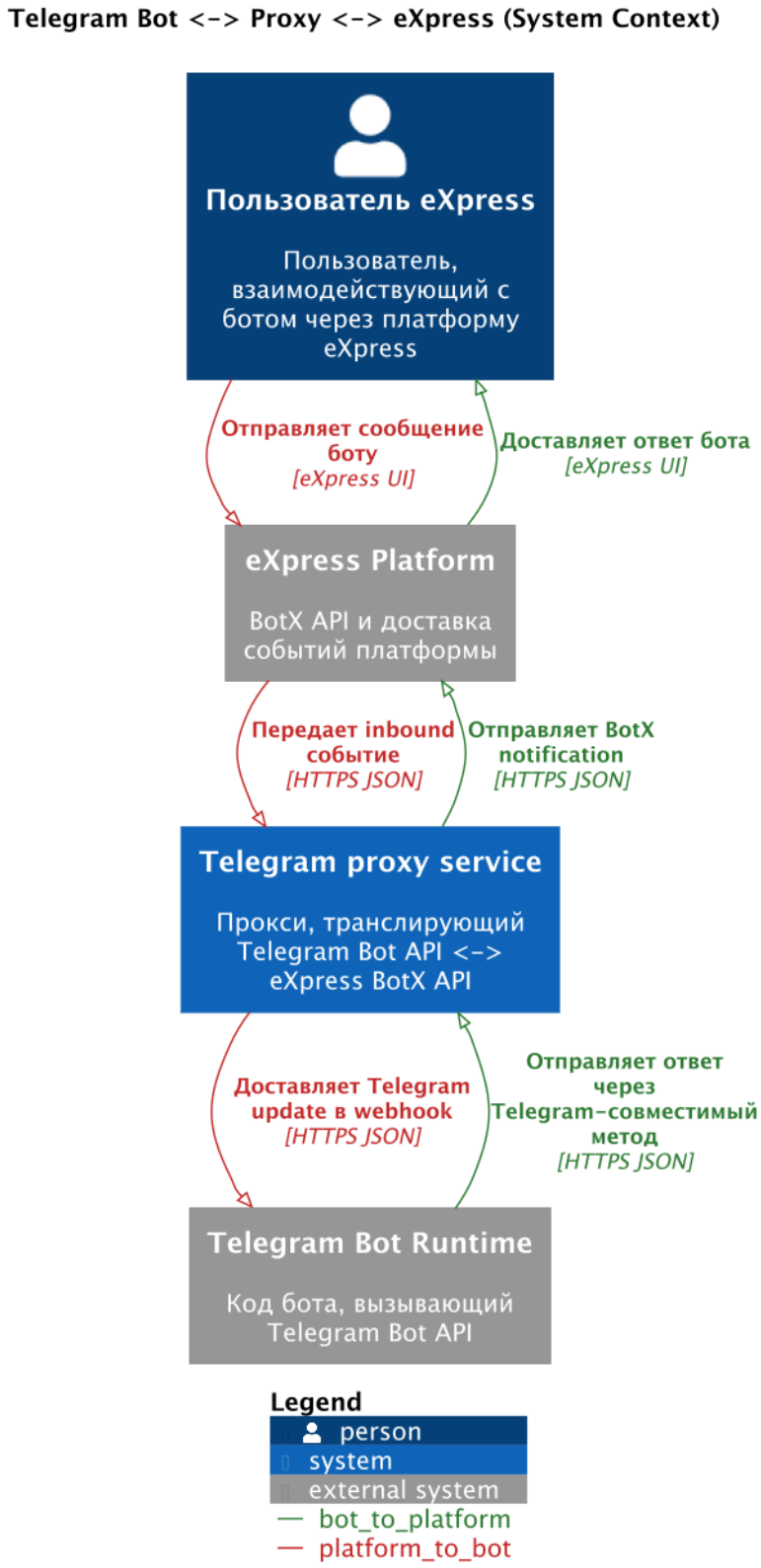
Пользовательская инструкция

1. Назначение сервиса



`telegram-proxy` предназначен для адаптации существующих Telegram-ботов к работе с платформой eXpress без полного переписывания кода интеграции.

Сервис выступает в качестве адаптера между Telegram Bot API и eXpress BotX API:

- код Telegram-бота продолжает вызывать привычные методы Telegram Bot API;
- `telegram-proxy` принимает эти запросы и преобразует их в вызовы BotX API;
- `telegram-proxy` принимает входящие события из eXpress, преобразует их в Telegram-совместимый формат и доставляет в webhook Telegram-бота;
- состояние ботов, маршруты чатов, индексы сообщений и служебные данные хранятся в Redis.

---

2. Статус проекта

Проект находится в стадии MVP.

Текущая версия покрывает основной сценарий интеграции между Telegram Bot API и eXpress BotX API, но пока не претендует на полную совместимость с Telegram Bot API и не должна рассматриваться как полностью production-ready решение.

Что это значит:

- реализован базовый end-to-end поток для ключевых сценариев;
- поддержка Telegram Bot API остается частичной;
- часть методов, режимов и edge cases еще не реализована;
- API-контракты и внутренняя архитектура могут уточняться по мере развития проекта.

---

3. Справочник поддерживаемых команд

Направление	Сценарий	Ограничения
Telegram bot -> eXpress	`getMe`	-
Telegram bot -> eXpress	`setWebhook`	-
Telegram bot -> eXpress	`deleteWebhook`	`drop_pending_updates` принимается, но не очищает накопленные события внутри proxy.
Telegram bot -> eXpress	`getFile` и `/file/bot...`	Может вернуть `503`, если временно недоступен

Направление	Сценарий	Ограничения
		eXpress/BotX или staged-файл.
Telegram bot -> eXpress	`sendMessage`	`parse_mode=HTML` и `entities` поддержаны частично: bold, italic, strikethrough, code, pre, text_link, url. `tg://openmessage...` превращается в eXpress mention только если proxy может найти пользователя или чат.
Telegram bot -> eXpress	`sendPhoto`	Поддержан multipart upload и synthetic proxy `file_id` (`pxf_*`).
Telegram bot -> eXpress	`sendDocument`	Требует multipart upload. URL и настоящий Telegram `file_id` не скачиваются автоматически.
Telegram bot -> eXpress	`sendVideo`	Требует multipart upload. URL и настоящий Telegram `file_id` не скачиваются автоматически.
Telegram bot -> eXpress	`sendAudio`	Доставляется в eXpress как обычный документ.
Telegram bot -> eXpress	`sendAnimation`	Доставляется в eXpress как обычный документ.
Telegram bot -> eXpress	`sendSticker`	Нативный eXpress-sticker не создается; используется fallback в текст или emoji.
Telegram bot -> eXpress	`sendChatAction`	Typing-like actions маппятся в BotX `typing` ; неизвестные actions завершаются успешно как no-op с warning.
Telegram bot -> eXpress	`deleteMessage`	Можно удалить только сообщение, которое раньше прошло через proxy и имеет mapping `message_id -> sync_id` .

Направление	Сценарий	Ограничения
Telegram bot -> eXpress	`editMessageText`	Можно изменить только сообщение, которое раньше прошло через proxy. Форматирование и `tg://openmessage` имеют те же ограничения, что и `sendMessage`.
Telegram bot -> eXpress	`editMessageReplyMarkup`	Поддержано для inline-кнопок у сообщения, которое раньше прошло через proxy.
Telegram bot -> eXpress	`answerCallbackQuery`	-
Telegram bot -> eXpress	`reply_markup` в исходящих сообщениях	Поддержаны inline-кнопки, обычная клавиатура с text-кнопками и удаление клавиатуры. `ForceReply` и `request_*`-кнопки не поддержаны.
eXpress -> Telegram bot	`POST /command`	Команда принимается API, ставится в Kafka и доставляется worker'ом в webhook.
eXpress -> Telegram bot	Inline base64-вложения в JSON	Файл временно сохраняется API-контейнером и отдается через `getFile` / `/file`. Worker не монтирует stage volume.
eXpress -> Telegram bot	`POST /notification/callback`	Используется для delivery state и callback-контекста.
eXpress -> Telegram bot	Доставка в Telegram webhook	Только webhook-модель. Polling через `getUpdates` не поддержан.
eXpress -> Telegram bot	`system:event_deleted` callback	Считается service ack и не отправляется Telegram-боту.

---

4. Ссылки на API

Основные спецификации:

- Telegram Bot API
- eXpress Bot API
- eXpress BotX API

---

5. Основные термины

Telegram bot token - токен Telegram-бота, который используется в Telegram-compatible URL вида `/bot<TOKEN>/sendMessage`.

eXpress account - набор параметров eXpress-бота: `express_bot_id`, `express_cts_url`, `express_secret_key` и внутренний `account_id`. Значения `express_bot_id` и `express_secret_key` берутся из созданного eXpress-бота.

Binding - связка одного Telegram-бота с одним или несколькими eXpress account. Один Telegram-бот может принимать события от нескольких eXpress-ботов.

DLQ - Kafka topic для входящих команд, которые не удалось обработать после всех retry или не удалось разобрать как корректную задачу.

---

6. Быстрый старт

1. Создать бота или ботов в eXpress.

Для каждого eXpress-бота нужны значения:

Значение	Где используется
`<EXPRESS_BOT_ID>`	В `express_bot_id` binding-конфигурации и во входящих событиях eXpress как `bot_id`.
`<EXPRESS_SECRET_KEY>`	В `express_secret_key` binding-конфигурации; в eXpress UI может называться secret key или secret id.
`<EXPRESS_CTS_URL>`	В `express_cts_url`; базовый URL CTS/eXpress-контура, куда proxy отправляет BotX-запросы.

Один Telegram-бот может быть связан с несколькими eXpress-ботами: в этом случае добавьте несколько объектов в `express_accounts`.

2. Создать binding-связку Telegram-бота с eXpress-ботами через `PRECONFIGURED_BOTS_FILE` или `PRECONFIGURED_BOTS`.

По умолчанию используется `BINDING_SOURCE=preconfigured`: binding-связки загружаются только из `PRECONFIGURED_BOTS_FILE` или `PRECONFIGURED_BOTS` при старте API/worker, не сохраняются в Redis и не доступны к редактированию.

Если binding нужно создать после старта через Admin API, включите mutable-режим:

```
export BINDING_SOURCE=redis
export BINDING_ADMIN_ENABLED=true
```

В этом режиме binding-связки хранятся в Redis, а `POST/PUT/DELETE /admin/bindings` доступны для управления ими.

Предупреждение: при `BINDING_SOURCE=redis` в Redis сохраняются `telegram_bot_token` и `express_secret_key` из binding-конфигурации. Используйте этот режим только если Redis защищен сетевыми правилами, аутентификацией и политиками доступа, подходящими для хранения секретов.

Через shell:

```
export PRECONFIGURED_BOTS='[{"telegram_bot_token":"
<TELEGRAM_BOT_TOKEN>","telegram_profile":{"first_name":"eXpress
Bot","username":"express_bridge_bot"},"express_accounts":
[{"express_bot_id":"<EXPRESS_BOT_ID>","express_cts_url":"
<EXPRESS_CTS_URL>","express_secret_key":"<EXPRESS_SECRET_KEY>"}]}]'
```

Тот же binding можно хранить в JSON-файле. Перед запуском измените `config/preconfigured_bots.example.json`: замените примерные `telegram_bot_token`, `express_bot_id`, `express_cts_url` и `express_secret_key` на значения своего Telegram-бота и eXpress-ботов.

```
[
  {
    "telegram_bot_token": "<TELEGRAM_BOT_TOKEN>",
    "telegram_profile": {
      "first_name": "eXpress Bot",
      "username": "express_bridge_bot"
    },
```

```
    "express_accounts": [
      {
        "express_bot_id": "<EXPRESS_BOT_ID>",
        "express_cts_url": "<EXPRESS_CTS_URL>",
        "express_secret_key": "<EXPRESS_SECRET_KEY>"
      }
    ]
  }
]
```

```
export PRECONFIGURED_BOTS_FILE=./config/preconfigured_bots.local.json
```

В стандартном `docker-compose.yml` в API и worker передаются `PRECONFIGURED_BOTS_FILE` и `PRECONFIGURED_BOTS`. Директория `./config` монтируется в контейнеры как `/code/config:ro`, поэтому путь вида `./config/preconfigured_bots.json` из `.env` доступен внутри контейнера. Если заданы и файл, и `PRECONFIGURED_BOTS`, записи из файла загружаются первыми.

3. Обеспечить сетевую связность между proxy-контейнерами и развернутым Telegram-ботом.

- Telegram-бот должен отправлять Telegram Bot API-compatible запросы в API-контейнер `telegram-proxy`, например на `http://<telegram-proxy-host>/bot<TELEGRAM_BOT_TOKEN>/sendMessage`.
- Каждый worker-контейнер `worker` должен иметь доступ к webhook URL Telegram-бота, который будет передан через `setWebhook`.

Webhook URL может быть публичным HTTPS-адресом или внутренним адресом, доступным из runtime `worker`.

4. Поднять API, worker, Redis, Kafka и Kafka UI:

```
docker compose up --build -d
```

5. Создать Kafka topics:

```
docker compose exec kafka /opt/bitnami/kafka/bin/kafka-topics.sh \
  --bootstrap-server kafka:9092 \
  --create \
  --if-not-exists \
  --topic telegram-proxy-commands \
  --partitions 3 \
```

```
  --replication-factor 1

docker compose exec kafka /opt/bitnami/kafka/bin/kafka-topics.sh \
  --bootstrap-server kafka:9092 \
  --create \
  --if-not-exists \
  --topic telegram-proxy-commands-dlq \
  --partitions 3 \
  --replication-factor 1
```

6. Проверить health endpoint:

```
curl http://localhost:8080/health
```

7. Открыть Swagger:

```
http://localhost:8080/docs
```

8. Открыть Kafka UI:

```
http://localhost:8082
```

9. При необходимости масштабировать worker'ы:

```
docker compose up --build -d --scale worker=3
```

---

7. Локальный запуск для разработки

Установить зависимости:

```
poetry install
```

Поднять только инфраструктуру:

```
docker compose -f docker-compose.local.yml up -d
```

Создать Kafka topics:

```
docker compose -f docker-compose.local.yml exec kafka
/opt/bitnami/kafka/bin/kafka-topics.sh \
  --bootstrap-server kafka:29092 \
  --create \
  --if-not-exists \
  --topic telegram-proxy-commands \
  --partitions 3 \
  --replication-factor 1

docker compose -f docker-compose.local.yml exec kafka
/opt/bitnami/kafka/bin/kafka-topics.sh \
  --bootstrap-server kafka:29092 \
  --create \
  --if-not-exists \
  --topic telegram-proxy-commands-dlq \
  --partitions 3 \
  --replication-factor 1
```

Запустить API:

```
poetry run uvicorn app.main:app --reload --host 0.0.0.0 --port 8080
```

Запустить worker:

```
poetry run python -m app.worker
```

Для локального запуска из IDE `.env` должен смотреть на сервисы с host-машины:

```
REDIS_URL=redis://localhost:6379/0
KAFKA_BOOTSTRAP_SERVERS=localhost:9092
```

Compose-варианты:

Файл	Назначение
`docker-compose.yml`	API, worker, Redis, Kafka, Kafka UI.
`docker-compose.local.yml`	Только инфраструктура: Redis, Kafka, Kafka UI.
`docker-compose.local.api.yml`	Инфраструктура + API; API получает named volume `inbound_attachment_stage`.

Файл	Назначение
`docker-compose.local.worker.yml`	Инфраструктура + worker.

---

8. Переменные окружения

Variable	Default	Description
`APP_NAME`	`telegram-proxy`	Имя приложения.
`APP_HOST`	`0.0.0.0`	Адрес, на котором API слушает входящие подключения.
`APP_PORT`	`8000`	Внутренний порт API.
`LOG_LEVEL`	`INFO`	Уровень логирования.
`LOG_FORMAT`	`console`	Формат логов: `console` или `json`.
`REDIS_URL`	`redis://redis:6379/0`	Адрес Redis.
`REDIS_MAX_CONNECTIONS`	`200`	Максимальное число одновременных подключений к Redis.
`REDIS_KEY_PREFIX`	`telegram_proxy`	Префикс ключей Redis.
`REDIS_CALLBACK_TTL_SECONDS`	`86400`	Сколько секунд хранить контекст inline-кнопок.
`REDIS_PROCESSED_COMMAND_TTL_SECONDS`	`86400`	Сколько секунд помнить уже обработанные входящие события.
`FILE_CACHE_TTL_SECONDS`	`172800`	Сколько секунд хранить служебную информацию о файлах.
`FILE_CLEANUP_INTERVAL_SECONDS`	`300`	Как часто запускать очистку устаревших файловых записей.
`FILE_CLEANUP_BATCH_SIZE`	`200`	Максимум файловых записей за один проход очистки.
`FILE_FETCH_MAX_PARALLELISM`	`8`	Сколько файлов можно читать одновременно.

Variable	Default	Description
`INBOUND_ATTACHMENT_STAGE_DIRECTORY`	`/tmp/telegram-proxy/inbound-attachments`	Директория временного хранения входящих inline base64-вложений.
`INBOUND_ATTACHMENT_STAGE_TTL_SECONDS`	`172800`	Сколько секунд хранить staged-файлы после terminal-статуса.
`KAFKA_BOOTSTRAP_SERVERS`	`kafka:9092`	Адрес Kafka.
`KAFKA_COMMANDS_TOPIC`	`telegram-proxy-commands`	Topic входящих команд от eXpress.
`KAFKA_COMMANDS_DLQ_TOPIC`	`telegram-proxy-commands-dlq`	Topic команд, которые не удалось обработать.
`WORKER_CONSUMER_GROUP`	`telegram-proxy-workers`	Kafka consumer group worker'ов.
`WORKER_CLIENT_ID_PREFIX`	`telegram-proxy-worker`	Префикс имени worker-клиента в Kafka.
`WORKER_BATCH_SIZE`	`10`	Сколько событий worker берет из Kafka за один раз.
`WORKER_POLL_TIMEOUT_MS`	`5000`	Сколько миллисекунд worker ждет новые события.
`WORKER_MAX_RETRIES`	`5`	Сколько раз worker пытается обработать событие перед DLQ.
`WORKER_RETRY_BACKOFF_MS`	`1000`	Начальная пауза перед повторной попыткой.
`WORKER_RETRY_BACKOFF_MULTIPLIER`	`2.0`	Множитель паузы между retry.
`WORKER_HEARTBEAT_TTL_SECONDS`	`30`	Сколько секунд Redis считает worker живым после heartbeat.
`BINDING_SOURCE`	`preconfigured`	Источник binding-связок: `preconfigured` хранит их в памяти процесса, `redis` хранит в Redis. В режиме `redis` вместе с binding сохраняются `telegram_bot_token` и `express_secret_key`, поэтому Redis должен

Variable	Default	Description
		быть защищен как хранилище секретов.
`BINDING_ADMIN_ENABLED`	`false`	Разрешает `POST/PUT/DELETE /admin/bindings`; для runtime-управления используйте вместе с `BINDING_SOURCE=redis`. В этом режиме Admin API может записывать в Redis Telegram bot tokens и BotX/eXpress secret keys из binding payload.
`PRECONFIGURED_BOTS_FILE`	`unset`	Путь к JSON-файлу со связками ботов. Файл содержит `telegram_bot_token` и `express_secret_key`, поэтому должен храниться как secret config.
`PRECONFIGURED_BOTS`	`[]`	Связки ботов как JSON в переменной окружения. Значение содержит `telegram_bot_token` и `express_secret_key`, поэтому должно передаваться как secret.
`EXPRESS_AUTH_HEADER`	`Authorization`	HTTP-заголовок авторизации eXpress.
`EXPRESS_AUTH_PREFIX`	`Bearer`	Префикс авторизационного токена eXpress.
`EXPRESS_DIRECT_SYNC_ENABLED`	`false`	Использовать синхронную отправку BotX notifications.
`HTTP_TIMEOUT_SECONDS`	`10.0`	Общий таймаут HTTP-запросов.
`EXPRESS_HTTP_MAX_CONNECTIONS`	`200`	Максимум HTTP-соединений к eXpress.
`EXPRESS_HTTP_MAX_KEEPALIVE_CONNECTIONS`	`50`	Сколько keep-alive соединений к eXpress держать открытыми.

Variable	Default	Description
`EXPRESS_HTTP_KEEPALIVE_EXPIRY_SECONDS`	`30.0`	Через сколько секунд закрывается неиспользуемое соединение к eXpress.
`WEBHOOK_HTTP_MAX_CONNECTIONS`	`200`	Максимум HTTP-соединений для доставки в webhook.
`WEBHOOK_HTTP_MAX_KEEPALIVE_CONNECTIONS`	`50`	Сколько webhook keep-alive соединений держать открытыми.
`WEBHOOK_HTTP_KEEPALIVE_EXPIRY_SECONDS`	`30.0`	Через сколько секунд закрывается неиспользуемое webhook-соединение.

---

9. Хранение данных

8.1. Redis

Redis хранит рабочее состояние proxy, а не только cache: webhook settings, chat routes, message mappings, delivery state, file registry, stage state и служебные ключи worker'ов. При `BINDING_SOURCE=redis` там также хранятся binding-связки с `telegram_bot_token` и `express_secret_key`.

Во всех compose-файлах Redis запускается с AOF persistence:

```
command: ["redis-server", "--appendonly", "yes", "--appendfsync",
"everysec"]
```

Состояние хранится в named volume `redis_data`. Обычный `docker compose restart redis` или `docker compose down && docker compose up` сохраняет данные. Команда `docker compose down -v` или удаление volume удаляет рабочее состояние proxy, включая chat/message mappings, webhook settings и Redis-backed bindings.

Данные	Срок хранения по умолчанию	Настройка	Как очищается
Контекст inline-кнопок: `user_id`,	24 часа	`REDIS_CALLBACK_TTL_SECONDS`	Redis TTL

Данные	Срок хранения по умолчанию	Настройка	Как очищается
`chat_id`, `express_account_id`			
Маркеры обработанных входящих событий	24 часа	`REDIS_PROCESSED_COMMAND_TTL_SECONDS`	Redis TTL
File registry metadata для `getFile` и `/file`	48 часов	`FILE_CACHE_TTL_SECONDS`	Фоновая очистка
Inline base64 attachment stage metadata/files	48 часов после успешной доставки или окончательной ошибки	`INBOUND_ATTACHMENT_STAGE_TTL_SECONDS`	Фоновая очистка
Worker heartbeat	30 секунд	`WORKER_HEARTBEAT_TTL_SECONDS`	Redis TTL
Bot bindings, webhook settings, chat routes, message mappings, delivery state	Бессрочно	-	Только явное удаление или перезапись

---

10. Структура проекта

```
app/
  domain/          # доменные сущности и контракты репозиториев
  application/     # сценарии, сервисы, DTO
  infrastructire/  # настройки, DI, Redis/Kafka/HTTP адаптеры
  presentation/    # FastAPI routers, handlers, schemas
  main.py          # FastAPI app
  worker.py        # Kafka worker


tests/
  unit/            # unit-тесты отдельных use cases, сервисов и адаптеров
  integration/     # интеграционные тесты с внешними зависимостями
```

---

11. Проверки

Минимальная проверка кода и тестов:

```
python3 -m compileall app tests
.venv/bin/pytest -q
```

Через Poetry:

```
poetry run python -m compileall app tests
poetry run pytest -q
```