инсайтов, нам необходимы агрегированные отчёты. SQL-функция GROUP BY позволяет сгруппировать данные и применить к ним агрегатные функции, такие как COUNT, SUM или AVG.

→ Ежедневная статистика

Мы можем узнать, сколько записей или задач было добавлено в определённый день, или сколько задач было закрыто.

→ Компактное представление

Агрегированные данные позволяют компактно отображать информацию, что делает отчёты более наглядными и удобочитаемыми.

SQL-запрос: Заметки за 7 дней

```
SELECT date(created_at) AS d, COUNT(*) AS total  
FROM notes  
WHERE user_id = ?  
GROUP BY date(created_at)  
ORDER BY d DESC  
LIMIT 7;
```

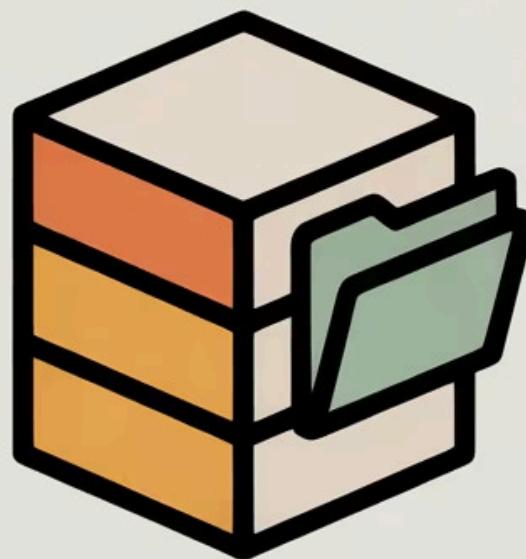
Этот запрос выбирает количество заметок, созданных пользователем за каждый день, за последние 7 дней, сортируя по дате в убывающем порядке.

Python: Визуализация «текстовой гистограммой»

```
def bar(n):  
    return "#" * min(n, 30)  
  
# Предполагается, что 'rows' получен из выполнения SQL-запроса выше  
lines = [f"{row['d']}: {bar(row['total'])} {row['total']}\" for row in rows]  
bot.reply_to(message, "Заметки за 7 дней:\n" + "\n".join(lines))
```

Простая функция `bar` создаёт текстовую "гистограмму" из символов, ограничивая длину до 30 символов, что помогает компактно визуализировать количество заметок прямо в чате бота.

Быстрые тесты db.py (без Telegram)



Для эффективной разработки и отладки важно иметь возможность тестировать логику работы с базой данных отдельно от интеграции с Telegram. Это не полнофункциональные юнит-тесты, а скорее быстрые проверки корректности функций CRUD.

→ Проверка CRUD в изоляции

Мы проверяем все основные операции (создание, чтение, обновление, удаление) на временной базе данных. Это гарантирует, что функционал работает как ожидается, без побочных эффектов или зависимостей от внешних компонентов.

→ Переопределение DB_PATH для теста

Для каждого тестового прогона мы временно меняем путь к файлу базы данных (DB_PATH), используя возможности модуля tempfile. Это позволяет запускать тесты в чистой, изолированной среде, не затрагивая реальные данные.

Пример кода: tests_db_demo.py

```
import tempfile, os
import db

def test_add_list():
    with tempfile.TemporaryDirectory() as d:
        db.DB_PATH = os.path.join(d, "test.db") # временная БД
        db.init_db()
        note_id = db.add_note(1, "тест")
        rows = db.list_notes(1, limit=5)
        assert any(r["id"] == note_id for r in rows)
```

Важно: Держите модули разделёнными (config, db, main) — это значительно упрощает тестирование и поддержку кода, позволяя проверять каждый компонент независимо.



Что дальше: семинары и практика

На семинарах изучим

- Расширение команд бота
- Простые тесты функций db.py
- Практика GitHub: коммиты, PR, ревью
- Мини-квиз по SQL и командам

Повторим ключевое

- CREATE/INSERT/SELECT/UPDATE/DELETE
- Интеграция SQLite + TeleBot
- Локальный запуск через long polling
- Безопасность: плейсхолдеры, фильтрация

❑ Если бот молчит — проверьте venv, TOKEN в .env и интернет-соединение

Вопросы?

Код доступен в слайдах и шпаргалках. Увидимся на семинарах!