ТЗ-ИНСТРУКЦИЯ: ИТЕРАЦИЯ 1 - "Скелет инфраструктуры"

PhotoHelper MVP - Для разработчика Mid+ на MacBook M1

Цель итерации: Создать рабочий фундамент системы с базовой инфраструктурой, деплоем и простым Telegram ботом.

Длительность: 3-5 дней

Критерий успеха: Бот отвечает на `/start`, все сервисы работают, деплой настроен.

📋 ПРЕДВАРИТЕЛЬНАЯ ПОДГОТОВКА

1. Создание структуры проекта

```bash
# Создаём корневую директорию
mkdir photohelper && cd photohelper
git init

# Создаём структуру
mkdir -p backend/app/{models,schemas,services,api,core,utils}
mkdir -p backend/app/api/v1
mkdir -p backend/alembic/versions
mkdir -p frontend/src/{components,pages,hooks,services,utils,types}
mkdir -p docs
mkdir -p nginx

# Создаём базовые файлы
touch backend/requirements.txt
touch backend/.env.example
touch backend/Dockerfile
touch frontend/package.json
touch docker-compose.yml
touch .gitignore
touch README.md
touch .cursorrules

# Открываем в Cursor
cursor .
```

2. Копирование .cursorrules

Скопируй содержимое файла `.cursorrules` в корень проекта.

🔨 ЭТАП 1: DOCKER COMPOSE + POSTGRESQL + REDIS

Задача

Настроить базовую инфраструктуру в Docker: PostgreSQL, Redis, подготовить контейнеры для FastAPI и Celery.

Промт для Cursor

@Codebase Создай docker-compose.yml для проекта PhotoHelper:

Сервисы:
1. postgres: PostgreSQL 16 Alpine
   - Порт: 5432
   - База: photohelper_db
   - Пользователь и пароль из .env
   - Volume для персистентности данных
   - Healthcheck

2. redis: Redis 7 Alpine
   - Порт: 6379
   - Volume для персистентности
   - Healthcheck

3. backend: FastAPI приложение (пока без Dockerfile, добавим позже)
   - Зависимости: postgres, redis
   - Порт: 8000
   - Volume для hot reload
   - Environment из .env

4. celery_worker: Celery worker (пока без Dockerfile, добавим позже)
   - Зависимости: postgres, redis, backend
   - Environment из .env

Используй networks для связи сервисов.
Добавь restart: unless-stopped для всех сервисов.

Также создай:
- backend/.env.example с необходимыми переменными
- .gitignore с типичными исключениями для Python/Node.js/Docker

Реализация

1. **Создай файлы через Cursor** (используй промт выше)
2. **Скопируй .env.example → .env** и заполни реальные значения
3. **Проверь структуру файлов**

ТРОЙНОЕ ТЕСТИРОВАНИЕ #1

✅ Тест 1: Проверка конфигурации Docker Compose

```bash
# Валидация docker-compose.yml
docker-compose config

# Ожидаемый результат: YAML валиден, нет ошибок
```

✅ Тест 2: Запуск только БД сервисов

```bash
# Запускаем только postgres и redis
docker-compose up -d postgres redis

# Проверяем статус
docker-compose ps

# Проверяем логи
docker-compose logs postgres
docker-compose logs redis

# Ожидаемый результат: оба сервиса healthy
```

✅ Тест 3: Подключение к БД и Redis

```bash
```

```
# Подключение к PostgreSQL
docker-compose exec postgres psql -U photohelper_user -d photohelper_db -c "SELECT version();"

# Подключение к Redis
docker-compose exec redis redis-cli ping

# Ожидаемый результат:
# PostgreSQL: выводит версию
# Redis: PONG
```

Критерии прохождения:

- ✅ docker-compose config без ошибок
- ✅ Контейнеры postgres и redis запущены и healthy
- ✅ Успешное подключение к обеим БД

Если тесты не прошли: исправь ошибки, повтори все 3 теста заново.

🔨 ЭТАП 2: FASTAPI БАЗОВОЕ ПРИЛОЖЕНИЕ

Задача

Создать минимальное FastAPI приложение с эндпоинтами `/health` и автогенерируемой документацией `/docs`.

Промт для Cursor

```
@backend/app Создай структуру FastAPI приложения для PhotoHelper:

1. backend/app/main.py:
   - FastAPI приложение с title="PhotoHelper API"
   - Подключение CORS middleware (разрешить все origins для разработки)
   - Роут GET /health -> {"status": "ok", "timestamp": "..."}
   - Роут GET / -> {"message": "PhotoHelper API v1.0"}

2. backend/app/core/config.py:
   - Класс Settings на основе pydantic BaseSettings
   - Загрузка переменных окружения:
     * DATABASE_URL
     * REDIS_URL
     * TELEGRAM_BOT_TOKEN
     * SECRET_KEY
```

- Валидация обязательных полей

3. backend/app/core/database.py:
   - Async SQLAlchemy 2.0 engine
   - Async session factory
   - Dependency для получения session
   - Base для моделей

4. backend/requirements.txt:
   - fastapi[all]
   - sqlalchemy[asyncio]
   - asyncpg
   - alembic
   - python-dotenv
   - pydantic[email]
   - pydantic-settings
   - redis[hiredis]
   - celery
   - python-telegram-bot==20.7
   - pillow
   - httpx

5. backend/Dockerfile:
   - Multi-stage build
   - Python 3.11-slim
   - Оптимизация для ARM64 (M1)
   - User non-root
   - Health check на /health

Следуй .cursorrules, используй async/await, type hints везде.

Реализация

1. **Генерируй код через Cursor** (используй промт выше)
2. **Обнови docker-compose.yml** чтобы использовать Dockerfile для backend
3. **Пересобери контейнеры**

```bash
docker-compose build backend
docker-compose up -d backend
```

ТРОЙНОЕ ТЕСТИРОВАНИЕ #2

✅ Тест 1: Проверка работы FastAPI

```bash
# Проверка health endpoint
curl http://localhost:8000/health

# Ожидаемый результат: {"status":"ok","timestamp":"..."}

# Проверка корневого endpoint
curl http://localhost:8000/

# Ожидаемый результат: {"message":"PhotoHelper API v1.0"}
```

✅ Тест 2: Проверка документации

```bash
# Открой в браузере
open http://localhost:8000/docs

# Ожидаемый результат: Swagger UI с документацией
# Проверь, что есть эндпоинты: GET /, GET /health
```

✅ Тест 3: Проверка подключения к БД

```bash
# Смотрим логи backend
docker-compose logs backend

# Ожидаемый результат: нет ошибок подключения к PostgreSQL
# Можно добавить временный endpoint для проверки коннекта

# Проверка через psql что база создана
docker-compose exec postgres psql -U photohelper_user -d photohelper_db -c "\dt"
```

Критерии прохождения:

- ✅ Эндпоинты /health и / отвечают корректно
- ✅ Swagger UI доступен и отображает API
- ✅ Backend успешно стартует без ошибок

Если тесты не прошли: проверь логи, исправь, повтори тесты.

🔨 ЭТАП 3: POSTGRESQL ПОДКЛЮЧЕНИЕ + ТЕСТОВАЯ ТАБЛИЦА

Задача

Настроить Alembic, создать первую миграцию с тестовой таблицей, проверить работу БД.

Промт для Cursor

```
@backend/app/models @backend/alembic Настрой Alembic и создай первую модель:

1. backend/alembic.ini:
   - Инициализация Alembic
   - sqlalchemy.url берётся из env

2. backend/alembic/env.py:
   - Async конфигурация
   - Импорт Base из app.core.database
   - Импорт всех моделей
   - target_metadata = Base.metadata

3. backend/app/models/__init__.py:
   - Импорт всех моделей

4. backend/app/models/test_model.py:
   - Модель TestConnection(Base)
   - Поля: id (UUID, primary_key), message (String), created_at (DateTime)
   - __tablename__ = "test_connections"

5. Скрипт backend/scripts/init_db.sh:
   - alembic revision --autogenerate -m "Initial test table"
   - alembic upgrade head

Используй SQLAlchemy 2.0 async стиль, type hints, UUID для id.
```

Реализация

1. **Генерируй код через Cursor**
2. **Инициализируй Alembic внутри контейнера:**

```
bash
```

```
# Зайди в контейнер backend
docker-compose exec backend bash

# Инициализируй Alembic
alembic init alembic

# Выйди из контейнера
exit
```

3. **Примени промт Cursor для настройки конфигурации**
4. **Создай и примени миграцию:**

```
bash

docker-compose exec backend alembic revision --autogenerate -m "Initial test table"
docker-compose exec backend alembic upgrade head
```

ТРОЙНОЕ ТЕСТИРОВАНИЕ #3

✅ Тест 1: Проверка создания миграции

```
bash

# Проверь что миграция создалась
ls backend/alembic/versions/

# Ожидаемый результат: файл миграции с timestamp

# Проверь содержимое миграции
cat backend/alembic/versions/*_initial_test_table.py

# Ожидаемый результат: код создания таблицы test_connections
```

✅ Тест 2: Проверка применения миграции

```
bash
```

```
# Проверь таблицы в БД
docker-compose exec postgres psql -U photohelper_user -d photohelper_db -c "\dt"

# Ожидаемый результат: таблицы test_connections и alembic_version

# Проверь структуру таблицы
docker-compose exec postgres psql -U photohelper_user -d photohelper_db -c "\d test_connections"

# Ожидаемый результат: колонки id, message, created_at
```

✅ Тест 3: Создание тестовой записи через API

```
bash

# Добавь временный endpoint в backend/app/main.py:
# POST /test-db -> создаёт запись в test_connections

# После добавления endpoint:
curl -X POST http://localhost:8000/test-db \
  -H "Content-Type: application/json" \
  -d '{"message":"Test connection works!"}'

# Проверь что запись создалась
docker-compose exec postgres psql -U photohelper_user -d photohelper_db \
  -c "SELECT * FROM test_connections;"

# Ожидаемый результат: одна запись с сообщением
```

Критерии прохождения:

- ✅ Миграция создана и применена
- ✅ Таблица test_connections существует в БД
- ✅ Можно создавать записи через API (если добавил endpoint)

Если тесты не прошли: проверь настройки Alembic, повтори миграцию.

🔨 ЭТАП 4: REDIS ПОДКЛЮЧЕНИЕ + ТЕСТОВЫЙ КЛЮЧ

Задача

Подключить Redis, создать сервис для работы с кешем, протестировать запись/чтение.

Промт для Cursor

```
@backend/app/core Создай Redis сервис для PhotoHelper:

1. backend/app/core/redis.py:
   - Async Redis client
   - Функция get_redis_client() -> Redis
   - Dependency для FastAPI

2. backend/app/services/cache_service.py:
   - Класс CacheService
   - Методы:
     * async set(key: str, value: str, expire: int = 3600)
     * async get(key: str) -> str | None
     * async delete(key: str)
     * async ping() -> bool

3. Обнови backend/app/main.py:
   - Добавь эндпоинт GET /test-redis
   - Использует CacheService.ping()
   - Возвращает {"redis": "ok"} или {"redis": "error"}

4. Добавь эндпоинт POST /cache:
   - Принимает {"key": "...", "value": "..."}
   - Сохраняет в Redis
   - Возвращает {"status": "saved"}

5. Добавь эндпоинт GET /cache/{key}:
   - Возвращает значение из Redis
   - Или 404 если ключ не найден

Используй redis.asyncio, type hints, обработку ошибок.
```

Реализация

1. **Генерируй код через Cursor**
2. **Перезапусти backend:**

```
bash

docker-compose restart backend
```

ТРОЙНОЕ ТЕСТИРОВАНИЕ #4

✅ Тест 1: Проверка подключения к Redis

```bash
# Тест ping
curl http://localhost:8000/test-redis

# Ожидаемый результат: {"redis":"ok"}
```

✅ Тест 2: Запись в Redis через API

```bash
# Сохрани значение
curl -X POST http://localhost:8000/cache \
  -H "Content-Type: application/json" \
  -d '{"key":"test_key","value":"Hello Redis!"}'

# Ожидаемый результат: {"status":"saved"}

# Проверь через Redis CLI
docker-compose exec redis redis-cli GET test_key

# Ожидаемый результат: "Hello Redis!"
```

✅ Тест 3: Чтение из Redis через API

```bash
# Получи значение
curl http://localhost:8000/cache/test_key

# Ожидаемый результат: {"key":"test_key","value":"Hello Redis!"}

# Попробуй несуществующий ключ
curl http://localhost:8000/cache/nonexistent

# Ожидаемый результат: 404 Not Found
```

Критерии прохождения:

- ✅ Redis ping успешен
- ✅ Запись в Redis работает

- ✅ Чтение из Redis работает

Если тесты не прошли: проверь REDIS_URL, настройки клиента.

🔨 ЭТАП 5: CELERY WORKER + ТЕСТОВАЯ ЗАДАЧА

Задача

Настроить Celery worker, создать тестовую задачу, проверить асинхронное выполнение.

Промт для Cursor

```
@backend/app/core @backend/app/services Настрой Celery для PhotoHelper:

1. backend/app/core/celery_app.py:
   - Создание Celery app
   - Broker: Redis
   - Backend: Redis
   - Автообнаружение задач из app.services.tasks
   - Конфигурация:
     * task_serializer = 'json'
     * result_serializer = 'json'
     * accept_content = ['json']
     * timezone = 'UTC'
     * enable_utc = True

2. backend/app/services/tasks.py:
   - Декоратор @celery_app.task
   - Задача test_task(message: str):
     * Логирует сообщение
     * Спит 5 секунд (имитация работы)
     * Возвращает {"status": "completed", "message": message}
   - Задача add_numbers(a: int, b: int):
     * Возвращает сумму

3. backend/app/celery_worker.py (точка входа):
   - Импорт celery_app
   - __name__ == "__main__" -> celery_app.start()

4. Обнови docker-compose.yml:
   - Для celery_worker укажи command: celery -A app.core.celery_app worker --loglevel=info

5. Добавь в backend/app/main.py эндпоинт POST /test-celery:
   - Принимает {"message": "..."}
   - Запускает test_task.delay(message)
```

- Возвращает {"task_id": "..."}

6. Добавь эндпоинт GET /task-status/{task_id}:
   - Проверяет статус задачи
   - Возвращает {"status": "...", "result": "..."}

Используй type hints, логирование, обработку ошибок.

Реализация

1. **Генерируй код через Cursor**
2. **Обнови docker-compose.yml**
3. **Пересобери и перезапусти:**

```bash
docker-compose down
docker-compose build
docker-compose up -d
```

ТРОЙНОЕ ТЕСТИРОВАНИЕ #5

✅ **Тест 1: Проверка запуска Celery Worker**

```bash
# Проверь логи worker
docker-compose logs celery_worker

# Ожидаемый результат:
# - Celery worker запущен
# - Подключение к Redis успешно
# - Задачи зарегистрированы (test_task, add_numbers)

# Проверь статус
docker-compose ps celery_worker

# Ожидаемый результат: Up
```

✅ **Тест 2: Запуск задачи через API**

```bash
```

```
# Запусти тестовую задачу
curl -X POST http://localhost:8000/test-celery \
  -H "Content-Type: application/json" \
  -d '{"message":"Testing Celery!"}'

# Ожидаемый результат: {"task_id":"..."}
# Сохрани task_id

# Сразу проверь статус
curl http://localhost:8000/task-status/{task_id}

# Ожидаемый результат: {"status":"PENDING",...} или {"status":"STARTED",...}

# Подожди 6 секунд и проверь снова
sleep 6
curl http://localhost:8000/task-status/{task_id}

# Ожидаемый результат: {"status":"SUCCESS","result":{...}}
```

✅ Тест 3: Проверка логов задачи

```
bash

# Смотри логи worker во время выполнения задачи
docker-compose logs -f celery_worker

# Ожидаемый результат в логах:
# - Получена задача test_task
# - Лог сообщения "Testing Celery!"
# - Задача завершена успешно
```

Критерии прохождения:

- ✅ Celery worker запущен и подключен к Redis
- ✅ Задачи регистрируются и выполняются
- ✅ Статус задачи отслеживается через API

Если тесты не прошли: проверь конфигурацию Celery, Redis URL.

🔨 ЭТАП 6: TELEGRAM БОТ - БАЗОВАЯ НАСТРОЙКА

Задача

Зарегистрировать бота через BotFather, настроить webhook, создать обработчик команды `/start`.

Предварительные действия

1. **Регистрация бота:**
   - Открой Telegram, найди @BotFather
   - Отправь `/newbot`
   - Укажи имя: `PhotoHelper Bot`
   - Укажи username: `photohelper_yourname_bot`
   - Скопируй токен: `1234567890:ABCdefGHIjklMNOpqrsTUVwxyz`
2. **Добавь токен в .env:**

```bash
TELEGRAM_BOT_TOKEN=1234567890:ABCdefGHIjklMNOpqrsTUVwxyz
WEBHOOK_URL=https://yourdomain.com/webhook
```

Промт для Cursor

```
@backend/app Создай Telegram бота для PhotoHelper:

1. backend/app/bot/__init__.py - пустой файл

2. backend/app/bot/handlers.py:
  - Обработчик команды /start:
    * Отправляет "👋 Привет! Я PhotoHelper бот."
    * Добавляет кнопки Inline:
      - "📸 Я фотограф"
      - "👤 Я клиент"

3. backend/app/bot/bot.py:
  - Инициализация бота (Application.builder())
  - Регистрация handlers
  - Функция setup_webhook(webhook_url: str)
  - Функция start_polling() для локальной разработки

4. backend/app/api/v1/webhook.py:
  - Router для webhook
  - POST /webhook:
    * Принимает Update от Telegram
    * Обрабатывает через бота
    * Возвращает 200 OK
```

5. Обнови backend/app/main.py:
   - Include webhook router
   - При startup:
     * Если WEBHOOK_URL установлен -> setup_webhook()
     * Иначе -> start_polling() в background

6. backend/app/core/config.py:
   - Добавь TELEGRAM_BOT_TOKEN: str
   - Добавь WEBHOOK_URL: str | None = None

Используй python-telegram-bot 20.7, async/await, type hints.

Реализация

1. **Генерируй код через Cursor**
2. **Для локальной разработки используй polling** (закомментируй WEBHOOK_URL в .env)
3. **Перезапусти backend:**

```
bash

docker-compose restart backend
```

ТРОЙНОЕ ТЕСТИРОВАНИЕ #6

✅ **Тест 1: Проверка запуска бота**

```
bash

# Проверь логи backend
docker-compose logs backend | grep -i telegram

# Ожидаемый результат:
# - "Bot started in polling mode" или "Webhook set to ..."
# - Нет ошибок авторизации
```

✅ **Тест 2: Отправка /start в Telegram**

```
bash
```

```
# Ручной тест в Telegram:
1. Открой чат с ботом
2. Отправь /start
3. Ожидаемый результат:
   - Бот отвечает "👋 Привет! Я PhotoHelper бот."
   - Видны кнопки "📸 Я фотограф" и "👤 Я клиент"

# Проверь логи обработки команды
docker-compose logs backend | tail -20
```

✅ Тест 3: Проверка callback кнопок

```
bash

# Ручной тест в Telegram:
1. Нажми кнопку "📸 Я фотограф"
2. Ожидаемый результат:
   - Бот реагирует (пока может быть заглушка)
   - Нет ошибок в логах

# Проверь логи callback
docker-compose logs backend | grep -i callback
```

Критерии прохождения:

- ✅ Бот запущен и отвечает на сообщения
- ✅ Команда /start работает корректно
- ✅ Кнопки отображаются (обработка - позже)

Если тесты не прошли: проверь токен, права бота, логи ошибок.

🔨 ЭТАП 7: ДЕПЛОЙ НА VPS

Задача

Настроить VPS, задеплоить приложение, настроить Nginx, SSL, автозапуск.

Предварительные требования

- VPS с Ubuntu 22.04+
- Доменное имя (photohelper.example.com)
- SSH доступ

Промт для Cursor

@docs Создай документацию по деплою PhotoHelper:

1. docs/deployment.md:

   Секция "VPS Setup":
   - Установка Docker & Docker Compose
   - Настройка firewall (ufw)
   - Создание пользователя deploy

   Секция "Application Deploy":
   - Клонирование репозитория
   - Копирование .env.production
   - docker-compose -f docker-compose.prod.yml up -d

   Секция "Nginx Configuration":
   - Установка Nginx
   - Конфигурация reverse proxy
   - Настройка для FastAPI (/api, /docs)
   - Настройка для React SPA (/, /assets)

   Секция "SSL Certificates":
   - Установка Certbot
   - Получение Let's Encrypt сертификата
   - Auto-renewal настройка

   Секция "Systemd Service":
   - Создание photohelper.service
   - Автозапуск при перезагрузке

   Секция "CI/CD":
   - Базовый deploy.sh скрипт:
     * git pull
     * docker-compose build
     * docker-compose up -d
     * docker-compose exec backend alembic upgrade head

2. docker-compose.prod.yml:
   - Отличия от dev версии:
     * Без volume для hot reload
     * Restart: always
     * Healthchecks
     * Логирование в файлы

3. nginx/photohelper.conf:
   - Конфигурация Nginx
   - Proxy pass для backend
   - Static файлы для frontend
   - WebSocket support для Telegram

Включи команды для Ubuntu 22.04, безопасные настройки.

Реализация

Важно для MacBook M1: Docker образы должны быть multi-platform или собраны для linux/amd64.

1. **Генерируй документацию через Cursor**
2. **Следуй инструкциям в docs/deployment.md**

Пошаговый деплой

```bash
# На VPS
ssh deploy@photohelper.example.com

# Клонируй репозиторий
git clone https://github.com/yourusername/photohelper.git
cd photohelper

# Настрой .env
cp backend/.env.example backend/.env
nano backend/.env  # Заполни production значения

# Запусти
docker-compose -f docker-compose.prod.yml up -d

# Проверь статус
docker-compose -f docker-compose.prod.yml ps

# Проверь логи
docker-compose -f docker-compose.prod.yml logs
```

Настройка Nginx

```bash
```

```
# Установи Nginx
sudo apt update
sudo apt install nginx

# Скопируй конфигурацию
sudo cp nginx/photohelper.conf /etc/nginx/sites-available/
sudo ln -s /etc/nginx/sites-available/photohelper.conf /etc/nginx/sites-enabled/

# Проверь конфигурацию
sudo nginx -t

# Перезапусти
sudo systemctl restart nginx
```

Настройка SSL

```
bash

# Установи Certbot
sudo apt install certbot python3-certbot-nginx

# Получи сертификат
sudo certbot --nginx -d photohelper.example.com

# Auto-renewal уже настроен через systemd timer
```

Настройка webhook для бота

```
bash

# Обнови .env на сервере
WEBHOOK_URL=https://photohelper.example.com/webhook

# Перезапусти backend
docker-compose -f docker-compose.prod.yml restart backend
```

ТРОЙНОЕ ТЕСТИРОВАНИЕ #7

✅ Тест 1: Проверка доступности сервисов

```
bash
```

```
# На VPS проверь статус контейнеров
docker-compose -f docker-compose.prod.yml ps

# Ожидаемый результат: все контейнеры Up и Healthy

# Проверь доступность API
curl https://photohelper.example.com/health

# Ожидаемый результат: {"status":"ok",...}
```

✅ Тест 2: Проверка Nginx и SSL

```
bash

# Проверь SSL сертификат
curl -I https://photohelper.example.com

# Ожидаемый результат: HTTP/2 200, сертификат валиден

# Проверь редирект с HTTP на HTTPS
curl -I http://photohelper.example.com

# Ожидаемый результат: 301 или 302 редирект на HTTPS

# Проверь документацию API
open https://photohelper.example.com/docs
```

✅ Тест 3: Проверка webhook бота

```
bash

# Отправь /start боту в Telegram
# Ожидаемый результат: бот отвечает через webhook

# Проверь логи webhook
docker-compose -f docker-compose.prod.yml logs backend | grep webhook

# Ожидаемый результат: POST /webhook 200 OK

# Проверь что webhook установлен
# В логах должно быть: "Webhook set to https://photohelper.example.com/webhook"
```

Критерии прохождения:

- ✅ Все сервисы запущены на VPS

- ✅ API доступен через HTTPS
- ✅ SSL сертификат валиден
- ✅ Nginx корректно проксирует запросы
- ✅ Telegram webhook работает

Если тесты не прошли: проверь логи, firewall, DNS настройки.

🔨 ЭТАП 8: SYSTEMD + АВТОЗАПУСК

Задача

Настроить автоматический запуск приложения при перезагрузке сервера.

Реализация

Создай systemd service на VPS:

```bash
# На VPS
sudo nano /etc/systemd/system/photohelper.service
```

Содержимое файла:

```ini
[Unit]
Description=PhotoHelper Application
Requires=docker.service
After=docker.service

[Service]
Type=oneshot
RemainAfterExit=yes
WorkingDirectory=/home/deploy/photohelper
ExecStart=/usr/local/bin/docker-compose -f docker-compose.prod.yml up -d
ExecStop=/usr/local/bin/docker-compose -f docker-compose.prod.yml down
User=deploy
Group=deploy

[Install]
WantedBy=multi-user.target
```

Активируй сервис:

```bash
sudo systemctl daemon-reload
sudo systemctl enable photohelper.service
sudo systemctl start photohelper.service
```

ТРОЙНОЕ ТЕСТИРОВАНИЕ #8

✅ Тест 1: Проверка статуса systemd service

```bash
# Проверь статус сервиса
sudo systemctl status photohelper.service

# Ожидаемый результат:
# - Active: active (running)
# - Enabled: enabled
```

✅ Тест 2: Тест автозапуска при перезагрузке

```bash
# Перезагрузи сервер
sudo reboot

# Подожди 2-3 минуты, затем подключись снова
ssh deploy@photohelper.example.com

# Проверь что все контейнеры запустились автоматически
docker ps

# Ожидаемый результат: все контейнеры Running

# Проверь доступность API
curl https://photohelper.example.com/health
```

✅ Тест 3: Проверка логов автозапуска

```bash
```

```
# Проверь логи systemd
sudo journalctl -u photohelper.service -n 50

# Ожидаемый результат: успешный запуск без ошибок

# Проверь что бот работает
# Отправь /start боту в Telegram
```

Критерии прохождения:

- ✅ Systemd service запущен и enabled
- ✅ После перезагрузки все контейнеры стартуют автоматически
- ✅ Приложение доступно после перезагрузки

Если тесты не прошли: проверь права пользователя deploy, пути в service файле.

🔨 ЭТАП 9: CI/CD СКРИПТ ДЕПЛОЯ

Задача

Создать простой скрипт для обновления приложения на сервере.

Промт для Cursor

```
@docs Создай скрипт автоматического деплоя:

1. scripts/deploy.sh:
   - Цветной вывод (green, red, yellow)
   - Шаги:
     * Проверка что запущено на VPS (не локально)
     * git pull origin main
     * Проверка изменений в requirements.txt
     * Если есть изменения -> docker-compose build
     * docker-compose -f docker-compose.prod.yml up -d
     * Применение миграций: docker-compose exec backend alembic upgrade head
     * Restart сервисов
     * Healthcheck: curl /health
     * Вывод статуса всех контейнеров
   - Обработка ошибок на каждом шаге
   - Rollback если что-то пошло не так

2. scripts/rollback.sh:
```

- git checkout HEAD~1
- docker-compose down
- docker-compose up -d
- Восстановление миграций БД

3. .github/workflows/deploy.yml (если используешь GitHub):
- Триггер: push в main
- Job: Deploy to VPS
- SSH в VPS
- Запуск deploy.sh

Скрипты для bash, безопасные, с логированием.

Реализация

1. **Генерируй скрипты через Cursor**
2. **Сделай их исполняемыми:**

```bash
chmod +x scripts/deploy.sh
chmod +x scripts/rollback.sh
```

ТРОЙНОЕ ТЕСТИРОВАНИЕ #9

✅ **Тест 1: Локальная проверка скрипта**

```bash
# Проверь синтаксис
bash -n scripts/deploy.sh

# Ожидаемый результат: нет ошибок синтаксиса
```

✅ **Тест 2: Тестовый деплой на VPS**

```bash
```

```
# На VPS
cd /home/deploy/photohelper

# Сделай небольшое изменение (например, в README)
echo "# Test deploy" >> README.md
git add README.md
git commit -m "Test deploy script"
git push

# Запусти deploy скрипт
./scripts/deploy.sh

# Ожидаемый результат:
# - Успешный git pull
# - Контейнеры перезапущены
# - Healthcheck passed
# - Статус контейнеров: все Running
```

✅ Тест 3: Проверка rollback

```
bash

# Сделай ещё один коммит
echo "# Another change" >> README.md
git add README.md
git commit -m "Test rollback"
./scripts/deploy.sh

# Откатись назад
./scripts/rollback.sh

# Проверь что изменения отменены
cat README.md

# Ожидаемый результат: последняя строка не должна быть видна

# Проверь что приложение работает
curl https://photohelper.example.com/health
```

Критерии прохождения:

- ✅ Скрипт deploy.sh успешно обновляет приложение
- ✅ Все этапы деплоя выполняются без ошибок
- ✅ Скрипт rollback.sh откатывает изменения

Если тесты не прошли: добавь больше логирования, проверь пути и права.

📊 ФИНАЛЬНАЯ ПРОВЕРКА ИТЕРАЦИИ 1

Чеклист завершения

Пройдись по всем критериям:

- [ ] **Docker Compose:**
- [ ] PostgreSQL работает и доступен
- [ ] Redis работает и доступен
- [ ] Backend запускается без ошибок
- [ ] Celery worker обрабатывает задачи
- [ ] **FastAPI:**
- [ ] Эндпоинты /health и / работают
- [ ] Swagger UI доступен на /docs
- [ ] Подключение к PostgreSQL успешно
- [ ] Подключение к Redis успешно
- [ ] **База данных:**
- [ ] Alembic настроен корректно
- [ ] Миграции применяются
- [ ] Тестовая таблица создана
- [ ] Можно создавать записи
- [ ] **Celery:**
- [ ] Worker запущен
- [ ] Тестовая задача выполняется
- [ ] Статус задачи отслеживается
- [ ] **Telegram Bot:**
- [ ] Бот зарегистрирован в BotFather
- [ ] Команда /start работает
- [ ] Кнопки отображаются
- [ ] Webhook настроен (production) или polling работает (dev)
- [ ] **Деплой:**
- [ ] VPS настроен
- [ ] Nginx проксирует запросы
- [ ] SSL сертификат валиден
- [ ] Systemd service запускает приложение
- [ ] Автозапуск при перезагрузке работает
- [ ] Скрипт deploy.sh обновляет приложение

Итоговое тестирование системы

```bash
# 1. Проверь все контейнеры
docker-compose ps

# 2. Проверь API
curl https://photohelper.example.com/health
curl https://photohelper.example.com/docs

# 3. Проверь БД
docker-compose exec postgres psql -U photohelper_user -d photohelper_db -c "SELECT COUNT(*) FROM test_connections

# 4. Проверь Redis
docker-compose exec redis redis-cli PING

# 5. Запусти тестовую Celery задачу
curl -X POST https://photohelper.example.com/test-celery \
  -H "Content-Type: application/json" \
  -d '{"message":"Final test"}'

# 6. Проверь бота в Telegram
# Отправь /start
```

Критерии успеха итерации:

- ✅ Все контейнеры работают стабильно
- ✅ API доступен и отвечает
- ✅ База данных принимает запросы
- ✅ Redis кеширует данные
- ✅ Celery выполняет задачи
- ✅ Telegram бот отвечает на команды
- ✅ Деплой на VPS успешен
- ✅ Автозапуск работает
- ✅ Логи пишутся корректно

📝 КОММИТ И ПОДГОТОВКА К ИТЕРАЦИИ 2

Финальный коммит

bash

```
git add .
git commit -m "feat: iteration 1 - infrastructure skeleton

- Setup Docker Compose with PostgreSQL, Redis, FastAPI, Celery
- Configure FastAPI with health check and docs endpoints
- Setup Alembic migrations
- Connect PostgreSQL with test table
- Connect Redis with cache service
- Setup Celery worker with test tasks
- Create Telegram bot with /start command
- Deploy to VPS with Nginx and SSL
- Configure systemd auto-start
- Create deploy and rollback scripts

All services tested and working in production."

git push origin main
```

Документация

Создай краткий отчёт:

markdown

Итерация 1 - Завершена ✅

Реализовано
- ✅ Docker Compose инфраструктура
- ✅ FastAPI базовое приложение
- ✅ PostgreSQL + Alembic
- ✅ Redis кеширование
- ✅ Celery worker
- ✅ Telegram бот (базовый)
- ✅ Деплой на VPS
- ✅ Nginx + SSL
- ✅ Systemd автозапуск
- ✅ CI/CD скрипты

Технические метрики
- Контейнеров: 4 (postgres, redis, backend, celery_worker)
- API endpoints: 8
- Моделей БД: 1 (test_connections)
- Celery задач: 2
- Время деплоя: ~3 минуты

Следующая итерация
Итерация 2: Регистрация пользователей
- Модель User
- Выбор роли (фотограф/клиент)
- Сохранение в БД
- API для работы с пользователями

🎯 ИТОГО

Что получилось в итерации 1:

- ✅ Рабочая инфраструктура на Docker
- ✅ FastAPI с документацией
- ✅ PostgreSQL с миграциями
- ✅ Redis для кеша и очередей
- ✅ Celery для фоновых задач
- ✅ Telegram бот отвечает на /start
- ✅ Приложение задеплоено на VPS
- ✅ Автоматический деплой настроен

Время выполнения: 3-5 дней

Готовность к итерации 2: ✅

📚 ПОЛЕЗНЫЕ КОМАНДЫ

Docker

```bash
# Пересобрать все контейнеры
docker-compose build --no-cache

# Перезапустить конкретный сервис
docker-compose restart backend

# Посмотреть логи
docker-compose logs -f backend

# Зайти в контейнер
docker-compose exec backend bash

# Остановить всё и удалить volumes
docker-compose down -v
```

База данных

```bash
# Создать миграцию
docker-compose exec backend alembic revision --autogenerate -m "Description"

# Применить миграции
docker-compose exec backend alembic upgrade head

# Откатить миграцию
docker-compose exec backend alembic downgrade -1

# Посмотреть историю миграций
docker-compose exec backend alembic history
```

Celery

```bash
```

```
# Посмотреть активные задачи
docker-compose exec celery_worker celery -A app.core.celery_app inspect active

# Очистить очередь
docker-compose exec redis redis-cli FLUSHALL
```

Nginx

```
bash

# Проверить конфигурацию
sudo nginx -t

# Перезапустить
sudo systemctl restart nginx

# Посмотреть логи ошибок
sudo tail -f /var/log/nginx/error.log
```

Успехов в разработке! 🚀

При проблемах:

1. Проверь логи контейнеров
2. Проверь .env файл
3. Проверь доступность портов
4. Проверь права доступа
5. Прогони тесты заново

Следующая итерация: Регистрация пользователей (User модель + роли)