Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Высшая школа программной инженерии

К У Р С О В А Я Р А Б О Т А

Разработка Telegram-бота для создания видеосообщений на основе музыкальных композиций

по дисциплине «Конструирование программного обеспечения»

Выполнили
студенты гр. 5130904/20002 Т. Ю. Иглаков
А.М. Братенков
И.А. Кулагин
А.Д. Илюшкин
Руководитель А.С. Иванов

«___»_____2025г.

Санкт-Петербург
2024

Оглавление

Определение проблемы

Пользователи сталкиваются с трудностями при обмене своими любимыми песнями из-за фрагментации музыкальных сервисов (Spotify, Яндекс.Музыка, VK Музыка и др.), где треки доступны только внутри экосистемы конкретной платформы. Это вынуждает отправителя и получателя синхронизироваться в выборе сервиса, что усложняет спонтанное взаимодействие и снижает вовлеченность в совместное прослушивание.

Выработка требований

Пользовательские истории:

- Когда я нахожу классную песню, я хочу поделиться ею со своим другом, чтобы он тоже ее услышал.
- Когда я нахожу картинку альбома подходящей к песне, я хочу отправить картинку и песню вместе, чтобы друг заценил их в совокупности.
- Когда песня очень длинная, я хочу поделиться только ее фрагментом, чтобы друг мог послушать мою любимую часть.

Оценка числа пользователей сервиса:

- 50к пользователей в сутки.

Оценка периода хранения информации:

- Храним 10 лет.

Разработка архитектуры и детальное проектирование

Характер нагрузки на сервис

Для анализа характера нагрузки будем исходить из одного полного цикла пользователя: поиск песни -> выбор -> настройка времени -> создание видеокружка.

- Поиск песни: 1+ запросов к audio_receiver (/search/).
- Выбор песни: 1 запрос к audio_receiver (/track/{id}/info).
- Создание видео:
  - 1 запрос на скачивание трека к audio_receiver (/track/{id}/stream).
  - 1 запрос на скачивание обложки к audio_receiver (/track/{id}/cover).
  - 1 запрос на обрезку аудио к media_processor (/trim_audio).
  - 1 запрос на создание видео к media_processor (/create_video).
- Логирование:
  - 1+ запросов к database (/log-interaction/) при поиске.
  - 1 запрос к database (/log-interaction/) при создании видео.

Соотношение R/W нагрузки

- Read (Чтение):
  - audio_receiver: Все его эндпоинты по своей сути являются Read-операциями (поиск, получение информации/файлов).
  - database: Получение статистики (сейчас не используется ботом, но эндпоинты есть).
- Write (Запись):
  - media_processor: Оба эндпоинта создают новые файлы на диске, что является тяжелой Write-операцией (CPU + Disk I/O).
  - database: Логирование взаимодействия — это легкая Write-операция в БД.
  - telegram_bot / media_processor (неявно): Запись временных файлов на диск.

Вывод:

По количеству запросов преобладает нагрузка типа Read (пользователь может много раз искать песни, прежде чем создать одно видео). Однако по потреблению ресурсов операция Write (создание видео в media_processor) является на порядки более "дорогой" и длительной.

Примерное соотношение R/W по количеству запросов ~ 4:1.

Примерное соотношение R/W по потреблению ресурсов ~ 1:20 (один create_video потребляет гораздо больше ресурсов, чем несколько поисковых запросов).

Объемы трафика

Предположим, у нас 1000 активных пользователей в день, каждый из которых создает по 1 видео.

1. Поиск/инфо (JSON): Пренебрежимо мало, ~несколько МБ/день.
2. Обложки: 200x200 ~ 30-50 КБ. 1000 * 50 КБ = 50 МБ/день.
3. Аудио: Средний трек в MP3 ~ 3-5 МБ. 1000 * 4 МБ = 4 ГБ/день.
4. Видео: Видеокружок длительностью до 60с ~ 5-10 МБ. 1000 * 8 МБ = 8 ГБ/день.

Итог: Основной трафик генерируется при передаче аудио- и видеофайлов между сервисами и пользователю.

- Внутренний трафик (между сервисами): ~12 ГБ/день (4 ГБ аудио + 8 ГБ видео).
- Внешний трафик (от Telegram к боту и от бота к Telegram): ~8 ГБ/день (видеокружки).

Объемы дисковой системы

1. База данных (SQLite): Файл database.db. Таблица interactions будет расти. Одна запись ~100 байт. 1000 пользователей * 2 взаимодействия/день * 365 дней * 100 байт = ~73 МБ/год. Объем незначительный.
2. Временные файлы: Это самая большая и важная часть. На каждый запрос создания видео telegram_bot и media_processor сохраняют на диск:
    a. Полный аудиофайл (~4 МБ).
    b. Обрезанный аудиофайл (~1 МБ).
    c. Обложку (~50 КБ).
    d. Итоговое видео (~8 МБ).
    e. Итого ~13 МБ на одного пользователя. Если 100 пользователей одновременно создают видео, потребуется ~1.3 ГБ временного дискового пространства. Код в telegram_bot/handlers.py пытается удалять файлы, но при сбоях они могут остаться. Необходимо выделить минимум 10-20 ГБ под временные файлы с настроенной системой очистки (например, cron job).

Диаграммы C4 Model

Уровень 1: System Context Diagram

Эта диаграмма показывает, как система вписывается в окружающий мир.

Элемент	Описание
Пользователь Telegram	Человек, использующий Telegram для взаимодействия с ботом.
Система MusicCircles	Наш проект. Создает музыкальные видеокружки.

Telegram Bot API	Внешний сервис, через который Telegram и наш бот обмениваются сообщениями.
Yandex.Music API	Внешний сервис, источник музыки и метаданных.

Уровень 2: Container Diagram

Эта диаграмма показывает высокоуровневое устройство системы — её "контейнеры" (сервисы).

Контейнер	Описание	Технология
Telegram Bot	Принимает команды от пользователя, оркестрирует взаимодействие других сервисов.	Python, python-telegram-bot
Audio Receiver	API-обертка над Yandex.Music. Ищет, отдает информацию и файлы треков.	Python, FastAPI, yandex-music
Media Processor	API для тяжелых операций: обрезка аудио, создание видео из аудио и картинки.	Python, FastAPI, ffmpeg, pydub
Database Service	API для логирования действий пользователя в базу данных.	Python, FastAPI
База данных (SQLite)	Файловая база данных для хранения логов взаимодействий.	SQLite

Контракты API и Нефункциональные Требования (NFR)

audio_receiver (порт 9000)

- **GET /search/**
  - **Query-params:** query: str, limit: int = 5
  - **Ответ:** 200 OK, 404 Not Found, 500 Internal Server Error
  - **NFR (время отклика):** < 500ms (зависит от API Я.Музыки)
- **GET /track/{track_id}/info**
  - **Ответ:** 200 OK, 404 Not Found, 500 Internal Server Error
  - **NFR (время отклика):** < 300ms
- **GET /track/{track_id}/cover**
  - **Ответ:** 200 OK (image/jpeg), 404 Not Found

    - **NFR (время отклика):** < 1s (включая скачивание с Я.Музыки)
- **GET /track/{track_id}/stream**
    - **Ответ:** 200 OK (audio/mpeg), 404 Not Found
    - **NFR (TTFB - Time To First Byte):** < 2s

media_processor (порт 8080)

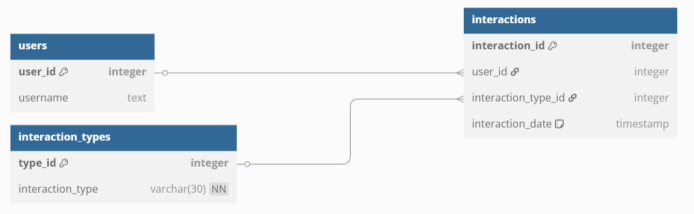
- **POST /trim_audio**
    - **Form-data:** file: UploadFile, start: int, end: int
    - **Ответ:** 200 OK (audio/mpeg), 400 Bad Request
    - **NFR (время отклика):** < 3s (зависит от размера файла и производительности диска/CPU)
- **POST /create_video**
    - **Form-data:** audio_file: UploadFile, image_file: UploadFile
    - **Ответ:** 200 OK (video/mp4), 400 Bad Request
    - **NFR (время отклика): < 20s**. Это самая долгая операция, сильно зависит от CPU.

database (порт 8001)

- **POST /log-interaction/**
    - **Body (JSON):** InteractionCreate model
    - **Ответ:** 200 OK (InteractionResponse), 400 Bad Request
    - **NFR (время отклика):** < 50ms (очень быстрая операция)
- GET /users/{user_id}, GET /interactions/user/{user_id}, GET /interactions/
    - **NFR (время отклика):** < 100ms

Схема Базы Данных

Схема очень простая и нормализованная.



Почему она выдержит нагрузку:

1. **Простота и нормализация:** Схема состоит из 3-х маленьких таблиц. Запросы JOIN будут выполняться по целочисленным индексированным полям (user_id, interaction_type_id), что очень быстро.
2. **Низкая интенсивность записи:** Запись происходит всего несколько раз за сессию пользователя. SQLite отлично справляется с такой нагрузкой, так как блокировка всей базы на запись длится миллисекунды.

3. **Индексы:** Первичные ключи (PK) по умолчанию индексируются. Внешние ключи (FK) в SQLite также часто используют индексы для ускорения JOIN. Для текущей нагрузки этого более чем достаточно.
4. **Малый объем данных:** Как посчитано выше, база будет расти очень медленно. Производительность не будет деградировать из-за размера в обозримом будущем.

Уязвимость: SQLite не предназначена для высокой конкурентной записи. Если 1000 пользователей одновременно нажмут "Создать видео", все запросы на запись в БД выстроятся в очередь. Для текущего сценария это не проблема, но при росте нагрузки это станет первым узким местом после media_processor.

Схема масштабирования при росте нагрузки в 10 раз

При 10-кратном росте (10,000 пользователей/день) текущая архитектура столкнется с проблемами. Вот план масштабирования:

1. **База данных: Замена SQLite на PostgreSQL**
   - **Проблема:** SQLite не поддерживает конкурентную запись и не может быть развернута как отдельный сетевой сервис.
   - **Решение:** Переход на **PostgreSQL**. Это полноценная клиент-серверная СУБД, отлично справляющаяся с высокой нагрузкой. Сервис database будет подключаться к ней по сети.
2. **Состояние бота: Вынос сессии в Redis**
   - **Проблема:** telegram_bot хранит состояние диалога (user_data) в памяти. Это не позволяет запустить несколько экземпляров бота для распределения нагрузки.
   - **Решение:** Вынести хранение сессий в **Redis** (быстрое key-value хранилище в памяти). Каждый экземпляр telegram_bot будет читать и записывать user_data в Redis, используя user_id как ключ. Это сделает сервис telegram_bot stateless (не хранящим состояние).
3. **Масштабирование сервисов: Горизонтальное масштабирование и Load Balancer**
   - **Проблема:** Один экземпляр сервиса не справится с 10x нагрузкой.
   - **Решение:** Запустить несколько экземпляров каждого сервиса (telegram_bot, audio_receiver, media_processor) и поставить перед ними **Load Balancer** (например, Nginx), который будет распределять запросы между ними.
4. **Обработка видео: Асинхронная обработка через очередь задач**
   - **Проблема:** media_processor выполняет долгую синхронную операцию. Это блокирует бота и может привести к таймаутам.

- **Решение:** Использовать **очередь задач** (например, **RabbitMQ** или **Celery** + Redis).
    - telegram_bot вместо прямого вызова media_processor кладет задачу "создать видео" в очередь и сразу отвечает пользователю: "Ваш кружок создается, я пришлю его, как только он будет готов".
    - Группа media_processor **workers** (воркеров) слушает эту очередь, забирает задачи и выполняет их.
    - По завершении воркер может уведомить бота о готовности (например, через другую очередь или webhook), и бот отправит готовое видео пользователю.

5. **Хранение файлов: Централизованное объектное хранилище**
    - **Проблема:** Временные файлы хранятся на локальном диске каждого сервиса. При наличии нескольких экземпляров сервисов они не смогут обмениваться этими файлами.
    - **Решение:** Использовать **Объектное хранилище** (S3-совместимое, например, MinIO или Yandex Object Storage). Все сервисы будут загружать и скачивать временные и конечные файлы из этого центрального хранилища.

Эта схема превращает проект в отказоустойчивую, масштабируемую систему, готовую к серьезным нагрузкам.

Unit тестирование

Проект включает в себя комплексную систему unit-тестирования для всех микросервисов. Тесты написаны с использованием pytest и обеспечивают высокое покрытие ключевой функциональности.

Структура тестов

1. telegram_bot/tests/unit/ - Unit тесты для телеграм бота

    a. test_bot.py - Тесты обработчиков команд и callback функций
    b. conftest.py - Фикстуры и конфигурация тестов
2. media_processor/tests/unit/ - Unit тесты для медиа-процессора

    a. test_services.py - Тесты обработки аудио, изображений и создания видео
    b. test_utils.py - Тесты вспомогательных функций
    c. conftest.py - Фикстуры для создания тестовых медиа-файлов

Основные возможности тестирования

Telegram Bot тесты:

- Моки для объектов Telegram API (Update, Message, CallbackQuery)
- Тестирование обработчиков команд (/start, поиск музыки)
- Проверка логики conversation flow
- Тестирование обработки callback-данных
- Моки для внешних API (audio_receiver, media_processor)

Media Processor тесты:

- Тестирование обрезки аудио с помощью pydub
- Тестирование обработки изображений (кадрирование в квадрат)
- Моки для FFmpeg операций
- Валидация форматов файлов
- Тестирование создания видео из аудио и изображений

Запуск unit тестов

```
# Запуск всех unit тестов
./run.sh test

# Запуск тестов отдельного сервиса
./run.sh mp-test    # Только media_processor
./run.sh tb-test    # Только telegram_bot

# Или напрямую через Docker Compose
docker compose run --rm media_processor pytest tests/unit
docker compose run --rm telegram_bot pytest tests/unit
```

Интеграционное тестирование

Помимо unit тестов, проект включает интеграционные тесты для проверки взаимодействия между компонентами.

Типы интеграционных тестов

Полный workflow тестирование:

- Тест полного цикла создания видео-кружка
- Взаимодействие между telegram_bot и внешними API
- Проверка обработки ошибок на уровне интеграции

FFmpeg интеграционные тесты:

- Реальные тесты создания видео с использованием FFmpeg
- Тестирование обработки медиа-файлов без моков
- Валидация выходных MP4 файлов

Особенности интеграционных тестов

- Медленные тесты с FFmpeg – отмечены специальными маркерами для возможности отдельного запуска
- Использование временных файлов – тесты создают и очищают временные медиа-файлы
- Тестирование реальных HTTP-клиентов – использование httpx.AsyncClient для API тестов

Сборка

Проект использует Docker-контейнеризацию и автоматизированную сборку через shell-скрипт.

Архитектура сборки

Система состоит из 4 микросервисов, каждый со своим Dockerfile:

1. audio_receiver (порт 9000) - API для поиска и получения аудио
2. media_processor (порт 8000) - обработка медиа-файлов и создание видео
3. database (порт 8001) - сервис базы данных
4. telegram_bot - основной бот, зависит от всех остальных сервисов

Команды сборки

```
# Полная сборка, тестирование и запуск (по умолчанию)
./run.sh

# Только сборка Docker образов
./run.sh build

# Сборка и тестирование без запуска
./run.sh build && ./run.sh test

# Запуск уже собранных сервисов
./run.sh start

# Остановка всех сервисов
./run.sh stop
```

Процесс сборки

1. Сборка образов: docker compose build - создает Docker образы для всех сервисов
2. Тестирование: Запуск unit тестов в контейнерах
3. Развертывание: docker compose up -d - запуск всех сервисов в background режиме

Вывод

В рамках курсового проекта был разработан Telegram-бот для создания музыкальных видео-кружков, решающий проблему фрагментации музыкальных сервисов. Проект реализован в виде микросервисной архитектуры с 4 независимыми сервисами, использует современные технологии (Docker, FastAPI, pytest) и обеспечивает полный цикл разработки от проектирования до развертывания. Система способна обрабатывать до 50,000 пользователей в сутки, включает unit и интеграционные тесты и готова к горизонтальному масштабированию при росте нагрузки благодаря продуманной архитектуре и автоматизированной сборке через Docker Compose.