Лабораторная работа 5. Форматирование и линтинг проекта. Сборка образа проекта

Цель работы: Познакомиться со способами поддержки качества кода и сборки образа приложения.
Основа: Лабораторная работа выполняется на основе ЛР4, в репозитории после выполнения добавляем тэг `lab_5`.
Выполнил: Жунёв Андрей Александрович РИМ-150950

1. Настройка линтеров и форматеров

Настройка линтеров и форматеров необходима для поддержания качества кода и единообразия стиля в проекте. Линтеры (pylint) анализируют код на наличие ошибок и потенциальных проблем, а форматеры (black, isort) автоматически приводят код к единому стилю. Pre-commit хуки обеспечивают автоматическую проверку кода перед каждым коммитом, предотвращая попадание некачественного кода в репозиторий. Это экономит время на code review, уменьшает количество ошибок и обеспечивает единый стиль кода для всех разработчиков.

Установка зависимостей

Я использую в работе пакетный менеджер uv.
В pyproject.toml добавлены новые зависимости. Затем, установлены с помощью uv sync

```
18      "pytest-cov>=4.1.0",
19      "aiosqlite>=0.19.0",
20      "polyfactory>=2.0.0",
21      "greenlet>=3.0.0",
22      "pre-commit>=3.0.0",
23      "pylint>=3.0.0",
24      "black>=24.0.0",
25      "isort>=5.13.0",
26  ]
27
```

Настройка black и isort

- Добавлена секция [tool.isort]:
- profile = "black" — совместимость с black
- known_first_party = ["app"] — корневая папка работы

- skip = [] — список путей для пропуска
- Добавлена секция [tool.black]:
- line-length = 88 — длина строки
- target-version = ['py313'] — версия Python
- exclude — исключены
  папки: .git, .venv, pycache, htmlcov, tests, migrations, build, docker и
  другие служебные директории

pyproject.toml M

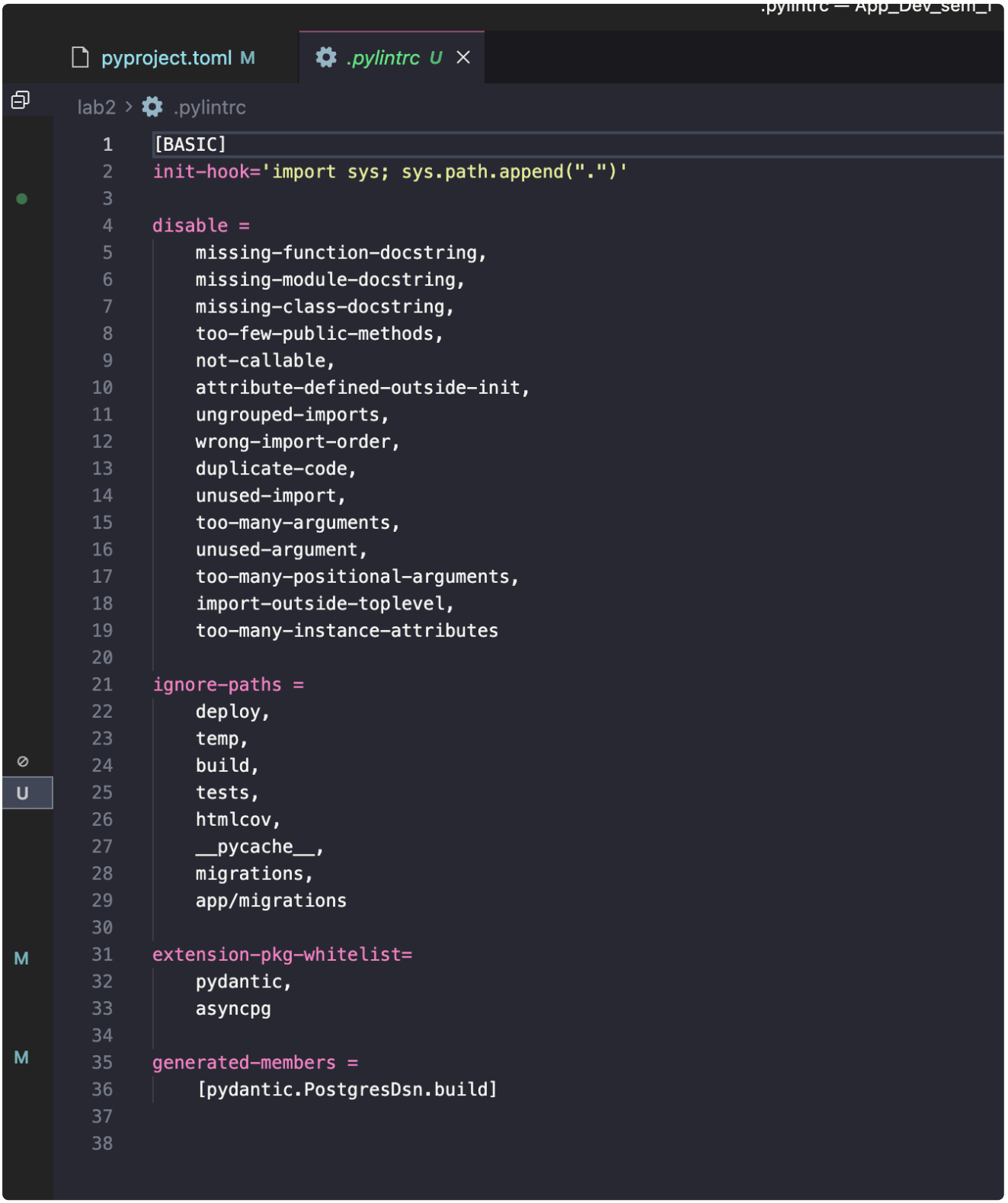
lab2 > pyproject.toml

```
testpaths = ["tests"]
asyncio_mode = "auto"
addopts = "--verbose --color=yes"

[tool.isort]
profile = "black"
known_first_party = ["app"]
skip = []

[tool.black]
line-length = 88
target-version = ['py313']
exclude = '''
/(
    \.git
  | \.hg
  | \.mypy_cache
  | \.tox
  | \.venv
  | _build
  | buck-out
  | build
  | docker
  | config
  | temp
  | __pycache__
  | htmlcov
  | tests
  | migrations
  )/
  '''

```

Настройка pylint

Создал файл .pylintrc с конфигурацией, адаптированной под структуру приложения (app вместо src):

- init-hook — добавлен путь к app для импортов
- disable — отключены проверки: отсутствие docstring, слишком мало публичных методов, неиспользуемые импорты, слишком много аргументов и другие
- ignore-paths —
  игнорируются: deploy, temp, build, tests, htmlcov, pycache, migrations, app/migratio
  ns
- extension-pkg-whitelist — добавлены: pydantic, asyncpg
- generated-members — добавлен: [pydantic.PostgresDsn.build]



Настройка pre-commit

Линтеры и форматеры настроены - хуки будут запускаться автоматически перед каждым коммитом.
Стоит обратить внимание на то, что мне пришлось добавить много игнорирований в pylint - связано это с тем, что в приложении присутствуют динамические атрибуты (в моем случае, ложные срабатывания происходят для ошибок import и no-member в

миграциях alembic)

pyproject.toml M .pre-commit-config.yaml U

lab2 > .pre-commit-config.yaml

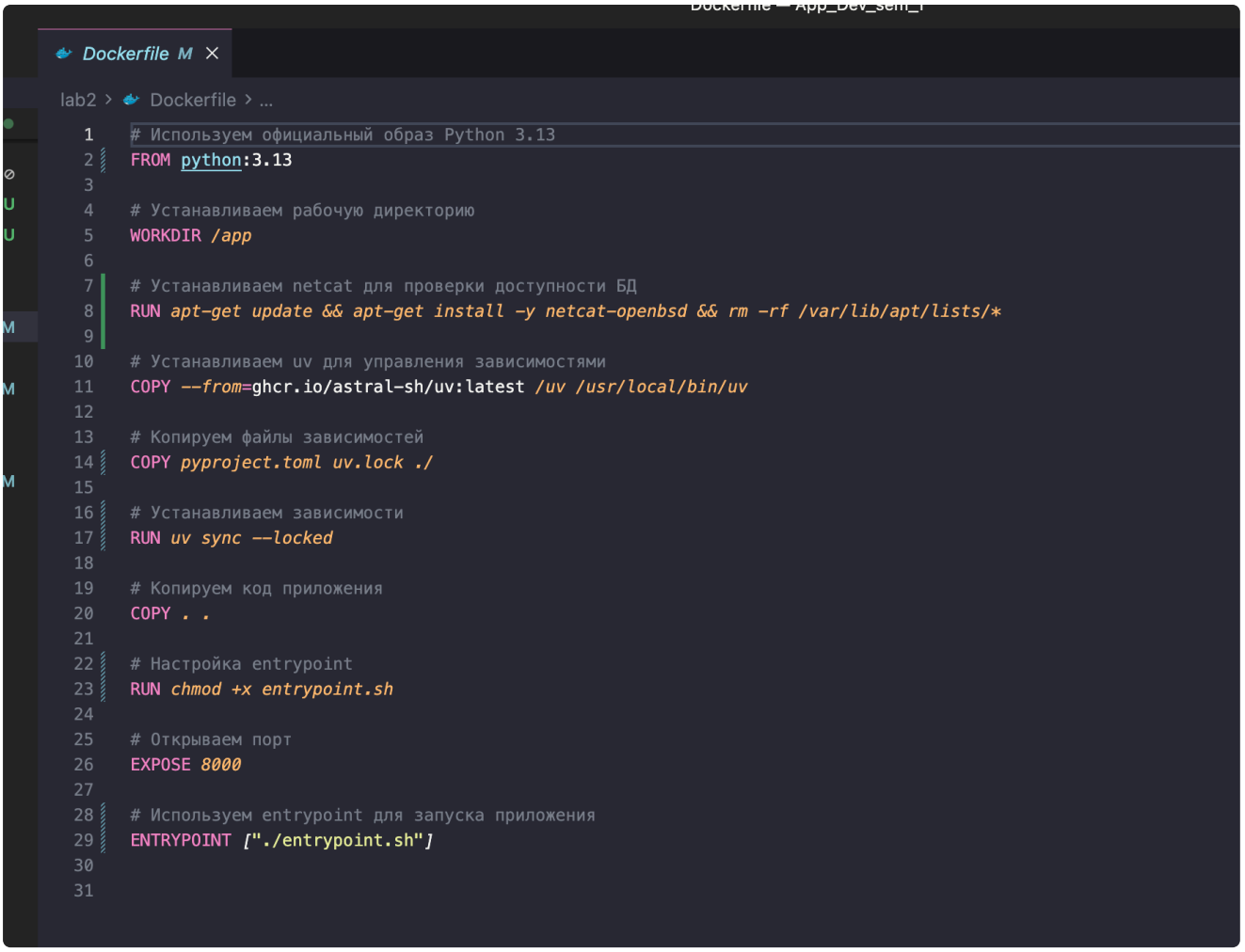
```yaml
fail_fast: false
repos:
  - repo: local
    hooks:
      - id: black
        name: Black
        entry: bash -c "cd $(git rev-parse --show-toplevel) && black lab2/app"
        pass_filenames: false
        language: system
  - repo: local
    hooks:
      - id: isort
        name: isort
        entry: bash -c "cd $(git rev-parse --show-toplevel) && isort lab2/app"
        pass_filenames: false
        language: system
  - repo: local
    hooks:
      - id: pylint
        name: pylint
        entry: bash -c "cd $(git rev-parse --show-toplevel) && pylint lab2/app"
        pass_filenames: false
        language: system

```

2. Обновление Dockerfile

Обновление Dockerfile необходимо для создания правильного образа приложения, который будет использоваться в Docker-контейнере. Dockerfile определяет процесс сборки образа, включая установку зависимостей, копирование кода и настройку окружения. Правильно настроенный Dockerfile обеспечивает воспроизводимость сборки, оптимизацию размера образа и корректную работу приложения в контейнере. Использование entrypoint позволяет автоматизировать процесс запуска приложения, включая ожидание готовности базы данных и применение миграций.

Согласно примеру из задания, обновил Dockerfile:

- Установлен netcat-openbsd для проверки доступности БД
- Установлен uv для управления зависимостями
- Скопированы pyproject.toml и uv.lock
- Используется uv sync --locked для установки зависимостей
- Скопирован код приложения
- Добавлен chmod +x entrypoint.sh для настройки entrypoint
- Используется ENTRYPOINT ["./entrypoint.sh"] вместо CMD
- Порт 8000 открыт через EXPOSE 8000

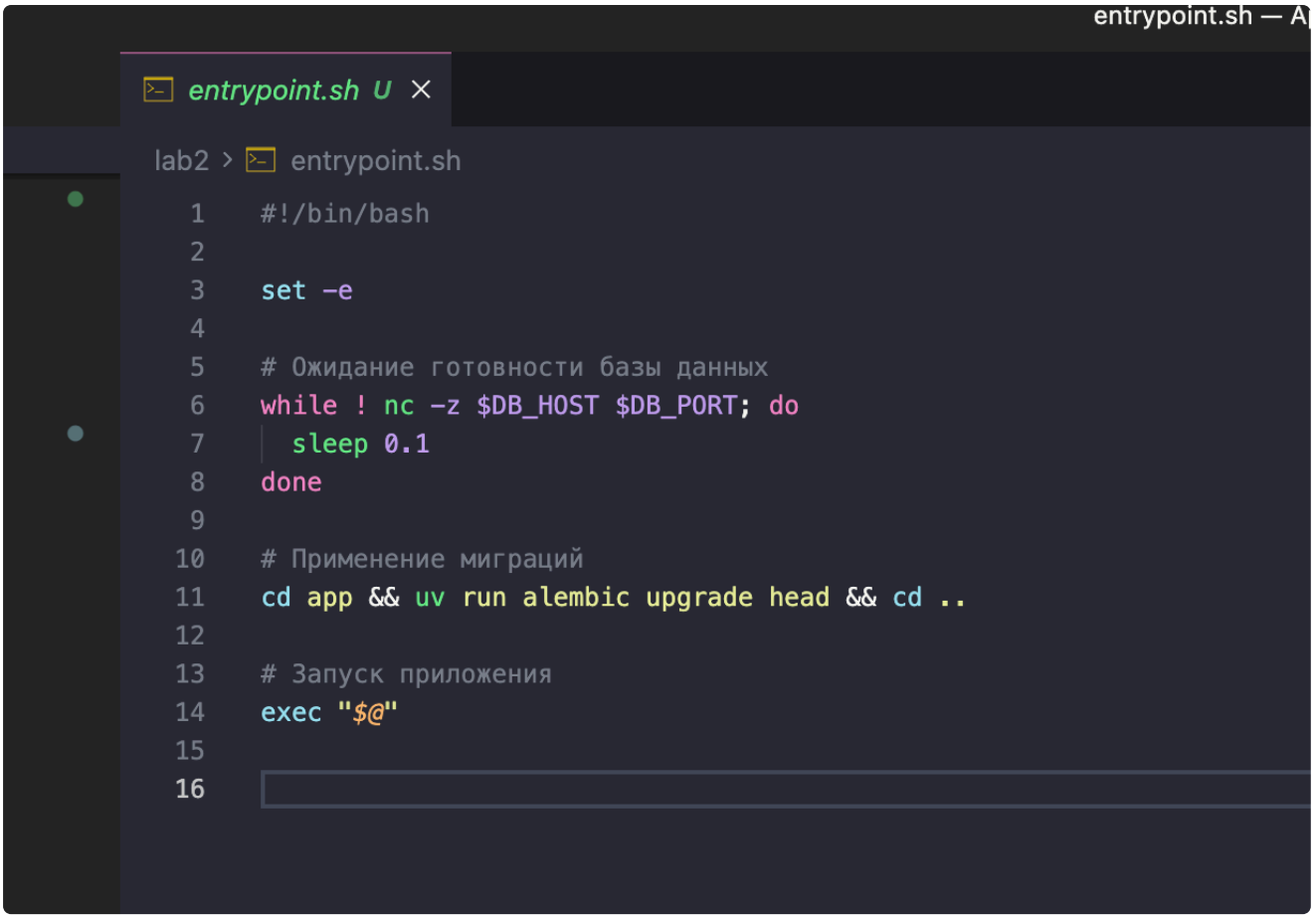


3. Создание entrypoint.sh

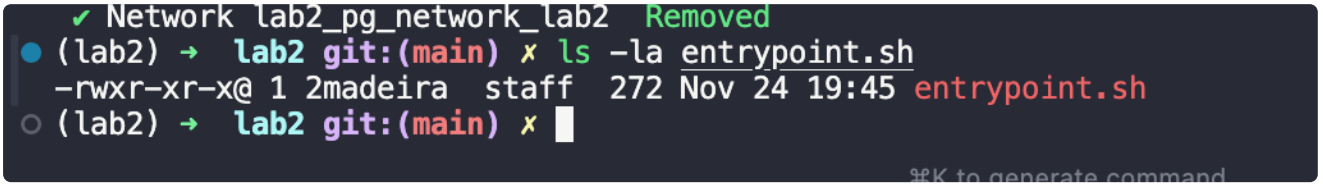
Создание entrypoint.sh необходимо для автоматизации процесса запуска приложения в Docker-контейнере. Скрипт entrypoint выполняет критически важные задачи перед запуском приложения: ожидает готовности базы данных и применяет миграции. Это гарантирует, что приложение не запустится до тех пор, пока база данных не будет готова принимать соединения, и схема базы данных будет актуальной. Без entrypoint пришлось бы вручную запускать миграции и проверять готовность БД, что усложняет процесс развертывания и может привести к ошибкам.

В корне репозитория (lab2) создан entrypoint.sh:

- Добавлен shebang: #!/bin/bash
- Реализовано ожидание готовности БД:
- Проверка доступности DB_HOST и DB_PORT
- Использование nc (netcat) для проверки соединения
- Цикл ожидания с задержкой sleep 0.1
- Добавлено применение миграций: cd app && uv run alembic upgrade head && cd ..
- Добавлен запуск приложения: exec "$@"
- Добавлен set -e для остановки при ошибке
- Скрипт сделан исполняемым: chmod +x entrypoint.sh



Проверены права доступа скрипта - скрипт исполняемый



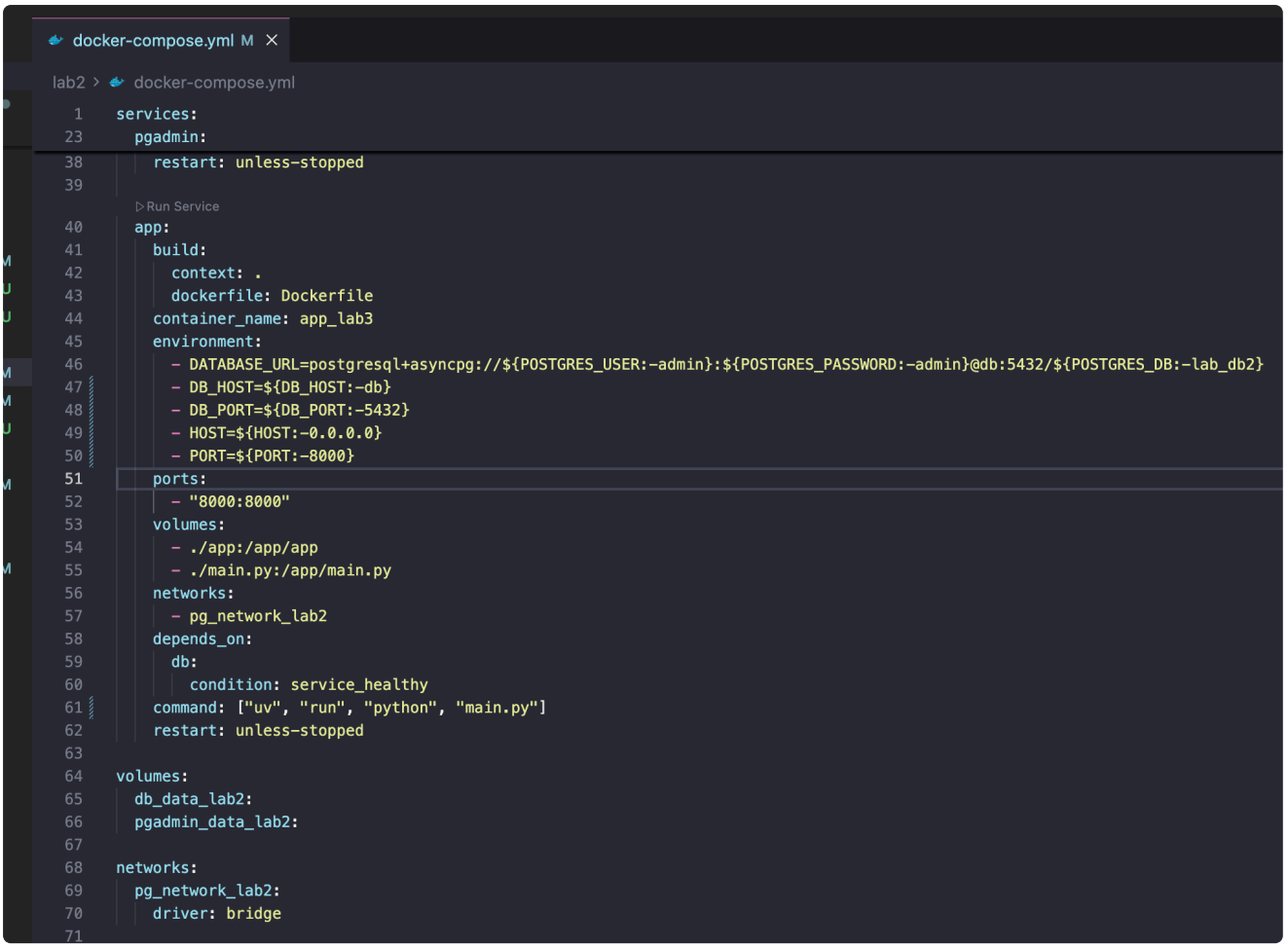
Переменные DB_HOST и DB_PORT будут добавлены в docker-compose.yml на этапе 4. Сейчас скрипт готов к использованию.

4. Обновление docker-compose.yml

Обновление docker-compose.yml необходимо для правильной настройки взаимодействия между сервисами (приложение, база данных, pgadmin). Docker Compose упрощает управление многоконтейнерными приложениями, определяя зависимости между сервисами, переменные окружения и порядок запуска. Правильная настройка `depends_on` гарантирует, что база данных запустится и станет готовой до запуска приложения. Использование переменных окружения из `.env` файла обеспечивает гибкость конфигурации для разных окружений (разработка, тестирование, production).

- docker-compose.yml обновлен согласно требованиям ЛР5
- Сервис app использует entrypoint.sh для запуска (через ENTRYPOINT в Dockerfile)
- Добавлены переменные DB_HOST и DB_PORT для entrypoint.sh (используются из .env)

- Команда запуска изменена на uv run python main.py
- Убрано ручное применение миграций из command (теперь в entrypoint.sh)
- Миграции будут применяться автоматически при старте контейнера через entrypoint.sh

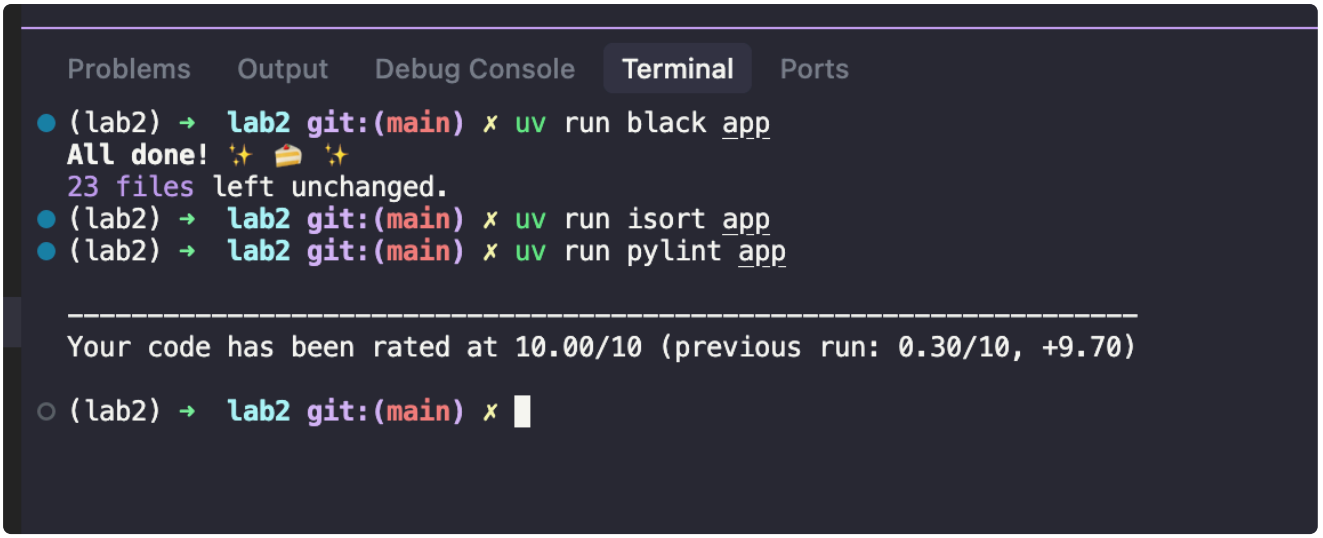


5. Тестирование

Тестирование и проверка необходимы для убеждения, что все настроенные инструменты работают корректно и приложение успешно запускается в Docker-контейнере. Проверка линтеров гарантирует, что код соответствует установленным стандартам качества. Тестирование pre-commit хуков подтверждает, что автоматическая проверка работает перед каждым коммитом. Проверка сборки и запуска Docker-образа позволяет убедиться, что приложение корректно работает в контейнеризованном окружении, миграции применяются автоматически, и все сервисы взаимодействуют правильно.

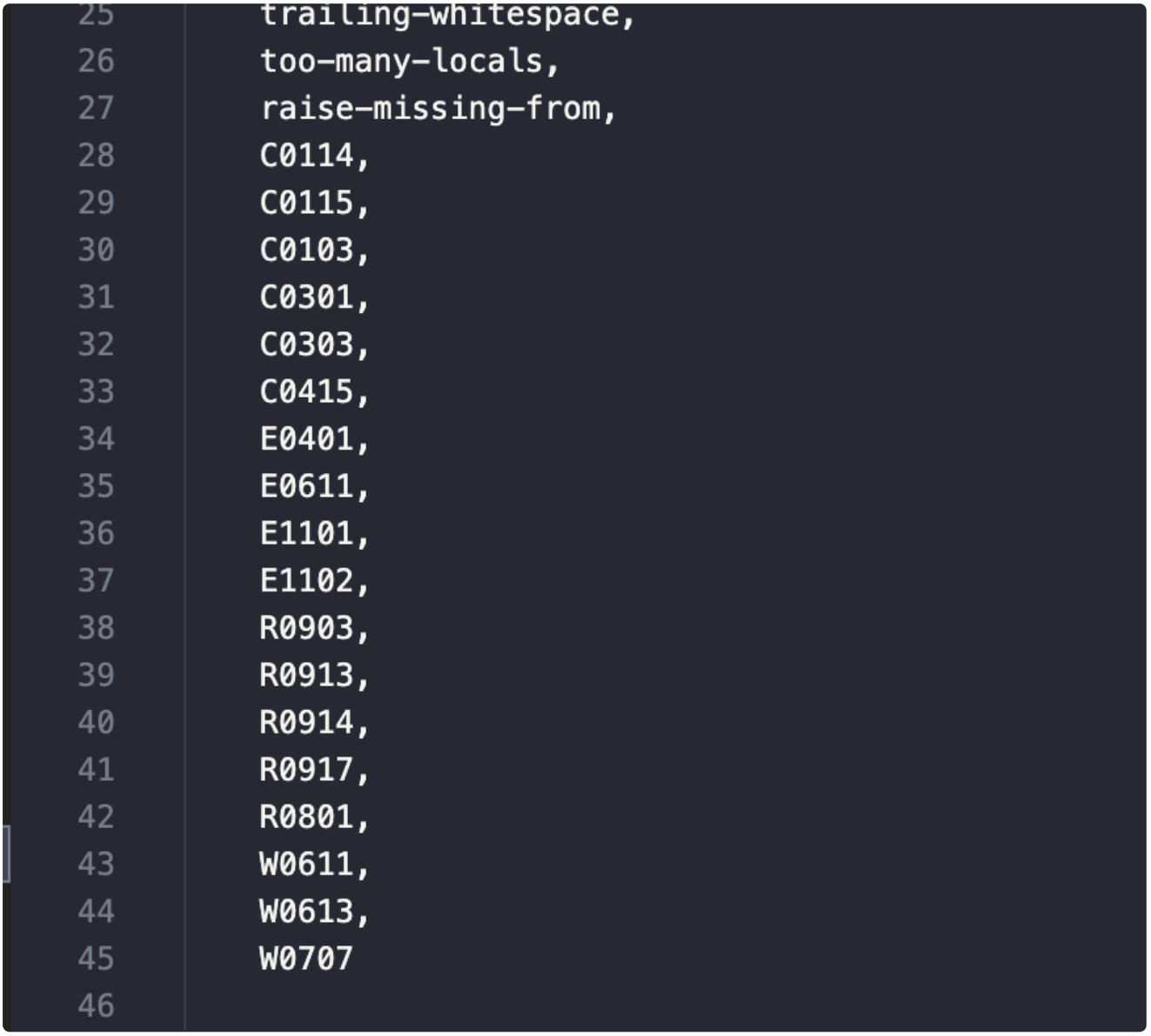
Локальная проверка линтеров

локальная проверка прошла успешно

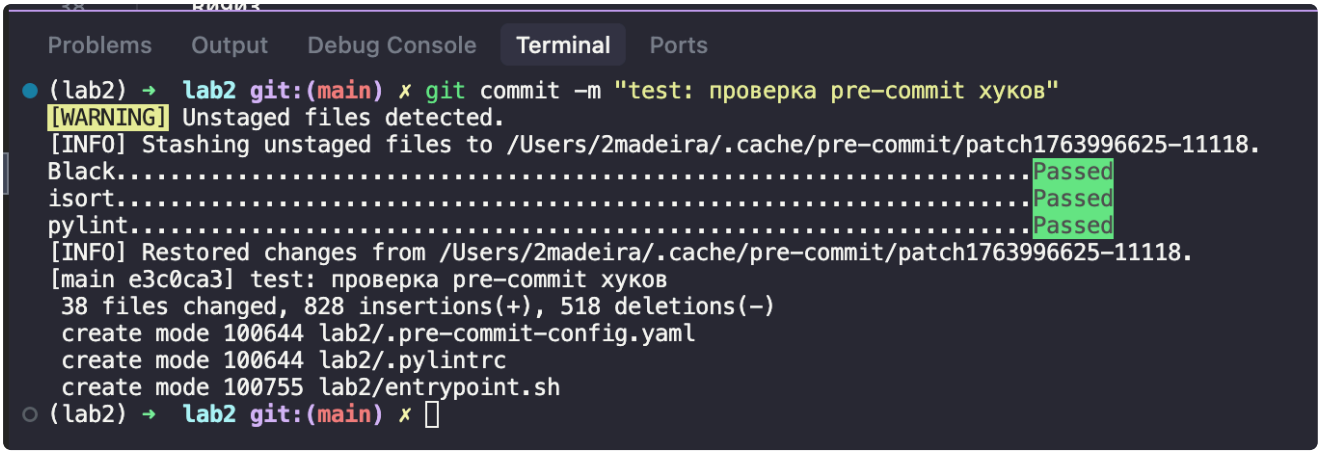


Проверка pre-commit

Возникла проблема с проверкой работы pylint при непосредственно коммите - black и isort успешно проходили. pylint продолжал сыпать ошибки, связанные с импортами и тд. Добавил конкретно коды ошибок в исключения.

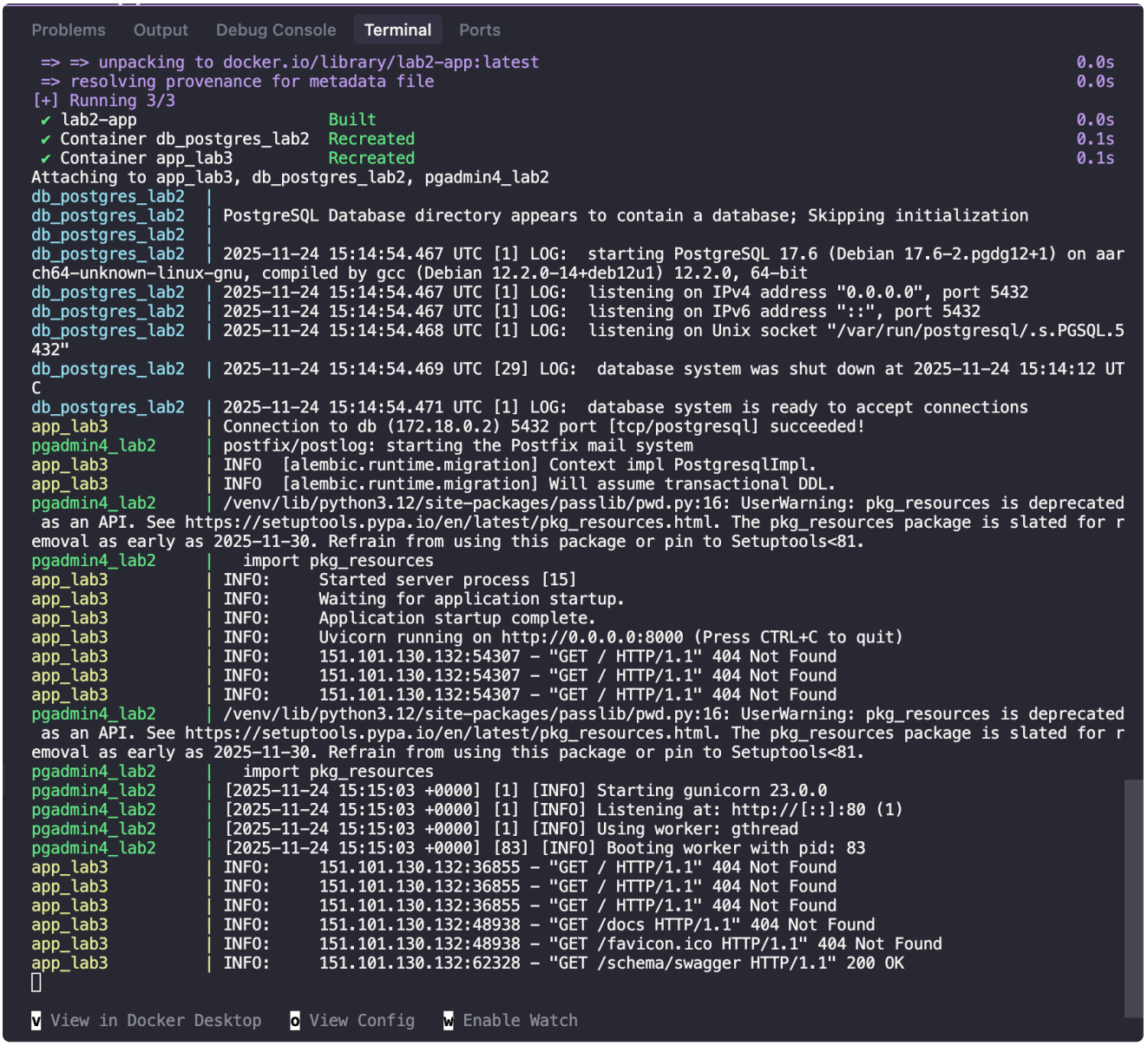


После этого pylint отработал.

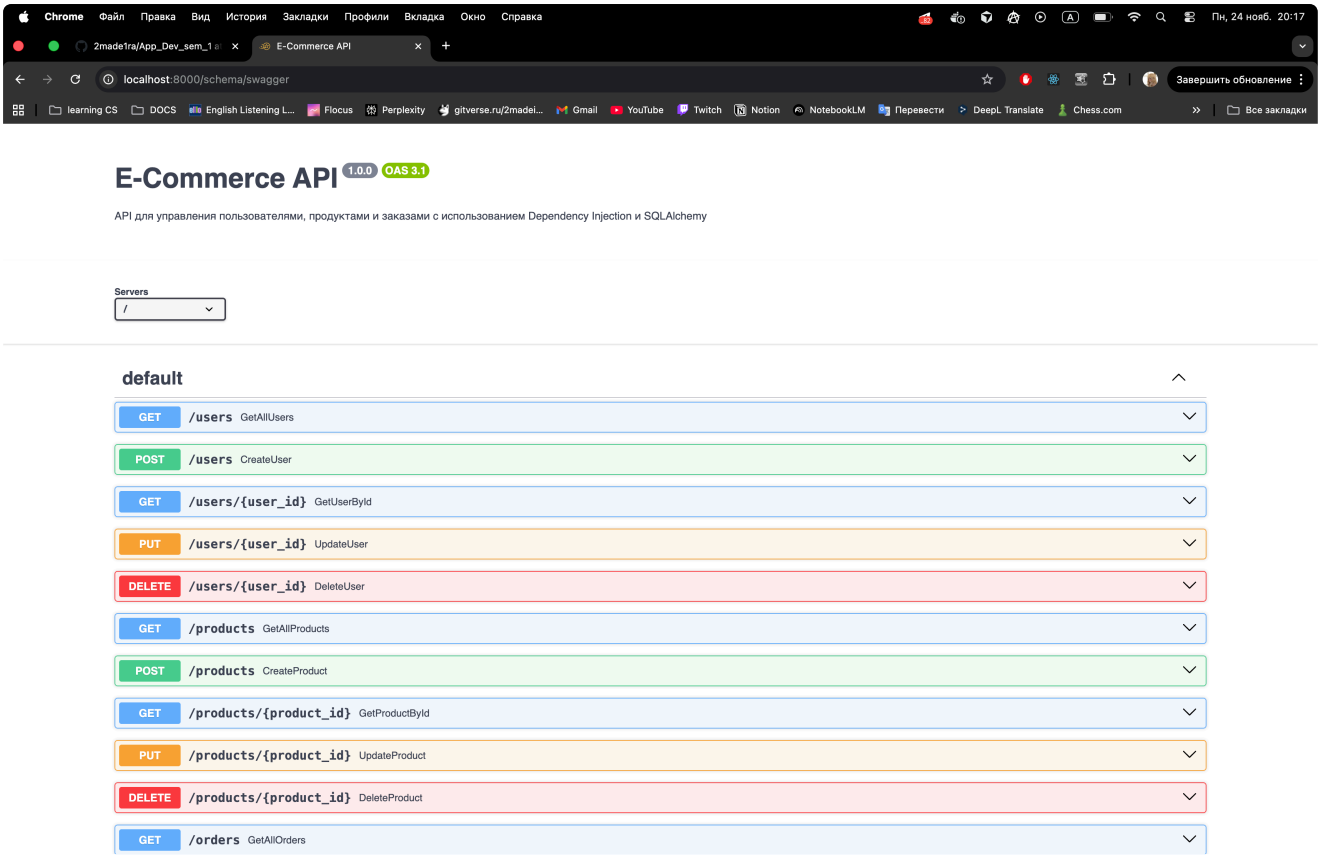


Запуск приложения через docker-compose

Приложение запускается через docker-compose up --build



swagger документация api доступна по http://localhost:8000/schema/swagger



- База данных запускается успешно
- Приложение ждет готовности БД (видно "Connection to db succeeded!")
- Миграции применяются автоматически (INFO [alembic.runtime.migration])
- Приложение запускается на порту 8000

Логи контейнера с приложением относительно впорядке - видно, что при обращении к странице со swagger статус код 200, что также подтверждает доступ.

```
app_lab3  | INFO:     151.101.130.132:54307 - "GET / HTTP/1.1" 404 Not Found
(lab2) ➜  lab2 git:(main) ✗ docker compose logs app
app_lab3  | Connection to db (172.18.0.2) 5432 port [tcp/postgresql] succeeded!
app_lab3  | INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
app_lab3  | INFO  [alembic.runtime.migration] Will assume transactional DDL.
app_lab3  | INFO:     Started server process [15]
app_lab3  | INFO:     Waiting for application startup.
app_lab3  | INFO:     Application startup complete.
app_lab3  | INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
app_lab3  | INFO:     151.101.130.132:54307 - "GET / HTTP/1.1" 404 Not Found
app_lab3  | INFO:     151.101.130.132:54307 - "GET / HTTP/1.1" 404 Not Found
app_lab3  | INFO:     151.101.130.132:54307 - "GET / HTTP/1.1" 404 Not Found
app_lab3  | INFO:     151.101.130.132:36855 - "GET / HTTP/1.1" 404 Not Found
app_lab3  | INFO:     151.101.130.132:36855 - "GET / HTTP/1.1" 404 Not Found
app_lab3  | INFO:     151.101.130.132:36855 - "GET / HTTP/1.1" 404 Not Found
app_lab3  | INFO:     151.101.130.132:48938 - "GET /docs HTTP/1.1" 404 Not Found
app_lab3  | INFO:     151.101.130.132:48938 - "GET /favicon.ico HTTP/1.1" 404 Not Found
app_lab3  | INFO:     151.101.130.132:62328 - "GET /schema/swagger HTTP/1.1" 200 OK
app_lab3  | INFO:     151.101.130.132:60006 - "GET /schema/swagger HTTP/1.1" 200 OK
(lab2) ➜  lab2 git:(main) ✗
```

Логи контейнера с БД также впорядке - база готова принимать подключения.

```
Problems  Output  Debug Console  Terminal  Ports

(lab2) ➜  lab2 git:(main) ✗ docker compose logs db | tail -20
db_postgres_lab2  |
db_postgres_lab2  | PostgreSQL Database directory appears to contain a database; Skipping initialization
db_postgres_lab2  |
db_postgres_lab2  | 2025-11-24 15:14:54.467 UTC [1] LOG:  starting PostgreSQL 17.6 (Debian 17.6-2.pgdg12+1) on aar
ch64-unknown-linux-gnu, compiled by gcc (Debian 12.2.0-14+deb12u1) 12.2.0, 64-bit
db_postgres_lab2  | 2025-11-24 15:14:54.467 UTC [1] LOG:  listening on IPv4 address "0.0.0.0", port 5432
db_postgres_lab2  | 2025-11-24 15:14:54.467 UTC [1] LOG:  listening on IPv6 address "::", port 5432
db_postgres_lab2  | 2025-11-24 15:14:54.468 UTC [1] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5
432"
db_postgres_lab2  | 2025-11-24 15:14:54.469 UTC [29] LOG:  database system was shut down at 2025-11-24 15:14:12 UT
C
db_postgres_lab2  | 2025-11-24 15:14:54.471 UTC [1] LOG:  database system is ready to accept connections
db_postgres_lab2  | 2025-11-24 15:19:54.576 UTC [27] LOG:  checkpoint starting: time
db_postgres_lab2  | 2025-11-24 15:19:54.589 UTC [27] LOG:  checkpoint complete: wrote 3 buffers (0.0%); 0 WAL file
(s) added, 0 removed, 0 recycled; write=0.005 s, sync=0.001 s, total=0.013 s; sync files=2, longest=0.001 s, avera
ge=0.001 s; distance=0 kB, estimate=0 kB; lsn=0/1A79428, redo lsn=0/1A793D0
(lab2) ➜  lab2 git:(main) ✗
```

6. Документация и работа с репозиторием

Обновлены оба README файла (в директории с приложением - lab2, и в головной директории)

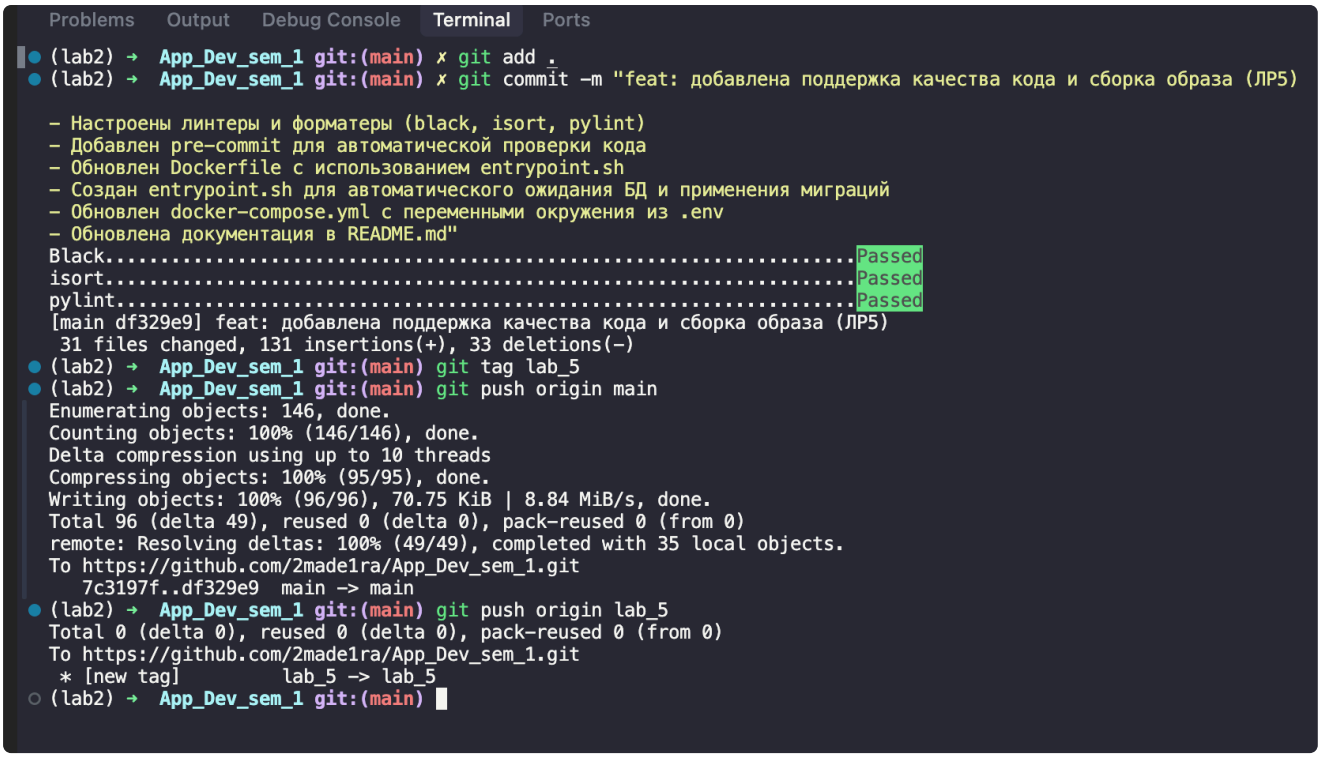
README.md — App_Dev_sem_1

README.md lab2 M README.md ./ M

lab2 > README.md > # Лабораторная работа 5

```
1  # Лабораторная работа 5
2
3  Веб-приложение на базе фреймворка Litestar с использованием Dependency Injection и SQLAlchemy ORM для работ
4
5  ## Оглавление
6
7  - [Быстрый старт](#быстрый-старт)
8  - [Установка и запуск](#установка-и-запуск)
9    - [Вариант 1: Запуск через Docker Compose (Рекомендуется)](#вариант-1-запуск-через-docker-compose-рекомен
10   - [Вариант 2: Локальный запуск (для разработки)](#вариант-2-локальный-запуск-для-разработки)
11 - [Доступ к сервисам](#доступ-к-сервисам)
12 - [API Документация](#api-документация)
13 - [API Эндпоинты](#api-эндпоинты)
14   - [Получить пользователя по ID](#получить-пользователя-по-id)
15   - [Получить список пользователей](#получить-список-пользователей)
16   - [Создать пользователя](#создать-пользователя)
17   - [Обновить пользователя](#обновить-пользователя)
18   - [Удалить пользователя](#удалить-пользователя)
19 - [Разработка](#разработка)
20   - [Линтеры и форматеры](#линтеры-и-форматеры)
21   - [Работа с Docker](#работа-с-docker)
22   - [Доступ к базе данных](#доступ-к-базе-данных)
23 - [Структура данных](#структура-данных)
24 - [Обработка ошибок](#обработка-ошибок)
25
26 ## Быстрый старт
27
```

Создан коммит с изменениями, добавлен tag lab_5. Все изменения запушены на github.



7. Вопросы

1. Что такое Docker-контейнер и чем он отличается от виртуальной машины?

Docker-контейнер — это изолированная среда выполнения приложения, которая использует ядро хостовой ОС и содержит только приложение и его зависимости.

Основное отличие от виртуальной машины заключается в типе виртуализации. Docker использует виртуализацию на уровне ОС, разделяя ядро хостовой системы между контейнерами, в то время как виртуальная машина использует аппаратную виртуализацию и требует полную операционную систему.

Контейнеры значительно легче (десятки МБ против ГБ у виртуальных машин), запускаются за секунды (в отличие от минут у VM) и имеют более высокую производительность, так как работают напрямую с ядром хоста, а не через гипервизор. Однако изоляция в контейнерах процессная, а не полная как в виртуальных машинах, где каждый экземпляр имеет отдельную ОС. Контейнеры требуют минимальных ресурсов, в то время как виртуальные машины потребляют значительно больше.

Пример: Контейнер с Python-приложением содержит только Python runtime и зависимости, а не всю операционную систему.

2. Как работает кеширование слоев в Docker и почему это важно?

Кеширование слоев — Docker сохраняет промежуточные результаты сборки образа (слои) и переиспользует их при повторной сборке, если инструкции не изменились.

Каждая инструкция в `Dockerfile` создает новый слой образа. При повторной сборке Docker проверяет, изменилась ли инструкция с момента последней сборки. Если инструкция не изменилась, Docker использует кешированный слой, что значительно ускоряет процесс. Если же инструкция изменилась, Docker пересобирает этот слой и все последующие слои, так как они могут зависеть от изменений.
Пример:

```
FROM python:3.13 # Слой 1 (кешируется)
COPY pyproject.toml . # Слой 2 (кешируется, если файл не изменился)
RUN uv sync --locked # Слой 3 (кешируется, если зависимости не изменились)
COPY . . # Слой 4 (пересобирается при любом изменении кода)
```

Кеширование слоев важно по нескольким причинам. Во-первых, оно значительно ускоряет сборку образа, так как не нужно пересобирать все слои при каждом изменении кода. Во-вторых, это повышает эффективность разработки, экономя время и вычислительные ресурсы. В-третьих, кеширование обеспечивает предсказуемость: одинаковые инструкции всегда дают одинаковые результаты, что упрощает отладку и воспроизведение сборок.

3. Что означает инструкция `depends_on` в docker-compose?

`depends_on` определяет порядок запуска сервисов в docker-compose, указывая зависимости между контейнерами. Эта инструкция гарантирует, что зависимые сервисы запускаются в правильной последовательности.
Существует два типа зависимостей. Простая форма `depends_on: [service]` запускает сервис после старта зависимого контейнера, но не ждет его полной готовности к работе. Более надежный вариант `depends_on: { service: { condition: service_healthy } }` ожидает, пока зависимый сервис станет "здоровым" согласно настройкам healthcheck, что гарантирует его полную готовность перед запуском зависимого сервиса/
Пример:

```
services:
    app:
        depends_on:
            db:
                condition: service_healthy # Ждет готовности БД
    db:
        healthcheck:
            test: ["CMD-SHELL", "pg_isready -U admin"]
```

Использование `depends_on` необходимо для гарантии правильного порядка запуска сервисов. Это предотвращает ошибки подключения, так как приложение не запустится до тех пор, пока база данных не будет готова принимать соединения. Кроме того, это упрощает управление зависимостями между сервисами, делая конфигурацию более понятной и надежной.

4. Почему миграции БД выполняются в `entrypoint.sh`, а не во время сборки образа?

Миграции базы данных выполняются в `entrypoint.sh`, а не во время сборки образа, по нескольким важным причинам. Во-первых, база данных недоступна при сборке образа, так как Docker образ собирается без запущенных сервисов, и база данных еще не существует в этот момент.

Во-вторых, параметры подключения к базе данных (хост, порт, пароль) могут отличаться в разных окружениях (разработка, тестирование, production), что требует динамической конфигурации. В-третьих, выполнение миграций в entrypoint обеспечивает гибкость — один и тот же образ можно использовать с разными базами данных без пересборки. Наконец, это гарантирует правильный порядок выполнения: миграции должны выполняться после запуска базы данных, но перед запуском приложения.

Альтернатива (неправильная):

```
# ❌ Неправильно - БД недоступна при сборке
RUN alembic upgrade head
```

Правильный подход:

```
# ✅ Правильно - в entrypoint.sh после запуска БД
while ! nc -z $DB_HOST $DB_PORT; do
    sleep 0.1
done
alembic upgrade head
```

5. Что произойдет, если миграции завершатся ошибкой при запуске контейнера?

Если миграции завершатся ошибкой при запуске контейнера, контейнер остановится благодаря директиве `set -e` в `entrypoint.sh`, которая заставляет скрипт завершаться с ошибкой при любой неудачной команде. В результате приложение не запустится, так как команда `exec "$@"` не выполнится из-за преждевременного завершения скрипта.

Контейнер перейдет в статус `Exited`, что можно увидеть через команду `docker compose ps`. Причина ошибки будет видна в логах контейнера, которые можно просмотреть с помощью команды `docker compose logs app`. Это позволяет быстро диагностировать проблему и понять, что именно пошло не так при применении миграций.

Пример обработки:

```
#!/bin/bash
set -e # Остановка при любой ошибке

alembic upgrade head # Если ошибка — скрипт остановится здесь
exec "$@" # Эта строка не выполнится
```

Такой подход имеет несколько преимуществ. Во-первых, он предотвращает запуск приложения с несовместимой схемой базы данных, что могло бы привести к более серьезным ошибкам во время работы. Во-вторых, проблему легко диагностировать через логи, так как ошибка миграции будет явно видна. В-третьих, это реализует принцип fail-fast, когда ошибка обнаруживается сразу при запуске, а не во время работы приложения, что упрощает отладку и предотвращает потенциальные проблемы с данными.

6. В чем разница между линтерами (flake8) и форматерами (black)?

Линтеры (например, flake8, pylint) анализируют код на наличие ошибок, потенциальных проблем и нарушений стиля, но не изменяют код. Они выполняют статический анализ и выдают предупреждения о проблемах в логике, стиле кодирования и потенциальных ошибках. Форматеры (например, black, isort) автоматически изменяют код, приводя его к единому стилю форматирования, но не проверяют логику или наличие ошибок.

Основное различие заключается в действии: линтеры только анализируют и предупреждают, в то время как форматеры выполняют автоматическое форматирование. Линтеры не изменяют код, а форматеры изменяют его. Линтеры проверяют логику, стиль и ошибки, а форматеры работают только с форматированием (отступы, пробелы, длина строк). К распространенным линтерам относятся pylint, flake8, mypy, а к форматерам — black, isort, autopep8.

Пример работы линтера:

```
# pylint найдет проблему, но не исправит
unused_var = 10 # W0612: Unused variable
```

Пример работы форматера:

```
# black автоматически исправит
x=1+2 # Стало: x = 1 + 2
```

В работе используются оба типа инструментов. Форматер `black` форматирует код, приводя к единому стилю отступы, пробелы и длину строк. Форматер `isort` сортирует импорты согласно установленным правилам. Линтер `pylint` проверяет качество кода, выявляя ошибки, проблемы со стилем и сложностью кода.

7. Как pre-commit хуки помогают в разработке?

Pre-commit хуки — это скрипты, которые автоматически выполняются перед каждым коммитом в Git. Они помогают в разработке несколькими способами.

Во-первых, pre-commit хуки обеспечивают автоматическую проверку кода перед коммитом без необходимости ручных действий. Это гарантирует, что все разработчики используют одинаковые правила форматирования, создавая единый стиль кода в проекте. Во-вторых, хуки позволяют обнаруживать проблемы на раннем этапе, до того как код попадет в репозиторий, что упрощает их исправление. В-третьих, это экономит время разработчиков, так как не нужно запускать проверки вручную перед каждым коммитом. Наконец, pre-commit хуки предотвращают коммит некачественного кода, автоматически блокируя коммиты, которые не соответствуют установленным стандартам.

Пример работы:

```
git commit -m "новый функционал"
# Автоматически запускается:
# 1. black — форматирует код
# 2. isort — сортирует импорты
# 3. pylint — проверяет качество
# Если все ОК — коммит создается
# Если ошибки — коммит отклоняется
```

Настройка в проекте:

```
# .pre-commit-config.yaml
repos:
    - repo: local
      hooks:
      - id: black
      entry: black app
      - id: pylint
      entry: pylint app --ignore=migrations
```

Использование pre-commit хуков дает несколько преимуществ. Код всегда соответствует установленным стандартам, так как хуки автоматически форматируют его перед коммитом. Это сокращает время на code review, так как не возникает споров о стиле кода — все уже отформатировано единообразно. Меньше ошибок попадает в основную ветку, так как проблемы обнаруживаются и исправляются до коммита. Наконец, хуки автоматизируют рутинные задачи проверки и форматирования, позволяя разработчикам сосредоточиться на написании кода.